

Índice

Tipos de datos estructurados y cadena.....	2
1 Tipos de datos estructurados	2
2 Arreglos Unidimensionales.....	2
2.1 Declaración de Arreglos Unidimensionales.....	2
2.1.1 Inicialización	3
2.2 Arreglos Unidimensionales como Parámetro.....	4
2.3 Arrays y Punteros (No dar)	7
2.4 Aritmética de Punteros (No dar)	7
3 Arreglos Bidimensionales	8
4 Tipo de Dato Cadena (string).....	8
4.1 Declaración de Cadenas	8
4.2 Inicialización	9
4.3 Funciones de biblioteca para el tratamiento de cadenas	9
4.3.1 Entrada y Salida de Cadenas con Formato	9
4.3.2 Entrada y Salida de Cadenas sin Formato	10
4.3.3 Lectura de Escalares y Cadenas	11
4.3.4 Funciones para Trabajar con Cadenas de Caracteres.....	13
4.3.5 Funciones para Conversión de Datos	18
4.3.6 Funciones para Conversión de Caracteres	21
5 Estructuras.....	21
5.1 Definición.....	21
5.2 Declaración	21
5.3 Definiciones de Variables de Estructuras	22
5.4 Inicialización de una Declaración de Estructura	22
5.5 Tamaño de una Estructura	22
5.6 Acceso a Estructuras.....	23
5.7 Creación de Sinónimos o alias “typedef”	23
5.8 Estructuras Anidadas	24
5.9 Estructuras como Parámetro.....	25
5.10 Estructuras con Arreglos.....	25
6 Uniones.....	26
6.1 Declaración de Uniones.....	26

Unidad 4

Tipos de datos estructurados y cadena

Temario: Tipos de datos estructurados: vectores, registros, uniones. Almacenamiento en memoria. Operaciones sobre tipos de datos estructurados. Cadena, concepto, almacenamiento en memoria. Funciones de biblioteca para el manejo de cadenas.

Bibliografía:

- [1] Luis Joyanes Aguilar e Ignacio Martínez - Programación en C. Metodología, algoritmos y estructuras de datos.
- [2] Cairó, Osvaldo. Fundamentos de Programación. Piensa en C.
- [3] Andrés Marzal e Isabel Gracia - Introducción a la programación con C .
- [4] Francisco Javier Ceballos Sierra - Curso de Programación en C/C++.
- [5]<http://www.taringa.net/posts/ciencia-educacion/14450390/Lectura-de-cadenas-en-C.html>.

1 Tipos de datos estructurados

Los tipos de datos se pueden clasificar en simples y estructurados. Los tipos de datos simples se vinculan a variables que pueden contener un solo dato por vez, en cambio, las variables asociadas a tipos de datos estructurados permiten registrar varios datos a la vez. En esta unidad se presentan algunos tipos de datos estructurados de uso frecuente.

2 Arreglos Unidimensionales

Definición: "Un arreglo unidimensional es una colección finita, homogénea y ordenada de datos, en la que se hace referencia a cada elemento del arreglo por medio de un índice, que indica su ubicación en el arreglo".

- *Finita:* Tiene un número limitado de elementos.
- *Homogénea:* Todos los elementos del arreglo son del mismo tipo.
- *Ordenada:* Se determina cual es el primer elemento, el segundo y así sucesivamente.

2.1 Declaración de Arreglos Unidimensionales

Los arreglos ocupan espacio en memoria que se reserva en el momento de realizar la declaración del arreglo. Por ejemplo:

```
int A[10]   define de un arreglo que contiene 10 números enteros.
```

```
Float B[5] define un arreglo de tipo real de 5 elementos.
```

Otra alternativa para la declaración de arreglos es definir un tipo de datos para luego asociar las variables unidimensionales al tipo de datos:

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

```
typedef float TLISTA[20]; /*Definición del tipo de datos TLISTA como un
arreglo de 20 número reales*/

int main() {
    TLISTA lista; /*Declaración de la variable lista de tipo TLISTA*/
    Lista[1]=12.9;
    ...
    return 0;
}
```

2.1.1 Inicialización

Luego de declarar los arreglos, éstos se pueden utilizar. Veamos algunos ejemplos:

- Todos los componentes del arreglo se inicializan en 0 (esto se da sólo para el 0).

```
int A[10] = {0}
```

Si se utiliza un tipo de datos declarado, la declaración e inicialización de la variable se realizaría de la siguiente forma:

```
typedef int TLISTA[20];

int main() {
    TLISTA A = {0};
    ...
    return 0;
}
```

- El primer componente del arreglo se inicializa con 5 el resto con 0

```
int B[5] = {5}
```

-Se asignan los cinco elementos del vector.

```
int C[5] = {6, 23, 8, 4, 11}
```

Nota: Si se omite el tamaño del arreglo, dicho tamaño será el número de elementos asignados en la declaración.

La variable C se denomina variable indizada, ya que cada valor del índice hace referencia a una dirección de memoria distinta. Así C[1] ocupa un lugar en la memoria distinto de C[2] (aunque C[2] es contigua a C[1]). Esto permite utilizar los ciclos de control repetitivos, para recorrer los arreglos. Gráficamente podemos representar al arreglo dimensional C (o variable indizada C) de la siguiente manera:

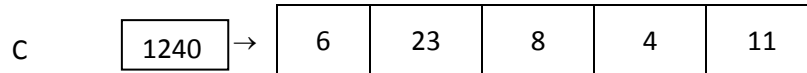
C	6	23	8	4	11
	↑	↑	↑	↑	↑
	C[0]	C[1]	C[2]	C[3]	C[4]

❗ Observe que el primer elemento se ubica en la posición 0.

Cada elemento del array C es un entero y puede ser usado en cualquier contexto donde un entero pueda usarse.

En este caso la variable indizada C es un puntero a su primer elemento. Por lo tanto es un puntero constante a un entero $C \equiv \&C[0]$.

Por ejemplo al declarar `int C[5] = {6, 23, 8, 4, 11}` se tendrá:



Donde 1240 es la dirección de memoria en donde comienza el vector; por lo tanto C apunta a `C[0]`.

El siguiente ejemplo registra la cantidad de votos que recibieron cinco candidatos.

