2.2 y 2.3 Datos Estructurados y Arreglos en Pascal

TIPOS ESTRUCTURADOS: En Pascal, se pueden definir , a partir de los datos *simples*, otros tipos más complejos conocidos como tipos *estructurados*.

Cuando se declara una variable como de un tipo estructurado, se puede manipular la estructura completa, o bien trabajar con los datos individuales que la forman.

A continuación, se describirán los diferentes tipos estructurados que pueden manejarse en Pascal.

- Cadenas (String)
- Arreglos (Array)
- Registros (Record)
- Conjuntos (Set)
- Archivos

TIPO CADENAS (strings) : Turbo Pascal proporciona el tipo string para el procesamiento de cadenas (secuencias de caracteres).

La definición de un tipo string debe especificar el número máximo de caracteres que puede contener, esto es, la máxima longitud para las cadenas de ese tipo. La longitud se especifíca por una constante entera en el rango de 1 a 255.

El formato para definir un tipo string es :

<identificador> = string [*limite_superior*];

Las variables de cadena se declaran en la sección Var o Type.

Declaración en Var:

```
Var
 nombre    : string[30];
 domicilio : string[30];
 ciudad    : string[40];
```

Declaración en Type:

```
Type
 cad30     : string[30];
 cad40     : string[40];
Var
 nombre    : cad30;
 domicilio : cad30;
 ciudad    : cad40;
```

Una Vez declaradas las variables se pueden realizar asignaciones u operaciones de lectura/escritura.

```
nombre    := 'Egrid Lorely Castro Gonzalez' ;
domicilio := 'Altamirano #220';
ciudad    := 'La Paz B.C.S.';
```

El contenido de la cadena se debe encerrar entre apóstrofes. Si se desea que figure un apóstrofe en una cadena, es preciso doblarlo en la cadena. Los procedimientos de Entrada/Salida son de la siguiente forma :

```
ReadLn (nombre);
WriteLn('Hola ',nombre);
```

Longitud de una cadena

Las variables de tipo cadena pueden ocupar la máxima longitud definida, más un octeto que contiene la longitud actual de la variable. Los caracteres que forman la cadena son numerados desde 1 hasta la longitud de la cadena.

Ejemplo:

```
Var
  nombre : string[10];
begin
  nombre := 'Susana';
end.
```

Número de byte 0 1 2 3 4 5 6

6	S	u	s	a	n	a	

longitud de la cadena

Obsérvese que el primer byte no es el carácter '6' si no el número 6 en binario (0000 0110) y los últimos bytes de la cadena hasta 10 (7-10) contienen datos aleatorios.

Una cadena en Turbo Pascal tiene dos longitudes :

1. *Longitud física :* Es la cantidad de memoria que ocupa realmente, está se establece en tiempo de compilación y nunca cambia

2. *Longitud lógica* : Es el número de caracteres almacenados actualmente en la varible cadena. Este dato puede cambiar durante la ejecución del programa.

Es posible acceder a posiciones individuales dentro de una variable cadena, mediante la utilización de corchetes que dentro de ellos se especifíca el número indice dentro de la cadena a utilizar así para el ejemplo anterior se tiene :

```
nombre[1] ==> 'S'
nombre[2] ==> 'u'
nombre[3] ==> 's'
nombre[4] ==> 'a'
nombre[5] ==> 'n'
nombre[6] ==> 'a'
```

Operaciones entre cadenas

Las operciones básicas entre cadenas son : *asignación, comparación y concatenación.* Es posible *asignar* una cadena a otra cadena, incluso aunque sea de longitud física más pequeña en cuyo caso ocurriría un truncamiento de la cadena.

Ejemplo:

```
Var
  nombre : String[21];
   .
   .
   .
  nombre := 'Instituto Tecnológico de La Paz';
```

El resultado de la asignación en la variable nombre será la cadena 'Instituto Tecnológico'.

Las comparaciones de las cadenas de caracteres se hacen según el orden de los caracteres en el código ASCII y con los operadores de relación.

'0' < '1'   '2' > '1'   'A' < 'B'   'm' > 'l'

Reglas de comparación de cadenas

Las dos cadenas se comparan de izquierda a derecha hasta que se encuentran dos caracteres diferentes. El orden de las dos cadenas es el que corresponde al orden de los dos caracteres diferentes. Si las dos cadenas son iguales pero una de ella es más corta que la otra, entonces la más corta es menor que la más larga.
Ejemplo :

```
'Alex'   > 'Alas'
{puesto que 'e' > 'a'}
'ADAN'   < 'adan'
{puesto que 'A' < 'a'}
'Damian'  < 'Damiana'
{'Damian' tiene menos caracteres que 'Damiana'}
```

Otra operación básica es la *concatenación*. La concatenación es un proceso de combinar dos o más cadenas en una sola cadena. El signo + se puede usar para concatenar cadenas (al igual que la función concat), debiendo cuidarse que la longitud del resultado no sea mayor que 255.

Ejemplos :

```
'INSTITUTO '+'TECNOLOGICO'='INSTITUTO TECNOLOGICO'
'CONTAB'+'.'+'PAS'= 'CONTAB.PAS'
```

Se puede asignar el valor de una expresión de cadena a una variable cadena, por ejemplo :

```
fecha := 'lunes';
y utilizar la variable fecha en :
frase:='El próximo '+fecha+' inician las clases';
```

Si la longitud máxima de una cadena es excedida, se pierden los caracteres sobrantes a la derecha. Por ejemplo, si fecha hubiera sido declarada del tipo string[7], después de la asignación contendría los siete primeros caracteres de la izquierda (CENTENA).

PROCEDIMIENTOS Y FUNCIONES DE CADENA INTERNOS

Turbo Pascal incorpora las siguientes funciones y procedimientos para el tratamiento de cadenas.

Procedimientos y funciones de cadena	
Procedimiento	**Función**
Delete	Concat
Insert	Copy
Str	Length
Val	Pos

Procedimiento Delete

Delete borra o elimina una subcadena de una cadena contenida en otra cadena de mayor longitud.
Formato :

Delete (*s*, *posición*, *número*)

S Cadena original o fuente
Posición Expresión entera que indica la posición del primer carácter a suprimir.
Número Cantidad de caracteres a suprimir.

Si posición es mayor que la longitud de la cadena, no se borra ningún carácter. Si número especifíca más caracteres que los existentes desde la posición inicial hasta el final de la cadena, sólo se borran tantos caracteres como estén en la cadena.

Ejemplo :

Cad := 'Tecnológico'
Delete(cad,1,5)
Resulta la cadena 'lógico'

Procedimiento Insert

Insert inserta una subcadena en una cadena.
Formato :

Insert (*cad1, s, posición*)

cad1 cadena a insertar
s cadena donde se inserta
posición carácter a partir del cual se inserta

Ejemplo:

cad:= 'México '
Insert (' lindo y querido',cad,7)
Resulta 'México lindo y querido'

Procedimiento Str

Este procedimiento efectúa la conversión de un valor númerico en una cadena.

Formato :

Str (*valor,s*)

Valor expresíon númerica
s cadena

Ejemplos :

```
Numero := 235.55;
Str(Numero,cadena);
Write(cadena);
Resulta la cadena '2.355000000E+02'
Numero := 12345;
Str(Numero,cadena);
Write(cadena);
Resulta la cadena '12345'
Numero := 12345;
Str(Numero+1000:10,cadena);
Write(cadena);
Resulta la cadena '     13345'
```

Procedimiento Val

Este procedimiento convierte una cadena en variable númerica. Para que esta conversión sea efectiva, el contenido de la cadena de caracteres debe corresponderse a las reglas de escritura de números: no debe de existir ningún blanco en la primera o última posición.
Formato:

Val (S, variable, código)

S cadena
Variable variable de tipo entero o real
código si la conversión ha podido ser efectuada toma el valor cero; en caso contrario contiene la primera posición del primer carácter de la cadena S que impide la conversión y en ese caso variable no queda definida

Ejemplo: ***Var***

```
 cad        : string[10];
 num1,codigo : integer;
 num2       : real;
begin cad:='22.25';
 Val(cad,num2,codigo);
 if codigo=0 then
 WriteLn(num2:2:2)
   {Produce 22.25}
 else
 WriteLn(codigo);
 cad:='12x45';
 Val(cad,num1,codigo);
 if codigo=0 then
 WriteLn(num1)
 else
   WriteLn(codigo)
   {Produce 3}
end.
```

Función Concat

Además del uso de '+' para la concatenación, Turbo Pascal tiene la función concat que permite concatenar una secuencia de caracteres.

Formato:

Concat (*S1*,*S2*,...,*Sn*)

S1,S2...cadenas o variables de caracteres (expresión tipo cadena)

Ejemplo :

```
Var
  cad1,cad2 : string[20];
destino   : string[50];
begin
  cad1:='Hola como '; cad2:='Estas ';
destino:=Concat(cad1,cad2,', estoy bien')
end.
```

Esto produce una cadena destino igual a 'Hola como estas, estoy bien'

Función Copy

Esta función devuelve una cadena de caracteres (subcadena) extraída de una cadena.

Formato:

Copy(*s*,*posición*,*número*)

s cadena (fuente)
posición primer carácter a extraer (tipo entero)
número total de caracteres a extraer (tipo entero)

Si posición es mayor que la longitud de S, se devuelve una cadena vacía; si número especifíca más caracteres que los indicados desde posición, sólo se devuelve el resto de la cadena.

Ejemplo:

```
cad  := 'Instituto Tecnológico de La Paz';
cad2 := Copy(cad,26,6);
Write(cad2);
```

Lo que produce la cadena 'La Paz' contenida en cad2.

Función Lenght (longitud)

La función Lenght proporciona la longitud lógica de una cadena de caracteres y devuelve un valor entero.
Formato :

Length (*s*)

s expresión tipo cadena

Ejemplo :

```
Var
  cad : string[20];
begin
cad:='Hola';
  WriteLn(Length ('Hola como estas'));  {devuelve el valor 15}
  WriteLn(Length (''));  {devuelve cero (cadena vacía)}
  WriteLn(Length (cad));  {devuelve el valor 4}
  WriteLn(Ord(cad[0]))  {devuelve el valor 4}
end.
```

Función Pos

Esta función permite determinar si una cadena está contenida en otra. En este caso, la función devuelve la posición donde comienza la cadena buscada en la cadena fuente, si la cadena no existe, se devuelve el resultado 0.
Formato :

Pos (*cadena buscada, cadena fuente*)

Ejemplo:

```
cad:= 'uno dos tres cuatro cinco seis';
WriteLn(Pos('dos',cad));
{Resulta 5 que es la posición de 'd'}
WriteLn(Pos('ocho',cad));
{Resulta 0 no existe la cadena 'ocho'}
```

Arreglos (array)

Un arreglo está formado por un número fijo de elementos contíguos de un mismo tipo. Al tipo se le llama *tipo base* del arreglo. Los datos individuales se llaman *elementos* del arreglo.
Para definir un tipo estructurado arreglo, se debe especificar el tipo base y el número de elementos.

Un array se caracteríza por :

1. Almacenar los elementos del array en posiciones de memoria contínua
2. Tener un único nombre de variable que representa a todos los elementos, y éstos a su vez se diferencían por un índice o subíndice.
3. Acceso directo o aleatorio a los elementos individuales del array.

Los arrays se clasifican en :

- Unidimensionales (vectores o listas)
- Multidimensionales (tablas o matrices)

El formato para definir un tipo array es :

nombre_array = array [*tipo subíndice*] of *tipo*

nombre_array identificador válido
tipo subíndice puede ser de tipo ordinal:
boolean o char, un tipo
enumerado o un tipo subrango.
Existe un elemento por cada
valor del tipo subíndice
tipo describe el tipo de cada elemento
del vector; todos los elementos
de un vector son del mismo tipo

Las variables de tipo array se declaran en la sección Var o Type.

Declaración en Var:

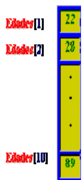
```
Var
  nombres  : array[1..30] of string[30];
  calif    : array[1..30] of real;
  numero   : array[0..100] of 1..100;
```

Declaración en Type:

```
Type
  nombres  : array[1..30] of string[30];
  calif    : array[1..30] of real;
numero   : array[0..100] of 1..100;
Var
  nom      : nombres;
  califica : calif;
  num      : numero;
```

Arrays unidimensionales

Un array de una dimensión (vector o lista) es un tipo de datos estructurado compuesto de un número de elementos finitos, tamaño fijo y elementos homogéneos.
Supongamos que desea conservar las edades de 10 personas. Para almacenar estas edades se necesita reservar 10 posiciones de memoria, darle un nombre al array, y a cada persona asignarle su edad correspondiente.



Nombre del vector edades
Subíndice [1],[2],...
Contenido edades[2]= 28
Ejemplo:

```
Program Vector_edades; {El siguiente programa captura 20 edades y las muestra en forma
ascendente}
Uses Crt;
Const
  MaxPersonas = 10;
Var
```

```
  edades   : array [1..MaxPersonas] of byte;
  i,j,paso : byte;
begin
  ClrScr;
  {lectura de array}
  for i:=1 to MaxPersonas do
  begin
    gotoxy(10,5);
    ClrEol;
    Write('Edad de la ',i,' persona : ');
    ReadLn(edades[i])
  end;
  {ordenación}
  for i:=1 to MaxPersonas-1 do
    begin
      for j:=i+1 to MaxPersonas do
        begin
          if edades[i]>edades[j] then
            begin
              paso     :=edades[i];
              edades[i]:=edades[j];
              edades[j]:=paso
            end
        end;
      WriteLn(edades[i]) {escritura del array}
    end;
   Readkey
end.
```

Arrays paralelos

Dos o más arrays que utilizan el mismo subíndice para referirse a términos homólogos se llaman arrays paralelos.
Basados en el programa anterior se tienen las edades de 'x' personas, para saber a que persona se refiere dicha edad se puede usar otro arreglo en forma paralela y asociarle los nombres de manera simultánea con las edades.

nombres[1] Fidel — edades[1] 22
nombres[2] Ely — edades[2] 28
nombres[10] Juan — edades[10] 89

Ejemplo:

```
Program Paralelo_edades;  {El siguiente programa captura 10 edades y nombres por medio de
arrays paralelos y los muestra ordenados en forma ascendente}
Uses Crt;
Const
  MaxPersonas = 10;
Var
  edades      :array [1..MaxPersonas] of byte;
  nombres     :array [1..MaxPersonas] of string [10];
  aux_nom     :string[10];
  i,j,aux_edad :byte;
begin
  ClrScr;
  {lectura de arrays paralelos de manera simultánea}
```

```
    for i:=1 to MaxPersonas do
      begin
        gotoxy(10,5);
        ClrEol;
        Write(i,'.- Nombre : ','Edad : ');
        gotoxy(23,5);ReadLn(nombres[i]) ;
        gotoxy(48,5);ReadLn(edades[i])
      end;
    {ordenación}
    for i:=1 to MaxPersonas-1 do
      begin
        for j:=i+1 to MaxPersonas do
          begin
            if edades[i]>edades[j] then
              begin
                aux_edad  :=edades[i];
                edades[i] :=edades[j];
                edades[j] :=aux_edad;
                aux_nom   :=nombres[i];
                nombres[i]:=nombres[j];
                nombres[j]:=aux_nom
              end
          end;
        WriteLn(nombres[i]:10,' ',edades[i]:3)
        {escritura de los arrays paralelos}
      end;
  Readkey
  end.
```

Arrays bidimensionales (tablas)

Un array bidimensional (tabla o matríz) es un array con dos índices, al igual que los vectores que deben ser ordinales o tipo subrango.

Filas \ Columnas	1	2	3	4	5
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]	A[1,5]
2					
3					
4	A[4,1]				A[4,5]

Para localizar o almacenar un valor en el array se deben especificar dos posiciones (dos subíndices), uno para la fila y otro para la columna.

