Capítulo 1

Dirección de Memoria y Apuntador

En este capítulo presentaremos un nuevo tipo de variable que llamaremos *apuntador* y algunos de sus usos. La memoria en una computadoras puede interpretarse como una secuencia de celdas cada una de un byte (8 bits contiguos) enumerados a partir de UNO,— no existe el byte CERO. Alerta esta secuencia de números no se representa como a los enteros. Salvo por el CERO, que representa a la celda NULA, es disjunta de los enteros. Una dirección de memoria no es más que el número de una celda. Por lo general para reprentar a una posisión de memoria se usan dos o cuatro bytes.

1.1. Apuntadores

Un apuntador es una variable que sirve para almacenar la posición (direccion) de la memoria donde se almacena alguna variable. También se les llama referencia, o *dirección de memoria* porque almacenan una dirección de memoria. Como variables que son se usa un identificador para manipularlas, hay que declararlas e inicializarlas antes de usarlas. Tienen una aritmética que mencionaremos luego, etc.

1.1.1. ¿Cómo se declaran?

En la declaración de un apuntador hay que indicar el tipo de dato que va a apuntar. Ello se logra colocando el tipo de dato al que se apunta seguido de un *asterisco* *, para indicar el tipo realmente es <**tipo**>* . Así la frase <**tipo**>* significa que la variable que sigue es un apuntador a una variable de tipo **tipo**; Recuerde que los tipos básicos son: **char, int, short, long, float, double, long double** y que **short** y **long** son abreviaturas para **short int** y **long**

int respectivamente. A continuación se muestra algunos ejemplos sensillos de declaración.

```
int * p;
char* s;
float * x;
```

Aquí la expresión "int * p" significa que p es una variable de tipo "int *". Y la instrucción "int * p;" significa que se declara una variable de tipo *apuntador a entero* de nombre p. No obstante, la expresion "int * p" tiene como intención que se asocie la expresión *p con un entero (el entero apuntado por p o el entero en la dirección p). Se aclarará luego.

Cuando se quieren declarar varios apuntadores del mismo tipo se coloca el tipo de dato seguido de una lista no vacía formada por asterisco * nombre de variable, separados por comas. La declaración es similar a la declaración de variables normales pero cada identificador (nombre de la variable) debe estar precedido del carácter *. Esto es:

```
<tipo> * <identificador>, ... , * <identificador>;
```

Puede interpretarse como si se hubiera sacado el <tipo>factor común. Si algún identificador no está precedido por el asterisco lo que se declara es una variable de tipo <tipo>, no de tipo <tipo>*.

Ejemplos:

```
int * p, * q, *otroP ;
float *r, x;
```

Note que cada identificador que se quiera sea un apuntador debe estar precedido por un *. En el segundo ejemplo se declara un apuntador a un punto flotante de nombre r y un punto flotante de nombre x.

Tabién se puede declarar un apuntador usando la palabra void. Un apuntador declarado de esta manera puede apuntar a cualquier tipo de dato. Por ejemplo, si declaramos a qq como un apuntador a **void** y a z como un char podemos asignarle a qq la dirección de memoria de z.

1.1.2. Inicialización y Uso

Una vez declarada una variable de tipo apuntador se debe inicializar antes de usarla en algún valor con sentido para nuestro propósito. Si bien es cierto que una variable apuntador puede apuntar a cualquiera de las celdas de la memoria, por consistencia debería apuntar sólo a celdas que sean realmente la dirección de memoria de alguna variable del mismo tipo que el que ella debe apuntar.

Aunque, se puede inicializar en el valor de cualquier cerda, los únicos valores con sentido son cero para expresar que no apunta a ninguna celda de memoria o alguna expresión que involucre las direcciones de memoria de datos previamente definidos y del tipo apropiado. Cuando se inicializa en cero se puede usar 0 o NULL. Conviene usar NULL por claridad. NULL es una constante que se define en la librería *stdlib.h*.

Antes de ilustrar la inicialización y modificación de las variables de tipo apuntador conviene presentar dos operadorres que actuan sobre las variables normales y las variable de tipo apuntador. La primera es el operador *direccional* $\& : ListVar \rightarrow DirMem$ que permite obtener la posición de memoria (direción de memoria—número de memoria) donde se almacena una variable dada. Si por ejemplo, a es una variable entera, entonces $\&a$ es el número (dirección de memoria) de la primera celda donde se almacena el valor de la variable a.

Por otro lado el operador de *desdireccional* $* : DirMem \rightarrow ListVar$ permite obtener el valor de la variable a la que apunta una variable de tipo apuntador. Si pi es un apuntador que apunta a una variable entera, entonces $*pi$ representa el valor de la variable entera almacenada en la casilla que indica pi.

Para ilustrar un poco el uso de apuntadores y en particular del un apuntador de tipo void, se muestra las siguientes líneas de código:

```
int a  = 8, b, c, * p = &a, * q = &b;
void *v = q;
*q = *p - 2;  c = 10**p+*q;
printf("a = %d, b = %d y c = %d.", a,*(int *)v,c);
```

En la primera línea se declaran tres variable enteras y dos apuntadores a enteros que se hacen apuntar a a y b respectivamente. En la segunda línea se declara un apuntador a cualquier cosa y se hace que apunte adonde apunta q, esto es, a b La siguiente línea usa los apuntadores correspondientes para inicializar a b y c. De esta forma b toma el valor 6, $a - 2$ y c toma el valor de 86, diez veces a más b. Finalmente la última línea imprime los valores de $a.b.c$, usando las variable a, c y el apuntador void v que apunta a b, pero para poder usarlo hay que hacerle un casting a apuntador a intero: (int *).

En el siguiente programa se muestra ...

```
int main(){
   int a, b, c, * p = &a, * q = &b, *r;
   *p = 8; r = NULL;
   void *v = q;
   *q = *p - 2;  c = 10**p+*(int *)v;
   printf("\na = %p, b = %p y c = %p.", &a,&b,&c);
   //printf("\np = %p, q = %p y r = %p.", &p,&q,&r);
   printf("\na = %d, b = %d y c = %d.", a,b,c);
   printf("\nr = %p, p = %p y v = %p.", r,p,v);
```

```
    return 0;
}
```

Inconcluso.....

Cuando en un programa se declara una variable, se le asigna una posición en la memoria donde se almacenará dicha variable y se agrega a una tabla su nombre, la posición asignada y su tipo. Los bits correspondientes a la variable por lo general no se alteran.—contienen la secuencia de bits que tenian antes de la asignación.

ListVar	DirMem	TipVar	*Valores*
a	860	int	8
b	864	int	6

Inconcluso.....

Todas las variable de un sub-programa son locales. Aparecen sólo después que se invoca el sub-programa y desaparecen cuando termina la ejecucion de la invocación del mismo. Por consiguiente, toda la información que se almacena en las variables de un subprograma desaparece cuando el mismo se acaba, para poder usar la información obtenida en un sub-programa en otro se debe o bien almacenarla en variables globales o bien almacenarla en alguna de las variables del programa que lo invoca,— en buena parte de los casos, del *main*. Para lograr esto, puesto que, los programas en C no comparten varables sino valores hay que usar un apuntador para compartir el valor de la dirección de memoria de las variables que se quieren modificar. Para ilustrar esto mostraremos algunos ejemplos.

El primero de ellos, que es el ejemplo que hay que presentar casi obligado, es el del procedimiento que intercambia los valores de dos variables enteras a y b. Sin usar un sub-programa las instrucciones serían, por ejemplo usando una variable auxiliar:

```
int aux = a; a = b; b = aux;
```

Pero si queremos usar un sub-programa, seleccionamos un nombre no muy largo que nos indique lo que hace el sub-programa, lo declaramos como void con parámetros a y b como sigue:

```
void swap(int a, int b){
   int aux = a; a = b; b = aux;
}
```

Nuestra sorpresa es que, si bien dentro del sub-programa se intercambiaron los valores, en el programa que lo invocó no pasó absolutamente nada.

Esto se resuelve pasandole al sub-programa swap las direcciones de memoria de de las variables que se quieren intercambiar, para ello debemos modificar los parametros de nuestra función; se deben cambiar a apuntadores para que alberguen direcciones de memoria en lugar de enteros. Además, puesto que ahora a y b no son enteros, sino direcciones de memoria, debemos convertir sus ocurrencias en los valores de las variables que apuntan usando el operador $*$. Recuelde de manera nemónica que: "*a es el valor de la variable apuntada por a, o el alias o sobrenombre de la variable apuntada por a". Ello nos produce el siguiente código:

```
void swap(int *a, int *b){
   int aux = *a; *a = *b; *b = aux;
}
```

Alerta rojo: por supuesto que si se quiere usar este sub-programa en algún otro sub-programa hay que definir en él las variable a las que se quiere intercambiar los valores. Como ejemplo, motramos el siguiente main().

```
void main(){
   int x = 8, y = 6;
   swap(&x, y&);
   printf("\nx = %d, b = %d.",x, y );
}
```

Note que antes de usar el sub-programa **swap** hemos debido haber decalrado dos variables enteras que son a las que se quiere intercambiar los valores y que puesto que **swap** espera como parámetros dos apuntadores debemos usar $\&a$ y $\&b$ en el momento de invocarlo.