Ejemplo 1:

```
1. #include<stdio.h>
2. int main() {
3.     int ELE[5]={0}; /*define e inicializa un arreglo que podrá contener 5 enteros */
4.     int I,VOT;
5.     printf ("Ingrese el n° del candidato elegido : \n");
6.     scanf ("%d",&VOT);
7.     while (VOT) /*si VOT es 0 implica false */
8.     {
9.         if ((VOT > 0) && (VOT < 6))
10.            ELE[VOT-1]++; /*candidatos del 1 al 5 arreglo VOT[0] a VOT[4]*/
11.        else
12.            printf ("\nEl voto ingresado es incorrecto\n");
13.        printf ("Ingrese el n° del candidato elegido : \n");
14.        scanf ("%d",&VOT);
15.    }
16.    printf ("\n Resultados de la Elección \n");
17.    for (I=0; I<= 4; I++)
18.        printf ("\n Candidato %d:  %d", I+1, ELE[I]);
19.    return 0;
20. }
```

Como puede verse en el programa se genera un arreglo con 5 elementos para registrar la cantidad de votos de cada uno de los candidatos, es decir, en la primera posición se guarda la cantidad de votos del candidato 1 y así sucesivamente (recuerde que el arreglo comienza en la posición 0). Una vez finalizado el ingreso, se muestran las cantidades obtenidas por cada candidato.

2.2 Arreglos Unidimensionales como Parámetro

Dado que un vector no tiene tamaño predefinido, pasar su contenido como un parámetro por valor es un costo que C no asume. Un vector completo se pasa a una función mediante un parámetro que contiene la primera dirección de memoria asignada a esta estructura, es decir que es un parámetro pasado por referencia.

La referencia a un vector o la dirección inicial, se especifica mediante su nombre, sin corchetes ni subíndices.

Por ejemplo, se puede definir un vector de 80 caracteres y usarlo para invocar al procedimiento `f00`, de la siguiente manera:

```
char caracteres[80] = "esta cadena es constante";
```

y usarlo como parámetro actual para invocar a la función `foo`, como sigue:

```
foo(caracteres, tamano)
```

El parámetro formal, que se define dentro del procedimiento, se escribirá con un par de corchetes vacíos, es decir, el tamaño del vector no se especifica:

```
void foo(char cadena_entrada[], int tam).
```

También puede especificarse un puntero del tipo del vector:

```
void foo(char *cadena_entrada, int tam).
```

Como puede verse, en este caso, se utiliza un puntero de tipo `char` como primer parámetro.

En C los arreglos son pasados como parámetros por referencia. Esto es, el nombre del arreglo es la dirección del primer elemento del arreglo. Así mismo, un elemento cualquiera de un arreglo puede ser pasado a una función por valor o por referencia, tal y como se hace con una variable simple.

Ejemplo 2:

```
1.  /* Paso de vector y elementos individuales del vector como parámetros*/
2.  #include<stdio.h>
3.  #define nroEltos 5
4.
5.  /*Prototipos*/
6.  void modificarVector(int [], int); /*Puede usarse(int *, int)*/
7.  void modificarValor(int);
8.  void modificarReferencia(int *);
9.
10. int main()
11. {
12.     int vector[nroEltos] = {0, 1, 2, 3, 4};
13.     int j;
14.     printf("Los valores del vector original son:\n");
15.     for (j = 0; j <nroEltos; j++)
16.         printf("%i\n",vector[j]);
17.     modificarVector(vector, nroEltos);/*el parámetro vector es pasado por referencia*/
18.     printf("Los valores del vector modificado son:\n");
19.     for (j = 0; j <= nroEltos - 1; j++)
20.         printf("%i\n",vector[j]);
21.     printf("Efectos de pasar un elemento de un vector como parámetro por valor\n");
22.     modificarValor(vector[3]);
23.     printf("El valor del cuarto elemento del vector es: %i\n",vector[3]);
24.     modificarReferencia(&vector[3]);
25.     printf("El valor del cuarto elemento del vector es: %i",vector[3]);
26.     return 0;
27. }
28. void modificarVector (int b[], int num) {
29.     int k;
30.     for (k = 0; k <= num - 1; k++)
31.         b[k] *= 2;
32. }
33. void modificarValor (int e) {
34.     e *= 2;
```

```

35.   printf("Valor modificado dentro del modulo = %i\n", e);
36. }
37. void modificarReferencia (int *e) {
38.     *e *= 2;
39.     printf("Valor modificado dentro del modulo = %i\n", *(e));
40. }

```

Ejemplo 3:

```

1.  /*Diseñar un programa que calcule el área y la circunferencia de n círculos. Se requiere
    conservar los valores del radio, del área y de la circunferencia.*/
2.  #include <stdio.h>
3.  #include <math.h>
4.  #define n 100
5.
6.  void leerRadios(float *, int *);
7.  void AreaCirc (float , float *, float *);
8.  void AreasCircs(float *, int , float [], float *); /*en array usar float * o float []*/
9.  void escribirAreasCircs (float *, int , float [], float *);
10.
11. int main()
12. {
13.     float R[n], A[n], C[n]; /*Declaración de los vectores*/
14.     int nRadios;
15.     leerRadios(R, &nRadios);
16.     AreasCircs(R, nRadios, A, C);
17.     escribirAreasCircs (R, nRadios, A, C);
18.     return 0;
19. }
20. void leerRadios(float radios[], int *cant)
21. {
22.     int j;
23.     printf("¿Cuántos radios seran introducidos ?\n");
24.     scanf("%i",cant);
25.     for (j = 0; j < *(cant); j++)
26.     {
27.         printf("radio %i = ",j);
28.         scanf("%f",&radios[j]);
29.     }
30. }
31. void AreaCirc (float radio, float *area, float *cir)
32. {
33.     float pi = 3.14159;
34.     *area = pi * pow(radio, 2);
35.     *cir = 2 * pi * radio;
36. }
37. void AreasCircs(float radios[], int cant, float areas[], float cirs[])
38. {
39.     int I;
40.     for (I = 0; I < cant; I++)
41.         AreaCirc(radios[I], &areas[I], &cirs[I]);
42. }
43. void escribirAreasCircs (float radios[], int cant, float areas[], float cirs[])
44. {
45.     int j;

```

```
46.   for (j = 0; j < numEltos; j++)
47.   {
48.       printf("Para la circunferencia de radio %f \t", radios[j]);
49.       printf("su area es: %f\n", areas[j]);
50.       printf("su circunferencia es: %f\n", cirs[j]);
51.   }
52. }
```

2.3 Arrays y Punteros

Hay una fuerte relación entre arrays y punteros, pero los arrays y los punteros NO son equivalentes.