Formato:

1. *identificador* = array [*índice1*, *indice 2*] of *tipo de elemento*
2. *identificador* = array [*índice 1*] of array [*indice 2*] *of tipo* de elemento

Supongase que se desea almacenar las calificaciones de 5 alumnos obtenidas en 3 examenes y mostrar en orden ascendente sus promedios respectivamente. En este caso se usará un array bidimensional (tabla o matríz) de 5 filas y 4 columnas en la cual se almacenará las calificaciones de 3 examenes en 3 columnas y la cuarta columna se utilizará para almacenar su promedio respectivo, además de un array unidimensional (vector) donde en forma paralela se almacenarán los nombres de los alumnos de la siguiente forma :

	alumno	Examen1	Examen2	Examen3	Promedio
alumno[1]	Fidel	89	88	85	87.33
alumno[2]	Ely	100	90	98	96
alumno[3]	Juan	100	58	89	82.33
alumno[4]	Daniel	90	70	98	86
alumno[5]	Paty	100	95	98	97.67

Ejemplo:

```
Program Matriz_Vector; {El siguiente programa captura las calificaciones de 5 alumnos en 3
examenes, y despliega en pantalla los promedios ordenados en forma descendente }
Uses Crt;
Const
 MaxAlumno = 5;
 MaxExamen = 4;{Columna 4 almacena el promedio}
Var
 Alumno    :array[1..MaxAlumno]            of string[10];
 examen    :array[1..MaxAlumno,1..MaxExamen] of real;
 aux_examen :array[1..MaxExamen]            of real;
{reserva 20 posiciones de memoria de datos reales: 5 filas por 4 columnas}
 promedio    :real;
 aux_alumno   :string [10];
 i,j,col,ren  :byte;
begin
 ClrScr;
 {lectura de arrays paralelos de manera simultánea}
 gotoxy(5,5);Write('Nombre');
 gotoxy(20,5);Write('Examen1 Examen2 Examen3 Promedio');
 col:=5;ren:=6;
 for i:=1 to MaxAlumno do
  begin
   gotoxy(col,ren);
   ReadLn(alumno[i]); {lectura de vector}
   col:=22;promedio:=0;
   for j:=1 to MaxExamen-1 do
    begin
     gotoxy(col,ren);
     ReadLn(examen[i,j]); {lectura de matríz}
     promedio:=promedio+examen[i,j];
     col:=col+10
    end; examen[i,j+1]:=promedio/3;
   gotoxy(col,ren);Write(promedio/3:3:2);
   inc(ren);
   col:=5
  end;
 {ordenación}
 for i:=1 to MaxAlumno-1 do
 for j:=i+1 to MaxAlumno do
 begin
   if examen[i,MaxExamen]<examen[j,MaxExamen] then
    begin
     {intercambio de nombres en vector}
     aux_alumno:=alumno[i];
     alumno[i] :=alumno[j];
     alumno[j] :=aux_alumno;
     {intercambio de calificaciones en matríz}
     move(examen[i],aux_examen,SizeOf(aux_examen));
    move(examen[j],examen[i],SizeOf(aux_examen));
    move(aux_examen,examen[j],SizeOf(aux_examen))
    end
  end;
 {recorrido de matríz y vector}
 gotoxy(25,14);Write('Datos ordenados');
 gotoxy(5,16);Write('Nombre');
 gotoxy(20,16);Write('Examen1 Examen2 Examen3 Promedio');
 col:=5;ren:=17;
 for i:=1 to MaxAlumno do
```

```
    begin
      gotoxy(col,ren);
      Write(alumno[i]);
      col:=22;
      for j:=1 to MaxExamen do
        begin
          gotoxy(col,ren);
          Write(examen[i,j]:3:2);
          col:=col+10
        end;
      col:=5;
      inc(ren)
  end;
  readkey
  end.
```

Arrays multidimensionales

Turbo Pascal no limita el número de dimensiones de un array, pero sí que debe estar declarado el tipo de cada subíndice.

Formato :

1. *identificador* = array [*índice1*] of array [*índice 2*].. of array [*índice n*] of *tipo de elemento*
2. *identificador* = array [*índice 1, índice 2,...,*índice n] *of tipo de elemento*

Ampliando el ejemplo anterior supongase que ahora deseamos capturar calificaciones para 3 materias en cuyo caso aplicaremos un array tridimensional. De la siguiente forma :

(nombre,examen,materia)

Ejemplo:

```
Program Tridimensional; {El siguiente programa captura calificaciones de 5 alumnos en 3
examenes de 3 materias distintas, y despliega en pantalla los promedios ordenados en forma
descendente }
Uses Crt;
Const
  MaxAlumno  = 5;
  MaxExamen  = 4; {Columna 4 almacena el promedio}
  MaxMateria = 3;
  materia    : array[1..3]of string[8]=('Fisica','Ingles','Historia');
Var
  Alumno     : array [1..MaxAlumno] of string[10];
  examen     : array [1..MaxAlumno,1..MaxExamen,1..MaxMateria] of real;
  aux_examen : array [1..MaxExamen]of real;
  {reserva 60 posiciones de memoria de datos reales :
   5 filas por 4 columnas y 3 dimensiones}
```

```
  promedio   :real;
  aux_alumno :string [10];
  i,j,k,col,ren : byte;
begin
  ClrScr;
  {lectura de arrays paralelos de manera simultánea}
  for k:=1 to MaxMateria do
    begin
      ClrScr;
      gotoxy(34,3);Write(materia[k]);
      gotoxy(5,5);Write('Nombre');
      gotoxy(20,5);Write('Examen1 Examen2 Examen3 Promedio');
      col:=5;ren:=6;
      for i:=1 to MaxAlumno do
        begin
          gotoxy(col,ren);
          if k=1 then
            ReadLn(alumno[i]) {lectura de vector}
          else Write(alumno[i]);
          col:=22;promedio:=0;
          for j:=1 to MaxExamen-1 do
            begin
              gotoxy(col,ren);
              ReadLn(examen[i,j,k]); {lectura de matríz}
              promedio:=promedio+examen[i,j,k];
              col:=col+10
            end;
          examen[i,j+1,k]:=promedio/3;
          gotoxy(col,ren);Write(promedio/3:3:2);
          inc(ren);
          col:=5 end;
      gotoxy(15,22);
      Write('Presione una tecla para continuar....');
      ReadKey
    end;
  {ordenación}
  for k:=1 to MaxMateria do for
    i:=1 to MaxAlumno-1 do for
    j:=i+1 to MaxAlumno do
        begin
          if examen[i,MaxExamen,k]<examen[j,MaxExamen,k] then
            begin
              {intercambio de nombres en vector}
              aux_alumno:=alumno[i];
              alumno[i] :=alumno[j];
              alumno[j] :=aux_alumno;
              {intercambio de calificaciones en matríz}
            move(examen[i,k],aux_examen,SizeOf(aux_examen));
            move(examen[j,k],examen[i,k],SizeOf(aux_examen));
            move(aux_examen,examen[j,k],SizeOf(aux_examen))
            end
        end;
  {recorrido de matríz y vector}
  for k:=1 to MaxMateria do
  begin
      ClrScr;
      gotoxy(35,4);Write(materia[k]);
```

```
        gotoxy(25,5);Write('Datos ordenados');
        gotoxy(5,6);Write('Nombre');
        gotoxy(20,6);Write('Examen1 Examen2 Examen3 Promedio');
        col:=5;ren:=7;
        for i:=1 to MaxAlumno do
          begin
            gotoxy(col,ren);
            Write(alumno[i]);
            col:=22;
            for j:=1 to MaxExamen do
              begin
                gotoxy(col,ren);
                Write(examen[i,j,k]:3:2);
                col:=col+10
              end;
            col:=5;
            inc(ren)
          end;
        gotoxy(15,22);
        Write('Presione una tecla para continuar....');
        readkey
      end
    end.
```

Registros (record)

Un *registro* es una estructura que consiste de un número fijo de componentes llamados *campos*. Los *campos* pueden ser de diferentes tipos y deben tener un *identificador de campo*.

La definición de un tipo registro debe consistir de la palabra reservada record, seguida de una lista de campos y terminada por el identificador reservado end.

formato:

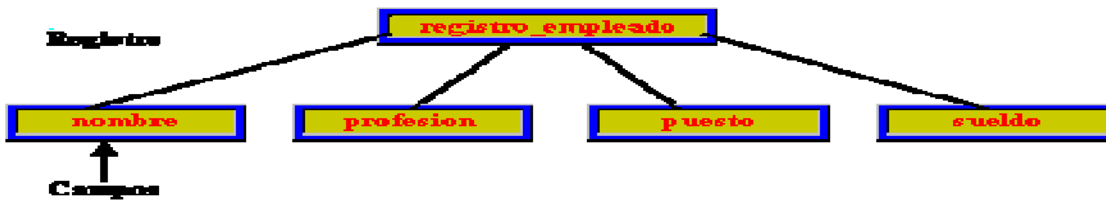
```
type
  tipo_reg = record
              lista id1:tipo 1;
              lista id2:tipo 2;
                  .
                  .
                  .
              lista idn:tipo n
            end;
```

tipo_reg nombre de la estructura o dato registro
lista id lista de uno o más nombres de campos separados por comas
tipo puede ser cualquier tipo de dato estándar o definido por el usuario

Ejemplo :

```
Type
  registro_empleado = Record
                        nombre    : string[30] ;
                        profesion : string[20] ;
                        puesto    : string[20] ;
                        sueldo    : real
                      end;
```

Un registro puede representarse gráficamente en función de sus campos.



El registro como un todo tiene el nombre registro_empleado. Este nuevo tipo que se acaba de definir puede utilizarse en la sección Var, para declarar variables como por ejemplo :

```
Var
  empleado : registro_empleado ;
```

Para asignar valores a cada campo del registro empleado, puede procederse de la siguiente manera :

```
......................
......................
......................
empleado.nombre    := 'MENDEZ ROMERO FEDERICÓ';
empleado.profesion := 'INGENIERO CIVIL';
empleado.puesto    := 'DIRECTOR';
empleado.sueldo    := 5000.00 ;
......................
......................
......................
```

Para simplificar la notación de los registros, se pueden utilizar instrucciones With Do.
Por ejemplo, la instrucción :

```
WriteLn(empleado.nombre:35,
        empleado.puesto:25,
        empleado.sueldo:15:0);
puede escribirse : With empleado Do
WriteLn(nombre:35,puesto:25,sueldo:15:0);
Asimismo, para leer datos usamos :
With empleado Do
begin
  Write('NOMBRE : ');
  ReadLn(nombre);
  Write('PROFESION : ');
  ReadLn(profesion);
  Write('PUESTO : ');
  ReadLn(puesto);
  Write('SUELDO MENSUAL : ');
  ReadLn(sueldo)
end;
```

Obsérvese el siguiente segmento de programa :

```
..............................
..............................
..............................
Type
 fecha = Record
          anio : 1900..1989 ;
          mes  : 1..12 ;
        dia  : 1..31
        end;
 datos = Record
          nombre       : string[30] ;
          sueldo       : real ;
        fecha_ingreso : fecha
        end;
Var
  personal : datos ;
  ..............................
  ..............................
  ..............................
```

La asignación del valor 15 al campo dia del registro personal se hace de la siguiente manera :

```
personal.fecha_ingreso.dia := 15 ;
```

donde :

```
personal.fecha_ingreso
```

hace referencia al campo fecha_ingreso del registro personal, mientras que :

```
dia
```

se refiere al campo dia del campo fecha_ingreso, que a su vez es un registro de tipo fecha.

Usando la forma With Do quedaría :

```
With personal Do With fecha_ingreso Do
  dia := 15 ;
```

que es equivalente a :

```
With personal, fecha_ingreso Do
  dia := 15 ;
```

o también a:

```
with personal Do
 with fecha_ingreso Do
   dia:=15;
```

Ejemplo:

```
Program Registro; {El siguiente programa captura 10 empleados y sus datos personales en un
arreglo con la utilización de registros}
Uses Crt;
Const
  MaxEmpleados=10;
Type
  registro_empleado = Record
                      nombre    : string[30];
                      profesion : string[20];
                      puesto    : string[20];
                      sueldo    : real
                    end;
Var
  registro  : registro_empleado;
  empleado  :array[1..MaxEmpleados] of registro_empleado;
  i,col,ren :byte;
begin
  ClrScr;
```

```
     Write('    Nombre            Profesión         Puesto          Sueldo');
     col:=1;ren:=2;
     for i:=1 to MaxEmpleados do
      begin
       With registro do
         begin
         gotoxy(col,ren);
           ReadLn(empleado[i].nombre);
           gotoxy(col+21,ren);
           ReadLn(empleado[i].profesion);
           gotoxy(col+40,ren);
           ReadLn(empleado[i].puesto);
           gotoxy(col+59,ren);
           ReadLn(empleado[i].sueldo);
           inc(ren);
           col:=1;
         end
   end;
   ReadKey
   end.
```

Conjuntos (sets)

Un conjunto es una colección de objetos relacionados. Cada objeto en un conjunto es llamado *miembro o elemento* del conjunto.

Aunque en matemáticas no hay restricciones para que los objetos puedan ser elementos de un conjunto, Pascal sólo ofrece una forma restringida de conjuntos, por lo que :

1. Los elementos de un conjunto deben ser del mismo tipo, llamado el *tipo base*.
2. El *tipo base* debe ser un tipo simple, excepto el tipo real.

Representación de conjuntos:

Elementos	Notación Matemática	Pascal
1,2,3,4,5	{1,2,3,4,5}	[1,2,3,4,5]
a,b,c	{a,b,c}	['a','b','c']

Aunque se puede utilizar notación de tipo subrango para especificar secuencia de valores que pertenezcan a un conjunto, los elementos del conjunto no tienen una ordenación interna particular. La única relación entre los miembros de un conjunto es: *existe o no existe* en el conjunto.
[5,5] y [5] son equivalentes (contienen un sólo elemento)

Ejemplos de conjuntos:

[1,3,5]	conjunto de tres enteros
[1..3,8..10]	conjunto de seis enteros [1,2,3,8,9,10]
[ ]	conjunto vacío (ningún elemento)
['a','b','A'..'D']	conjunto de seis elementos ['a','b','A','B','C','D']

Un conjunto especial, denominado conjunto vacío, es aquel que no contiene ningún elemento.
El formato para la definición de un tipo conjunto es :

```
type
  <identificador> = set of <tipo_base>
```

Ejemplos :

```
Type
  dias_mes   = set of 0..31;
  mayusculas = set of 'A'..'Z';
  caracteres = set of char;
  equipos    = (Chivas,Santos,Pumas,
               Toluca,Tigres,America,
               Leon);
  futbol     = set of equipos;
Var
  GrupoA,GrupoB : futbol;
```

En Turbo Pascal, el máximo número de elementos permitidos en un conjunto es 256, y los valores ordinales del tipo base deben estar en el rango de 0 a 255 .

Asignación en conjuntos

Los elementos se ponen en los conjuntos utilizando una sentencia de asignación.

```
GrupoA := [Chivas,Santos,Toluca];
GrupoB := [Tigres..Leon];
```

Si dos tipos de conjuntos son compatibles (tienen los tipos de base compatibles: igual tipo de dato, o uno subrango de otro, o ambos del mismo tipo patrón), sus variables representativas se pueden utilizar en sentencias de asignación.

```
 GrupoA := GrupoB;
 GrupoB := [];{asignación del conjunto vacío}
```

La relación In

El operador relacional In y una expresión relacional nos permite conocer si un elemento dado pertenece a un conjunto dado.

Formato:

elemento in [*lista de elementos*]

El resultado de evaluar la expresión relacional puede ser true o false. El tipo de datos de elemento y la lista de elementos deben ser compatibles.

Ejemplo:

```
Program Hasta_Si;
 {El siguiente programa obtiene del teclado un nombre
  hasta que se presiona la tecla 's' o 'S' o Esc}
Uses Crt;
Const
  Esc  =#27;
Var
  nombre : string[30];
  tecla  : char;
begin
  Repeat
  ClrScr;
    Write('Escribe tu nombre : ');
    ReadLn(nombre); Write('Desea
    Salir S/N ? : '); Tecla:=Readkey
  Until tecla in['s','S',Esc];
  ClrScr
end.
```

Operaciones con conjuntos

Una vez creados los conjuntos y las variables tipo conjunto es posible realizar tres operaciones binarias sobre ellos: unión, intersección y diferencia. La siguiente tabla resume el funcionamiento de las operaciones con conjuntos A y B.

Operaciones sobre dos conjuntos A y B		
Operación	**Notación Pascal**	**Conjunto resultante**
Unión	+	A+B es el conjunto que contiene todos los elementos que están en A, en B o en ambos.
Intersección	*	A*B es el conjunto cuyos elementos pertenecen a A y B simultáneamente.
Diferencia	-	A-B es el conjunto cuyos elementos son de A pero no de B.

Ejemplo:

Operación	Resultado
[1,3,4,5]+[1,2,4]	[1,2,3,4,5]
[1,3,4,5]*[1,2,4]	[1,4]
[1,3,4,5]-[1,2,4]	[3,5]
[1,2,3]+[ ]	[1,2,3]
[1,2,4]*[3,5,6]	[ ]
[1,2,3]-[ ]	[ ]

Reglas de prioridad

Cuando una expresión de conjuntos contiene dos o más operadores de conjunto, éstos se evalúan de acuerdo a la siguiente prioridad:

*	Prioridad más alta
+,-	Prioridad más baja

En caso de operaciones con igual prioridad se evalúan de izquierda a derecha.

Comparación de conjuntos (operadores de relación)

Los conjuntos se pueden comparar entre sí mediante el uso de los operadores relacionales (==, <>,<=,>=). Los operandos deben ser del mismo tipo base. El resultado de la comparación es un valor lógico: true o false.

Operadores de relación de conjuntos A y B		
Operador	**Nombre del operador**	**Resultado**
< =	Subconjunto	El valor de A< = B es true. Si cada elemento de A es también de B. En caso contrario es false.
=	Igualdad	El valor de A = B es true si cada elemento de A está en B y cada elemento de B está en A. En caso contrario A = B es falso, A=B equivale a (A<=B) and (B<=A).
< >	Desigualdad	El valor de A < > B es true si el valor de A = B es false y viceversa.
> =	Superconjunto	A < > B equivale a not (A = B), A >= B es true si B<=A es true.

