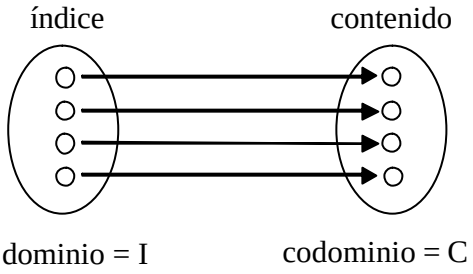
17. ARREGLOS. Array

17.1. Concepto.

El arreglo es un tipo estructurado de datos, y representa, en un ambiente de programación, a las entidades matemáticas denominadas vectores y matrices.

Sin lugar a dudas, es la estructura de datos más empleada por los programadores; y en algunos lenguajes de programación suele ser la única estructura explícita disponible.

Se entiende por estructura de datos, la acción de agrupar elementos primitivos en cierta forma. La forma más simple es agrupar componentes de igual tipo y asociarle un número de orden a cada componente; éste es el caso del arreglo.



En términos matemáticos abstractos la transformación (mapeo) puede anotarse:

$$A: I \dashrightarrow C$$

En Pascal puede anotarse, la definición del nuevo tipo A según:

```
type A = array [I] of C;
```

I se denomina tipo del índice, y debe ser un tipo ordinal.

C es el tipo del contenido, o de las componentes. También suele llamarse tipo base. Importa insistir en que todas las componentes deben ser de igual tipo.

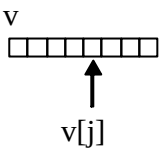
El tipo estructurado A queda completamente definido, si están previamente definidos los tipos I y C.

17.2. Notación.

Es preciso definir como denotar, o accesar, a una componente del arreglo.

Una visión esquemática útil del arreglo es la siguiente, asumiendo declarada una variable vector v de tipo A.

var v: A;



Es decir, visualizar un objeto de tipo A como un vector renglón. Para denotar la componente de orden j se emplea el nombre de la variable, y luego encerrado entre paréntesis cuadrados el índice j.

v es una variable de tipo A, y está formada por una colección (secuencia o arreglo) de componentes de tipo C. La secuencia está indexada por los elementos de I; es decir para cada elemento de I existe un valor asociado de la secuencia, de tipo C.

17.3. Ejemplos de Definición de tipo arreglo.

```
Type vector = array [1..10] of real;
     bitvector = array [1..32] of boolean;
     linea = array [1..80] of char;
     tabla = array [1..n] of integer;
```

17.4. Arreglos Bidimensionales. Matrices.

Si en el arreglo unidimensional o vector, definido antes, se escogen las componentes tal que éstas, a su vez, sean de tipo arreglo unidimensional se tendrá:

```
type  matriz = array [R] of array [C] of T;
```

Donde R es dominio de los renglones y C el dominio de las columnas. T es el tipo de las componentes o tipo base.

Con: var m : matriz;

Se tendrá que una componente puede anotarse:

m[i][j]

Para simplificar la notación se acepta:

Type matriz = array [R, C] of T;

Y puede accesarse a una componente según:

m[i,j]

Ejemplo: Dos matrices, m y n, de 100 por 100 de valores enteros puede declararse:

```
const  dimension = 100;
type   matriz = array [1..dimension,1..dimension] of integer;
var    m, n: matriz;
```

17.5. Arreglos Multidimensionales.

Pueden definirse según:

Type M = array [T1,T2,...TN] of T;

Donde: T1 a TN son los tipos de los índices.
T es el tipo base.

En general:

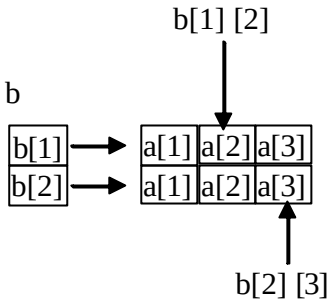
<tipoarreglo>::=

'array [' {<tipo ordinal>* ',' } '] of' <tipobase>

Ejemplo: Sean las declaraciones:

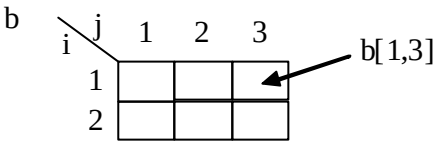
```
a: array [1..3] of T;
b: array [1..2] of a;
```

Se puede interpretar gráficamente las estructuras, según:



De esta forma pueden verse los arreglos multidimensionales como árboles.

Para el caso de matrices, puede representarse como es tradicional, según:



17.6. Arreglos estáticos.

Es importante destacar que los arreglos en Pascal son estructuras estáticas; es decir, el número de sus componentes queda determinado en el momento en que el programa es traducido a lenguaje de máquina. Esto está de acuerdo con la estrategia de diseño de Pascal, de que todos los tipos deben estar completamente especificados en el momento de la compilación.