Declarar un puntero reserva memoria SÓLO para el puntero.

La variable puntero NO está inicializada para apuntar a ningún espacio existente, por lo que inicialmente tiene cualquier valor no válido. Para indicar que un puntero no apunta a ningún sitio se le puede asignar el valor NULL.

Los arrays y los punteros usan diferentes notaciones de indexación.

Por ejemplo si se define un array que contiene 5 números elementos de tipo float:

```
float arr[5];
```

Entonces, las siguientes expresiones son equivalentes:

USANDO ÍNDICES	USANDO PUNTEROS
arr[0]	*(arr)
arr[1]	*(arr + 1)
arr[2]	*(arr + 2)
arr[3]	*(arr + 3)
arr[4]	*(arr + 4)
arr[índice]	*(arr + índice)

Ejemplos:

```
int arr[10];      Declara el array arr con 10 elementos.
int *ptr;         Declara un puntero a entero.
arr[1]=5;         Asigna 5 al segundo elemento del array arr.
*(arr+1)=5;       Equivalente a la instrucción anterior. Recordar que *arr equivale a arr[0]
ptr=&arr[2];       Asigna a ptr la dirección del tercer elemento:
```

2.4 Aritmética de Punteros

Considerando el siguiente arreglo de enteros:

```
int array[10];
```

La posición del primer elemento puede ser extraída como:

```
int * p_array = &array[0];
```

El n-esimo elemento se extrae como

```
array[n-1]
```

pero también puede utilizarse la aritmética de punteros:

```
*(p_array + n - 1).
```

Se puede observar que `p_array` almacena la dirección de memoria del primer elemento y como el resto de los elementos son consecutivos, puede hacerse una simple suma o resta para llegar al elemento de interés. Notar también que las unidades sumadas son relativas al tamaño.

Los apuntadores de memoria se mueven por bloques, por lo tanto, cuando se hace alguna operación aritmética con algún apuntador de memoria, se suma un bloque que está formado por `m` bytes.

3 Arreglos Bidimensionales

Definición: “Un arreglo bidimensional es una colección finita, homogénea y ordenada de datos, en la que se hace referencia a cada elemento del arreglo por medio de dos índices. El primero de los índices se utiliza para indicar la fila, y el segundo para indicar la columna”.

Un arreglo bidimensional permite almacenar `NxM` elementos del mismo tipo y acceder a cada uno de ellos.

Donde `N` es la cantidad de filas y `M` la cantidad de columnas del arreglo. Por ejemplo:

```
int A[4][5];
```

 declaración de un arreglo bidimensional de 4 filas y 5 columnas.

La matriz, al igual que el array, se usa como parámetro pasado por referencia.

El prototipo para ingresar una matriz puede ser de la siguiente forma:

```
void ingresa_matriz(int MATRIZ[][100], unsigned int *, unsigned int *);
```


Note que es necesario introducir en el prototipo, la cantidad de columnas declara.

Dentro de `main`, se declara una matriz de 1000 elementos enteros (100 filas y 100 columnas), y dos variables simples para la cantidad de filas y de columnas:

```
int M, N, MATRIZ[100][100];
```

La invocación a la función será:

```
ingresa_matriz(MATRIZ, &M, &N);
```

 Capítulo 6 Sección 6.1 de la Página 216-217 del libro Fundamentos de Programación. Piensa en C de Cairó, Osvaldo.

4 Tipo de Dato Cadena (string)

4.1 Declaración de Cadenas

En C una cadena se define como un array de caracteres (`char`) que termina en un carácter nulo (`'\0'`). Al declarar una cadena se debe considerar el espacio para el carácter nulo al final de la cadena.

Sintaxis: `char<identificador>[<longitud máxima>];`

Por ejemplo: `char a[5];`

4.2 Inicialización

Se puede inicializar la cadena con un valor en el momento de su declaración, por ejemplo:

<code>char a[5]="Hola";</code>	a →	<table border="1"><tr><td>h</td><td>o</td><td>l</td><td>a</td><td>\0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	h	o	l	a	\0	0	1	2	3	4
h	o	l	a	\0								
0	1	2	3	4								
<code>char b[5]={'o','l','l','a','\0'};</code>	b →	<table border="1"><tr><td>o</td><td>l</td><td>l</td><td>a</td><td>\0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	o	l	l	a	\0	0	1	2	3	4
o	l	l	a	\0								
0	1	2	3	4								
<code>char c[5]={97, 98, 99, 100, 0};</code>	c →	<table border="1"><tr><td>a</td><td>b</td><td>c</td><td>d</td><td>\0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	a	b	c	d	\0	0	1	2	3	4
a	b	c	d	\0								
0	1	2	3	4								

El ejemplo muestra tres maneras distintas de inicializar un vector de 5 caracteres y su representación en memoria.

La variable de tipo cadena llamada **a**, se inicializó con la constante `Hola`. Observar que el carácter nulo de terminación lo añade C automáticamente debido a que la inicialización se hizo con una cadena, por ello se utilizó comillas dobles para delimitar la constante "Hola".

La variable de tipo cadena **b** se inicializó con el vector de caracteres que contiene la palabra olla, en este caso si es necesario añadir el carácter nulo (`'\0'`). El vector de inicialización, en este ejemplo, contiene en cada posición, un elemento de tipo carácter, por lo cual, cada elemento es delimitado con comilla simple `'o'`, `'l'`, `'l'`, `'a'`, `'\0'`.

La variable **c**, también de tipo cadena, se inicializó con el vector de enteros que contiene la palabra `abcd`, en este caso sí es necesario añadir el carácter nulo con el entero cero (0). Esto es así porque un array de caracteres es un array de enteros (los datos de tipo `char` son un subconjunto de enteros); cada carácter tiene asociado un entero entre 0 y 255, que corresponde a su correspondiente código en la tabla `ascii`.

La representación gráfica de la memoria muestra que internamente los caracteres se almacenan en posiciones consecutivas de memoria. Así también se puede observar que el identificador de la cadena es la dirección de memoria de comienzo del array de caracteres.

Al ser una cadena un array de caracteres, se pueden usar como parámetros por referencia exclusivamente.

 Andrés Marzal e Isabel - Gracia Estructuras de datos en C-pag.91.