Ejemplos :

Comparación	Resultado
[3,5] = [3,5]	true
[ ] = [1]	false
[ ] <= [1,5]	true
[1,2] >= [1,2,3,4]	false
[ ] >= [1,2]	false
[1,4,5] = [4,5,1]	true

Prioridad de operadores :

Operador	Prioridad
not	1 (más alta)
*,/,div,mod,and	2
+,-,or	3
=,<>,<,<=,>,>=,in	4 (más baja)

Lectura de conjuntos

Algunas operaciones no se pueden ejecutar sobre variables de conjunto. Por ejemplo, *no se puede leer cinco ciudades en un conjunto Mundo con la siguiente sentencia:*

ReadLn(Mundo)

La razón es que la computadora no puede saber cuantos elementos existen en el conjunto.
Si se desea leer y almacenar datos en un conjunto se debe utilizar un bucle.

```
for elemento:=1 to 5 do
  begin
    {leer un dato}
    {almacenar el valor en el
     siguiente elemento del conjunto}
  end;
```

Reglas :

1. Inicializar A al conjunto vacío.
   A := [ ];
2. Leer cada elemento x del conjunto y añadirlo al conjunto A con la operación unión (+)
   ReadLn(x);
   A := A + [x];

Escritura de conjuntos

Los valores de conjuntos no se pueden visualizar con la sentencia Write.
Para visualizar los elementos de un conjunto A se debe utilizar el siguiente algoritmo:

1. Copiar los elementos de A en un conjunto auxiliar Aux que tenga un tipo base compatible con el de A.
2. Declarar x una variable del tipo base de Aux e inicializar x al primer elemento de esté tipo base.
3. Mientras x es diferente del último elemento de esté tipo base y Aux no está vacía, hacer :

   I. Si x pertenece a Aux, entonces visualizar x y eliminarlo de Aux
   II. Sustituir x con su sucesor.

Ejemplo:

```
Program Impares; {El siguiente programa encuentra y muestra todos los números impares
menores o igual a un número dado n que esté entre el límite 1..255}
Uses Crt;
Type
  numeros = set of 1..255;
Var
  impares,Aux :numeros;
  x,MaxNum,i  :byte;
begin
  ClrScr;
  Write('Escribe un número entero : ');
  ReadLn(MaxNum);
  impares:=[];   {inicializa a conjunto vacío}
  for i:=1 to MaxNum do
   begin
     if odd(i) then
       impares:=impares + [i]
  {añadir elementos al conjunto}
   end;
  {visualizar elementos del conjunto}
  Aux:=impares;
  x:=1;
  while (x<>MaxNum+1) and (Aux<>[]) do
    begin
      if x in Aux then
        begin
        WriteLn(x);
          Aux:=Aux-[x]
        end;
  x:=succ(x)
  end;
  ReadKey;
  ClrScr
end.
```

Archivos (file)

Un tipo archivo se define con los identificadores reservados FILE OF, seguidas por el tipo de los componentes del archivo.

Por ejemplo :

```
Type
  empleado = file of Record
                nombre:string[30];
                sueldo:real;
             end;
Var
  nomina : empleado ;
```

también puede escribirse así :

```
Type
  empleado = Record
              nombre:string [30];
              sueldo:real;
             end;


Var
  nomina : file of empleado;
```

Archivos

Conceptos básicos
Un archivo es el módulo básico de información manejado por el Sistema Operativo. El Sistema Operativo es un conjunto de programas cuya función es administrar los recursos del Sistema de Cómputo. Por ejemplo, un programa fuente es almacenado como un archivo.

Primero es introducido en la memoria RAM por medio de un programa editor, y después es almacenado como un *archivo de texto* en un medio de almacenamiento permanente (cinta, disco flexible, disco duro). Una vez que el programa fuente ha sido compilado, el programa resultante de la compilación viene a ser un archivo binario.

En Pascal, un archivo es una secuencia de elementos que pertenecen al mismo tipo o estructura, esto es que un archivo puede ser una secuencia de caracteres, números o registros, por lo que su representación lógica puede hacerse como una secuencia de módulos del mismo tamaño, tal como se presenta en la siguiente figura.

Elemento1	Elemento2	Elemento3	Elemento4	EOF

En el vocabulario de manejo de archivos, a cada elemento de un archivo se le llama *registro*. En Pascal, la numeración de los registros empieza con el número CERO , por lo que al elemento_1 se le llamará *registro* 0, al elemento_2 *registro* 1, y así sucesivamente hasta llegar a la marca de fin de archivo EOF.

Turbo Pascal difiere significativamente de Pascal estándar por la forma en que maneja los archivos.

En Pascal estándar, los archivos son formalmente definidos independientemente del medio en que residan.

Este método de definición fue inspirado por los archivos de tarjetas perforadas y cintas magnéticas, las cuales eran los medios de almacenamiento comúnmente usados cuando Pascal fue definido por primera vez.

Como resultado, todo acceso a cualquier archivo en Pascal estándar es secuencial(registro por registro) tal como se realiza en las tarjetas perforadas y cintas magnéticas.

En Turbo Pascal los archivos son definidos como *archivos de disco*. Los discos son actualmente los dispositivos de almacenamiento más utilizados en las microcomputadoras.

Los mecanismos de acceso secuencial proporcionados por Pascal estándar son algunas veces inconvenientes e insuficientes para los archivos basados en discos de acceso aleatorio, por lo que Turbo Pascal provee nuevas estructuras y mecanismos de acceso a los archivos.

La primera gran diferencia entre Turbo Pascal y Pascal estándar, es la forma en que enlazan los archivos a un programa. En Pascal estándar, se abren los archivos referenciando su nombre de archivo en el encabezado del programa, y se cierran cuando el programa termina. En Turbo Pascal, los archivos de disco deben enlazarse a una *variable de archivo* particular con el procedimiento:

Assign(variable_archivo,nombre_archivo);

y deben ser preparados para procesarse (abiertos) con: reset(variable_archivo) o rewrite(variable_archivo) antes de ser utilizados.

Además, los archivos deben ser explícitamente cerrados por medio de close(variable_archivo), después de que han sido utilizados, o se perderán los datos que estén en la memoria auxiliar (variable_archivo) .

variable_archivo es el nombre de la memoria auxiliar (*buffer*), por medio de la cual el programa manejará los datos hacia o desde el archivo en disco.

nombre_archivo es el nombre que identifica al archivo en el disco.

reset abre un archivo existente para procesamiento y coloca *el apuntador de registro* en el primer registro (0).

rewrite crea un nuevo archivo (o sobre-escribe en uno existente) y lo abre para procesamiento con el *apuntador de registro* colocado en el registro 0.

En el caso de un archivo de tipo text, reset hará que el archivo sólo pueda ser usado para operaciones de *lectura* , mientras que rewrite sólo permitirá operaciones de *escritura*.

Los nombres de archivo válidos son cadenas de 1 a 8 caracteres seguidos por una extensión opcional consistente de un punto y hasta tres caracteres. A estos nombres también se les llama "*nombres de archivo externo*", puesto que son los nombres con que se graban en el disco.

Tipos de archivos

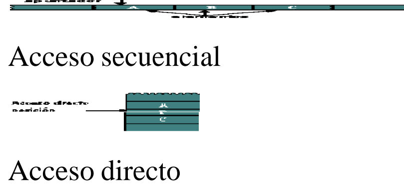
Existen tres tipos de archivos de datos en Turbo Pascal :

1. texto (text) o secuenciales (acceso secuencial),
2. tipeados (tipificados) o con tipo (file of) (acceso aleatorio), aleatorios,
3. no tipeados (no tipificados) o sin tipo (file).

Archivos de texto (secuenciales) :	Son archivos que contienen texto (carácter ASCII)
Archivos con tipo (aleatorios) :	Archivos que contienen datos de cualquier tipo como integer, real, byte, record, datos con estructuras.
Archivos sin tipo :	Archivos en los que no se conoce su estructura ni su contenido; están concebidos para acceso de bajo nivel a los datos de un disco (E/S de bytes).

Tipos de acceso a un archivo

Existen dos modalidades para acceder a un archivo de datos : acceso secuencial y acceso directo o aleatorio. El acceso secuencial exige elemento a elemento, es necesario una exploración secuencial comenzando desde el primer elemento. El acceso directo permite procesar o acceder a un elemento determinado haciendo una referencia directamente por su posición en el soporte de almacenamiento. Pascal estándar sólo admite el acceso secuencial, pero Turbo Pascal permite el acceso directo.



Acceso secuencial

Acceso directo

Declaración de archivos

La declaración de un archivo consta de dos pasos :

1. Declaración del tipo de archivo adecuado :
   - 1.1 file of *char* archivo de texto
     file of text
   - 1.2 file of <tipo> archivo con tipo
   - 1.3 file archivo sin tipo
2. Declaración de una variable archivo de un tipo de archivo declarado.

Declaración de un tipo archivo (file)

Un tipo archivo se declara de igual modo que cualquier otro tipo de dato definido por el usuario: en la sección de declaración de tipos (type).

Formato:

```
Type
  nombre = file of tipo de datos
```

nombre identificador que se asigna como nombre del tipo archivo

tipo de datos tipo de datos de los elementos del archivo

Ejemplos:

```
type ArchEntero = file of integer;
{archivo de enteros}
type ArchCarac  = file of char;
{archivo de carácteres}
type nomina     = file of empleado;
{archivo de registros}
type ciudad     = file of string[20];
{archivo de cadenas}
```

Variable tipo archivo (file)

Para definir un archivo con tipos, simplemente declare una variable archivo.

Ejemplo:

```
Var
  arch_nomina : nomina;
  enteros     : ArchEntero;
```

O también

```
Var
  arch_nomina : file of empleado;
  enteros     : file of integer;
```

Variables de tipo texto y sin tipo
Este tipo de variables no requiere ninguna declaración de tipos; así pues, se puede declarar con un identificador predefinido (Text,File):

```
Var
  texto   : text;
  Archivo : file;
```

Gestión de archivos
La siguiente tabla recopila todos los procedimientos y funciones estándar para tratamiento y manipulación de archivos en Turbo Pascal.

Todos tipos de archivos	Archivos de texto	Archivos sin tipo
Procedimientos		
Assign	Append	BlockRead
ChDir	Flush	BlockWrite
Close	Read	
Erase	ReadLn	
GetDir	SetTexBuf	
MkDir	Write	
Rename	WriteLn	
Reset		
Rewrite		
RmDir		
Funciones		
Eof	Eoln	
IOResult	SeekEof	
	SeekEoln	
Procedimientos y funciones sobre cualquier tipo de variable excepto archivos de texto		
FilePos		
FileSize		
Seek		
Truncate		

Assign

Éste procedimiento realiza la operación de asignar un archivo mediante una correspondencia entre una variable tipo archivo con un archivo externo situado en un disco.

Formato :assign (f,nombre)

f nombre interno del archivo por el que se conoce el archivo dentro del programa(por ejemplo, el utilizado en la declaración

nombre nombre externo con el que se conoce el archivo por el sistema operativo (por ejemplo : 'a:program4.pas'); es una cadena que se define como una constante en el programa o bien una variable cuyo valor se lee interactivamente; puede ser una unidad:nombre.extensión o bien la ruta completa a un archivo.

Ejemplo :

```
Const
  ruta ='c:\tp\lista.pas';
Var
  ArchText : Text;
  ArchBin  : File;
begin
  assign (ArchText,ruta);
  assign (ArchBin,'alumnos.dbf')
    .
    .
    .
```

ChDir

Formato:

ChDir(s)

Cambia el subdirectorio actual al especificado por la variable de cadena s.

Ejemplo (asignación del archivo lista.pas ubicado en c:\tp\lista.pas)

```
Var
  ArchText : Text;
begin
ChDir('C:\tp\');
  assign (ArchText,'lista.pas');
    .
```

CLOSE

Éste procedimiento nos permite cerrar los archivos después que han sido utilizados, si los archivos no son cerrados se perderán los datos que se encuentran en la memoria auxiliar.

Formato :

Close (f)

f Variable de archivo.

Ejemplo:

```
Var
  ArchText : Text;
begin
  assign (ArchText,'c:\tp\lista.pas');
  rewrite(ArchText);
  close(ArchText);
    .
```

Erase

Éste procedimiento nos permite borrar un archivo, el archivo a borrar no debe estar abierto. Para borrar un archivo se debe realizar lo siguiente :

1. Asignar el archivo externo a una variable de archivo.
2. Llamar al procedimiento erase

Formato:

erase(s)

Borra (elimina) un archivo cuya ruta de acceso está especificada por s.

Ejemplo:

```
Var
 ArchText : Text;
begin
 assign (ArchText,'c:\tp\lista.pas');
 erase(ArchText);
      -
```

GetDir

Formato :

GetDir(i,s)

Busca el nombre del subdirectorio actual. El primer parámetro i es un entero que representa la unidad : i=0 (unidad de disco actual, por defecto), i=1 (unidad A), i=2 (unidad B), etc. Después de la llamada, la variable de cadena s contiene el subdirectorio actual de esa unidad.

Ejemplo:

```
Var s:string[80];
begin
ChDir('c:\tp\');
GetDir(0,s);
 WriteLn(s);   {Resulta c:\tp}
  .
```

MkDir

Formato:

MkDir(s)

Crea un nuevo subdirectorio de nombre s.

Ejemplo:

```
begin
 MkDir('C:\nuevo\');
  .
```

Rename

Éste procedimiento renombra (cambia el nombre) un archivo externo. Para renombrar un archivo se debe hacer los siguiente :

1. Asignar un archivo a la variable archivo
2. Llamar al procedimiento rename

Formato:

Rename(f,nombre_archivo)

f	variable de archivo
nombre_archivo	nuevo nombre del archivo

Ejemplo :

```
Var
 ArchText : Text;
begin
 Assign (ArchText,'viejo.pas');
 Rename (ArchText,'nuevo.pas');
  .
```

Reset

Éste procedimiento abre un archivo existente para una operación de lectura. Si se intenta llamar a Reset y el archivo especificado no existe, se producirá un error de E/S (entrada/salida).

Formato :

Reset(f)

f variable de tipo archivo

Ejemplo:

```
Var
 ArchText : Text;
begin
 Assign (ArchText,'lista.pas');
 Reset(ArchText);
  .
```

Rewrite

Crea y abre un nuevo archivo. Si el archivo ya existe, Rewrite borra su contenido; en caso contrario, el archivo queda abierto para una operación de escritura.

Formato:

Rewrite(f)

f variable de archivo

Ejemplo:

```
Var
 ArchText : Text;
begin
 Assign (ArchText,'lista.pas');
 Rewrite(ArchText);
  .
  .
  .
```

RmDir

Formato

RmDir(s)

Elimina el subdirectorio de nombre s.

```
begin
RmDir('C:\nuevo\');
  .
  .
```

Append

El procedimiento Append abre un archivo existente para añadir datos al final del mismo.

Formato:

Append(f)

f variable de archivo de texto que debe haber sido asociada con un archivo externo por medio de Assign. Si el archivo no existe, se produce un error; y si ya estaba abierto, primero se cierra y luego se reabre.

Ejemplo:

```
Var
 ArchText : Text;
begin
 Assign (ArchText,'lista.pas');
 Rewrite(ArchText);
 WriteLn(ArchText,'Primera línea');
 Close(ArchText);
 Append(ArchText);
 WriteLn(ArchText,'Agrega esto al último');
 Close(ArchText)
end.
```

Flush

El procedimiento Flush vacía el *buffer* de un archivo de texto abierto para salida. Cuando un archivo de texto es abierto para escritura usando Rewrite o Append, la llamada a Flush llenará el *buffer* asociado al archivo. Esto garantiza que todos los carácteres escritos en el archivo en ese tiempo sean escritos en el archivo externo. Flush no afecta los archivos abiertos para entrada.

Usando la directiva {$I}, IOResult devuelve 0 si esta operación fue realizada con éxito de lo contrario retonará un código de error.

Formato:

Flush(f)

f variable de archivo

Ejemplo:

```
Var
 ArchText : Text;
begin
 Assign (ArchText,'lista.pas');
 Rewrite(ArchText);
 WriteLn(ArchText,'Primera línea');
 WriteLn(ArchText,'Segunda línea');
 Flush(ArchText);
 WriteLn(ArchText,'tercera línea');
 WriteLn(ArchText,'Agrega esto al último');
 Close(ArchText)
end.
```

Read

El procedimiento Read se utiliza para la lectura de datos situados en un archivo de tipo texto.

Formato:

Read(f,v1,v2,..)

f variable de archivo de texto
v1,v2,... variable de tipo char,integer, real o string

Ejemplo:

```
Var
 ArchText : Text;
 datos    : string[20]; begin
Assign(ArchText,'lista.pas');
Reset(ArchText);
Read(ArchText,datos);
WriteLn(ArchText,datos);
Close(ArchText)
end.
```

ReadLn

EL procedimiento ReadLn se utiliza para la lectura de datos situados en un archivo de tipo texto. A diferencia de Read, ReadLn salta al principio de la siguiente línea del archivo. Este salto de línea se produce cuando se han asignado valores a la lista de varibles del procedimiento; en caso contrario, el procedimiento hace caso omiso del control de línea y sigue asignando información.