Cuando queremos hacer un sub-programa que nos permita modificar más de un valor del espacio de estados de un programa—me explico más de una variable del programa que lo invoca, es necesario devolverlos o modificarlos usando apuntadores—si se quiere: "pasando los parámetros por referencia".

A continuación presentamos un programa que lee tres enteros de la entrada estandard y escribe en la salida estandard el valor del mínimo y del máximo de dichos enteros. Además lo hace usando tres sub-programas esclavos que leen los datos, determinan el mínimo y el máximo y finalmente otro que ecribe los valores del mínimo y del máximo. Note que puesto que x, y son enteros y **swap** espera dos apuntadores hay que hacer la invocación con $\&x$ y $\&y$ y no con x, y.

```
#include <stdio.h>

void leeData(int *a, int *b, int *c){
   printf("\nAmo dame tres enteros: ");
   scanf("%d%d%d", a, b, c);
}

void maxMin(int a, int b, int c, int *min, int *max){
   *min = a;
   if (b < *min) *min = b; if (c < *min) *min = c;
   *max = a;
   if (b > *max) *max = b; if (c > *max) *max = c;
}

void escribeReporte(int min, int max){
   printf("\nEl minimo es %d y el maximo es %d.", min, max);
}

int main(){
   int x,y,z,m,M;
   leeData(&x,&y,&z);
   maxMin(x,y,z,&m,&M);
   escribeReporte(m,M);
```

```
   return 0;
}
```

En el programa main() sólo se declaran las variable necesarias y se invoca adecuadamente a los sub-programas esclavos. Observe lo simple de las primeras tres líneas del código del sub-programa *maxMin*; El algoritmo consiste en tomar al primer elemento en este caso a como el mínimo, y comparar uno tras otro los que faltan con el mínimo, actualizando el mínimo si alguien es más pequeño que el mínimo actual. Este algoritmo es optimal en el número de comparaciones que hace, que significa que no hay uno mejor, ni lo habrá. Sin embargo se puede complicar tremendamente para que haga sólo una asignación. A continuación se muestra el código que sustituye las primera tres líneas.

```
if (a < b) if (c < a) *min = c; else *min = a;
else if (c < b) *min = c; else *min = b;
```

Lo aclaro, si $a < b$, entonces b no puede ser el mínimo, luego basta comparar c con a para decidir quien es el mínimo. De lo contrario, $a \geq b$ y basta comparar a b con c para saber cuál es el mínimo.

Para hallar tanto el máximo como el mínimo se puede mejorar a cambio de perder simplicidad y pensar un pooooco más.

A continuación se muestra una versión mejorada del código anterior. Este hace sólo tres comparaciones y a lo sumo cuatro asignaciones. El anterior hacía cuatro comparaciones y a lo sumo cuatro asignaciones también. ¿Por qué?

```
void maxMin(int a, int b, int c, int *min, int *max){
   if (a<b){*min = a; *max = b;} else {*min = b; *max = a;}
   if (c < *min) *min = c;
   if (c > *max) *max = c;
}
```

A continuación se muestra otro ejemplo que permite hallar el mínimo, la mediana y el máximo de tres enteros. Dicho programa usa entre sus subprogramas a **swap**.

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

void leeData(int *a, int *b, int *c){
   printf("\nAmo dame tres enteros: ");
   scanf("%d%d%d", a, b, c);
}

void swap(int *a, int *b){
   int aux = *a; *a = *b; *b = aux;
```

```
}

void minMedMax(int a, int b, int c,int *min,int *med,int *max){
   if (a<b){*min = a; *max = b;} else {*min = b; *max = a;}
   *med = c;
   if (*med < *min) swap(med,min);
   if (*med > *max) swap(med,max);
}

void reporte(int a, int b, int c){
   printf("\nEl orden es %d, %d, %d.", a,b,c);
}

int main(){
   int a,b,c, x,y,z; // x,y,z tiene los valores en orden.
   leeData(&a,&b,&c);
   minMedMax(a,b,c,&x,&y,&z);
   reporte(x,y,z);
   return 0;
}
```

Pongale especial cuidado a las declaraciones de los parámetros y a la posterior invocación según el caso. Analice el algoritmo usado en el procedimiento *minMedMax*.

1.2. Problemas Resueltos

1.3. Problemas Propuestos

1.

2.

Capítulo 2

Arreglos y Cadenas de Carácteres

En este capítulo presentaremos un nuevo tipo de dato que se ha dado por llamar arreglo. Formalmente un arreglo es un conjunto de variables del mismo tipo que se almacenan en memorias consecutivas y que se acceden con un mismo nombre más un índice. Veremos más adelante que en C al declarar un arreglo lo que se declara es en el fondo un apuntador más algunas reglas de juego. La idea es mantener simple la manipulación sin pensar en los apuntadores. Pero en ocaciones conviene saber lo que está detrás de bastidores.

Así para mantenerlo simple, por ahora, conviene pensar que un arreglo es un conjunto de cajitas en cada una de las cuales se puede almacenar un dato, pero cada uno los datos del mismo tipo.

2.1. Arreglos

Al declarar una variable de tipo arreglo se le solicita al sistema operativo que reserve las memorias suficiente para almacenar un determinado número de variables de un tipo determinado debidamente especificado en la declaración. A continuación lo mostraremos con algunos ejemplos.

```
int a[5]; // Arreglo de enteros de tamaño 5 de nombre a, pero no inicializado.
int bb[6] = {1,2,3}; //Declara e inicializa un arreglo de tamaño 6, de nombre bb.
```

Es importante hacer notar que en el primer caso se declara un arreglo de enteros de tamaño 5, pero por no haber sido inicializadas, las cinco cajitas contienen cualquier valor entero (basura), mientras en el segundo caso se decclara un arreglo de enteros de tamaño 6 y las primeras tres entradas contienen los números indicados en el orden indicado y las restante tres contienen ceros. Así el arreglo del segundo ejemplo luce como sigue:

bb[0]	bb[1]	bb[2]	bb[3]	bb[4]	bb[5]
1	2	3	0	0	0

Por supueto que esto es sólo una muestra del los tipos de declaraciones que podemos hacer. Mosremos otros:

```
int n = 4;
float b[2*n]; // Arreglo de flotantes de tamaño 8 de nombre b.
char s[] = {'H','o', 'l','a', '\0'};//Arreglo de carácteres, de nombre s.
char c[] = "Mundo";
```

En el primer caso de este ejemplo se declara un arreglo de tipo punto flotante de nombre *b* y no se inicializa. En el segundo caso se muestra una manera de declarar un arreglo de carácters de nombre *s* que se inicializa con los carácteres correspondientes a las letras de la palabra **Hola** más el carácter `'\0'`. Y finalmente, el tercer caso declara y inicializa un arreglo de carácters de nombre *c* y se inicializa con los carácteres correspondientes a las letras de la palabra **Mundo**, más al carácter `'\0'`. Nótese que en el primer caso el carácter `'\0'` hay que indicarlo explicitamente mientras que el segundo caso el mismo está implicito.

A continuación se muestra el resultado de estas dos declaraciones.

s[0]	s[1]	s[2]	s[3]	s[4]
'H'	'o'	'l'	'a'	'\0'

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]
'M'	'u'	'n'	'd'	'o'	'\0'

Note se agrega el carácter `'\0'` como medio de indicar que si el arreglo de carácteres fuera más largo que lo usado los carácteres después del carácter `'\0'` no son importantes. Así que cuando se manejan cadenas, para que las mismas estén formadas deben terminar (tener en algún lugar) un carácter `'\0'`. El arreglo se pudo haber declarado, por ejemplo, de tamaño 20 y sólo estar usando, por ejemplo, ocho de sus casillas.

Es muy importante notar que al declarar un arreglo se debe indicar directa o indirectamente el tamaño del arreglo para que el sistema operativo pueda reservar la memoria adecuada. Así que la declaración `int c[];` es incorrecta salvo al declarar los parámetros de una función, en cuyo caso es equivalente a `int *c;`. Y en dicho caso debe interpretarse como: "Lo que se espera es un arreglo de tipo entero, esto es, la dirección de memoria de un arreglo de enteros, sin importar su tamaño. Por ello es totalmente equivalente al declarar una función que recibe un arreglo *a* anotar `int a[];` o `int *a;` , aunque por claridad, al menos como principiante, es conveniente anotar la primera de ellas; así se indica que no es cualquier apuntador sino uno a un arreglo de enteros.

Otra forma de declarar un arreglo es declararlo con un apuntador y luego solicitarle memorria usando la función *malloc*. Por ejemplo:

```
int a [] = {1, 2, 3, 4},  *b, k, n = 4;
```

```
b = malloc(n*sizeof(int));
for(k=0; k<n; k++) b[k] = a[k];
```

En la primera íinea se declara una variable de tipo arreglo de enteros de nombre a de tamaño cuartro y sus cuatro elementos valen respectivamente: $a[0] = 1, a[1] = 2, a[2] = 3, a[3] = 4$ y también se declara un apuntador a un entero de nombre b y un entero. La segunda solicita memoria para n enteros y la hace que b apunte a su primer elemento, para ello se usa la función **malloc**. Con ello hace que b se comporte casi como un arreglo. La siguiente instrucción hace que en el arreglo b sea inicializado con los valoves de a, esto es copia uno por uno los elementos de a en b.

Note que no se puede pretender copiarlos todos de una sola vez, hay que hacerlo uno por uno. Si se intente la instrucción `a = b;` lo que logra es poner al apuntador b a apuntar a donde apunta a. con lo cual si se modifica a $b[i]$ se modifica a $a[i]$.

También se puede inicializar un arreglo solicitando los valores al usuario por la entrada estandard. Ejemplo de ello veremos luego. Además también cuando estudiemos los archivos veremos como inicializar los valores de un arreglo a partir de valores almacenados en memoria secundaria—entiendase: en un disco duro, por ejemplo.

Al declarar un arreglo de denteros de tamaño, digamos n, y de nombre a es como si se declarara n variables enteras con nombres $a[0], a[1], \cdots a[n-1]$, que se almacenan en posiciones consecutivas, pero a su vez se declara una variable de tipo apuntador de nombre a que contiene la dirección de la primera de estas variables.

Para ilustrar cómo se programa en C usando arreglos enpezaremos mostrando cómo se lee y escribe los arreglo y para ello usaremos definiremos las funciones apropiadas.

2.1.1. Lectura y Escriturade Arreglos

Si queermos leer los primeros n elementos de un arreglo, el primer paso es definir una función que modifique alguna variable de tipo arreglo del programa que la invoque o alguna variable global; si por ejemplo su nombre queremos que sea *leeArreglo* anotamos:

```
void leeArreglo(int a[], int n){...}
```

`int a[]` es la manera de decir que la función recibe un arreglo de enteros, y el `int n` es la manera de decir que sól nos importa leer sus primeras n entradas. Es la forma de laograr generalidad.

Como debemos leer uno por uno los elementos de a, necesitamos una variable local que indique que cassilla del arreglo estamos leyendo. y hacer un ciclo con dos intrucciones a saber: perdir el valor y leerlo. Una forma es como sigue:

```
   int k;
   for(k=0; k<n; k++){
      printf("\na[%d] = ",k);
      scanf("%d", &a[k]);
   }
```

Con lo que nos queda el siguiente sub-programa:

```
//Pre: n > 0; Post: Lee n enteros en a
void leeArreglo(int a[], int n){
   int k;
   for(k=0; k<n; k++){
      printf("\na[%d] = ",k);
      scanf("%d", &a[k]);
   }
}
```

Si ahora queremos escribir un sub-programa que escriba los n primero elementos de un arrglo a de nuevo tenemos que empezar por darle un nombre y decir cuales son sus parámetros de entrada y de salida.

```
void escribeArreglo(int a[], int n){...}
```

Ahora tanto a como n están "donados por Dios"—significa: están dados, no tenemos que leerlos. Como antes necesitamos una variable para recorrer los elementos del arreglos imprimiendolos. Los *printfs* fuera del ciclo son para asegurar un formato de salida adecuado. El resultado es:

```
//Imprime los primeros n elementos del arreglo a
void escribeArreglo(int a[], int n){
   int k;
   printf("\n[ ");
   for(k = 0; k < n-1; k++) printf("%d, ",a[k]);
   printf("%d ]",a[k]);
}
```

Si por último queremos probar que esto, en efecto, funciona debemos escribir un programa que los use. En él básicamente debemos declarar las variables necesarias e invocar adecuadamente a los sub-programa en el orden adecuado y con los parámetros adecuados. Un ejemplo puede ser el siguiente *main()*.

```
void main() {
   int n = 10, b[n];
   leeArreglo(b,5);
   escribeArreglo(b,5);
}
```

Esto declara un arreglo de tamaño 10 y lee las cinco primeras entradas del arreglo b y posteriormente para corroborar que, en efecto, las lee bien las manda a escribir. Todo esto puesto en un sólo archivo luce como:

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

//Pre: n > 0; Post: Lee n enteros en a
void leeArreglo(int a[], int n){
   int k;
   for(k=0; k<n; k++){
      printf("\na[%d] = ",k);
      scanf("%d", &a[k]);
   }
}

//Imprime los primeros n elementos del arreglo a
void escribeArreglo(int a[], int n){
   int k;
   printf("\n[ ");
   for(k = 0; k < n-1; k++) printf("%d, ",a[k]);
   printf("%d ]",a[k]);
}

//Lee y escribe los primeros cinco elemento de un arreglo.
void main() {
   int n = 10, b[n];
   leeArreglo(b,5);
   escribeArreglo(b,5);
}
```

A continuación se muestran algunas de los subprogramas desarrollados en clase y un programa principal que permite probar algunos de ellos. Se dan con el fin de qué los analice, pruebe los que faltan y pueda construir otros similares...

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

//Pre: n > 0; Post: Lee n enteros en a
void leeArreglo(int a[], int n){
   int k;
   for(k=0; k<n; k++){
      printf("\na[%d] = ",k);
      scanf("%d", &a[k]);
   }
}

//Imprime los primeros n elementos del arreglo a
void escribeArreglo(int a[], int n){
   int k;
   printf("\n[ ");
   for(k = 0; k < n-1; k++) printf("%d, ",a[k]);
   printf("%d ]",a[k]);
}

//Post: suma los elementos de a
int sumaArreglo(int *a, int n){
   int k, s = 0;
   for(k= 0; k<n; k++) s = s + a[k];
   return s;
}

//Halla el minimo del arreglo
int minArreglo(int a[], int n){
   int k = 0, min = a[0];
   while(k<n) if (a[k]< min) min = a[k];
   return min;
}

//Determina si x pertenece a a.
int estaEn(int x, int a[], int n){
   int k = 0;
   while (k<n-1 && a[k] != x) k++;
   return x == a[k];
}

//Intercambia los valores en las memorias a y b
```

```
void swap(int *a, int *b){
   int aux = *a; *a = *b; *b = aux;
}

//Ordena los elementos de a en forma no decreciente.
void ordena(int a[], int n){
   int k,j,m;
   for(k = 0; k< n-1; k++){
      m = k;
      for (j = k+1; j < n; j++) if (a[j]< a[m]) m = j;
      swap(&a[m],&a[k]);
   }
}

int main(){
   int n;
   printf("\nDime un entero positivo: "); scanf("%d", &n);
   int a[n];
   leeArreglo(a,n);
   escribeArreglo(a,n);
   ordena(a,n);
   escribeArreglo(a,n);
   return 0;
}
```

2.2. Cadenas de Carácteres

Empecemos por hacer énfasis en que existe una gran diferencia entre un arreglo de carácteres y una cadena de carácteres. El primero es simplemente un dato concreto que ya definimo antes. Una cadena de carácteres es un dato abstracto que se representa en C usando un arreglo de carácteres que tiene el carácter `'\0'` en alguna de sus entradas para indicar que la cadena la constituyen los carácteres antes de él. Así la cadena vacía se representa por ejemplo con `""` o por `int s[10] = {'\0'};` El primer caso se trata de una constante de tipo cadena mientras que en el segundo caso es un arreglo de caracteres de tamaño 10 en el que se alnacena en su primera casilla un carácter `'\0'` para representar a la cadena vacía.

Hay también que tener presente `'a', ';', '\t'` etc, representa un entero y que por lo tanto se pueden usa en cualquier lugar donde se requiera una expresión entera. además que al declarar una variable entera a la misma se le puede asignar cualquiera de estos carácteres, como `int k = 'a';`.

a continuación se muestra una tabla con... carácteres especiales... etc...
Falta tabla

Para leer o escribir una cadena se puede usar las funciones siguientes:

```
char s[80];
scanf("\%s", s);
printf("\%s",s);
s = gets();
puts(s);
```

Con la limitación que `scanf("%s",s);` sólo puede leer palabras; si se introduce un espacio en blanco se interrumpe la lecctura. `gets(s)` y `puts(s)` toman de la entrada estandard o colocan en la salida estandard respectivamente a la cadena de caracteres *s*.

Así la siguientes líneas piden al usuario su nombre y lo escriben de nuevo en la salida estandard.