Aguilar y Martínez - Programación en C. Metodología, algoritmos y estructuras de datos- desde pag. 380.

4.3 Funciones de biblioteca para el tratamiento de cadenas

Se sabe que una función es un fragmento de código que realiza una tarea bien definida. Existen funciones que realizan diversas tareas ya definidas en el estándar ANSI C y que pueden ser utilizadas por el programador. Este tipo de funciones predefinidas son denominadas funciones de biblioteca. Nos centraremos en las siguientes funciones:

4.3.1 Entrada y Salida de Cadenas con Formato

La función `printf()` permite mostrar una cadena por consola con la marca de formato `%s`.

La función `scanf()` permite leer una “palabra” por teclado, es decir, una secuencia de caracteres no blancos, usando la marca de formato `%s`. **Termina la lectura cuando encuentra un espacio en blanco ó un fin de línea.** Esta característica implica que sólo se ingresará una palabra en cada invocación a `scanf`, por lo que, si el usuario escribe caracteres luego de un espacio en blanco, la función `scanf` tomará los caracteres previos al espacio y los otros caracteres serán tomados en futuras invocaciones a la función `scanf`.

El ejemplo a continuación muestra la variable `cad` carácter a carácter, recuerda que la marca de formato asociada a un carácter es `%c`:

Ejemplo 5:

```
1. #include<stdio.h>
2. int main()
3. {
4.     int i;
5.     char cad[11]="abcd";
6.     for (i=0; i<5; i++)
7.         printf("cad[%d]=%c\n", i, cad[i]);
8.     return 0;
9. }
```

El siguiente ejemplo captura una cadena en la variable `cad` y luego muestra su contenido:

Ejemplo 6:

```
1. #include<stdio.h>
2. int main()
3. {
4.     char cad[11];
5.     scanf("%s", cad);
6.     printf("La cadena leída es %s\n", cad);
7.     return 0;
8. }
```



Andrés Marzal e Isabel - Gracia Estructuras de datos en C desde pag.93

4.3.2 Entrada y Salida de Cadenas sin Formato

Las funciones principales que realizan la entrada y salida sin formato son: `puts()` y `gets()`.

La función `puts()` escribe su argumento de tipo cadena, en la salida estándar, `stdout`, y reemplaza el carácter nulo de terminación de la cadena (`'\0'`) por el carácter nueva línea (`'\n'`), provocando un salto de línea.

```
int *puts(const char *var);
```

Ejemplo 7:

```
1. #include<stdio.h>
2. int main(void)
3. {
4.     char cad[11];
5.     printf("Ingrese una cadena\n");
6.     gets(cad);
7.     puts(cad);
```


```

8.  return 0;
9.  }
10.

```

11.  Francisco Javier Ceballos Sierra - Curso de Programación en C/C++ desde pág. 188.

La función `gets()` tiene el inconveniente de que, al ser una lectura sin formato, puede provocar desbordamiento de memoria, ya que toma todos los caracteres hasta el fin de la cadena, pero si la variable de tipo cadena no tiene reservado espacio suficiente para la cantidad de caracteres ingresados, desborda la memoria, pudiendo afectar a otras variables.

 <http://www.taringa.net/posts/ciencia-educacion/14450390/Lectura-de-cadenas-en-C.html>.

4.3.3 Lectura de Escalares y Cadenas

La lectura alterna de valores escalares y cadenas, es decir, la mezcla de llamadas a `scanf()`, a `getchar()` y a `gets()`, produce efectos curiosos que derivan de la combinación de su diferente comportamiento frente a los espacios en blanco o al efecto que produce el carácter nueva línea que queda en el buffer de entrada al ejecutar la función `scanf()` o `getchar()`.

Solución 1

Leer línea completa con `gets()` en una variable auxiliar y extraer de ella los valores escalares con la función `sscanf()`. La función `sscanf()` obtiene (captura) datos de una cadena.

```

...
int a;
char frase[30];
char linea[30];
printf("Ingrese un entero: ");
gets(linea);
sscanf(linea, "%d", &a);
printf("Ingrese una frase: ");
gets(frase);
...

```

 Andrés Marzal e Isabel Gracia - Introducción a C - pag 96.

Solución 2

Ejemplo:

Usar la función `scanf()` con modificador carácter.

```

int a, b;
char frase[31];
printf("Ingrese entero: ");
scanf("%d%c", &a);
printf("Ingrese una frase: ");
gets(frase);
...

```

Usando `%c` la función `scanf()` no dejará en el buffer el carácter de fin de línea. Y la función `gets()` leerá correctamente la frase.

📖 Andrés Marzal e Isabel Gracia - Introducción a C - pag 96.

Aguilar y Martínez - Programación en C. Metodología, algoritmos y estructuras de datos- desde pag. 401.

Solución 3

Usar la función `fflush()` que limpia el buffer de entrada.

```
...
int a;
char frase[31];
printf("Ingrese un entero: ");
scanf("%d",&a);
fflush(stdin);
printf("Ingrese una frase: ");
gets(frase);
...
```

La función `fflush()` en Linux y sistemas tipo Unix no funciona.

Consejo:

Se recomienda, para resolver el problema de la función `gets()`, realizar una función propia que permita leer cadenas considerando todos los inconvenientes mencionados.

La función propuesta tiene la siguiente cabecera:

Sintaxis:

```
void leeCad(char *, int);
```

El módulo `leeCad()` acepta dos parámetros: la variable donde se almacena la cadena, y su tamaño (incluyendo el carácter nulo). Además, devuelve un valor de tipo entero, que permite verificar si hubo un error.