Formato:

ReadLn(f,v1,v2,..)

f variable de archivo de texto
v1,v2,... variable de tipo char,integer, real o string

Ejemplo:

```
Var
  ArchText : Text;
datos    : string[20];
begin
  Assign (ArchText,'lista.pas');
  Reset(ArchText);
  ReadLn(ArchText,datos);
  WriteLn(ArchText,datos);
  Close(ArchText)
end.
```

SetTextBuf

El procedimiento SetTextBuf asigna un *buffer* de entrada/salida a un archivo de texto. SetTextBuffer podría nunca ser usado en la apertura de un archivo, aunque éste puede ser llamado inmediatamente después de Reset, Rewrite, y Append.

Si se invoca a SetTextBuf en una apertura de archivo las operaciones de entrada/salida son tomadas por éste procedimiento, y se pueden perder datos como resultado de el cambio de *buffer*.

formato :

SetTextBuf(f,buffer)

Ejemplo:

```
Var
  ArchText: Text;
  Ch: Char;
  buffer: array[1..4096] of Char; { buffer de 4K}
begin
  Assign(ArchText,'memoria.txt');
  {Se utiliza un buffer grande para
 lecturas más rápidas}
 SetTextBuf(ArchText,buffer);
 Reset(ArchText);
  {Visualiza el archivo en la pantalla}
  while not Eof(ArchText) do
  begin
    Read(ArchText,Ch);
    Write(Ch)
  end;
  Close(ArchText)
end.
```

Write

EL procedimiento Write sirve para escribir datos en un archivo.

Formato:

Write(f,v1,v2,...)

f variable de tipo archivo

v1,v2,... variables de tipo de datos

Ejemplo:

```
Var
  ArchText : Text;
begin
  Assign (ArchText,'nuevo.txt');
  Rewrite(ArchText);
  Write(ArchText,'Está es la primera línea.');
  Write(ArchText,'Fin del texto.');
  Close(ArchText)
end.
```

El contenido del archivo será :

Está es la primera línea.Fin del texto.

WriteLn

EL procedimiento WriteLn sirve para escribir datos en un archivo. A diferencia de Write, WriteLn incluye un salto de línea para separar el texto.

Formato:

WriteLn(f,v1,v2,...)

f variable de tipo archivo
v1,v2,... variables de tipo de datos

Ejemplo:

```
Var
  ArchText : Text;
begin
  Assign (ArchText,'nuevo.txt');
  Rewrite(ArchText);
  WriteLn(ArchText,'Está es la primera línea.');
  WriteLn(ArchText,'Fin del texto.');
  Close(ArchText)
end.
```

El contenido del archivo será :

Está es la primera línea. Fin del texto.

BlockRead

Transfiere un bloque de datos de un archivo sin tipo hacia un *buffer*.

Formato:

BlockRead(arch,buffer,bloques,resul)

arch archivo sin tipo
buffer variable de cualquier tipo de longitud suficiente para acoger los datos transferidos
bloques expresión que corresponde al número de bloques de 128 bytes a transferir.
resul parámetro opcional que indica el número de bloques que se leyeron o escribieron realmente.

BlockWrite

Transfiere un buffer de datos hacia un archivo sin tipo.

Formato:

BlockWrite(arch,buffer,bloques,resul)

arch archivo sin tipo
buffer variable de cualquier tipo de longitud suficiente para acoger los datos transferidos
bloques expresión que corresponde al número de bloques de 128 bytes a transferir.
resul parámetro opcional que indica el número de bloques que se leyeron o escribieron realmente.

Ejemplo:

```
Program CopiaDeArchivos;
Var
  fuente,destino : file; {archivo sin tipo}
  buffer        : array[1..512] of byte;
  leidos        : integer;
begin
  Assign(fuente,'original.txt');
  Assign(destino,'copia.txt');
  Reset(fuente,1);
  Rewrite(destino,1);
  BlockRead(fuente,buffer,SizeOf(buffer),leidos);
  while leidos>0 do begin
    BlockWrite(destino,buffer,SizeOf(buffer),leidos);
    BlockRead(fuente,buffer,SizeOf(buffer),leidos)
    end
  close(fuente);
close(destino)
end.
```

Eof

La función eof(*end of file*), fin de archivo, devuelve el estado de un archivo. Es una función de tipo lógico que indica si el fin de archivo se ha encontrado; devuelve true si se encontró, false en casi contrario.

Formato:

Eof(f)

f variable de archivo

Ejemplo:

```
Var
  ArchText: Text;
  Ch: Char;
begin
  Assign(ArchText,'memoria.txt');
  Reset(ArchText);
  {Visualiza el archivo en la pantalla}
  while not Eof(ArchText) do
  begin
    Read(ArchText,Ch);
    Write(Ch)
  end;
  Close(ArchText)
end.
```

Eoln

La función Eoln devuelve el estado de fin de línea de un archivo. Es una función de tipo lógico. Devuelve true si la posición actual del archivo (puntero) esta en la marca de fin de línea, o bien si Eof(f) es true, el caso contrario devuelve false.

Formato:

Eoln(f)

f variable de archivo de texto

```
Var
  car   :char;
ArchText:text;
begin
```

```
  Clrscr;
  Assign(ArchText,'nuevo.txt');
  Reset(ArchText);
  {muestra sola la primera línea}
  While not(Eoln(ArchText)) do
  begin
    Read(ArchText,car);
    Write(car)
   end
end.
```

SeekEof

Retorna el estado de fin de archivo. Es una función de tipo lógico. Sólo para archivos de texto. Si la posición actual del puntero de archivo se encuentra en la marca de fin de archivo devuelve true, de lo contrario retorna false. SeekEof es una función que avanza siempre al siguiente carácter, ignorando por completo los espacios en blanco.

Formato:

SeekEof(f)

f variable de archivo de texto

Ejemplo:

```
Var
  car   :char;
ArchText:text;
begin
  Clrscr;
  Assign(ArchText,'nuevo.txt');
  Reset(ArchText);
  {muestra el contenido del archivo}
  While  not(SeekEof(ArchText))  do
  begin
    Read(ArchText,car);
    Write(car)
   end
end.
```

SeekEoln

Retorna el estado de fin de línea. Es una función de tipo lógico. Sólo para archivos de texto. Si la posición actual del puntero de archivo se encuentra en la marca de fin de línea devuelve true, de lo contrario retorna false. SeekEoln es una función que avanza siempre al siguiente carácter, ignorando por completo los espacios en blanco.

Formato:

SeekEoln(f)

f variable de archivo de texto

Ejemplo:

```
Var
  car   :char;
ArchText:text;
begin
  Clrscr;
  Assign(ArchText,'nuevo.txt');
  Reset(ArchText);
  {muestra el contenido del archivo}
  While not(SeekEoln(ArchText)) do
  begin
    Read(ArchText,car);
    Write(car)
   end;
  Close(ArchText)
end.
```

IOResult

Es una función que devuelve un número entero que indica el código del error de la última operación de E/S; si el valor de retorno es cero, esto nos indica que la operación se ha desarrollado sin problema.
Formato:

IOResult

Ejemplo:

```
Var
ArchText:text;
Nerror  :integer;
begin
  Clrscr;
  Assign(ArchText,'nuevo.txt');
   {$I-}
   Reset(ArchText);
   Nerror:=IOResult;
   {$I+}
   if Nerror<>0  then
    WriteLn('Código de error # ',Nerror)
   else
    Close(ArchText)
end.
```

FilePos

Está función devuelve la posición actual del archivo (número de registro), en forma de un entero largo (longint).
Formato:

FilePos (f)

f varible de tipo archivo
Notas :

1. Devuelve 0 si esta al principio del archivo.
2. Es igual a FileSize(f) si el puntero de posición del archivo esta al final del archivo.

FileSize

Está función devuelve el tamaño actual del archivo(número de registros existentes en el archivo). Si el archivo esta vacío devuelve cero.
Formato:

FileSize(f)

f varible de tipo archivo

Seek

Sitúa el puntero de posición del archivo en el número de registro correspondiente.
Formato:

Seek(f,numreg)

f nombre de archivo (no de tipo text)
numreg número de posición del registro, el primer registro es cero; si numreg es menor que 0 o mayor que n (para un archivo de n registros), se producirán resultados impredecibles.

Ejemplo:

```
Seek(empleados,FileSize(empleados));
{está sentecia posiciona
el puntero al final del archivo}
```

Truncate

Está función trunca (corta, mutíla) un archivo a partir de la posición actual del puntero del archivo.
Formato:

Truncate(f)

f varible de tipo archivo

Si Truncate se ha ejecutado con éxito IOResult tendrá un valor de cero. El archivo debe estar abierto. Truncate no trabaja en archivos de texto.

Ejemplo:

```
Var
  f: file of integer;
  i,j: integer; begin
Assign(f,'prueba.int');
Rewrite(f);
  for i := 1 to 5 do
    Write(f,i);
  Writeln('Archivo antes de ser truncado :');
  Reset(f);
  while not Eof(f) do
  begin
    Read(f,i);
    Writeln(i)
  end;
  Reset(f);
  for i := 1 to 3 do
    Read(f,j); { lectura de 3 registros }
  Truncate(f); { Trunca el archivo apartir de aquí }
  Writeln;
  Writeln('Archivo despues de ser truncado:');
  Reset(f);
  while not Eof(f) do
  begin
    Read(f,i);
    Writeln(i)
  end;
  Close(f);
end.
```

Archivos de texto (secuenciales)

Un archivo de texto está constituido por elementos que son caracteres Pascal (pertenecientes al código ASCII) .

Un archivo de texto consta de una serie de líneas y separadas por una marca de fin de línea (eoln, "*end of file*"). La marca de *fin de línea* es una secuencia de caracteres CR(carriage return) y LF (line feed), que se conoce como retorno de carro y avance de línea. La combinación CR/LF (códigos ASCII 10 y 13) se conoce como delimitador y se obtiene pulsando la tecla Intro (Enter o Return).

Un archivo de texto está constituido por una serie de líneas de caracteres separados por CR/LF (pulsación de la tecla Enter,).

Esto es una prueba de un archivo de texto
Donde cada línea de texto finaliza con CR/LF,
{línea vacía}
Última línea del archivo de texto...

Los archivos de texto se terminan con una marca de final de archivo CTRL-Z (eof, *end of file*).

El Turbo Pascal el delimitador eoln se trata como caracteres independientes: un *retorno de carro*, que posiciona el cursor (puntero) a la primera columna de la línea actual; un *avance de línea,* que mueve el cursor a la siguiente línea .

Ejemplo:
archivo de entrada Fin.
Tiene 6 caracteres (F, i, n, . ,#10, #13)
Nota: Un archivo de texto similar a un archivo de caracteres (char). La única diferencia es que un archivo de texto se divide en líneas y un archivo de caracteres no. Los archivos de caracteres se leen y escriben de carácter en carácter, mientras que los archivos de texto se leen línea a línea. La declaración es de la siguiente manera :

```
Var
  Arch : file of char;
```

Declaración de un archivo
El proceso de archivos de texto exige los siguientes pasos:

1. Declaración del archivo.
2. Apertura del archivo.
3. Leer datos del archivo o escribir datos en él.
4. Cierre del archivo.

La declaración de un archivo consta de dos operaciones:

1. Definir una variable de tipo archivo Text.
2. Asociar a esta variable el nombre de un archivo en disco (sentencia Assign).

Formato para la definición de una variable tipo Text:

```
Var
  nombre:Text;
```

Como cualquier otra variable el tipo Text se puede definir local o globalmente; pero al contrario que otras estructuras, la longitud de una variable archivo no se precisa.

Asignación de archivos
Este procedimiento realiza la operación de asignar un archivo mediante una correspondencia entre una variable tipo archivo con un archivo externo situado en un disco.
Formato :

```
assign (f,nombre)
```

f nombre interno del archivo por el que se conoce el archivo dentro del programa, por ejemplo, el utilizado en la declaración

nombre nombre externo con el que se conoce el archivo por el sistema operativo (por ejemplo : 'a:program4.pas'); es una cadena que se define como una constante en el programa o bien una variable cuyo valor se lee interactivamente; puede ser una unidad: nombre.extensión o bien la ruta completa a un archivo.

Ejemplo :

```
Const
  ruta ='c:\tp\lista.pas';
Var
  ArchText : Text;
ArchBin  : File;
begin
  assign (ArchText,ruta);
  assign (ArchBin,'alumnos.txt')
   .
```

Después que se ha asignado un identificador de archivo, se prepara el archivo con una de estas tres órdenes (procedimientos): Reset, Rewrite, o Append.

Apertura de un Archivo

Después de haber asignado, un archivo debe ser abierto. Esta operación se realiza por uno de los dos procedimientos predefinidos: rewrite y reset.

Reset

Este procedimiento abre un archivo existente para una operación de lectura. Si se intenta llamar a Reset y el archivo especificado no existe, se producirá un error de E/S (entrada/salida).

Formato :

Reset(s)

s variable de tipo archivo

Ejemplo:

```
Var
  ArchText : Text;
begin
  Assign (ArchText,'lista.pas');
  Reset(ArchText);
    .
    .
    .
```

Rewrite

Crea y abre un nuevo archivo. Si el archivo ya existe, Rewrite borra su contenido; en caso contrario, el archivo queda abierto para una operación de escritura.

Formato:

Rewrite(s)

s variable de archivo

Ejemplo:

```
Var
  ArchText : Text;
begin
  Assign (ArchText,'lista.pas');
  Rewrite(ArchText);
    .
```

Notas:

- Si al abrir un archivo de texto que ya existe en el disco con la sentencia Rewrite lo reescribirá (el contenido anterior a la ejecución de la sentencia Rewrite se perderá).
- Por el contrario las sentencias assign y reset suponen la existencia del archivo llamado en el disco. Si el archivo no existe se producirán errores de ejecución (Runtime error 002 at 0BE2:0039). Mediante la directiva {$I-} se puede desactivar la detección de errores de Entrada/Salida y prevenir una parada en la ejecución del programa. Una posible solución es la siguiente:

```
Var
  ArchText : Text;
begin
  Assign (ArchText,'lista.pas');
   {$I-}
    Reset(ArchText);
   {$I+}
  if IOResult = 0  then
     {todo bien}
  else
     {error en apertura de archivo}
    .
    .
    .
```

La valor IOResult será cero si Reset se ha ejecutado correctamente.

Escritura de un archivo
Una vez que se ha abierto un archivo para escritura, las sentencias (procedimientos) Write y WriteLn sirven para escribir datos en el nuevo archivo.

Procedimiento Write.
Formato:

Write(f,v1,v2,...)

f variable de tipo archivo
v1,v2,... variables de tipo de datos

Ejemplo:

```
Var
  ArchText : Text;
begin
  Assign (ArchText,'nuevo.txt');
  Rewrite(ArchText);
  Write(ArchText,'Esta es la primera línea.');
  Write(ArchText,'Fin del texto.')
  Close(ArchText)
end.
```

El contenido del archivo será :
Esta es la primera línea.Fin del texto.

Procedimiento WriteLn
A diferencia de Write, WriteLn incluye un salto de línea para separar el texto.
Formato:

WriteLn(f,v1,v2,...)

f variable de tipo archivo
v1,v2,... variables de tipo de datos

Ejemplo:

```
Var
  ArchText : Text;
begin
  Assign (ArchText,'nuevo.txt');
  Rewrite(ArchText);
  WriteLn(ArchText,'Esta es la primera línea.');
  WriteLn(ArchText,'Fin del texto.');
  Close(ArchText)
end.
```

El contenido del archivo será :
Esta es la primera línea. Fin del texto.

Lectura de un archivo
Los procedimientos Read y ReadLn se utilizan para la lectura de los datos situados en un archivo de tipo texto.

Procedimiento Read.
Formato:

Read(f,v1,v2,..)

f variable de archivo de texto
v1,v2,... variable de tipo char,integer, real
o string

Ejemplo:

```
Var
  ArchText : Text;
datos    : string[20];
begin
  Assign (ArchText,'lista.pas');
  Reset(ArchText);
  Read(ArchText,datos);
  WriteLn(ArchText,datos);
  Close(ArchText)
end.
```

Procedimiento ReadLn

EL procedimiento ReadLn se utiliza para la lectura de datos situados en un archivo de tipo texto. A diferencia de Read, ReadLn salta al principio de la siguiente línea del archivo. Esta salto de línea se produce cuando se han asignado valores a la lista de varibles del procedimiento; en caso contrario, el procedimiento hace caso omiso del control de línea y sigue asignando información.