```
char s[80];
printf("\nAmo como te llamas? ");
puts(gets(s));
```

Por supuesto que una cadena se puede también leer caraçter por carácter de la entrada estandar sin usar *gets*, usando *getc* Ello se puede hacer, por ejemplo usando el siguiente código:

```
int k; char s[80];
while ((s[k] = getc()) !='\n')   k++;
s[k] = '\0';
```

Uno podría estar interesado en saber la longitud de una cadena para ello puede usar el siguiente código: Que se basa en el hecho de que una cadena es un arreglo de carácteres que termina en un '\0'.

```
int longCad(char s[]){
   int n = 0;
   while (s[k] != '\0') n++;
}
```

También podría estar interesado en invertir la cadena o producir una copia de la misma. De nuevo para copiar la cadena s en el arreglo t y producir la cadena t nos vasamos en el hecho de que eventualmente una de las entradas de s es el carécter '\0'. El siguiente subprograma copia la cadena s en la cadena t:

```
//Post: Copia la cadena s en la cadena t
void copia(char s[], t[]){
   int k = 0;
   while (s[k] != '\0') {t[k] = s[k]; k++;}
   t[k] = '\0';
}
```

Para invertir las cadena se puede, por ejemplo, usar el siguiente sub-programa, invoca al sub-programa **swap** estudiado con anterioriada. Usamos el hecho de que en C un **char** es también un entero .

```
vooid invertir(char s[]){
   int n = 0, k;
   while (s[k] != '\0') n++;
   for (k = 0; k<= n/2; k++) swap(s[k],s[n-1-k]);
}
```

Otras funciones útilies para leer carácteres son `getchar()` y `putchar()` Para ilustrar su uso motramos el siguiente sub programa que copia de la entrada estandard a la salida estandard haciendo un eco.

```
void eco(){
   int c;
   c = getchar();
   while(c != EOF){
      putchar(c);
      c = getchar();
   }
}
```

2.2.1. Librería string.h

En la librería string.h se encuentra un conjunto de sub-programas que facilitan la manipukación de las cadenas de carácteres. Se recominda al lector echarle un vistazo. Inconcluso.....

Una frese es palíndrome si fuera de los espacios en blanco se lee igual de izquierda a derecha que de derecha a izquierda. A continuación se muestra un programa que permita decidir si una frase es palíndrome.

```
#include <stdio.h>
#include <stdlib.h>
//Lee cadena
void leeCadena(char s[]){
    printf("\nDame una cadena: ");
    gets(s);
}

//Canbia las mayusculas a minusculas
void lowCase(char s[], char t[]){
  //Completar
}

//Copia s en t sin los blancos
void quitaBlancos(char s[], char t[]){
   int j = 0, k = 0;
   while(s[k] != '\0'){
      if (s[k] != ' ') {t[j] = s[k]; j++;}
      k++;
   }
   t[j] = '\0';
}

int palindrome(char s[]){
   int k,n;
   for (n=0; s[n] != '\0'; n++);
   for (k=0; k<n/2; k++) if (s[k] != s[n-1-k]) return 0;
   return 1;
}

void reporte(char s[], char t[]){
   if (palindrome(t)) printf("\nLa frase: %s, es palindrome",s);
   else printf("\nLa frase %s No es palindrome.",s);
}

int main(){
   char s[80], t[80];
   leeCadena(s);
   quitaBlancos(s,t);
   reporte(s,t);
   return 0;
}
```

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

int r;

int * fac(int n){
   int k;
   r = 1;
   for(k = 0; k<n; k++) r *= k+1;
   return &r;
}

int main(){
   int n = 5;
   printf("\n%d! = %d.",n, *fac(n));
   return 0;
}
```

Copia de la entrada estandard a la salida estandard...

```
void copia(){
   int c;
   c = getchar();
   while(c != EOF){
      putchar(c);
      c = getchar();
   }
}
```

2.3. Arreglos Bidimensionales

Un arreglo bidimencional de elementos de tipo <tipo> es un arreglo cuyos elementos son arreglos de tipo <tipo>. Al declarar, por ejemplo: `int a[2][3];` estamos declarando un erreglo de tamaño dos cuyos elementos son cada uno arreglos de enteros de tamaño tres.

Al igual que con los arreglos unidimensionales se puede inicializar sólo en la declaración, pero además, si se desea inicializarlo se debe indicar explicitamente el número de elementos de cada uno de los arreglos que contituyen el arreglo que se declara. Por ejemplo:

```
int a[][3] = {{1,2,3}, {4,5,6}};
```

Declara un arreglo que contiene dos elementos que a su vez son arreglos. Un arreglo de arreglos se puede ver como una "matriz" en la que cada fila de la matriz es uno de los elementos del arreglo. Por supuesto que se pudo haber indicado explicitamente que la matriz tiene dos filas, pero ello no es necesario porque táxitamente se están asignando dos arreglos. Para visualizar lo anterior, la declaración anterior se pudu haber escrito como sigue:

```
int a[][3] = {{1,2,3},
              {4,5,6}
                    };
```

Con el fin de escribir menos, omitiendo las llaves internas, también se pudo haber anotado como:

```
int a[][3] = {1,2,3,4,5,6};
```

Por supuesto que si omitimos algunas entradas, las mismas se completan con ceros para completar la fila imcompleta. así `int a[][3] = {1,2,3,4};` es equivalente a `int a[][3] = {1,2,3,4,0,0};`. Y si deseamos inicializar un arreglo de 2×3 con ceros podemos anotar: `int a[2][3] = {0};`

Lo que no se permite es en una declaración donde se inicializa indicar las dimensines del arreglo usando una variable o una constante aún cuado se haya declarado e inicializado, por ejemplo los siguientes declaraciones son incorrectas:

```
int m = 5, b[m] = {1,2,3,4,5}; // Debería ser b[5] = {1,2,3,4,5};
int n = 3, a[][n] = {1,2,3,4,5,6}; // a[][3] = {1,2,3,4,5,6}
```

si no se sabe de antemano el tamaño de arreglo conviene declarar variables que representen el tamaño del mismo leerlas y luego declarar el arreglo, pero de esta forma no se puede inicializar en conjunto sino uno por uno los elementos del arreglo usando dos ciclos anidados. Por ejemplo:

```
int m, n;
// Código para leer m y n
int a[m][n];
// Código para inicializar el arreglo
```

Cuando se declara un arreglo bidimensional como parámetro de una función se debe indicar al menos el tamaño de la segunda dimención, ello se puede hacer de una de las siguientes tres formas:

1. Indicar con un valor entero constante (no sirve una contante entera), por ejemplo: `<tipo> f(<tipo> a[][5]){...}`
2. Indicar con una variable que se haya definido antes en un parámetro anterior, como por ejemplo: `<tipo> f(int n, <tipo> a[][n]){...}`
3. O usar una variable global que haya sido inicializada adecuadamente en el valor deseado como tamaño del arreglo. Ejemplo si n es una variable globar `<tipo> f(<tipo> a[][n]){...}` .

2.4. Problemas

1. Escriba una función en C que permita determinar posición de la primera ocurrencia de un entero x en las primeras n entradas de un arreglo de enteros a. Si x no ocurre en dichas n primeras entradas debe devolver n.

2. Escriba una función que reciba un arreglo de enteros a y un número natural $n > 0$ y devuelva la posición de la primera ocurrencia del mínimo del arreglo.

Capítulo 3

Registros y Archivos

En este capítulo presentaremos una forma de construir nuevos tipos de datos a partir de dados más simples ya existentes. Además mostraremos como se puede almacenar y recuperar la información usando memoria secundaria. (Permanente... explicarlo...)

3.1. Registros

Un registro es una plantilla que define un nuevo tipo de datos. La noción de registro en computación aparece por primera vez en el lenguajes Pascal.[1] Sin embargo la noción de registro en la vida cotidiana es muy anterior a su aparición en computación. Antes de la aparición de las computadoras las tiendas de casi cualquier tipo de mercancía tenían por costumbre utilizar unas fichas por cada uno de los productos que se vendían en la tienda donde se incluía, por ejemplo, el nombre del producto, su código, su precio, el número de unidades en existencia, el número promedio de salida por mes, etc.

Un registro es una plantilla en la que se pueden anotar los valores espécificos de un determinado objeto. Por ejemplo, si es en estudiante, uno podria estar interesado en: su carnet, su nombre, su indice academico, su carrera, etc. Si usaramos una ficha para ello, ella luciría como:

Plantilla de Tipo Estudiante

Campos	Valores
Carnet:	
Nombre:	
Indice Academico:	
Carrera:	

Así, declarar un registro es definir una plantilla.

[1] El lenguaje Pascal fue desarrollado por el Profesor Suizo Niklaus Wirth (Instituto tecnológico de Zurich, Suiza) en 1968, aunque su primer compilardor completo no apareció sino hasta finales de 1970.