El siguiente es un ejemplo de invocación a la función `leeCad()`:

```
...
char cade[30];
printf("Escribe un cadena: ");
leeCad(cade, 30);
...
```


Definición de la función `leeCad()`:

```
1. void leeCad(char cadena[], int tam){
2.     int j, m;
3.     j=0;
4.     while(j<tam-1 && (m=getchar())!=EOF && m!='\n'){
5.         cadena[j]=m;
6.         j++;}
7.     cadena[j]='\0';
8.     if(m != EOF && m != '\n') /*limpia el buffer*/
9.         while((m=getchar())!=EOF && m!='\n');
10. }
```

4.3.4 Funciones para Trabajar con Cadenas de Caracteres

La biblioteca de C proporciona un amplio número de funciones que permiten realizar diversas operaciones con cadenas de caracteres. A continuación se listan algunas de las funciones más usadas, indicando la biblioteca a la que pertenece:

Propósito	Biblioteca	Funciones
Entrada y salida	stdio.h	getchar()
Longitud	string.h	strlen()
Copiar	string.h	strcpy strncpy()
Concatenación	string.h	strcat() strncat() sprintf()
Comparación Los operadores (==, !=, <, <=, >, >=) NO funcionan con cadenas.	string.h	strcmp() strncmp()
Búsqueda	string.h	strstr() strchr() strrchr()
Conversión de tipos	stdlib.h	atoi() atof() atol() fcvt()
	ctype.h	tolower() toupper()
	string.h	strupr() strlwr()

 Puedes encontrar una descripción de cada función consultando Francisco Javier Ceballos Sierra - Curso de Programación en C/C++ pág. 171.

strcat()

```
#include<string.h>
```

Prototipo: `char * strcat (char *destino, const char *origen)`

La función estándar `strcat()` permite añadir una cadena “origen” (o bloque de memoria) a otra “destino”. Las dos cadenas deben terminar con un carácter nulo. Finaliza la cadena resultante “destino” con un carácter nulo y devuelve un puntero a “destino”. El nombre `strcat()` es una abreviación de “string concatenate” (concatenación de cadena).

Por ejemplo:

```
char str1[100] = "; Hola,"; /* 100: se reserva espacio extra */
strcat (str1, " mundo !\n");
printf (str1); /* muestra "; Hola, mundo !" en la salida estándar */
```

La función `strcat()` puede ser peligrosa porque si la cadena a añadir es demasiado larga entonces sobrescribirá la memoria adyacente.

NOTA: Recuerde que para utilizar la función `printf` debe usar la biblioteca `stdio.h`

strncat()

```
#include<string.h>
```

Prototipo: `char * strncat(char*s1, const char *s2, size_t n);`

Añade no más de `n` caracteres (un carácter nulo y los demás caracteres siguientes no son añadidos) de la cadena apuntada por `s2` al final de la cadena apuntada por `s1`. El carácter inicial de `s2` sobrescribe el carácter nulo al final de `s1`. El carácter nulo siempre es añadido al resultado.

Por ejemplo:

```
char s1[12] = "Hola ";
char s2[7] = "amigos";
strncat( s1, s2, 3 );
printf( "%s\n", s1 ); /*muestra 'Hola ami'*/
```

strcpy()

```
#include<string.h>
```

Prototipo: `char *strcpy (char *destino, const char *origen)`

Copia la cadena apuntada por origen en la cadena apuntada por destino.

Por ejemplo:

```
/*Se inicializa la variable origen con el valor indicado*/
char *origen = "Hola mundo";

/*El puntero destino debe contener suficiente espacio.*/
char destino[strlen(origen)+1];

/*Después de realizar esta operación la variable destino tendrá el valor "Hola mundo"*/
strcpy (destino, origen);
```

strncpy()

```
#include<string.h>
```

Prototipo: `char * strncpy(char *s1, const char *s2, size_t n);`

Copia no más de *n* caracteres (caracteres posteriores al carácter nulo no son copiados) de la cadena apuntada por *s2* a la cadena apuntada por *s1*.

La función retorna el valor de *s1*. Si al copiar una cadena a la otra se superponen, entonces el comportamiento no está definido. Si el array/arreglo apuntado por *s2* es una cadena que es más corta que *n* caracteres, entonces caracteres nulos son añadidos a la copia en el array apuntado por *s1*.

Por ejemplo:

```
char s2[8] = "abcdefg";
char s1[8];
strncpy( s1, s2, 3 );
printf( "%s\n", s1 );      /*Muestra la cadena abc.*/
```

strchr()

```
#include<string.h>
```

Prototipo: `char *strchr(char *s, char c);`

Localiza la primera aparición de *c* (convertido a `unsigned char`) en la cadena apuntada por *s* (incluyendo el carácter nulo).

La función retorna un puntero a partir del carácter encontrado. Si no se ha encontrado el carácter contenido en la variable *c*, entonces retorna un puntero `null`. El carácter en la variable *c* puede ser el valor nulo `'\0'`.

Por ejemplo:

```
char s[12] = "Hola amigos";
char c = 'a';
printf( "%s\n", strchr( s, c ) );      /*muestra "a amigos"*/
```

strrchr()

```
#include<string.h>
```

Prototipo: `char *strrchr(char *s, char c);`

Localiza la última aparición del carácter en la variable *c* (convertido a `unsigned char`) en la cadena apuntada por *s* (incluyendo el carácter nulo).

La función retorna un puntero a partir del carácter encontrado. Si no se ha encontrado el carácter de la variable *c*, retorna un puntero nulo.

Por ejemplo:

```
char s[16] = "Hola mis amigos";
char c = 'a';
printf( "%s\n", strrchr( s, c ) );    /* muestra "amigos"*/
```

strcmp()

```
#include<string.h>
```

Prototipo: `int strcmp(const char *s1, const char *s2);`

Compara la cadena apuntada por `s1` con la cadena apuntada por `s2`.

La función retorna un número entero mayor, igual, o menor que cero, apropiadamente según si la cadena apuntada por `s1` es mayor, igual, o menor (según orden alfabético) que la cadena apuntada por `s2`. La función diferencia entre mayúsculas y minúsculas

Por ejemplo:

```
char s1[6] = "Abeja";
char s2[6] = "abeja";
int i;
i = strcmp( s1, s2 );
printf( "s1 es " );
if( i < 0 ) printf( "menor que" );
else if( i > 0 ) printf( "mayor que" );
else printf( "igual a" );
printf( " s2\n" ); /*muestra: s1 es mayor que s2*/
```

strncmp()

```
#include<string.h>
```

Prototipo: `int strncmp(const char *s1, const char *s2, size_t n);`

La función `strncmp()` compara no más de `n` caracteres (caracteres posteriores al carácter nulo no se tienen en cuenta) de la cadena apuntada por `s1` con la cadena apuntada por `s2`.

La función retorna un número entero mayor, igual, o menor que cero, apropiadamente según la cadena apuntada por `s1` es mayor, igual, o menor que la cadena apuntada por `s2`.