Formato:

ReadLn(f,v1,v2,..)

f variable de archivo de texto
v1,v2,... variable de tipo char,integer, real o string

Ejemplo:

```
Var
  ArchText : Text;
  datos    : string[20];
begin
  Assign (ArchText,'lista.pas');
  Reset(ArchText);
  ReadLn(ArchText,datos);
  WriteLn(ArchText,datos);
  Close(ArchText)
end.
```

Añadir datos a un archivo

El procedimiento Append abre un archivo existente para añadir datos al final del mismo.

Formato:

Append(f)

f variable de archivo de texto que debe haber sido asociada con un archivo externo por medio de Assign.

Si el archivo no existe, se produce un error; y si ya estaba abierto, primero se cierra y luego se reabre.

Ejemplo:

```
Var
  ArchText : Text;
begin
  Assign (ArchText,'lista.pas');
  Rewrite(ArchText);
  WriteLn(ArchText,'Primera línea');
  Close(ArchText);
  Append(ArchText);
  WriteLn(ArchText,'Agrega esto al último');
  Close(ArchText)
end.
```

Archivos de acceso secuencial (con tipo)
Dependiendo de la manera en que se accesen los registros de un archivo, se le clasifica como SECUENCIAL o como DIRECTO.
En el caso de los archivos de ACCESO SECUENCIAL, para tener acceso al registro localizado en la posición N, se deben haber accesado los N-1 registros previos, en un orden secuencial.
Cuando se tienen pocos registros en un archivo, o que los registros son pequeños, la diferencia entre los tiempos de acceso de forma secuencial y directa puede no ser perceptible para el usuario; sin embargo, la diferencia viene a ser significativa cuando se manejan archivos con grandes cantidades de información.
La forma de manejar los archivos de acceso secuencial es más sencilla en la mayoría de los lenguajes de programación, por lo que su estudio se antepone al de los archivos de acceso directo.
El manejo secuencial de un archivo es recomendable cuando se deben procesar todos o la mayoría de los registros, como por ejemplo en los casos de una nómina de empleados o en la elaboración de reportes contables.

Declaración y asignación de Archivos
La declaración de un archivo con tipo se efectúa con la ayuda de las palabras reservadas file of.
El procedimiento de asignación es idéntico al utilizado anteriormente.
Ejemplo:

```
Type
  datos = record
          clave  : string[3];
          nombre : string[30];
          puesto : string[20];
          sueldo : real;
         end;
Var
 archivo:file of datos; begin
Assign(archivo,'empleado.dat');
  .
  .
  .
```

Escritura de un archivo
Una vez que se ha abierto un archivo para escritura, con el procedimiento Rewrite o Append (caso de añadir registros) se utilizan las sentencias (procedimientos) Write para escribir datos en el archivo.
Como ejemplo, veamos el siguiente procedimiento que sirve para escribir en el archivo registros que están formados por los campos :
clave :de tipo cadena (3 caracteres) ,
nombre :de tipo cadena (30 caracteres),
puesto :de tipo cadena (20 caracteres), y
sueldo :de tipo real (6 octetos) .

El siguiente ejemplo que se manejará a lo largo de está unidad es idéntico al realizado en la unidad siete de listas enlazadas (Listas_enlazadas) . Con la única diferencia que el lugar de listas enlazadas utilizaremos archivos.

```
procedure altas;
Var
 otro   :char;
 CodError:integer;
begin
  {$I-} Reset(archivo);
   CodError:=IOResult;
  {$I+}
  Case CodError Of
```

```
{Caso cero el archivo ya existe,
 ubicarse en el último registro,
 para añadir registros nuevos}
  0: Seek(archivo,FileSize(archivo));
{caso dos el archivo no existe,
 se crea con Rewrite, listo
 para escritura}
  2: Rewrite(archivo)
{caso contrario existe un error}
 else
   begin
      error(CodError);
      exit; {Salta al fin del procedimiento}
   end
end;
With registro Do
begin
 Repeat
   ClrScr;
   gotoxy(30,5);Write('Altas de empleados');
   gotoxy(25,7);Write('Clave : ');
   ReadLn(clave);
   gotoxy(25,8);Write('Nombre : ');
   ReadLn(nombre);
   gotoxy(25,9);Write('Puesto : ');
   ReadLn(puesto);
   Repeat
   {$I-}      {validación de entrada de datos}
    gotoxy(25,10);write('Sueldo : ');
    ReadLn(sueldo);
   {$I+}
   until IOResult=0;
   Write(archivo,registro);
   {Escribe los datos del registro en el archivo}
   gotoxy(20,22);write('Desea dar otra alta s/n? ');
   otro:=ReadKey
 until otro in ['n','N',Esc]
end;
Close(archivo)
end;
```

Nota : Al final de la unidad se presentará todo el código completo del programa.

Lectura de un archivo

Normalmente la lectura de un archivo no tiene sentido por si sola, sino que adquiere significado cuando se asocia con la actualización del archivo o con la generación de un reporte cualquiera.

De alguna manera podemos pensar en la lectura de un archivo como un proceso inverso al de su creación, puesto que ahora se pasará la información desde el disco hacia la memoria auxiliar.

Veamos un procedimiento que lea todos los registros del archivo "empleados.dat", y los despliegue en la pantalla.

En este procedimiento se asocia la lectura de los registros con la generación de un reporte a través de la pantalla.

```
procedure consultas;
Var
CodError:integer;
begin
  {$I-}
   reset(archivo);
```

```
 CodError:=IOResult;
{$I+}
if CodError<>0 then
  error(CodError)
else
  begin
    With registro Do
      begin
        while not(Eof(archivo)) Do
          begin
            ClrScr;
            Read(archivo,registro);
            gotoxy(30,5);Write('Consulta de empleados');
            gotoxy(25,7);Write('Clave : ');
            Write(clave);
            gotoxy(25,8);Write('Nombre : ');
            Write(nombre);
            gotoxy(25,9);Write('Puesto : ');
            Write(puesto);
            gotoxy(25,10);write('Sueldo : ');
            Write(sueldo:6:2);
            gotoxy(20,22);Write('Presione una tecla...');
            ReadKey
          end
      end;
    Close(archivo)
  end
end;
```

Nota : Al final de la unidad se presentará todo el código completo del programa.

Actualización de un archivo

La actualización (modificación) de un archivo incluye las acciones de:

1. Borrar registros.
2. Agregar registros (visto anteriormente 8.7.3).
3. Modificar los datos contenidos en los campos de un registro.

En un acceso secuencial, toda modificación de un archivo requiere la aplicación de un método padre-hijo (sólo para borrar y modificar datos), el cual se realiza con los siguientes pasos:

1. Se abre el archivo a modificar (padre) en el modo de lectura
2. Se abre un nuevo archivo (hijo) en el modo de escritura, para guardar en él la versión modificada.
3. Se copian de padre a hijo los registros que permanecerán inalterados.
4. Se leen del padre los registros a modificar, los cuales una vez modificados se graban en hijo.
5. Se cierran los archivos padre e hijo.
6. Se borra el archivo padre.
7. Se renombra el hijo con el nombre del padre.

A continuación de presenta un procedimiento para borrar un registro del archivo "empleado.dat" utilizando el método padre-hijo.

```
function busca_clave(clave:string):Boolean;
Var
  CodError:integer;
begin
  {$I-}
   Reset(archivo);
   CodError:=IOResult;
  {$I+}
  if CodError<>0 then
   begin
     error(CodError);
     busca_clave:=false
```

```
   end
  else
  begin
     Read(archivo,registro);
     While (not(Eof(archivo)) and (clave<>registro.clave)) Do
        Read(archivo,registro);
     if(clave=registro.clave) then
     busca_clave:=true
     else
     busca_clave:=false;
     Close(archivo)
   end;
end;
{tipo 1 eliminar un registro,
 tipo 2 se efectúo una modificación}
procedure padre_hijo(clave:string;tipo:byte;aux:datos);
Var
  copia : file of datos;
begin
  Assign(copia,'hijo.dat');
  Rewrite(copia);
  Reset(archivo);
  {proceso de intercambio padre hijo}
    While not(Eof(archivo)) Do
     begin
       Read(archivo,registro);
       if ((registro.clave<>clave)) then
       Write(copia,registro);
       if((registro.clave=clave) and (tipo=2)) then
       {caso modificación}
         Write(copia,aux); {escritura en hijo}
     end;
  Close(copia);Close(archivo);
  Erase(archivo); {borra el padre}
  Rename(copia,'empleados.dat')
  {renombra 'hijo.dat'con 'empleado.dat'}
end;
procedure bajas;
Var
  otro    :char;
  clave   :string[3];
CodError:integer;
begin
  {$I-} Reset(archivo);
   CodError:=IOResult;
  {$I+}
  if CodError<>0 then
   begin
   error(CodError);
     exit
   end;
  Close(archivo);
  Repeat
    ClrScr;
    gotoxy(30,5);Write('Bajas de empleados');
    gotoxy(25,7);Write('Clave : ');
    ReadLn(clave);
    if busca_clave(clave) then
```

```
    begin
      gotoxy(25,8);Write('Nombre : ');
      Write(registro.nombre);
      gotoxy(25,9);Write('Puesto : ');
      Write(registro.puesto);
      gotoxy(25,10);Write('Sueldo : ');
      Write(registro.sueldo:6:2);
      gotoxy(20,15);Write('Desea eliminarlo s/n? ');
      otro:=ReadKey;Write(otro);
      if otro in['s','S'] then begin
        padre_hijo(clave,1,registro);
          gotoxy(20,17);Write('Registro Eliminado...')
        end;
    end
  else
  begin
      gotoxy(20,10); Write('La clave no existe...')
    end;
  gotoxy(20,22);Write('Desea dar otra baja s/n? ');
  otro:=ReadKey
 until otro in ['n','N',Esc]
end;
{Procedimiento para modificación de
los campos de un archivo}
procedure cambios;
Var
 otro    :char;
 clave   :string[3];
 CodError:integer;
 aux     :datos;
begin
 {$I-} Reset(archivo);
  CodError:=IOResult;
 {$I+}
 if CodError<>0 then
  begin
  error(CodError);
  exit
  end;
 Close(archivo);
 Repeat
  ClrScr;
  gotoxy(30,5);Write('Modificaciones de empleados');
  gotoxy(25,6);Write('Si no desea modificar presione Enter');
  gotoxy(25,7);Write('Clave : ');
  ReadLn(clave);
  if busca_clave(clave) then begin
    move(registro,aux,SizeOf(aux));
      gotoxy(1,10);Write('Nombre : ');Write(registro.nombre);
      gotoxy(35,10);Write('Nuevo nombre : ');ReadLn(aux.nombre);
      if(length(aux.nombre)=0) then
        begin aux.nombre:=registro.nombre;
        gotoxy(50,10);WriteLn(aux.nombre);
        end;
      gotoxy(1,11);Write('Puesto : ');Write(registro.puesto);
```

```
        gotoxy(35,11);Write('Nuevo puesto : ');ReadLn(aux.puesto);
        if(length(aux.puesto)=0) then
          begin
          aux.puesto:=registro.puesto;
           gotoxy(50,11);WriteLn(aux.puesto);
          end;
         gotoxy(1,12);write('Sueldo : ');Write(registro.sueldo:6:2);
        Repeat
        {$I-}      {validación de entrada de datos}
         gotoxy(35,12);Write('Nuevo sueldo : ');ReadLn(aux.sueldo);
        {$I+}
        until (IOResult=0);
       if(aux.sueldo=0) then begin
          aux.sueldo:=registro.sueldo;
           gotoxy(50,12);WriteLn(aux.sueldo:6:2);
          end;
        gotoxy(20,15);Write('Las modificaciones están correctas s/n? ');
        otro:=ReadKey;Write(otro);
        if otro in['s','S'] then begin
          padre_hijo(clave,2,aux);
            gotoxy(20,17);Write('Registro modificado...')
          end;
      end
     else
     begin
        gotoxy(20,10); Write('La clave no existe...')
      end;
     gotoxy(20,22);write('Desea realizar otra modificación s/n? ');
     otro:=ReadKey
   until otro in ['n','N',Esc]
  end;
```

Archivos de acceso directo (con tipo)

Los archivos *tipeados* (con tipo), también llamados *archivos binarios*, contienen datos de tipo simple o estructurado, tales como integer, real , record, etc., excepto otro tipo de archivos.

Los archivos con tipos están estructurados en elementos o registros (record) cuyo tipo puede ser cualquiera.

A los elementos de estos archivos se accede directamente, al no situarse éstos en posiciones físicamente consecutivas, sino en posiciones lógicas. Esta es la razón por la cual se les denomina archivos de acceso aleatorio o directo. Los elementos de los archivos aleatorios son de igual tamaño y el término acceso directo significa que es posible acceder directamente a un elemento con solo especificar su posición.

Declaración y asignación de archivos

La declaración de un archivo con tipo se efectúa con la ayuda de las palabras reservadas file of.

El procedimiento de asignación es idéntico al utilizado anteriormente.

Ejemplo:

```
Type
  datos = record
          clave  : integer;
          nombre : string[30];
          puesto : string[20];
          sueldo : real;
          estado : boolean;
          {true activo,false baja lógica}
         end;
Var
 archivo:file of datos; begin
Assign(archivo,'empleado.dat');
  .
```

Escritura de un archivo

Una vez que se ha abierto un archivo para escritura, con el procedimiento Rewrite o Append (caso de añadir registros) se utilizan las sentencias (procedimientos) Write para escribir datos en el archivo.

Como ejemplo, veamos el siguiente procedimiento que sirve para escribir en el archivo registros que están formados por los campos :

clave :de tipo cadena (3 caracteres) ,
nombre :de tipo cadena (30 caracteres),
puesto :de tipo cadena (20 caracteres), y
sueldo :de tipo real (6 octetos) .

El siguiente ejemplo que se manejará a lo largo de esta unidad es identico al realizado en en el cápitulo anterior . La unica diferencia es que se tendrá un acceso directo en todas las operaciones.

```
procedure altas;
Var
  otro    :char;
CodError:integer;
begin
   {$I-} Reset(archivo);
    CodError:=IOResult;
   {$I+}
   Case CodError Of
    {Caso cero el archivo ya existe,
     ubicarse en el último registro,
     para añadir registros nuevos}
      0: Seek(archivo,FileSize(archivo));
    {caso dos el archivo no existe,
     se crea con Rewrite, listo
     para escritura}
      2: Rewrite(archivo)
    {caso contrario existe un error}
     else
       begin
          error(CodError);
          exit
         {Salta al fin del procedimiento}
       end
   end;
   With registro Do
   begin
     Repeat
       ClrScr;
       gotoxy(30,5);Write('Altas de empleados');
       clave:=FileSize(archivo);
       gotoxy(25,7);Write('Clave : ',clave);
       gotoxy(25,8);Write('Nombre : ');
       ReadLn(nombre);
       gotoxy(25,9);Write('Puesto : ');
       ReadLn(puesto);
       Repeat
       {$I-} {validación de entrada de datos}
        gotoxy(25,10);write('Sueldo : ');
        ReadLn(sueldo);
       {$I+}
       until IOResult=0;
       estado:=true;
       Write(archivo,registro);
       {Escribe los datos del registro en el archivo}
```

```
      gotoxy(20,22);write('Desea dar otra alta s/n?');
      otro:=ReadKey
     until otro in ['n','N',Esc]
    end;
    Close(archivo)
  end;
```
Nota : Al final de la unidad se presentará todo el código completo del programa.

Lectura de un archivo

Normalmente la lectura de un archivo no tiene sentido por si sola, sino que adquiere significado cuando se asocia con la actualización del archivo o con la generación de un reporte cualquiera.

De alguna manera podemos pensar en la lectura de un archivo como un proceso inverso al de su creación, puesto que ahora se pasará la información desde el disco hacia la memoria auxiliar.

Veamos un procedimiento que lea todos los registros del archivo "empleados.dat", y los despliegue en la pantalla.

En este procedimiento se asocia la lectura de los registros con la generación de un reporte a través de la pantalla.

```
procedure consultas;
Var
 CodError:integer;
begin
  {$I-} reset(archivo);
   CodError:=IOResult;
  {$I+}
  if CodError<>0 then
    error(CodError)
  else
    begin
      With registro Do
        begin
          while not(Eof(archivo)) Do
            begin
              ClrScr;
              Read(archivo,registro);
              gotoxy(30,5);Write('Consulta de empleados');
              gotoxy(25,7);Write('Clave : ');
              Write(clave);
              gotoxy(25,8);Write('Nombre : ');
              Write(nombre);
              gotoxy(25,9);Write('Puesto : ');
              Write(puesto);
              gotoxy(25,10);write('Sueldo : ');
              Write(sueldo:6:2);
              gotoxy(20,22);Write('Presione una tecla...');
              ReadKey
            end end;
      Close(archivo)
    end
  end;
```
Nota : Al final de la unidad se presentará todo el código completo del programa.

Actualización de un archivo

La actualización (modificación) de un archivo incluye las acciones de:

1. Borrar registros.
2. Agregar registros (visto anteriormente 8.8.3).
3. Modificar los datos contenidos en los campos de un registro.