En en el lenguaje C, hay varias maneras de declarar (si se quiere definir) un registro. A continuación lo explicaremos con varios ejemplos.

En Pascal se llaman **record**.

Para declarar un registro en C se usa la palabra reservada **struct**[2] seguida de un bloque donde se anotan las declaraciones de las variables que conforman el registro y éste esté seguido de un punto y coma. Entre la palabra **struct** y el bloque se puede—no es obligado, pero si, a veces, conveniente— agregar un identificador que llamaremos "un alias". Conviene que ese identificador capture el fin de la plantilla y no sea muy largo. También conviene, por cuestion de estilo, que el identificador para el *alias* se escriba todo en letras minusculas. Entre la llave que cierra el bloque y el punto y coma se pueden anotar intancias de la plantilla que se define.

Para el ejemplo anterior la declaración en lenguaje C luce como:

```
struct student {
   char carnet[9]; // 11-10010  7 números el guión y el '\0'
   char name[26];
   float indA;
   int mayor;
} a, b ;
```

Una vez declarado Observaciones importantes:

1. Esta declaración es análoga a la declaración `int a, b;`. En la primera se declaran dos variables con esa estructura nueva de nombres a y b respectivamente, mientras que en el segundo caso se declaran dos variables enteras de nombres a y b. Todo lo anterior a: "`a, b;`" haces las veces del `int`

2. Un registro se puede declarar en cualquier parte de un programa. Su alcance por supuesto es el bloque donde se ha definido. Si se define por ejemplo en el main(), ni la plantilla ni la variables definidas con él se pueden usar dentro de otros sub-programas. Esto hace que para que tenga un uso global, se deba declarar la plantilla antes de todos los sub-programas.

3. Lo bueno no omitir el alias es poder declarar las variables en donde las necesite y que no sean todas globales. Si se dispone de un alias el mismo se puede usar para definir variables de dicho tipo en cualquer sub-programa. Por ejemplo dentro de cualquier sub-programa se pueden declarar variables de este nuevo tipo como sigue:

   ```
   struct estudent x, y, *p, z[20];
   ```

   que declara dos variables de tipo "struct student" de nombres x y y, un apuntador a variables de tipo "struct student" y un arreglo de tamaño 20 de elementod de tipo "struct student".

[2] La elección de este vocablo se debe posiblemente a que ello es el primer paso para definir algo que se suele llamar una **estructura de datos**.

4. El alias es particularmente útil si queremos pasarle una variable, o un arreglo, o un apuntador a una variable de tipo estructura como parámetro a un sub-programa.

5. En la mayoría de los casos conviene declarar la palntilla antes de todos los sub-programas, darle un alias y no declarar las variables sino cuando convenga, pues de esta manera las variables no son globales y se tiene más control sobre ellas.

Si no se usa el alias las declaraciones de las instancias de un **struct** sólo se pueden hacer junto con la declaración del struct y justo despúes de la llave que cierra. Si se usa el alias se pueden hacer en cualquier parte usando la frase **struct** $<$ *alias* $>$. Además se pueden inicializar en la declaración dandole los valores a los campos en el mismo arden en el que aparecen en la declaración como se muestra en el siguiente ejemplo.

```
struct contacto x ={"Maria Lopez", "04148686868", "mflopez@hotmail.com", 'F', 19};
struct contacto y ={"Pedro Perez", "04128882566", "pp@gmail.com", 'M', 20};
struct contacto c, d, *p, f[10];
```

Se declaran las valiables x, y, c, d, p, f. Todas salvo la p del tipo struct contacto, la p es un apuntador a un struct contacto. Las dos primeras se inicializan. Y la f es un arreglo para dar albergue a 10 bariables de tipo struct contacto.

Las diferentes sub-variables de un registro se suelen llamar miembros, campos o atributos y se tiene acceso a ellas usando el operador punto (.). Por ejemplo, en el ejemplo anterior la variable x.name tiene el valor "Maria Lopez", mientras que y.sexo tiene el valor 'M'. Si quisieramos saber el valor del registro almacenado en la primera casilla del arreglo f usaríamos f[0].name. Es importante tener en ccuenta que un miembro de un registro puede a su vez ser un arreglo o un registro o un apuntador.

Además, si el apuntador a un struct contacto p se le asigna la dirección de memoria de un registro, por ejemplo y, entonces otra manera de usar algún atributo del registro y es es usando el operador flechita (->). Así p->mail es el email de la varable y. Por supuesto que si p apunta a NULL no tiene sentido hablar de p->mail.

Es importante hacer notar que los registros a diferencia de los arreglos no son apuntadores, pero que como cualquier otra variable se le puede solicitar su dirección de memoria y hacer que un apuntador apunte a ella. Además, a diferencia de los arreglos, se pueden asignar y pueden ser devueltos por una función. Así por ejemplo la instrucción `c = x` tiene el efecto de darle a c los mismo valores que tenía x, produciendo así una copia de x.

A continuación se muestra un programa que completo que permite ilustrar el uso básico de un registro.

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

struct contacto {
   char name[26];
   char cell[17];
   char mail[26];
   char sexo;
   int  edad;
} ;

// Lee un contacto desde la consola
void leeContacto(struct contacto *b){
   printf("\nNombre? "); gets((*b).name);
   printf("\nNo. Cell? "); gets((*b).cell);
   printf("\nEmail? "); gets((*b).mail);
   printf("\nSexo M o F? "); (*b).sexo = getchar();
   printf("\nEdad? "); scanf("%d",&(*b).edad);
   getchar();//Toma el último return introducido
}

// Escribe un contacto en un archivo
void escribeContacto(struct contacto a){
   printf("%-25s%-16s%-25s%c %d\n", a.name, a.cell, a.mail, a.sexo, a.edad);
}

int main(){
   struct contacto a={"Maria Lopez", "04148686868", "mfl@hotmail.com",'F',19};
   struct contacto b={"Pedro Perez", "04128882566", "pp@gmail.com", 'M', 20};
   struct contacto c, d, f[10];    int k;
   d = a; //Hace una copia de a en d.
   printf("Primera Prueba--uso lectura y escritura--:\n\n");
   printf("Dame un Contacto: \n");
   leeContacto(&c);
   printf("Imprimiendo los contactos a, b, c, d\n\n");
   escribeContacto(a); escribeContacto(b);
   escribeContacto(c); escribeContacto(d);
   system("pause"); system("cls");
   printf("Segunda Prueba==uso de arreglos--\n\n");
   printf("Introduzca los contactos que le solicitan:\n");
   for (k=0; k<3; k++)  leeContacto(&f[k]); //Lee tres contactos en f
   system("pause"); system("cls");
   printf("Los contactos leidos fueron:\n");
   for (k=0; k<3; k++)  escribeContacto(f[k]);
   return 0;
}
```

3.1.1. La función sizeof

La función sizeof permite determinar el número de bytes que se requieren para almacenar un dato de cualquier tipo. Por ejemplo, sizeof(int) da como salida el número de bytes que usa el sistema donde se está ejecutando para representar a un entero. En nuestro caso problemente es 4. Si se declara `int n`, entonces sizeof(n) tiene el mismo efecto que el anterior.

En particular es interesante que Si se declara `int a[10]` y se pide sizeof(a) el resultado es justo $10 * sizeof(int)$, o sea que $sizeof(a)/sizeof(a[0])$ es justamente el tamaño del arreglo a.

Pues bien tambien podemos usar la función sizeof para obtener el número de bytes que se usan para almacenar un registro, la solpresa es que puede resultar ligeramente superior a la suma de lo nesesario para almacenar sus atributos. ¿'Por qué? En el caso de **struc contacto** la suma da 74 y la función dice que es 76—dos bytes de más. A propósito como el tipo es **struc contacto**, debe invocarse como: sizeof(struc contacto) o usar el nombre de una variable de dicho tipo.

3.2. Unión

En algunas ocasiones es necesario tener una variable que pueda almacenar dos o mas tipos de datos, por ejemplo un entero o un flotante o un char. Por supuesto que en dicha variable se almacenará un sólo dato, pero lo que se quiere es que pueda ser de un de esos tipos. Esto es que si le asigno cualquiera de ellos se pueda representar bien. el lenguaje C permite definir este tipo de variables usando la palabra resevada **union** y dando una lista de los posibles tipos de datos que se quieran almacenar. La sintaxis es similar a la de un registro. A continuación de de un ejemplo:

```
union  numero  {
   int ent;
   float flo;
};
```

Al declarar `union numero m,n;` se están declarando y reservando espacio de memoria para almacenar un número que puede ser tanto entero como punto flotante. Es responsabilidad del programador saber cuál fué el último tipo de dato que almacenó para extraerlo adecuadamente. Al igual que con los registros se pudo haber declarado las variables en la definición del tipo, pero de nuevo con las misma restricciones de alcance.

Al igual que con registro se acceden usando el operador punto.

3.3. Definición de Constantes Enumeradas

El lenguaje C permite definir un conjunto de **constantes enteras** que, en principio, guarden entre sí alguna relación. Para ello usa la palabra reservada **enum**, por enumeración, Seguida de un identificador opcional y de un bloque que contiene una lista de constantes. Si no se indica ningún valor la primera de dichas constantes toma el valor cero y las restantes toman los valores consecutivos: $1, 2, 3, \ldots$, a menos que se indique su valor explicitamente. Si sólo se indica el primer valor los siguientes toman los valores consecutivos, a menos que se indiquen sus valores. Las no indicadas toman sus valores a partr de la última indicada. En el sifuiente ejemplo se indic'p que el primer valor es cero, pero no haber indicado ninguno hubiese tenido el mismo efecto.