Por ejemplo:

```
char s1[9] = "artesano";
char s2[8] = "artista";
int i;
i = strncmp( s1, s2, 3 );
printf( "Las 3 primeras letras de s1 son " );
if( i < 0 ) printf( "menores que" );
else if( i > 0 ) printf( "mayores que" );
else printf( "iguales a" );
printf( " s2\n" ); /*muestra: Las 3 primeras letras de s1 son iguales a s2*/
```

strcspn()

```
#include<string.h>
```

Prototipo: `size_t strcspn(const char *s1, const char *s2);`

La función retorna el número de caracteres leídos de la subcadena hasta encontrar alguno de los caracteres de `s2`. El carácter nulo no se cuenta.

Por ejemplo:


```
char s1[13] = "Hola a todos";
char s2[6] = "bcade";
printf( " %d\n", strcspn( s1, s2 ) );
/* muestra el número 3, ya que los tres primeros caracteres (H, o y l) no se
encuentran en s2. */
```

strlen()

```
#include<string.h>
```

Prototipo: `size_t strlen(const char *s);`

Calcula el número de caracteres de la cadena apuntada por `s`.

La función retorna la cantidad de caracteres hasta el carácter nulo, éste no se incluye.

Por ejemplo:

```
char s[13] = "Hola a todos";
printf( " %d\n", strlen( s ) ); /*muestra el número 12 */
```

strstr()

```
#include<string.h>
```

Prototipo: `char * strstr(const char *s1, const char *s2);`

Localiza la primera aparición en la cadena apuntada por `s1` de la secuencia de caracteres (excluyendo el carácter nulo) en la cadena apuntada por `s2`.

La función retorna un puntero a la cadena encontrada, o un puntero nulo si no se encontró la cadena. Si `s2` apunta a una cadena de longitud cero, la función retorna `s1`.

Por ejemplo:

```
char s1[13] = "Hola a todos";
char s2[3] = "la";
printf( "%s\n", strstr( s1, s2 ) ); /* Muestra 'la a todos' */
```

strtok()

```
#include <string.h>
```

Prototipo: `char * strtok(char *s1, const char *s2);`

Rompe la cadena `s1` en segmentos o tokens. Esta ruptura destruye `s1` en el proceso. La forma de romper la cadena depende de la secuencia de caracteres de la cadena `s2`. Estos caracteres se denominan caracteres delimitadores. La función recorrerá la cadena en busca de alguno de los delimitadores de la cadena `s2`. Cuando lo encuentre, el proceso se detiene, ya que tiene un token. Posteriores llamadas a `strtok` romperán la cadena `s1` en otros tokens. Estas llamadas pueden tener otra secuencia de delimitadores.

La primera llamada a `strtok` determina la cadena a romper, retornando el puntero al comienzo del primer token. Si se recorrió la cadena `s1` sin haber encontrado un delimitador, y aún no se ha obtenido el primer token, la función retornará un puntero nulo.

Posteriores llamadas retornarán más tokens. Si ya no encuentra más delimitadores, entonces retornará todos los caracteres desde el último delimitador para ser el último token. Si ya se retornó el último token, entonces retornará un puntero nulo en las siguientes llamadas a la función.

Por ejemplo:

```
char s1[49] = "Esto es un ejemplo para usar la funcion strtok()";
/*Aquí se definen dos delimitadores, el espacio y fin de línea. */
char s2[3] = " \n";
char *ptr;
/* Primera llamada => Primer token.*/
ptr = strtok( s1, s2 );
printf( "%s\n", ptr );    /*Muestra "Esto"*/
/* Posteriores llamadas */
while( (ptr = strtok( NULL, s2 )) != NULL )
    printf( "%s\n", ptr );
```

La cadena `s1` se va disgregando en cada espacio o fin de línea (delimitadores definidos en `s2`). Por eso, la salida, muestra cada palabra de `s1` en un renglón diferente.

Una vez leído el primer elemento de `s1`, para leer el siguiente elemento se llama a `strtok` pasando como argumento `NULL`.

strlwr()

```
#include <string.h>
```

Prototipo: `char * strlwr(char *cadena);`

Convierte las letras mayúsculas de cadena, en minúsculas. El resultado es la propia cadena en minúsculas.

Por ejemplo:

```
char s1[49] = "Esto es UN ejemplo";
printf( "%s\n", strlwr(s1));    /*Muestra 'esto es un ejemplo'*/
```

strupr()

```
#include<string.h>
```

Prototipo: `char * strupr(char *cadena);`

Convierte las letras minúsculas de cadena, en mayúsculas. El resultado es la propia cadena en mayúsculas.

Por ejemplo:

```
char s1[49] = "Esto es UN ejemplo";
printf( "%s\n", strupr(s1));    /*Muestra 'ESTO ES UN EJEMPLO'*/
```

4.3.5 Funciones para Conversión de Datos

atof()

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

Prototipo: `double atof(const char *numPtr);`

Convierte la porción inicial de la cadena apuntada por numPtr a una representación de double.

Por ejemplo:

```
char numPtr[11] = "123.456789";
printf("Convirtiendo la cadena \"%s\" en un numero: %f\n", numPtr, atof(numPtr) );
```

atoi()

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

Prototipo: `int atoi(const char *numPtr);`

Convierte la porción inicial de la cadena apuntada por numPtr a una representación de int.

Por ejemplo:

```
Char numPtr[5] = "1234";
printf( "Convirtiendo la cadena \"%s\" en un numero: %d\n", numPtr, atoi(numPtr) );
```

atol()

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

Prototipo: `long int atol(const char *numPtr);`

Convierte la porción inicial de la cadena apuntada por numPtr a una representación de long.

Por ejemplo:

```
char numPtr[11] = "1234567890";
printf("Convirtiendo la cadena \"%s\" en un numero: %u \n", numPtr, atol(numPtr));
```

fcvt()

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

Prototipo: `char * fcvt (double valor, int decs, int * pdec, int * signo);`

Convierte un número real a una cadena de caracteres, la cual será finalizada con un carácter nulo. La función devuelve un puntero a la cadena de caracteres.

valor: Es el número a convertir.

decs: Número de dígito después del punto decimal.

pdec: Devuelve un puntero a un valor entero que especifica la posición del punto decimal.

signo: Devuelve un puntero a un valor 0, si el número es positivo; o un valor distinto de 0, si el número es negativo.