Agregar

Para poder agregar datos a un archivo es preciso que éste exista, es decir, se necesita: si el archivo no existe, se crea por primera vez para escritura (Rewrite), si el archivo ya existe se posiciona al final del archivo (Seek).

Borrar / Modificar

El procedimiento para borrar o modificar requiere los siguientes pasos:

1. Situarse en la posición del registro con Seek.
2. Leer el registro.
3. Borrar/Modificar.
   3.1 Modificar
   - Leer el nuevo registro modificado (Read).
   - Situarse en la posición correspondiente (Seek).
   - Escribir el nuevo registro en el archivo.

   3.2 Borrar
   - La eliminación será lógica por lo que quedarán huecos en el archivo.

La baja lógica que se aplica en el siguiente ejemplo se realiza por medio de una bandera de estado donde true significa activo y false no activo.

Si no se desea que queden huecos en el archivo, entonces la baja que se requiere es física. La solución a esto es aplicar el proceso padre-hijo visto anteriormente en los archivos con acceso secuencial.

```
procedure bajas;
Var
  otro    :char;
  clave   :integer;
CodError:integer;
begin
  {$I-} Reset(archivo);
   CodError:=IOResult;
  {$I+}
  if CodError<>0 then
   begin
   error(CodError);
     exit
   end;
  Repeat
  ClrScr;
    gotoxy(30,5);Write('Bajas de empleados');
    Repeat
       {$I-} {validación de entrada de datos}
         gotoxy(25,7);Write('Clave : ');
         ReadLn(clave);
       {$I+}
    until IOResult=0; registro.estado:=false;
    if((clave>=0)and(clave<=FileSize(archivo)-1)) then
    {validación para seek}
      begin
      seek(archivo,clave);
      Read(archivo,registro)
      end;
```

```
    if registro.estado=true then {si está activo}
       begin
         gotoxy(25,8);Write('Nombre : ');
         Write(registro.nombre);
         gotoxy(25,9);Write('Puesto : ');
         Write(registro.puesto);
         gotoxy(25,10);write('Sueldo : ');
         Write(registro.sueldo:6:2);
         gotoxy(20,15);Write('Desea eliminarlo s/n? ');
         if otro in['s','S'] then
         begin
           Seek(archivo,clave); registro.estado:=false;
           Write(archivo,registro); {datos actualizados}
           gotoxy(20,17);Write('Registro Eliminado...')
         end
     end
     else
      begin
        gotoxy(20,10); Write('La clave no existe...')
      end;
    gotoxy(20,22);write('Desea dar otra baja s/n? ');
    otro:=ReadKey
  until otro in ['n','N',Esc];
  Close(archivo)
end;

{Procedimiento para modificación de
 los campos de un archivo}
procedure cambios;
Var
  otro    :char;
  clave   :integer;
  CodError:integer;
  aux     :datos;
begin
  {$I-} Reset(archivo);
   CodError:=IOResult;
  {$I+}
  if CodError<>0 then
   begin
   error(CodError);
   exit
   end;
  Repeat
  ClrScr;
    gotoxy(30,5);Write('Modificaciones de empleados');
    gotoxy(25,6);Write('Si no desea modificar presione Enter');
    Repeat
       {$I-} {validación de entrada de datos}
         gotoxy(25,7);Write('Clave : ');
         ReadLn(clave);
       {$I+}
    until IOResult=0;
       registro.estado:=false;
    if((clave>=0)and(clave<=FileSize(archivo)-1)) then
    {validación para seek}
      begin
```

```
      seek(archivo,clave);
     Read(archivo,registro)
     end;
   if registro.estado=true then {si está activo}
      begin move(registro,aux,SizeOf(aux));
        gotoxy(1,10);Write('Nombre : ');Write(registro.nombre);
        gotoxy(35,10);Write('Nuevo nombre : ');ReadLn(aux.nombre);
        if(length(aux.nombre)=0) then
         begin
         aux.nombre:=registro.nombre;
           gotoxy(50,10);WriteLn(aux.nombre);
         end;
        gotoxy(1,11);Write('Puesto : ');Write(registro.puesto);
        gotoxy(35,11);Write('Nuevo puesto : ');ReadLn(aux.puesto);
        if(length(aux.puesto)=0) then
        begin aux.puesto:=registro.puesto;
        gotoxy(50,11);WriteLn(aux.puesto);
        end;
        gotoxy(1,12);write('Sueldo : ');Write(registro.sueldo:6:2);
        Repeat
        {$I-}       {validación de entrada de datos}
         gotoxy(35,12);Write('Nuevo sueldo : ');ReadLn(aux.sueldo);
        {$I+}
       until (IOResult=0);
       if(aux.sueldo=0) then
       begin
           aux.sueldo:=registro.sueldo;
         gotoxy(50,12);WriteLn(aux.sueldo:6:2);
         end;
       gotoxy(20,15);Write('Las modificaciones están correctas s/n? ');
       otro:=ReadKey;Write(otro);
      if otro in['s','S'] then
       begin
       Seek(archivo,clave);
       Write(archivo,aux);
         gotoxy(20,17);Write('Registro modificado...')
       end
     end
   else
     begin
       gotoxy(20,10); Write('La clave no existe...')
     end;
   gotoxy(20,22);write('Desea realizar otra modificación s/n? ');
   otro:=ReadKey
  until otro in ['n','N',Esc]
end;
```

Archivos sin tipo

Todos los archivos utilizados hasta ahora suponen algún tipo de estructura. Si no se conoce la estructura del registro se debe utilizar un archivo sin tipo. Los archivos sin tipo son canales de E/S de bajo nivel, principalmente utilizados para acceso directo a cualquier archivo de disco con independencia del tipo y estructura.

Cualquier archivo de disco puede ser declarado como sin tipo. Turbo Pascal permite tratar un archivo como una serie de bloques sin necesidad de conocer el tipo de datos que contiene.

Declaración de un archivo

La declaración de un archivo sin tipo omite el tipo de archivo :

```
Var
  nombre: file;
```

Usando la declaración de archivos sin tipo, no importa como se haya escrito originalmente el archivo. Los archivos de texto, archivos binarios, archivos de programas o cualquier otro, pueden ser declarados y abiertos como archivos sin tipo.
Los archivos sin tipo tienen una longitud de registro por defecto de 128 bytes, y se transfieren directamente entre la variable registro y el archivo.
El formato de los procedimientos Reset y Rewrite en archivos sin tipo difiere del ya conocido para archivos de texto y con tipo.

```
Reset  (nombre,bloque)
Rewrite(nombre,bloque)
```

nombre variable de tipo archivo
Bloque un número de tipo word

Ejemplo:
Reset(archivo,1);
Prepara a un archivo para ser leído y específica que la longitud de registro es de 1 byte.
Acceso a los archivos (sin tipo)
El acceso al los archivos se realiza con los siguientes procedimientos:

BlockRead	BlockWrite

Formatos:

BlockRead
Transfiere un bloque de datos de un archivo sin tipo hacia un buffer.
Formato:

```
BlockRead(arch,buffer,bloques,resul)
```

arch archivo sin tipo
buffer variable de cualquier tipo de longitud suficiente para acoger los datos transferidos
bloques expresión que corresponde al número de bloques de 128 bytes a transferir.
resul parámetro opcional que indica el número de bloques que se leyeron o escribieron realmente.

BlockWrite
Transfiere un buffer de datos hacia un archivo sin tipo.
Formato:

```
BlockWrite(arch,buffer,bloques,resul)
```

arch archivo sin tipo
buffer variable de cualquier tipo de longitud suficiente para acoger los datos transferidos
bloques expresión que corresponde al número de bloques de 128 bytes a transferir.
resul parámetro opcional que indica el número de bloques que se leyeron o escribieron realmente.

Los archivos sin tipo son más rápidos, debido a que los datos no necesitan ser organizados en líneas o estructuras; son simplemente bloques que se transfieren entre el disco y un buffer.

El siguiente ejemplo muestra una de las aplicaciones que se les puede dar a los archivos sin tipo. La función del siguiente ejemplo es el de copiar el contenido de un archivo a una ruta específica (idéntico al comando Copy de MS-DOS).

Ejemplo:

```
Program Copiar; Var
 fuente,destino : file; {archivo sin tipo}
 buffer        : array[1..512] of byte;
 temp          : string;
 leidos        : integer;

            begin
 if ((ParamCount<2)or (ParamCount>2)) then begin
 if (ParamCount<2)then WriteLn('Faltan
 párametros...') else
  WriteLn('Exceso de párametros...');
  WriteLn('Ejemplo: Copiar [Nombre de archivo a copiar]',
                '[Ruta destino del archivo]');
  WriteLn('//Copiar datos.doc a://'); Halt(1);
 end;
 Assign(fuente,ParamStr(1));
 temp:=ParamStr(2)+ParamStr(1);
 {archivo destino con ruta}
 Assign(destino,temp);
 {$I-} Reset(fuente,1);
 {$I+}
 if IOResult <> 0 then begin
   WriteLn('Error en archivo fuente..'); Halt(1);
  end;
 {$I-} Rewrite(destino,1);
 {$I+}
 if IOResult <> 0 then begin
   WriteLn('Error en la ruta destino..'); Halt(1);
  end;
 BlockRead(fuente,buffer,SizeOf(buffer),leidos);
 while leidos>0 do begin
   BlockWrite(destino,buffer,SizeOf(buffer),leidos);
   BlockRead(fuente,buffer,SizeOf(buffer),leidos) end;
 close(fuente);       close(destino)
end.
```

2.2 y 2.3 Datos Estructurados y Arreglos en Lenguaje C

TIPOS DE DATOS DERIVADOS.

Además de los tipos de datos fundamentales vistos en la Sección 2, en C existen algunos otros tipos de datos muy utilizados y que se pueden considerar derivados de los anteriores. En esta sección se van a presentar los ***punteros***, las ***matrices*** y las ***estructuras***.

Punteros

CONCEPTO DE PUNTERO O APUNTADOR

El valor de cada variable está almacenado en un lugar determinado de la memoria, caracterizado por una ***dirección*** (que se suele expresar con un número hexadecimal). El ordenador mantiene una ***tabla de direcciones*** (ver Tabla 6.1) que relaciona el nombre de cada variable con su dirección en la memoria. Gracias a los nombres de las variables (identificadores), de ordinario no hace falta que el programador se preocupe de la dirección de memoria donde están almacenados sus datos. Sin embargo, en ciertas ocasiones es más útil trabajar con las direcciones que con los propios nombres de las variables. El lenguaje C dispone del ***operador dirección*** (&) que permite determinar la dirección de una variable, y de un tipo especial de variables destinadas a contener direcciones de variables. Estas variables se llaman ***punteros*** o ***apuntadores*** (en inglés *pointers*).

Así pues, un ***puntero*** es una variable que puede contener la ***dirección*** de otra variable. Por supuesto, los *punteros* están almacenados en algún lugar de la memoria y tienen su propia dirección (más adelante se verá que existen *punteros a punteros*). Se dice que un *puntero apunta a una variable* si su contenido es la dirección de esa variable. Un *puntero* ocupa de ordinario 4 bytes de memoria, y *se debe declarar o definir de acuerdo con el tipo del dato* al que apunta. Por ejemplo, un *puntero* a una variable de tipo ***int*** se *declara* del siguiente modo:

```
int *direc;
```

lo cual quiere decir que a partir de este momento, la variable ***direc*** podrá contener la dirección de cualquier variable entera. La regla nemotécnica es que el valor al que apunta ***direc*** (es decir ****direc***, como luego se verá), es de tipo *int*. Los *punteros* a *long, char*, *float* y *double* se definen análogamente a los *punteros* a *int*.

OPERADORES DIRECCIÓN (&) E INDIRECCIÓN (*)

Como se ha dicho, el lenguaje C dispone del ***operador dirección*** (&) que permite hallar la dirección de la variable a la que se aplica. Un *puntero* es una verdadera variable, y por tanto puede cambiar de valor, es decir, puede cambiar la variable a la que apunta. Para acceder al valor depositado en la zona de memoria a la que apunta un *puntero* se debe utilizar el ***operador indirección*** (*). Por ejemplo, supóngase las siguientes declaraciones y sentencias,

```
int i, j, *p;       // p es un puntero a int
p = &i;             // p contiene la dirección de i
*p = 10;            // i toma el valor 10
p = &j;             // p contiene ahora la dirección de j
*p = -2;            // j toma el valor -2
```

Las constantes y las expresiones no tienen dirección, por lo que no se les puede aplicar el operador (&). Tampoco puede cambiarse la dirección de una variable. Los valores posibles para un puntero son las direcciones posibles de memoria. Un puntero puede tener valor 0 (equivalente a la

constante simbólica predefinida NULL). No se puede asignar una dirección absoluta directamente (habría que hacer un *casting*). *Las siguientes sentencias son ilegales:*

```
p = &34;            // las constantes no tienen dirección
p = &(i+1);         // las expresiones no tienen dirección
&i = p;             // las direcciones no se pueden cambiar
p = 17654;          // habría que escribir p = (int *)17654;
```

Para imprimir *punteros* con la función ***printf()*** se deben utilizar los formatos **%u** y **%p**, como se verá más adelante.

No se permiten asignaciones directas (sin *casting*) entre punteros que apuntan a distintos tipos de variables. Sin embargo, existe un tipo indefinido de punteros (***void ****, o ***punteros a void***), que puede asignarse y al que puede asignarse cualquier tipo de puntero. Por ejemplo:

```
int     *p;
double  *q;
void    *r;
p = q;              // ilegal
p = (int *)q;       // legal
p = r = q;          // legal
```

ARITMÉTICA DE PUNTEROS

Como ya se ha visto, los *punteros* son unas variables un poco especiales, ya que guardan información –no sólo de la dirección a la que apuntan–, sino también del *tipo* de variable almacenado en esa dirección. Esto implica que *no van a estar permitidas las operaciones que no tienen sentido con direcciones de variables*, como multiplicar o dividir, pero sí otras como sumar o restar. Además estas operaciones se realizan de un modo correcto, pero que no es el ordinario. Así, la sentencia:

```
p = p+1;
```

hace que **p** apunte a la dirección siguiente de la que apuntaba, teniendo en cuenta el tipo de dato. Por ejemplo, si el valor apuntado por **p** es *short int* y ocupa 2 bytes, el sumar 1 a **p** implica añadir 2 bytes a la dirección que contiene, mientras que si **p** apunta a un *double*, sumarle 1 implica añadirle 8 bytes.

También tiene sentido la *diferencia de punteros* al mismo *tipo* de variable. El resultado es la *distancia* entre las direcciones de las variables apuntadas por ellos, no en *bytes* sino en *datos* de ese mismo tipo. Las siguientes expresiones tienen pleno sentido en C:

```
p = p + 1;
p = p + i;
p += 1;
p++;
```

Tabla 6.1. Tabla de direcciones.

Variable	Dirección de memoria
a	00FA:0000
b	00FA:0002
c	00FA:0004
p1	00FA:0006
p2	00FA:000A

p	00FA:000E

El siguiente ejemplo ilustra la aritmética de punteros:

```
void main(void) {
     int   a, b, c;
     int   *p1, *p2;
     void  *p;

     p1 = &a;    // Paso 1. La dirección de a es asignada a p1
     *p1 = 1;    // Paso 2. p1 (a) es igual a 1. Equivale a a = 1;
     p2 = &b;    // Paso 3. La dirección de b es asignada a p2
     *p2 = 2;    // Paso 4. p2 (b) es igual a 2. Equivale a b = 2;
     p1 = p2;    // Paso 5. El valor del p1 = p2
     *p1 = 0;    // Paso 6. b = 0
     p2 = &c;    // Paso 7. La dirección de c es asignada a p2
     *p2 = 3;    // Paso 8. c = 3
     printf("%d %d %d\n", a, b, c);    // Paso 9. ¿Qué se imprime?

     p = &p1;    // Paso 10. p contiene la dirección de p1
     *p = p2;    // Paso 11. p1= p2;
     *p1 = 1;    // Paso 12. c = 1
     printf("%d %d %d\n", a, b, c);    // Paso 13. ¿Qué se imprime?
}
```

Supóngase que en el momento de comenzar la ejecución, las direcciones de memoria de las distintas variables son las mostradas en la Tabla 6.1.

La dirección de memoria está en hexadecimal, con el *segmento* y el *offset* separados por dos puntos (:); basta prestar atención al segundo de estos números, esto es, al *offset*.

La Tabla 6.2 muestra los valores de las variables en la ejecución del programa paso a paso. Se muestran en negrita y cursiva los cambios entre paso y paso. Es importante analizar y entender los cambios de valor.

Tabla 6.2. Ejecución paso a paso de un ejemplo con punteros.