Esto produce dificultades cuando se desea trabajar con arreglos de largo variable. En estos casos suele definirse un espacio igual al mayor largo que se desee manipular; con la consiguiente mala utilización de memoria para los casos en que el largo sea menor que el definido. Si el largo variable máximo no es conocido, no podrá usarse esta estructura; y quizá convendría emplear un lenguaje que admitiera arreglos dinámicos.

Por esta razón los valores máximos de los índices suelen definirse como constantes. Si se tiene un programa funcionando con una determinada dimensión para un arreglo, y si se desea modificar la dimensión, deberá modificarse la constante que establece la máxima dimensión, y volver a compilar.

17.7. Acceso Aleatorio.

En la representación interna, las diversas componentes del arreglo permanecen en memoria principal; por esta razón se dice que es una estructura de acceso aleatorio. Es decir, el tiempo necesario para accesar una componente cualquiera es el mismo; también el tiempo para asignar un valor a cualesquiera de las componentes.

17.8. Manipulación.

17.8.1.- Sintaxis de variable de arreglo.

En general la sintaxis de una componente de una variable de tipo arreglo es la siguiente:

<variable componente arreglo> ::=

<identificador> { '[' {<expresión>* ',' } ']' * }

La notación abreviada usa como índices una lista de expresiones, separadas por comas. El lazo exterior representa la notación extendida.

La variable componente de arreglo debe ser un identificador declarado como una variable de tipo arreglo.

Nótese que, en general, el índice puede ser una expresión. En el caso más simple de expresión puede usarse una variable; ésta se denomina variable índice, o más simplemente índice.

17.8.2.- Operaciones básicas.

a) Lectura.
Se dice que se efectúa una lectura, o acceso a la estructura, cuando una variable de tipo arreglo figura como factor en una expresión. La evaluación del factor se logra, previamente calculando el valor del índice, para luego accesar en la estructura el valor de la componente correspondiente.

b) Escritura.
Se dice que se efectúa una escritura, o asignación selectiva, si una variable de tipo arreglo aparece a la izquierda en una instrucción de asignación. En este caso la expresión de la derecha, debe tener igual tipo que la base del arreglo. También deben calcularse las

expresiones que figuran como índices de la variable, para asignarle posteriormente el valor ya calculado de la expresión a la derecha del símbolo de asignación.

Si m y n son variables arreglos de igual tipo puede efectuarse una asignación global:

```
m := n;
```

{n debe tener valores asignados en sus componentes}

17.8.3. Operaciones usuales.

17.8.3.1. Inicialización

Consiste en dar valores a una estructura, antes de realizar otros cálculos.

Ejemplos: Sean

```
a: array [1..20] of char;
b: array ['a'..'z'] of integer;
```

Nótese que se han empleado tipos anónimos.

a) Iniciar a con espacios. Corresponde a una línea en blanco.

```
....
i:=1;
repeat a[i] := ' '; i:=i+1 until i = 20;
....
```

Pero es preferible emplear la instrucción for:

```
for i:=1 to 20 do a[i]:=' ';
```

b) Iniciar b con ceros.

```
for ch:='a' to 'z' do b[ch]:=0;
```

17.8.3.2. Copia.

Dados dos arreglos de igual tipo, se desea copiar los valores del arreglo a en el arreglo b:

Esto puede escribirse:

```
for i:=1 to N do b[i]:=a[i];
```

Pero como se dijo antes, es más simple:

```
b := a;
```

17.9. Operaciones Especiales.

En los siguientes ejemplos se asume:

```
var a:array [1..N] of integer;
```

17.9.1. Sumar las componentes.

```
...
suma:=0;
for i:=1 to N do suma:=suma+a[i];
...
```

17.9.2. Búsqueda de valor extremo.

Encontrar el mínimo valor, y también el índice correspondiente:

Sea h el índice al mínimo.

```
....
min := a[1]; h := 1;   {se apunta, con el índice, al primero}
for i := 2 to N do      {se revisa del segundo hasta el final}
  begin
  if a[i] <min          {si se encuentra una menor}
  then
    begin
    h:=i; min:=a[h]    {se cambia índice al menor}
    end
  end
```

Puede alterarse levemente para encontrar el mayor.

Otro esquema de búsqueda del índice, que apunte al mayor es:

```
...    {se asume que el primero es el mayor}
j:=1;
for k:=2 to n do if a[k]>a[j] then j:=k;
      {j queda apuntando al mayor}
....
```

17.9.3. Ordenamiento de las componentes.

Existen numerosos algoritmos para ordenar. A continuación, y a modo de introducción al tema, se verán dos esquemas simples de ordenamiento de las componentes de un arreglo. Esta tarea ocurre frecuentemente y quizá por esta razón, en algunos países se denomina ordenadores a los computadores digitales.