Por ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    double valor = 3.141592653;
    int puntodecimal, signo;
    char *cadena;
    cadena = fcvt (valor, 8, &puntodecimal, &signo);
    printf ("Valor: %2.9f, cadena: '%s', puntodecimal: %d "
           "signo: %d\n", valor, cadena, puntodecimal, signo);
    return 0;
}
```

Muestra: Valor: 3.141592653, cadena: 3141592653, puntodecimal: 1 signo: 0

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    char *buffer;
    double value = 365.249;
    int precision = 5;
    int decimal, sign;

    buffer = fcvt (value, precision, &decimal, &sign);
    printf ("%c%c.%s x 10^%d\n", sign? '-' : '+', buffer[0], buffer+1, decimal-1);
    return 0;
}
Output:
+3.6525 x 10^2
```

sprintf()

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

Prototipo: `int sprintf(char *buffer, const char *formato [, argumento]...);`

Convierte los valores de los argumentos especificados a una cadena de caracteres que se almacena en buffer, la cual finaliza con un carácter nulo. Cada argumento es convertido y almacenado de acuerdo con el formato correspondiente que se haya especificado. La descripción de formato, es la misma que se especificó en la función `printf()`.

Devuelve como resultado un entero correspondiente al número de caracteres almacenados en buffer sin contar el carácter nulo de terminación.

Por ejemplo:

```
char nombre[20], mensaje[81];
unsigned int edad=0;
printf( "Escriba su nombre: " );
scanf( "%s", nombre );
printf( "Escriba su edad: " );
scanf( "%u", &edad );
sprintf( mensaje, "\nHola %s. Tienes %d años.\n", nombre, edad );
```

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

4.3.6 Funciones para Conversión de Caracteres

tolower()

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

Prototipo: int tolower(int c);

Convierte la letra contenida en *c* a una letra minúscula, si corresponde.

Por ejemplo:

```
char car;
do
{
printf("\¿Desea  continuar? s/n ");
car = getchar();
fflush(stdin);
}
while (tolower(car) != 'n' &&tolower(car) != 's' );
```

El ejemplo admite como respuesta la letra *s* o *n* en minúscula o mayúscula, pero la comparación se hace en minúscula.

toupper()

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

Prototipo: int toupper(int c);

Convierte la letra contendida en *c*, a una letra mayúscula, si corresponde.

5 Estructuras

5.1 Definición

Las estructuras, conocidas como registros, son un tipo de datos estructurado. Se usan para resolver problemas que involucran tipos de datos estructurados heterogéneos.

Una estructura es una colección de uno o más tipos de elementos denominados miembros cada uno de los cuales puede ser un tipo de dato diferente. (Joyanes Capítulo 9: Estructuras y Uniones. Pag 296)

Una estructura es una colección de elementos finita y heterogénea. Cada componente de una estructura se denomina campo. Cada campo puede ser a la vez simple o estructurado. (Cairó, Capítulo 8, Estructuras y Uniones. Pag 288)

5.2 Declaración

```
Struct<nombre de la estructura>
```

```
{  
<tipo de dato miembro><nombre miembro>;  
...  
};
```

5.3 Definiciones de Variables de Estructuras

Se presentan dos modos de definir las variables estructuradas a través de un ejemplo en que se declara una estructura alumno y se definen las variables alum1, alum2 y alum3:

1)

```
struct alumno  
{  
    int matricula;  
    char nombre[20];  
    char carrera[20];  
    float promedio;  
    char direccion[20];  
};  
  
struct alumno alum1, alum2, alum3;
```

2)

```
struct alumno  
{  
    int matricula;  
    char nombre[20];  
    char carrera[20];  
    float promedio;  
    char direccion[20];  
} alum1, alum2, alum3;
```

"alum1, alum2 y alum3" son instancias de "alumno".

5.4 Inicialización de una Declaración de Estructura

```
struct alumno  
{  
    int matricula;  
    char nombre[20];  
    char carrera[20];  
    float promedio;  
    char direccion[20];  
};  
struct alumno alum1={120, "Maria", "Contabilidad", 8.9, "La Madrid 54"};
```

5.5 Tamaño de una Estructura

El operador `sizeof` determina el tamaño que ocupa en memoria una estructura. Por ejemplo:
`printf("%d", sizeof(alum1));`

5.6 Acceso a Estructuras

Podemos acceder a una estructura mediante:


- Usando el operador punto (.)
- Usando el operador puntero -> (ver el ejemplo 4 en la sección uniones)

Por ejemplo para ingresar un valor en la variable alum1 del tipo alumno haríamos:

```
struct alumno alum1={120, "Maria", "Contabilidad", 8.9, "La Madrid 54"};
```

O podemos hacer:

```
printf("Ingrese matricula:\t");  
scanf("%d", &alum1.matricula);  
fflush(stdin);  
printf("\n Ingrese nombre :\t");  
gets(alum1.nombre);  
fflush(stdin);  
printf ("\n Matrícula = %d , Nombre = %s ",alum1.matricula, alum1.nombre);
```

 Capítulo 8. Estructura y Uniones - ejemplo 8.3 pág 291 a pág 293 en el libro Fundamentos de Programación. Piensa en C de Cairó, Osvaldo.

5.7 Creación de Sinónimos o alias “typedef”

Por ejemplo para declarar variables de tipo enteras podemos hacer:

```
int a1, a2, a3;
```

o en su defecto:

```
typedef int contador;  
int main(void) { contador a1,a2,a3;.....
```

Es un poco redundante con tipos de datos simples. En cambio en los estructurados cobra mayor sentido. Por ejemplo:

```
typedef struct {  
    int matricula;  
    char nombre[20];  
    char carrera[20];  
    float promedio;  
    char direccion[20];  
} alumno;  
  
int main(void) {  
    alumno a0 = {120, "Maria", "Contabilidad", 8.9, "Lamadrid 203"},*a3, *a4, *a5, a6;
```

Declara las variables a0, a3, a4, a5 y a6 de tipo alumno e inicializa a0.

Las variables a3, a4 y a5 son punteros a la estructura alumno.

La sentencia `typedef` identifica a un tipo de dato con un nuevo nombre con el fin de que sea más legible el código fuente.

Capítulo 9: Estructuras y Uniones. Joyanes Aguilar e Ignacio Martínez - Programación en C. Metodología, algoritmos y estructuras de datos.