Paso	**a** 00FA:0000	**b** 00FA:0002	**c** 00FA:0004	**p1** 00FA:0006	**p2** 00FA:000A	**p** 00FA:000E
1				***00FA:0000***		
2	***1***			00FA:0000		
3	1			00FA:0000	***000FA:0002***	
4	1	***2***		00FA:0000	000FA:0002	
5	1	2		***000FA:0002***	000FA:0002	
6	1	***0***		000FA:0002	000FA:0002	
7	1	0		000FA:0002	***000FA:0004***	
8	1	0	***3***	000FA:0002	000FA:0004	
9	1	0	3	000FA:0002	000FA:0004	
10	1	0	3	000FA:0002	000FA:0004	***000FA:0006***
11	1	0	3	***000FA:0004***	000FA:0004	000FA:0006
12	1	0	***1***	000FA:0004	000FA:0004	000FA:0006

13	1	0	1	000FA:0004	000FA:0004	000FA:0006

Vectores, matrices y cadenas de caracteres

Un ***array*** (también conocido como *arreglo*, *vector* o *matriz*) *es un modo de manejar una gran cantidad de datos del mismo tipo bajo un mismo nombre o identificador*. Por ejemplo, mediante la sentencia:

```
double a[10];
```

se reserva espacio para 10 variables de tipo ***double***. Las 10 variables se llaman **a** y se accede a una u otra por medio de un ***subíndice***, que es una *expresión entera* escrita a continuación del nombre entre corchetes [...]. La forma general de la declaración de un vector es la siguiente:

```
tipo nombre[numero_elementos];
```

Los elementos se numeran desde 0 hasta (numero_elementos-1). El tamaño de un vector puede definirse con cualquier expresión constante entera. Para definir tamaños son particularmente útiles las *constantes simbólicas*. Como ya se ha dicho, para acceder a un elemento del vector basta incluir en una expresión su nombre seguido del *subíndice* entre corchetes. *En C no se puede operar con todo un vector o toda una matriz como una única entidad*, sino que hay que tratar sus elementos uno a uno por medio de bucles ***for*** o ***while***. Los vectores (mejor dicho, los elementos de un vector) se utilizan en las expresiones de C como cualquier otra variable. Ejemplos de uso de vectores son los siguientes:

```
a[5] = 0.8;
a[9] = 30. * a[5];
a[0] = 3. * a[9] - a[5]/a[9];
a[3] = (a[0] + a[9])/a[3];
```

Una ***cadena de caracteres*** no es sino un vector de tipo ***char***, con alguna particularidad que conviene resaltar. Las cadenas suelen contener texto (nombres, frases, etc.), y éste se almacena en la parte inicial de la cadena (a partir de la posición cero del vector). Para separar la parte que contiene texto de la parte no utilizada, se utiliza un *carácter fin de texto* que es el carácter nulo ('\0') según el código ASCII. Este carácter se introduce automáticamente al leer o inicializar las cadenas de caracteres, como en el siguiente ejemplo:

```
char ciudad[20] = "San Sebastián";
```

donde a los 13 caracteres del nombre de esta ciudad se añade un decimocuarto: el '\0'. El resto del espacio reservado –hasta la posición **ciudad[19]**– no se utiliza. De modo análogo, una cadena constante tal como "**mar**" ocupa 4 bytes (para las 3 letras y el '\0').

Las ***matrices*** se declaran de forma análoga, con corchetes independientes para cada subíndice. La forma general de la declaración es:

```
tipo nombre[numero_filas][numero_columnas];
```

donde tanto las *filas* como las *columnas* se numeran también a partir de 0. La forma de acceder a los elementos de la matriz es utilizando su nombre, seguido de las expresiones enteras correspondientes a los dos subíndices, entre corchetes.

En C tanto los vectores como las matrices admiten los *tipos* de las variables escalares (***char***, ***int***, ***long***, ***float***, ***double***, etc.), y los *modos de almacenamiento* ***auto***, ***extern*** y ***static***, con las mismas características que las variables normales (escalares). No se admite el modo ***register***. Los

arrays ***static*** y ***extern*** se inicializan a cero por defecto. Los arrays ***auto*** pueden no inicializarse: depende del compilador concreto que se esté utilizando.

Las matrices en C *se almacenan por filas*, en posiciones consecutivas de memoria. En cierta forma, una matriz se puede ver como un *vector de vectores-fila*. Si una matriz tiene N filas

(numeradas de 0 a N-1) y M columnas (numeradas de 0 a la M-1), el elemento (i, j) ocupa el lugar:

posición_elemento(0, 0) + i * M + j

A esta fórmula se le llama *fórmula de direccionamiento* de la matriz.

En C pueden definirse *arrays* con tantos subíndices como se desee. Por ejemplo, la sentencia,

```
double a[3][5][7];
```

declara una ***hipermatriz*** con tres subíndices, que podría verse como un conjunto de 3 matrices de dimensión (5x7). En la fórmula de direccionamiento correspondiente, el último subíndice es el que varía más rápidamente.

Como se verá más adelante, los *arrays* presentan una especial relación con los *punteros*. Puesto que los elementos de un vector y una matriz están almacenados consecutivamente en la memoria, la *aritmética de punteros* descrita previamente presenta muchas ventajas. Por ejemplo, supóngase el código siguiente:

```
int  vect[10], mat[3][5], *p;
p = &vect[0];
printf("%d\n", *(p+2));      // se imprimirá vect[2]
p = &mat[0][0];
printf("%d\n", *(p+2));      // se imprimirá mat[0][2]
printf("%d\n", *(p+4));      // se imprimirá mat[0][4]
printf("%d\n", *(p+5));      // se imprimirá mat[1][0]
printf("%d\n", *(p+12));     // se imprimirá mat[2][2]
```

RELACIÓN ENTRE VECTORES Y PUNTEROS

Existe una relación muy estrecha entre los vectores y los punteros. De hecho, el *nombre de un vector es un puntero* (un puntero constante, en el sentido de que no puede apuntar a otra variable distinta de aquélla a la que apunta) a la dirección de memoria que contiene el primer elemento del vector. Supónganse las siguientes declaraciones y sentencias:

```
double vect[10];  // vect es un puntero a vect[0]
double *p;
...
p = &vect[0];     // p = vect;
...
```

El identificador **vect**, es decir *el nombre del vector*, es un *puntero* al primer elemento del vector **vect[ ]**. Esto es lo mismo que decir que el valor de **vect** es **&vect[0]**. Existen más puntos de coincidencia entre los vectores y los punteros:

– Puesto que el nombre de un vector es un *puntero*, obedecerá las leyes de la aritmética de punteros. Por tanto, si **vect** apunta a **vect[0]**, (**vect+1**) apuntará a **vect[1]**, y (**vect+i**) apuntará a **vect[i]**.

– Recíprocamente (y esto resulta quizás más sorprendente), a los *punteros* se les pueden poner *subíndices*, igual que a los vectores. Así pues, si **p** apunta a **vect[0]** se puede escribir:

```
p[3]=p[2]*2.0;      // equivalente a vect[3]=vect[2]*2.0;
```

– Si se supone que **p=vect**, la relación entre *punteros* y *vectores* puede resumirse como se indica en las líneas siguientes:

```
*p     equivale a vect[0], a *vect     y a p[0]
*(p+1) equivale a vect[1], a *(vect+1) y a p[1]
*(p+2) equivale a vect[2], a *(vect+2) y a p[2]
```

Como ejemplo de la relación entre vectores y punteros, se van a ver varias formas posibles para sumar los N elementos de un vector **a[ ]**. Supóngase la siguiente declaración y las siguientes sentencias:

```
int a[N], suma, i, *p;

for(i=0, suma=0; i<N; ++i)           // forma 1
     suma += a[i];

for(i=0, suma=0; i<N; ++i)           // forma 2
     suma += *(a+i);

for(p=a, i=0, suma=0; i<N; ++i)      // forma 3
     suma += p[i];

for(p=a, suma=0; p<&a[N]; ++p)       // forma 4
     suma += *p;
```

RELACIÓN ENTRE MATRICES Y PUNTEROS

En el caso de las *matrices* la relación con los *punteros* es un poco más complicada. Supóngase una declaración como la siguiente

```
int mat[5][3], **p, *q;
```

El *nombre de la matriz* (**mat**) es un *puntero* al primer elemento de un *vector de punteros* **mat[ ]** (por tanto, *existe un vector de punteros que tiene también el mismo nombre que la matriz*), cuyos elementos contienen las direcciones del primer elemento de cada fila de la matriz. El nombre **mat** es pues un *puntero a puntero*. El *vector de punteros* **mat[ ]** se crea automáticamente al crearse la matriz. Así pues, **mat** es igual a **&mat[0]**; y **mat[0]** es **&mat[0][0]**. Análogamente, **mat[1]** es **&mat[1][0]**, **mat[2]** es **&mat[2][0]**, etc. La dirección base sobre la que se direccionan todos los elementos de la matriz no es **mat**, sino **&mat[0][0]**. Recuérdese también que, por la relación entre vectores y punteros, (**mat+i**) apunta a **mat[i]**. Recuérdese que la fórmula de direccionamiento de una matriz de N filas y M columnas establece que la dirección del elemento (i, j) viene dada por:

dirección (i, j) = dirección (0, 0) + i*M + j

Teniendo esto en cuenta y haciendo ****p = mat;** se tendrán las siguientes formas de acceder a los elementos de la matriz:

```
*p     es el valor de mat[0]       **p          es mat[0][0]
*(p+1) es el valor de mat[1]       **(p+1)      es mat[1][0]
                                   *(*(p+1)+1)  es mat[1][1]
```

Por otra parte, si la matriz tiene M columnas y si se hace **q = &mat[0][0]** (dirección base de la matriz. Recuérdese que esto es diferente del caso anterior **p = mat**), el elemento **mat[i][j]** puede ser accedido de varias formas. Basta recordar que dicho elemento tiene por delante **i** filas completas, y **j** elementos de su fila:

```
*(q + M*i + j)      // fórmula de direccionamiento
*(mat[i] + j)       // primer elemento fila i desplazado j elementos
(*(mat + i))[j]     // [j] equivale a sumar j a un puntero
*((*(mat + i)) + j)
```

Todas estas relaciones tienen una gran importancia, pues implican una correcta comprensión de los punteros y de las matrices. De todas formas, hay que indicar que las *matrices* no son del todo idénticas a los *vectores de punteros*: Si se define una matriz explícitamente por medio de vectores de punteros, las filas pueden tener diferente número de elementos, y no queda garantizado que estén contiguas en la memoria (aunque se puede hacer que sí lo sean). No sería pues posible en este caso utilizar la fórmula de direccionamiento y el acceder por columnas a los elementos de la matriz. La Figura 6.1 resume gráficamente la relación entre matrices y vectores de punteros.

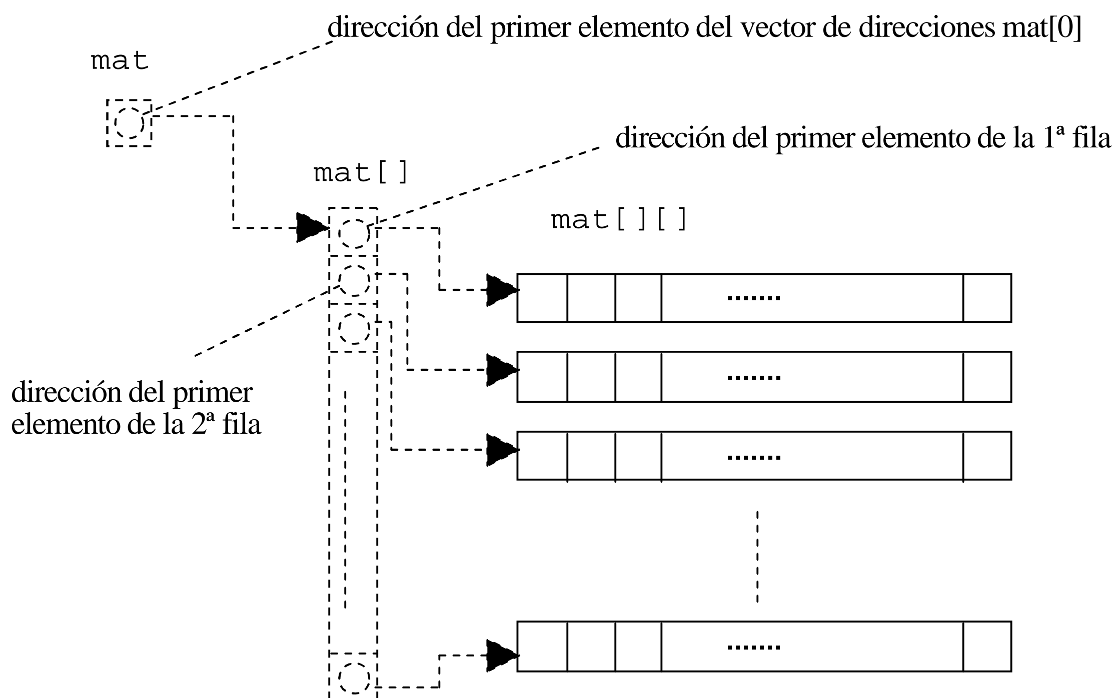


Figura 6.1. Relación entre matrices y punteros.

INICIALIZACIÓN DE VECTORES Y MATRICES

La inicialización de un *array* se puede hacer de varias maneras:

– Declarando el array como tal e inicializándolo luego mediante lectura o asignación por medio de un bucle ***for***:

```
double vect[N];
...
for(i = 0; i < N; i++)
     scanf(" %lf", &vect[i]);
...
```

– Inicializándolo en la misma declaración, en la forma:

```
double   v[6] = {1., 2., 3., 3., 2., 1.};
float     d[] = {1.2, 3.4, 5.1};           // d[3] está implícito
int    f[100] = {0};                       // todo se inicializa a 0
int     h[10] = {1, 2, 3};                 // restantes elementos a 0
int mat[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

donde es necesario poner un punto decimal tras cada cifra, para que ésta sea reconocida como un valor de tipo *float* o *double*.

Recuérdese que, al igual que las variables escalares correspondientes, los arrays con modo de almacenamiento *external* y *static* se inicializan a cero automáticamente en el momento de la declaración. Sin embargo, esto no está garantizado en los arrays *auto*, y el que se haga o no depende del compilador.

Estructuras

Una ***estructura*** es *una forma de agrupar un conjunto de datos de distinto tipo bajo un mismo nombre o identificador*. Por ejemplo, supóngase que se desea diseñar una *estructura* que guarde los datos correspondientes a un alumno de primero. Esta *estructura*, a la que se llamará ***alumno***, deberá guardar el nombre, la dirección, el número de matrícula, el teléfono, y las notas en las 10 asignaturas. Cada uno de estos datos se denomina ***miembro*** de la estructura. El *modelo* o *patrón* de esta estructura puede crearse del siguiente modo:

```
struct alumno {
     char nombre[31];
     char direccion[21];
     unsigned long no_matricula;
     unsigned long telefono;
     float notas[10];
};
```

El código anterior crea el *tipo* de dato **alumno**, pero aún no hay ninguna variable declarada con este nuevo tipo. Obsérvese la necesidad de incluir un carácter (;) después de cerrar las llaves. Para declarar dos variables de tipo **alumno** en C se debe utilizar la sentencia incluyendo las palabras ***struct*** y ***alumno*** (en C++ basta utilizar la palabra ***alumno***):

```
struct alumno alumno1, alumno2;     // esto es C
alumno alumno1, alumno2;            // esto es C++
```

donde tanto **alumno1** como **alumno2** son una estructura, que podrá almacenar un nombre de hasta 30 caracteres, una dirección de hasta 20 caracteres, el número de matrícula, el número de teléfono y las notas de las 10 asignaturas. También podrían haberse definido **alumno1** y **alumno2** al mismo tiempo que se definía la estructura de tipo **alumno**. Para ello bastaría haber hecho:

```
struct alumno {
     char nombre[31];
     char direccion[21];
     unsigned long no_matricula;
     unsigned long telefono;
     float notas[10];
} alumno1, alumno2;
```

Para acceder a los miembros de una estructura se utiliza el ***operador punto*** (.), precedido por el nombre de la *estructura* y seguido del nombre del *miembro*. Por ejemplo, para dar valor al **telefono** del alumno **alumno1** el valor 903456, se escribirá:

```
alumno1.telefono = 903456;
```

y para guardar la dirección de este mismo alumno, se escribirá:

```
alumno1.direccion = "C/ Penny Lane 1,2-A";
```

El tipo de estructura creado se puede utilizar para definir más variables o estructuras de tipo **alumno**, así como vectores de estructuras de este tipo. Por ejemplo:

```
struct alumno nuevo_alumno, clase[300];
```

En este caso, **nuevo_alumno** es una estructura de tipo **alumno**, y **clase[300]** es un *vector de estructuras* con espacio para almacenar los datos de 300 alumnos. El número de matrícula del alumno 264 podrá ser accedido como **clase[264].no_matricula**.

Los *miembros* de las estructuras pueden ser variables de cualquier tipo, incluyendo vectores y matrices, e incluso otras estructuras previamente definidas. Las estructuras se diferencian de los *arrays* (vectores y matrices) en varios aspectos. Por una parte, los *arrays* contienen información múltiple pero homogénea, mientras que los *miembros* de las estructuras pueden ser de naturaleza muy diferente. Además, *las estructuras permiten ciertas operaciones globales que no se pueden realizar con arrays*. Por ejemplo, la sentencia siguiente:

```
clase[298] = nuevo_alumno;
```

hace que se copien todos los miembros de la estructura **nuevo_alumno** en los miembros correspondientes de la estructura **clase[298]**. Estas operaciones globales no son posibles con *arrays*.

Se pueden definir también ***punteros a estructuras***:

```
struct alumno *pt;
pt = &nuevo_alumno;
```

Ahora, el puntero **pt** apunta a la estructura **nuevo_alumno** y esto permite una nueva forma de acceder a sus miembros utilizando el ***operador flecha*** (->), constituido por los signos (-) y (>). Así, para acceder al teléfono del alumno **nuevo_alumno**, se puede utilizar cualquiera de las siguientes sentencias:

```
pt->telefono;
(*pt).telefono;
```

donde *el paréntesis es necesario* por la mayor prioridad del operador (.) respecto a (*).

Las estructuras admiten los mismos modos *auto*, *extern* y *static* que los *arrays* y las variables escalares. Las reglas de inicialización a cero por defecto de los modos *extern* y *static* se mantienen. Por lo demás, una estructura puede inicializarse en el momento de la declaración de modo análogo a como se inicializan los vectores y matrices, por medio de valores encerrados entre llaves {}. Por ejemplo, una forma de declarar e inicializar a la vez la estructura **alumno_nuevo** podría ser la siguiente:

```
struct alumno {
     char nombre[31];
     char direccion[21];
     unsigned long no_matricula;
     unsigned long telefono;
     float notas[10];
} alumno_nuevo = {"Mike Smith", "San Martín 87, 2º A", 62419, 421794};
```

donde, como no se proporciona valor para las notas, éstas se inicializan a cero.

Las estructuras constituyen uno de los aspectos más potentes del lenguaje C. En esta sección se ha tratado sólo de hacer una breve presentación de sus posibilidades. C++ generaliza este concepto incluyendo *funciones miembro* además de *variables miembro*, llamándolo ***clase***, y convirtiéndolo en la base de la programación orientada a objetos.

Manejo de archivos en Lenguaje C

Veremos ahora la entrada y/o salida de datos utilizando ficheros, lo cual será imprescindible para un gran número de aplicaciones que deseemos desarrollar.

Ficheros

El estándar de C contiene funciones varias para la edición de ficheros, estas están definidas en la cabecera *stdio.h* y por lo general empiezan con la letra f, haciendo referencia a file. Adicionalmente se agrega un tipo **FILE**, el cual se usará como *apuntador a la información del fichero*. La secuencia que usaremos para realizar operaciones será la siguiente:

- Crear un apuntador del tipo **FILE ***
- Abrir el archivo utilizando la función **fopen** y asignándole el resultado de la llamada a nuestro apuntador.
- Hacer las diversas operaciones (lectura, escritura, etc).
- Cerrar el archivo utilizando la función **fclose**.

fopen

Ésta función sirve para abrir y crear ficheros en disco.

El prototipo correspondiente de **fopen** es:

```
FILE * fopen (const char *filename, const char *opentype);
```

Los parámetros de entrada de **fopen** son:

filename: una cadena que contiene un nombre de fichero válido. opentype: especifica en tipo de fichero que se abrirá o se creará.

Una lista de parámetros opentype para la función fopen son:

- "r" : abrir un archivo para lectura, el fichero debe existir.
- "w" : abrir un archivo para escritura, se crea si no existe o se sobreescribe si existe.
- "a" : abrir un archivo para escritura al final del contenido, si no existe se crea.
- "r+" : abrir un archivo para lectura y escritura, el fichero debe existir.
- "w+" : crear un archivo para lectura y escritura, se crea si no existe o se sobreescribe si existe.
- "a+" : abrir/crear un archivo para lectura y escritura al final del contenido

Adicionalmente hay tipos utilizando "b" (*binary*) los cuales no serán mostrados por ahora y que solo se usan en los sistemas operativos que no pertenecen a la familia de unix.

fclose

Esta función sirve para poder cerrar un fichero que se ha abierto.

El prototipo correspondiente de **fclose** es:

```
int fclose (FILE *stream);
```

Un valor de retorno cero indica que el fichero ha sido correctamente cerrado, si ha habido algún error, el valor de retorno es la constante EOF.

Un ejemplo pequeño para abrir y cerrar el archivo llamado fichero.in en modo lectura:

```
#include <stdio.h>

int main(int argc, char** argv)
{
        FILE *fp;
        fp = fopen ( "fichero.in", "r" );
        fclose ( fp );

        return 0;
}
```

Como vemos, en el ejemplo se utilizó el *opentype* "r", que es para la lectura.

Otra cosa importante es que el lenguaje C no tiene dentro de si una estructura para el manejo de excepciones o de errores, por eso es necesario comprobar que el archivo fue abierto con éxito "if (archivo == NULL)". Si **fopen** pudo abrir el archivo con éxito devuelve la referencia al archivo (FILE *), de lo contrario devuelve **NULL** y en este caso se debera revisar la direccion del archivo o los permisos del mismo. En estos ejemplos solo vamos a dar una salida con un retorno de 1 que sirve para señalar que el programa termino por un error.

feof

Esta función sirve para determinar si el cursor dentro del archivo encontró el final (**e**nd **of** **f**ile). Existe otra forma de verificar el final del archivo que es comparar el caracter que trae **fgetc** del archivo con el macro **EOF** declarado dentro de *stdio.h*, pero este método no ofrece la misma seguridad (en especial al tratar con los archivos "binarios"). La función **feof** siempre devolverá cero (Falso) si no es encontrado **EOF** en el archivo, de lo contrario regresará un valor distinto de cero (Verdadero).

El prototipo correspondiente de feof es:

```
int feof(FILE *fichero);
```

rewind

Literalmente significa "rebobinar", sitúa el cursor de lectura/escritura al principio del archivo.

El prototipo correspondiente de rewind es:

```
void rewind(FILE *fichero);
```

Lectura

Un archivo generalmente debe verse como un string (una cadena de caracteres) que esta guardado en el disco duro. Para trabajar con los archivos existen diferentes formas y diferentes funciones. Las funciones que podríamos usar para leer un archivo son:

- int fgetc(FILE *archivo)
- char *fgets(char *buffer, int tamano, FILE *archivo)
- size_t fread(void *puntero, size_t tamano, size_t cantidad, FILE *archivo);
- int fscanf(FILE *fichero, const char *formato, argumento, ...);

Las primeras dos de estas funciones son muy parecidas entre si. Pero la tercera, por el numero y el tipo de parámetros, nos podemos dar cuenta de que es muy diferente, por eso la trataremos aparte junto al **fwrite** que es su contraparte para escritura.

fgetc

Esta función lee un caracter a la vez del archivo que esta siendo señalado con el puntero ***archivo**. En caso de que la lectura sea exitosa devuelve el caracter leído y en caso de que no lo sea o de encontrar el final del archivo devuelve **EOF**.

El prototipo correspondiente de **fgetc** es:

```
        int fgetc(FILE *archivo);
```

Esta función se usa generalmente para recorrer archivos de texto. A manera de ejemplo vamos a suponer que tenemos un archivo de texto llamado "prueba.txt" en el mismo directorio en que se encuentra el fuente de nuestro programa. Un pequeño programa que lea ese archivo será:

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

int main()
{
        FILE *archivo;
        char caracter;

        archivo = fopen("prueba.txt","r");

        if (archivo == NULL)
                exit(1);

        printf("\nEl contenido del archivo de prueba es \n\n");

        while (feof(archivo) == 0)
        {
                caracter = fgetc(archivo);
                printf("%c",caracter);
        }

        return 0;
}
```

fgets

Esta función está diseñada para leer cadenas de caracteres. Leerá hasta n-1 caracteres o hasta que lea un retorno de línea. En este último caso, el carácter de retorno de línea también es leído.

El prototipo correspondiente de **fgets** es:

```
char *fgets(char *buffer, int tamano, FILE *archivo);
```

El primer parámetro buffer lo hemos llamado así porque es un puntero a un espacio de memoria del tipo char (podríamos usar un arreglo de char). El segundo parámetro es tamano que es el limite en cantidad de caracteres a leer para la función **fgets**. Y por ultimo el puntero del archivo por supuesto que es la forma en que **fgets** sabrá a que archivo debe escribir.

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

int main()
{
        FILE *archivo;

        char caracteres[100];

        archivo = fopen("prueba.txt","r");

        if (archivo == NULL)
                exit(1);

        printf("\nEl contenido del archivo de prueba es \n\n");
        while (feof(archivo) == 0)
        {
                fgets(caracteres,100,archivo);
                printf("%s",caracteres);
        }

        return 0;
}
```

Este es el mismo ejemplo de antes con la diferencia de que este hace uso de **fgets** en lugar de **fgetc**. La función **fgets** se comporta de la siguiente manera, leerá del archivo apuntado por archivo los caracteres que encuentre y a ponerlos en buffer hasta que lea un caracter menos que la cantidad de caracteres especificada en tamano o hasta que encuentre el final de una línea (\n) o hasta que encuentre el final del archivo (**EOF**).

El beneficio de esta función es que se puede obtener una línea completa a la vez. Y resulta muy útil para algunos fines como la construcción de un parser de algún tipo de *archivo de texto*.

fread

Para la lectura de ficheros se utilizará la función **fread**, la cual sirve para leer contenidos de un fichero.

El prototipo correspondiente de **fread** es:

```
size_t fread (void *data, size_t size, size_t count, FILE *stream);
```

En estas definiciones se usa el tipo **size_t**, el cuál está definido en *stddef.h* y sirve para definir tamaños de objetos. Lo que recibe esta función es un puntero donde almacenaremos los datos leídos (comúnmente llamado buffer), el tamaño de los datos a leer, la cantidad de esos datos a leer y el apuntador al fichero. Aquí hay un ejemplo simple para leer los primeros 100 caracteres de un fichero y almacenarlos en un buffer:

```
#include <stdio.h>

int main ( int argc, char **argv )
{
        FILE *fp;

        char buffer[100];

        fp = fopen ( "fichero.in", "r+" );

        fread ( buffer, sizeof ( char ), 100, fp );

        printf("%s", buffer);

        fclose ( fp );

        return 0;
}
```

fscanf

La función fscanf funciona igual que scanf en cuanto a parámetros, pero la entrada se toma de un fichero en lugar del teclado.

El prototipo correspondiente de **fscanf** es:

```
        int fscanf(FILE *fichero, const char *formato, argumento, ...);
```

Podemos ver un ejemplo de su uso, abrimos el documento "fichero.txt" en modo lectura y leyendo dentro de el.

```
#include <stdio.h>

int main ( int argc, char **argv )
{
        FILE *fp;

        char buffer[100];

        fp = fopen ( "fichero.txt", "r" );

        fscanf(fp, "%s" ,buffer);
        printf("%s", buffer);

        fclose ( fp );

        return 0;
}
```

Escritura

Así como podemos leer datos desde un fichero, también se pueden crear y escribir ficheros con la información que deseamos almacenar, Para trabajar con los archivos existen diferentes formas y diferentes funciones. Las funciones que podríamos usar para escribir dentro de un archivo son:

- int fputc(int caracter, FILE *archivo)
- int fputs(const char *buffer, FILE *archivo)
- size_t fwrite(void *puntero, size_t tamano, size_t cantidad, FILE *archivo);
- int fprintf(FILE *archivo, const char *formato, argumento, ...);

fputc

Esta función escribe un carácter a la vez del archivo que esta siendo señalado con el puntero ***archivo**. El valor de retorno es el carácter escrito, si la operación fue completada con éxito, en caso contrario será **EOF**.

El prototipo correspondiente de **fputc** es:

```
        int fput(int carácter, FILE *archivo);
```

Mostramos un ejemplo del uso de **fputc** en un "fichero.txt", se escribira dentro del fichero hasta que presionemos la tecla **enter**.

```
#include <stdio.h>

int main ( int argc, char **argv )
{
        FILE *fp;

        char caracter;

        fp = fopen ( "fichero.txt", "r+" );

        printf("\nIntrouce un texto al fichero: ");

        while((caracter = getchar()) != '\n')
        {
                printf("%c", fputc(caracter, fp));
        }

        fclose ( fp );

        return 0;
}
```

fputs

La función **fputs** escribe una cadena en un fichero. No se añade el carácter de retorno de línea ni el carácter nulo final. El valor de retorno es un *número no negativo* o **EOF** en caso de error. Los parámetros de entrada son la cadena a escribir y un puntero a la estructura **FILE** del fichero donde se realizará la escritura.

El prototipo correspondiente de **fputs** es:

```
        int fputs(const char *buffer, FILE *archivo)
```

para ver su funcionamiento mostramos el siguiente ejemplo:

```
#include <stdio.h>

int main ( int argc, char **argv )
{
        FILE *fp;

        char cadena[] = "Mostrando el uso de fputs en un fichero.\n";

        fp = fopen ( "fichero.txt", "r+" );

        fputs( cadena, fp );

        fclose ( fp );

        return 0;
}
```

fwrite

Esta función está pensada para trabajar con registros de longitud constante y forma pareja con **fread**. Es capaz de escribir hacia un fichero uno o varios registros de la misma longitud almacenados a partir de una dirección de memoria determinada. El valor de retorno es el número de registros escritos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura **FILE** del fichero del que se hará la lectura.

El prototipo correspondiente de **fwrite** es:

```
        size_t fwrite(void *puntero, size_t tamano, size_t cantidad, FILE
*archivo);
```

Un ejemplo concreto del uso de **fwrite** con su contraparte **fread** y usando *funciones* es:

```
/* FicheroCompleto.c */

#include <stdio.h>

void menu();
void CrearFichero(FILE *Fichero);
void InsertarDatos(FILE *Fichero);
void VerDatos(FILE *Fichero);

struct sRegistro {
   char Nombre[25];
   int Edad;
   float Sueldo;
} registro;

int main(int argc, char** argv)
{
        int opcion;
        int exit = 0;
        FILE *fichero;
```

```
        while (!exit)
        {
                menu();
                printf("\nOpcion: ");
                scanf("%d", &opcion);

                switch(opcion)
                {
                        case 1:
                                CrearFichero(fichero);
                        break;
                        case 2:
                                InsertarDatos(fichero);
                        break;
                        case 3:
                                VerDatos(fichero);
                        break;
                        case 4:
                                exit = 1;
                        break;
                        default:
                                printf("\nopcion no valida");
                }
        }

        return 0;
}

void menu()
{
        printf("\nMenu:");
        printf("\n\t1. Crear fichero");
        printf("\n\t2. Insertar datos");
        printf("\n\t3. Ver datos");
        printf("\n\t4. Salir");
}

void CrearFichero(FILE *Fichero)
{
        Fichero = fopen("fichero", "r");

        if(!Fichero)
        {
                Fichero = fopen("fichero", "w");
                printf("\nArchivo creado!");
        }
        else
        {
                printf("\nEl fichero ya existe!");
        }

        fclose (Fichero);

        return;
}
```

```
void InsertarDatos(FILE *Fichero)
{
        Fichero = fopen("fichero", "r+");

        if(Fichero == NULL)
        {
                printf("\nFichero no existe! \nPor favor creelo");
                return;
        }

        printf("\nDigita el nombre: ");
        scanf("%s", registro.Nombre);

        printf("\nDigita la edad: ");
        scanf("%d", &registro.Edad);

        printf("\nDigita el sueldo: ");
        scanf("%f", &registro.Sueldo);

        fwrite(&registro, sizeof(struct sRegistro), 1, Fichero);

        fclose(Fichero);

        return;
}

void VerDatos(FILE *Fichero)
{
        int numero = 1;

        Fichero = fopen("fichero", "r");

        if(Fichero == NULL)
        {
                printf("\nFichero no existe! \nPor favor creelo");
                return;
        }

        fread(&registro, sizeof(struct sRegistro), 1, Fichero);

        printf("\nNumero \tNombre \tEdad \tSueldo");

        while(!feof(Fichero))
        {
                printf("\n%d \t%s \t%d \t%.2f", numero, registro.Nombre,
                registro.Edad, registro.Sueldo);
                fread(&registro, sizeof(struct sRegistro), 1, Fichero);
                numero++;
        }

        fclose(Fichero);

        return;
}
```

fprintf

La función **fprintf** funciona igual que printf en cuanto a parámetros, pero la salida se dirige a un fichero en lugar de a la pantalla.

El prototipo correspondiente de **fprintf** es:

```
int fprintf(FILE *archivo, const char *formato, argumento, ...);
```

Podemos ver un ejemplo de su uso, abrimos el documento "fichero.txt" en modo lectura/escritura y escribimos dentro de el.

```
#include <stdio.h>
 int main ( int argc, char **argv )
{
        FILE *fp;

        char buffer[100] = "Esto es un texto dentro del fichero.";

        fp = fopen ( "fichero.txt", "r+" );

        fprintf(fp, buffer);
        fprintf(fp, "%s", "\nEsto es otro texto dentro del fichero.");

        fclose ( fp );

        return 0;
}
```