```
enum bool { no = 0, si};
```

Esto declara dos constantes de nombres no y si cuyos valores son respectivamente 0 y 1. En el siguiente ejemplo se definen cuatro contantes que pretenten contener los valores de la primera y última letra de los rangos a..z y A..Z.

```
enum letras {a = 'a', z = "z",  A = 'A', Z = 'Z'};
```

```
enum ds {Lun =0, Mar, Mie, Jue, Vie, Sab, Dom};
```

Para definir variable que guarde uno de dichos valores se puede bien anotarlas después del bloque y antes del punto y coma (;) separadas poe comas; o bien usar el nombre del tipo.

```
enum ds hoy, ayer = Mar;
enum bool esPalindrome;
```

Ejercicio 3.1 *Cuánto vale cada una de las constantes definidas por la siguiente instrucción:*

```
enum  {C1 =-2, C2, C3, C4, C5 = 5, C6, C7 = 12 };
```

3.4. Definición de Tipos

El lenguaje C provee un mecanismo para definir nuevos nombres a tipos de datos. El mismo usa la palabra reservada **typedef** tiene la siguiente sintaxis.

```
typedef <el_tipo_del_objeto> <nuevo_nombre_de_tipo>;
```

Por ejemplo

```
typedef float Medida;
```

Hace de la palabra Medida un sinónimo de la palabra float. El tipo Medida puede a partir de ahora usarse en una declaración que requiera un float, en un casting, para solicitar tamaño de una variable de tipo flotante o saber el tamaño de un flotante.

Otro ejemplo típico es definir un String como un apuntador a un char.

```
typedef char* String;
```

Esto define a **String** como un apuntador a un caracter, o si se quiere a una cadena de carácteres. Más adelante podremos usar la palabra String para declarar un apuntador a una cadena de carácteres y posteriormente solicitar memoria para la misma. (Por aclarar...)

Un ejemplo más interesante es definir un tipo en base a un registro. Usando el alias del registro y por supuesto asumiendo que ya definimos a **struct contacto** podemos escribr simplemente:

```
typedef struct contacto Contacto;
```

Para definir el nuevo nombre del tipo **struct contacto**. Y en adelante usar Contacto en cuarquier parte donde se requiera **struct contacto**.

Pero también podemos hacerlo desde cero y no usar si no queremos el alias en la definición.

Otra manera de definir Contacto sería:

```
typedef struct   {
   char name[26];
   char cell[17];
   char mail[26];
   char sexo;
   int  edad;
}  Contacto;
```

Note que no se uso el alias "contacto", pero se pudo haber usado. Ya no es necesario. Con cualquiera de estas declaraciones del tipo Contacto, se puede substituir las ocurrencias de **struct contacto** por Contacto. Por ejemplo:

```
Contacto v, z[12];
```

Definen respectivamente un Contacto de nombre v y un arreglo de 12 Contacto de nombre z.

Note que hemos usado mayúsculas para enfatizar que es un nuevo tipo. Se recomienda usar mayusculas para los tipos definidos con typedef.

3.5. Problemas Propuestos

1. Escribir un programa en C que permita imprimir el contenido de un archivo de texto en al salida estandard. El mismo debe solicitar al usuario el nombre del archivo que se desea mostrar, leer dicho nombre y escribir el contenido del mismo de 40 en cuarenta lineas. Si el archivo no existe debe dar un mensaje indicandolo. nombre

Capítulo 4

Manipulación de Archivos

En este capítolo presentaremos los conceptos básicos necesarios para manipular archivos de texto. Un archivo es un conjunto de información que se ha almacenado usando algún medio físico para para preservar su integridad. En particular un archivo de texto se almacena en algún dispositivo electrónico, pero la información se almacena en bytes que representan carácteres codificados.

Para almacenar o extraer la información en un dispositivo físico se asocia al dispositivo con un flujo—un flujo es una secuencia formada por los carácteres almacenados en, o que se quieren almacenar en el dispositivo, junto con un conjunto de operaciones que permiten manipular dicha secuencia.

4.1. Operaciones Básicas

Para manipular la información almacenada en un archivo se requiere de una estructura de datos adicional que si bien ya la hemos usado hasta los momentos no era necesario que nos percataramos de su existencia. se trata del tipo de datos abstracto ARCHIVO(o más bien FLUJO DE DATOS) que el lenguaje C denota como FILE. Es importante hacer énfasis que al ejecutar un programa cualquiera que requiera de entrada o salida de datos por cónsola se crean de manera automática un flujo de entrada llamado **stdin** y un flujo de salida que se llama **stdout**. Hemos venido trabajando con ellos casi que desde nuestra primera clase. Los flujos de datos se manipulan mediante una variable de tipo apuntador a un FILE (a un flujo)— nunca se declara una variable de tipo FILE. A continuación se mostrarán las opeciones básicas que se usan sobre un archivo—debería decir sobre un flujo de datos. no?.

Las operaciones básicas sobre un archivo son:

- crear un apuntador a un FILE.
- abrir el FILE.
- tomar o poner información en el FILE según sea el caso (leer o escribir).

- cerrar el FILE, al terminar.

4.1.1. Declaración de un Flujo

Para crear un apuntador se declara el mismo como sigue:

```
FILE *pArc, *e;
```

lo cual declara dos apuntadores a flujos de datos.

4.1.2. Inicialización de un Flujo

Para leer o escribir información en el flujo se requiere asociar al flujo con un archivo que se encuentre o que se creará en el sistema de archivos de la computadora donde se ejecutará el programa. Dicho archivo tiene un nombre con o sin extensión. Este proceso se conoce con el nombre de **abrir el archivo**. Es importante notar que un archivo se puede necesitar abrir para fines distintos, leer, escribir, escribir al finar del mismo, o alguna combinación de éstas. La sintexis del proceso de apertura es como sigue:

```
FILE * fopen(const char *fileName, const char *mode );
```

donde *fileName* es una cadena de carácteres que representa el nombre con el que aparece o aparecerá el archivo en el sistema de directorios y *mode* es una cadena de carácteres con uno de los siguientes valores.

- `"r"` abre el archivo para leer al principio de él.
- `"w"` crea y abre el archivo para escribir en él,
  si había algún contenido previo lo descarta.
- `"a"` abre o crea el archivo para escribir al final de él.
- `"r+"` abre el archivo en su principio para actualizar(i.e., leer o escribir).
- `"w+"` crea el archivo para escribir o leerl,
  si había algún contenido previo lo descarta.
- `"a+"` abre o crea el archivo para escribir al final de él y actualizar.

Los modos de actualización permiten tanto leer como escribir a la vez en un archivo, pero para ello deben usarse las funciones de posicionamiento y *fflush* entre lecturas y escrituras. (Lo aclararemos luego...)

Al invocar a la función fopen(), por ejemplo, con el fin de abrir al archivo *pepita* para leer de él con la siguiente instrucción `e = fopen("pepita", "r" );`, si se pudo abrir el archivo, en el apuntador e está la posición de memoria donde

se almacena el flujo de datos que contiene los carácteres presentes en el archivo físico pepita, e es su nombre lógico que debemos usar para refirienos a él de ahora en adelante. "?se entiende?, en caso contrario e toma el valor NULL (definido como cero). El intento de abrir puede fracasar, por ejemplo, debido a que el archivo no exista, o no esté en el directorio adecuado, o porque esté protegido para lectura, etc. Es conveniente preguntar si se pudo abrir el archivo antes de continuar.

4.1.3. Lectura de un Flujo

Para leer los carácteres de un archivo(de un flujo) se usan las siguientes funciones:

- `int fgetc(FILE *flujo)`
  retorna el siguiente carácter de flujo como un carácter sin signo, (convertido en un entero), o EOF si se está al final del archivo o ocurrió un error.
- `int getc(FILE *flujo)`
  es equivalente a fgetc excepto por el hecho de que si es un macro, puede evaluar flujo más de una vez.
- `char *fgets(char *s, int n, FILE *flujo)`
  fgets lee a lo sumo $n-1$ carácteres en el areglo s; se detiene si se encuentra al carácter *nuevalínea*, el carácter nuevalínea se incluye en el arreglo que se debe terminar con el carácter `'\0'`. Retorna s o NULL si ocurre EOF o error.
- `int fscanf(FILE *flujo, const char *formato, ... )`
  fscanf lee de flujo bajo el control de formato, y le asigna los valores convertidos a cada uno de sus argumentos correspondientes, cada uno de los cuales debe ser un apuntador. Termina cuando se termina de procesar formato. Retorna el número de items convertidos y asignados, o EOF en caso de llegar al fin del archivo o de error.

4.1.4. Escritura en un Flujo

Para escribir en un archivo se usan las siguientes funciones...

- `int fputc(int c, FILE *flujo)`
  fputc escribe el carácter c, (convertido en un unsigned char) en flujo. Retorna el carácter escrito, o EOF en caso de error.
- `int putc(int c, FILE *flujo)`
  Es equivalente a fputc excepto por el hecho de que si es un macro, puede evaluar flujo más de una vez.

- `int fputs(const char *s, FILE *flujo)`
  fputs escribe el string *s*, (que necesita **no tener** el carácter `'\n'`) en flujo. Retorna un entero no negativo, o EOF si ocurre un error.
- `int fprintf(FILE *flujo, const char *formato, ... )`
  fscanf convierte las directices de formato y escribe el string formateado en flujo. Retorna el número de caráctere escrito, o un entero negativo si ocurre un error.

4.1.5. Control de Fin de Flujo

Cuando estamos leyendo de un archivo, para saber si hemos llegado o no al fin del archivo se puede usar el valor retornado por la función que estamos usando para leer. Note que todas devuelven EOF en caso de error o fin del archivo. Tambien se puede usar la siguiente funión:

```
int foef(FILE *flujo)
```

que devuelve un valor no nulo si se está en el fin del archivo y cero si no.

4.1.6. Cerrar un Flujo

Finalmente cuando se termina de leer o escribir en un archivo hay que cerrar el archivo. Ello se hace mediante la función siguiente:

```
int fclose(FILE *flujo)
```

Es de hacer notar que al terminar el programa de forma normal se cierran todos los archivos que esten abiertos en dicho momento, pero si termina porque ocurre algún error los archivos que no se han cerrado no se actualizan.

4.2. Lectura y Escritura Básica en un Archivo

Para ilustrar lo hasta aquí presentado mostraremos varios ejenplos de programas que requieren el uso de archivos.

Ejemplo 4.1 *Escriba un programa en C que muestre en pantalla el contenido de un archivo de texto cuyo nombre se solicita al usuario, si el archivo existe y se pudo abrir, y en caso contrario dé un mensaje indicando que qno se pudo abrir el archivo. ¿Cómo puede modificar dicho programa para que se copie en otro archivo cuyo nombre también se le solicita al usuario.*

Explicación: Primero declaramos un String para leer el nombre del archivo y un carácter para leer cada uno de los carácteres del archivo. Luego pedimos y leemos el nombre, e intentamos abrir el archivo. Si pudimos abrir el archivo, mostramos su contenido con el **while**, que lee uno por uno los carácteres del

archivo *e* en *c* y los coloca en la salida standard (**stdout**). Finalmente cerramos el archivo. Si no lo pudimos abrir damos mostramos un error. A continuación se muestra el código en C.

```
#include <stdio.h>

int main(){
   char nameAr[30], c;
   printf("\nMuestro  un archivo.");
   printf("\nDime el nombre del archivo: ");
   gets(nameAr);
   FILE *e = fopen(nameAr, "r");
   if (e != NULL){
      printf("\nEl contenido del archivo es:\n\n");
      while ( (c = fgetc(e)) != EOF) fputc(c,stdout);
      fclose(e);
      printf("\n\nMision cumplida!!!");
   } else printf("\nEl archivo %s no se pudo abrir.",nameAr);
    return 0;
}
```

•

Otra versión más compacta de este programa y que usa la función *feof* es la siguiente:

```
#include <stdio.h>

int main(){
   char nameAr[30];
   printf("\nDime el nombre del archivo que quieres vizualizar: ");
   gets(nameAr);
   FILE *pe = fopen(nameAr, "r");
   if (pe != NULL){
      while ( (!feof(pe)) putchar(getc(pe));
      fclose(pe);
   } else printf("\nNo se pudo abrir el archivo %s.", nameAr);
    return 0;
}
```

Ejemplo 4.2 *Escriba un programa en C que escriba en un archivo de texto todo lo que se escribe en la consola. El nombre del archivo donde se escribirá se le solicita al usuario. Alerta rojo el usuario debe procurar dar un nombre que no exista pues de lo contrario se dañará su contenido.*

Explicación: Por ahora, tengan el resultado sin anestesia!!!

```
#include <stdio.h>

int main(){
   char nameAr[30], c;
   printf("\nEscribo en un archivo.");
   printf("\nDime el nombre del archivo: ");
   gets(nameAr);
   FILE *s = fopen(nameAr, "w");
   if (s != NULL){
      printf("\nAhora introduce el texto:\n\n");
      while ( (c = fgetc(stdin)) != EOF) fputc(c,s);
      fclose(s);
      printf("\n\nMision cumplida!!!Revisa en Management:Files.");
   } else printf("\nNo se pudo abrir el archivo %s.", nameAr);
    return 0;
}
```

•

Ejercicio 4.1 *¿Cómo puede modificar el programa anterior para que se chequee la no existencia del archivo donde se pretende escribir?*

Ejercicio 4.2 *¿Cómo puede modificar el programa anterior para que, si el archivo existe, en lugar de descartar su contenido, agregue el nuevo contenido al final?*

Ejemplo 4.3 *Escriba un programa que permita contar el número de líneas, palabras y carácteres que contiene un archivo de texto. El mismo debe pedir al usuario el nombre del archivo y usar dicho nombre para inspeccionar el archivo, si el archivo existe debe escribir por pantalla el número de líneas, palabras y carácteres que contiene, en caso contrario debe dar un mensaje indicando que el archivo no existe.*

Explicación:

Empecemos por analizar el problema; Puesto que nos piden que solicitemos al usuarioe el nombre del archivo debemos declarar una variable para almacenar el nombre solicitado, y una variable de tipo char y un apuntador a un archivo para poder leer los carácters del archivo. Está claro que dicho archivo lo debemos abrir para lectura. También debe ser fácil entender que debemos declarar tres variables enteras para ir acumulando el número de carácteres, palabras y líneas que contiene el archivo a medida que vayamos leyendo sus carácteres. Dichas variables se deben inicializar en cero.

Esto hace que las primeras líneas de nuestro programas sean:

```
int main(){
   char nameAr[30], c; int nl, nw, nc;
```

```
   printf("\nCuento las lineas, palabras y caracteres de un archivo.");
   printf("\nDime el nombre del archivo: ");
   gets(nameAr);
   FILE *e = fopen(nameAr, "r");
}
```

Puesto que se nos pide que el programa dé un mensaje si se fracasa al intentar abrir el archivo un esqueleto de lo que nos falta debe lucir como sigue. Que significa que vamos a ir tomando en en la variable *c* uno a uno los caracteres del archivo y actualizando los valores de los acumuladores nl,nw y nc según haga falta.

```
   if (e != NULL){
      nl = nw = nc = 0;
      while ( (c = fgetc(e)) != EOF){
         Actualizar los valores de nl, nw y nc;
      }
      printf("\nHay %d lineas, %d palabras y %d caracteres.\n",nl,nw,nc);
   } else printf("\nEl archivo %s no se pudo abrir.",nameAr);
    return 0;
```

Puesto que cada vez que entro al ciclo es porque se ha leido un carácter, cada vez que entre al ciclo se debe incrementar *nc* en uno; pero como se sabe que tenemos una nueva línea solo aparece el carácter `'\n'`, entonces cada vez que aparezca un carácter `'\n'` se debe incrementar en uno la variable *nl*. Esto hace que sólo nos falte analizar como resolver el problema de contar las palabras.

Lo primero que tenemos que entender que una palabra es una secuencia de carácteres que no contiene ninguno de los carácteres `' '`, `'\t'`, `'\n'` (blancos, tabuladores o saltos de línea).

La observación crucial es que sé que tengo una nueva palabra cuando estando en alguno de los tres carácterea anteriores paso a uno distonto de ellos, esto es, cuando estando fuera de una palabra entro en una. Para ello usando el tipo enumerado definiremos una variable de nombre *estado* que tome dos valores constantes OUT o IN que indiquen que estoy fuera o dentro de una palabra.

A parte de la declaración y la inicialización de la variable *estado*, esto agrega las siguientes cuatro líneas a nuestro código:

```
        nc++;
        if (c == '\n') nl++;
        if (c == ' ' || c == '\n' || c == '\t') estado = OUT;
        else if (estado == OUT) { estado = IN; nw++;}
```

Si ponemos esto junto resulta el siguiente programa en C

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