5.8 Estructuras Anidadas

Consideremos el caso en que una empresa requiere almacenar la siguiente información de cada empleado:

- Nombre del empleado (cadena de caracteres)
- Departamento de la empresa (cadena de caracteres)
- Sueldos (real)
- Domicilio:
 - Calle (cadena de caracteres)
 - Número (entero)
 - Código Postal (entero)
 - Localidad (cadena de caracteres)

Empleado						
Nombre	Departamento	Sueldo	Domicilio			
			Calle	Número	CP	Localidad

La definición del tipo es:

```
typedef struct {  
    char calle[20];  
    int numero;  
    int cp;  
    char localidad[20];  
} domicilio;  
typedef struct {  
    char nombre[20];  
    char departamento[20];  
    float sueldo;  
    domicilio direccion;  
} empleado;  
...  
int main() {  
    empleado a;
```

Como puede observarse en este ejemplo, el tipo de datos empleado define una estructura con un campo llamado dirección que es de tipo domicilio. La variable **a**, al ser del tipo empleado, contiene el campo dirección. Para acceder, por ejemplo al código postal de la variable **a**, se accede primero al campo dirección y desde allí al campo cp:

```
a.direccion.cp=4400;
```


5.9 Estructuras como Parámetro

Puede usarse las estructuras como parámetros pasados por valor o por referencia. Para pasar por referencia debe usarse el operador &.

```
struct info {
    int campol ;
    ...
};
void entrada (struct info*);
void salida(struct info);
...
int main(void) {
    struct info reg;
    ...
    entrada(&reg);
    salida(reg);
    ...
}
```

En el caso en que la estructura sea pasada por referencia, el parámetro formal contiene la dirección de la estructura, es un puntero a la misma, por lo que, el acceso a cada campo se realiza mediante el operador ->, como se muestra en el siguiente ejemplo:

```
void entrada (struct info* p){
    printf("Ingrese un entero ");
    scanf("%d", &(p->campol));
}
```

En cambio, si la estructura fuera pasada por valor, el parámetro formal contiene los datos de la estructura, por lo que, el acceso a cada campo se realiza mediante el operador . (punto), como se muestra en el siguiente ejemplo

```
void salida(struct info t){
    printf("El número entero es %d", t.campol);
}
```

5.10 Estructuras con Arreglos

Se presenta a continuación el uso de array de registros (estructuras) a través del siguiente ejemplo, el cual permite cargar los datos de alumnos.


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

```
typedef struct {
    int matricula;
    char nombre[20];
    float cal[5]; /*se guardan 5 calificaciones por cada alumno*/
} alumno;
```

```
void Lectura(alumno[], int T);
```

```
int main() {
    int TAM;
    alumno ARRE[50]; /*se define un array de alumnos (array de registro)*/
```

```
...
Lectura (ARRE, TAM); /*ARRE, por ser un array es un parám pasado por ref*/
...
}
void Lectura (alumno A[], int T) {
    int I, J;
    for (I=0; I < T; I++){
        printf ("\nIngrese los datos del alumno %d: ", I+1);
        printf ("\nIngrese la matricula del alumno : ");
        scanf ("%d", &A[I].matricula);
        fflush (stdin);
        printf ("\nIngrese el nombre del alumno : ");
        gets (A[I].nombre); /*se aconseja el uso de leeCad*/
        for (J=0; J<5; J++) {
            printf ("\nIngrese la calificacion %d del alumno %d: ", J+1, I+1);
            scanf ("%f", &A[I].cal [J]);
        }
    }
}
```

 Ver desarrollo del programa en Cairó, Osvaldo. Fundamentos de Programación. Piensa en C. Ed. Pearson Educación. 2006 pág 298 a pág.300.

6 Uniones

Las uniones representan también un tipo de datos estructurados. Son similares a las estructuras.

Se distinguen fundamentalmente de las estructuras en que sus miembros comparten el mismo espacio de almacenamiento en la memoria.

Son útiles para ahorrar memoria. Sin embargo, es necesario considerar que sólo pueden utilizarse en aquellas aplicaciones en que sus componentes no reciben valores al mismo tiempo. Es decir sólo uno de sus componentes puede recibir valor a la vez.

El espacio de memoria reservado para una unión corresponde a la capacidad del campo de mayor tamaño.

Definición: Una unión es una colección de elementos finita y heterogénea en la que sólo uno de sus componentes puede recibir valor a la vez.

6.1 Declaración de Uniones

Veamos un ejemplo para entender el concepto de unión:

Se desea almacenar la siguiente información de cada alumno de una universidad:

- Nombre del alumno (cadena de caracteres).
- Promedio del alumno (real).
- Teléfono celular (cadena de caracteres).
- Correo electrónico (cadena de caracteres).

Nota: De cada alumno se guarda el teléfono o el correo (no ambos datos), por ello se utilizará el tipo estructurado unión.

El programa sugerido es el siguiente:

Ejemplo 4:

```
1. #include <stdio.h>
2. #include <string.h>
3. union datos {
4.     char  celular[15];
5.     char  correo[20];
6. };
7. typedef struct {
8.     char nombre[20];
9.     float promedio;
10.    union datos personales; /* Esta unión reserva 20 caracteres para el campo personales*/
11. } alumno;
12.
13. /*prototipo*/
14. void Lectura(alumno *);
15.
16. int main() {
17.     alumno a1 = {"Maria", 8.9, "154-747653"};
18.     printf("\n %s", a1.personales.celular);
19.     printf("\n %s", a1.personales.correo); /* muestra el mismo valor que celular*/
20.
21.     Lectura(&a1); /*Ingresa el valor del correo, no el del celular*/
22.
23.     printf("\n %s", a1.personales.celular);/* muestra el mismo valor que correo*/
24.     printf("\n %s", a1.personales.correo);
25.
26.     return 0;
27. }
28. void Lectura(alumno *a){ /*parámetro "a" pasado por referencia*/
29.     printf("\n Ingrese el nombre del alumno : ");
30.     gets(a->nombre); /*uso del operador puntero -> para acceder a la estructura*/
31.     printf("\n Ingrese el promedio del alumno : ");
32.     scanf("%f",&(a->promedio));
33.     fflush(stdin);
34.     printf("Ingrese el correo :");
35.     gets(a->personales.correo); /*se aconseja el uso de leeCad*/
36. }
```

En la línea 32 surge la pregunta porqué usar el símbolo "&" si "a" es un puntero. La respuesta es que "promedio" no es un puntero, si "promedio" fuese del tipo puntero, entonces sería `scanf("%f",a->promedio);`. Advierta además que como se accede a la estructura por medio de un puntero, entonces se debe usar el operador puntero "->" para acceder a los miembros (campos) del registro.