```
enum {OUT, IN} estado;

int main(){
   char nameAr[30], c; int nl, nw, nc;
   printf("\nCuento las lineas, palabras y caracteres de un archivo.");
   printf("\nDime el nombre del archivo: ");
   gets(nameAr);
   FILE *e = fopen(nameAr, "r");
   if (e != NULL){
      nl = nw = nc = 0; estado = OUT;
      while ( (c = fgetc(e)) != EOF){
        nc++;
        if (c == '\n') nl++;
        if (c == ' ' || c == '\n' || c == '\t') estado = OUT;
        else if (estado == OUT) { estado = IN; nw++;}
      }
      printf("\nHay %d lineas, %d palabras y %d caracteres.\n",nl,nw,nc);
      fclose(e);
   } else printf("\nEl archivo %s no se pudo abrir.",nameAr);
    return 0;
}
```

•

Ejemplo 4.4 *Escriba un programa en C que permita leer un archivo de texto, que luce como el ejemplo que se muestra luego, y escriba en otro archivo los mismos datos con dos columnas adicionales: la suma de segunda más la tercera columna, y el promedio de dichas columnas. El formato debe ser como se muestra en el ejemplo. Los nombres de los archivos de entrada y salida debe suministrarlo el usuario.*

Ejemplo de archivo de entrada:

```
Petra 12 18
Carolina 16 16
Jose 20 12
Ana 18 17
Juan 15 16
```

El archivo de salida para el archivo de entrada anterior debe lucir como sigue:

```
Nombre P1 P2 su mean
Petra 12 18 30 15.00
Carolina 16 16 32 16.00
Jose 20 12 32 16.00
```

```
Ana 18 17 35 17.50
Juan 15 16 31 15.50
```

Explicación: Las tareas básicas son: leer los nombres de los archivos de entrada y saliada, abrir dichos archivos, y si se pudieron abrir, mientras no se haya acabado el archivo de entrada, leer una linea en él y escribirla en el archivo de salida con las dos columnas adicionales que se piden. Por último se deben cerrar los archivos.

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

int main(){
   FILE *pe, *ps;
   char nEnt[30], nSal[30], name[15];
   int n1, n2;
   printf("Amo dime los nombres de los archivos de entrada y de salida: ");
   scanf("%s%s", nEnt, nSal);
   pe = fopen(nEnt, "r");
   ps = fopen(nSal, "w");
   if (pe != NULL && ps != NULL) {
      fprintf(ps, "%-15s%s\n", "Nombre", " P1 P2 su mean" );
      while (!feof(pe)) {
         fscanf(pe, "%s%d%d", name, &n1, &n2);
         fprintf(ps,"%-15s%3d%3d%3d%6.2f\n",name,n1,n2,n1+n2,(float)(n1+n2)/2);
      }
   } else printf("Error abriendo algun archivo!!!");
   fclose(pe);
   fclose(ps);
   return 0;
}
```

•

Ejemplo 4.5 *Escriba un programa en C que permita leer de un archivo cuyo nombre lo sumisnistra el usuario los datos de varios estudiantes, y escribirlos en otro archivo, cuyo nombre también debe suministrar el usuario, los datos ordenados de mayor a menor en base a la suma de la segunda y la tercera columna. La salida, para la entrada del ejercicio anterior, debería ser:*

```
Nombre P1 P2 su mean
Ana 18 17 35 17.50
Jose 20 12 32 16.00
Carolina 16 16 32 16.00
Juan 15 16 31 15.50
Petra 12 18 30 15.00
```

Explicación: A diferencia del ejemplo anterior, no podemos leer los datos en variables sencillas sino que tenemos que definir una estructura para almacenar los datos que llamaremos Estudiante. Estudiante es un nuevo tipo de datos. En el main declaramos un arreglo de Estudiantes en el que leeremos los datos, luego lo ordenaremos, y finalmente lo escribiremos en otro archivo. Para que sea más fácil conviene que el *struct* además del nombre y las dos notas tenga la suma de las dos notas.

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

struct est {char nom[15]; int n1, n2, s;};

typedef struct est Estudiante;

int main(){
   FILE *pe, *ps;
   char nEnt[30], nSal[30];
   Estudiante e[30], aux;
   int n = 0, i, j;
   printf("Amo dime los nombres de los archivos de entrada y de salida: ");
   scanf("%s%s", nEnt, nSal);
   pe = fopen(nEnt, "r");
   ps = fopen(nSal, "w");
   if (pe != NULL && ps != NULL) {
      fprintf(ps, "%-15s%s\n", "Nombre", " P1 P2 su mean" );
      while (!feof(pe)) {
         fscanf(pe, "%s%d%d", e[n].nom, &e[n].n1, &e[n].n2);
         e[n].s = e[n].n1 + e[n].n2;
         n++;
      }
      for (i = 0; i < n-1; i++)
         for (j = i+1; j < n; j++)
            if (e[j].s > e[i].s) {
               aux = e[j]; e[j] = e[i]; e[i] = aux;
            }
      for (i = 0; i< n; i++)
      fprintf(ps,"%-15s%3d%3d%3d%6.2f\n",e[i].nom,e[i].n1,e[i].n2,e[i].s,(float)e[i
   } else printf("Error abriendo algun archivo!!!");
   fclose(pe);
   fclose(ps);
   return 0;
}
```

•

En el siguiente ejemplo mostraremos cómo ordenar un arreglo de estructuras en base a un capo que sea una cadena de carácteres. Además mostraremos cómo separar el programa en sub-programas: uno para cada sub-tarea.

Ejemplo 4.6 *Escriba un programa en C que permita leer un archivo de texto, que luce como el ejemplo que se muestra luego, en una estructura de datos para luego ordenarlos en base a los apellidos y volverlo a escribir en otro archivo de texto. Los nombres de los archivos de entrada y de salida deben perdirsele al usuario. Debe definirse un registro adecuado, y declararse un tipo de datos concreto para almacenar en memoria principal los registros leidos del archivo de entrada. Todas las sub-tareas deben ejecutarse usando sub-programas adecuados, de tal manera que en el programa principal sólo se tenga que declarar las estructuras de datos necesarias e invocar a los sub-programas adecuados.*

El archivo de entrada luce como:

```
Zixia Pan 20
Pedro Perez 15
Luis Campos 13
```

Explicación: Hay tres tareas básicas que ejecutar; ellas son:

- Leer el archivo de entrada
- Ordenar los datos
- Escribir los datos ordenados en el archivo de salida

Ellas se van a ejecutar usando tres sub-programas, pero antes hay que definir la estructuta de datos que usaremos. Necesitamos definir un *struct* con los campos: nom, ape y not, por nombre, apellido y nota respectivamente. Note que se incluye la librería string.h. Note también que el programa principal sólo declara como variables un arreglo de Items y un entero n en el que se almacenará el tamaño del archivo, e invoca a los sub-programas. A continuación se muestra el código. Analicelo cuidadosamente.

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

typedef struct {
   char nom[12], ape[12];
   int not;
} Item;
```

```
void leeArc(Item e[], int *n){
   FILE *pe; char nomA[20];
   printf("Archivo de entrada: ");
   scanf("%s", nomA);
   pe = fopen(nomA, "r");
   if (pe != NULL){
      *n = 0;
      while (!feof(pe)) {
         fscanf(pe, "%s%s%d", e[*n].nom, e[*n].ape, &e[*n].not);
         (*n)++;
      }
      (*n)--;
   } else printf("Error abriendo el archivo %s!!!", nomA);
   fclose(pe);
}

void ordena(Item e[], int n){
   int i,j; Item aux;
   for (i = 0; i < n-1; i++){
      int m = i;
      for (j = i + 1; j < n; j++)
         if (strcmp(e[j].ape, e[m].ape) < 0) m = j;
      if (m != i) {aux = e[i]; e[i] = e[m]; e[m] = aux;}
   }
}

void escribeArc(Item e[], int n){
   FILE *ps; char nomA[20];
   printf("\nArchivo de salida: ");
   scanf("%s", nomA);
   ps = fopen(nomA, "w");
   if (ps != NULL){
      int k;
      for (k = 0; k < n; k++)
         fprintf(ps, "%-12s%-12s%3d\n", e[k].ape, e[k].nom, e[k].not);
   } else printf("Error abriendo el archivo %s!!!", nomA);
   fclose(ps);
}

int main(){
   Item e[30]; int n;
   leeArc(e,&n);
   ordena(e,n);
   escribeArc(e,n);
   printf("Mision cumplida... revisa el archivo de salida.\n");
   return 0;
}
```

•

```
Item max(Item e[], int n){
   Item m = e[0]; int k;
   for (k = 1; k < n; k++) if (e[k].not < m.not) m = e[k];
   return m;
}
```