17.9.3.1. Ordenamiento descendente buscando el mayor.

En un arreglo de largo n se busca el mayor y se lo coloca en la primera posición. Luego en un arreglo de largo n-1 se vuelve a buscar el mayor y se lo coloca en la segunda posición. Y así sucesivamente hasta ordenar todos los elementos. Cuando se obtiene el elemento ubicado en n-1; el último queda determinado automáticamente.

```
....
for i:=1 to n-1 do              {ubica los primeros n-1.}
  begin                         {la ubicación del i-avo}
  j:=i;                         {asume que j apunta al mayor}
  for k:=i+1 to n do            {recorre el subarreglo}
    if a[k]>a[j] then j:=k;     {dejando en j el mayor}
  b:=a[i]; a[i]:=a[j]; a[j]:=b {posiciona el i-avo}
  end;
....
```

Se emplea b para el intercambio de valores entre a[i] y a[j]. Nótese que este intercambio podría condicionarse a realizarse cuando i<>j.

El ordenamiento consiste en cambiar los contenidos del arreglo, tal que al final los valores de las componentes estén ordenadas (en este caso en forma descendente).

Pueden efectuarse, leves modificaciones, para el ordenamiento ascendente.

17.9.3.2. Ordenamiento ascendente buscando el mínimo.

Una variación del algoritmo para buscar el valor mínimo y su índice, nos permite plantear un algoritmo de ordenamiento ascendente.

```
...                {ordenamiento ascendente}
for i:=1 to n-1 do
  begin
  min:=a[i]; k:=i;
  for j:=i to n do
     begin
     if a[j]<min
     then
       begin k:=j; min:=a[k] end
     end;
  a[k]:=a[i]; a[i]:=min
  end
```

17.9.3.3. Ordenamiento burbuja (Bubblesort).

```
Program burbuja;
const n=10;
type  vector=array [1..n] of integer;
var
   i, j, k:integer;
   v:vector;
begin
{llena el arreglo; sin prompts para simplificar}
for i:=1 to n do read(v[i]);
for i:=n downto 2 do
  for j:=1 to i-1 do
    if v[j]>v[j+1] then
      begin    {intercambio}
      k:=v[j]; v[j]:=v[j+1]; v[j+1]:=k
      end;
{escribe arreglo ordenado}
for i:=1 to n do write (v[i])
end.
```

Se comparan dos elementos adyacentes, al menor se lo ubica con índice menor. Es decir, los valores menores van ascendiendo; de aquí viene el nombre de burbuja. Asumiendo que las burbujas de aire ascienden dentro de un líquido.

```
        |
        |
j ---> v[j]
       v[j+1]
        |
```

El for interno revisa las componentes, de a pares, desde el inicio hasta una posición final variable. Debe notarse que una primera revisión del vector entrega el mayor (más pesado) en la última posición. Por esta razón puede ir acortándose el largo de las futuras revisiones. Esto se logra con el for externo, que está implementado en forma regresiva.

17.9.4. Búsqueda (Search).

Otra tarea frecuente es buscar el índice de un elemento igual a un valor dado b.

Como no conocemos, por adelantado, cuántos elementos deben inspeccionarse, la repetición conviene plantearla en términos de while (o repeat) y no emplear for.

17.9.4.1. Búsqueda lineal o secuencial (Linear Search).

Con: var a: array [1..n] of T {n>0}

Consiste en revisar, en forma ordenada y partiendo desde un extremo, si el valor buscado coincide con alguno de los elementos del vector. Se usará la variable booleana estaba para indicar si el valor b está o no en el arreglo a.

```
....
i:=1;
while (i<n) and (a[i]<>b) do i:=i+1;
estaba:=a[i]=b;
.....
```

Se sale del lazo cuando (a[i]=b) o (i=n); es decir si se llegó al final o se encontró el valor. Nótese que la condición de término corresponde a la negación de la condición del while; ésta puede obtenerse aplicando la Ley de De Morgan.

Debe notarse que en el lazo no se revisa el último lugar, pero se incrementa i. Entonces al salir, sin haber encontrado el valor b en las primeras (n-1) componentes de a se revisa el último en la asignación de la variable estaba.

Una variante de la búsqueda lineal se logra con la instrucción repeat:

```
....
i:=0;
repeat
  i:=i+1
until (i=n) or (a[i]=b);
estaba:=a[i]=b;
....
```

Nota sobre **Expresiones booleanas.**

Una observación importante tiene relación con la evaluación de expresiones booleanas; ya que en algunos casos puede conocerse el valor antes de evaluar la expresión completa. Los casos más característicos son:

a) p and q
b) p or q

Si se conoce que p tiene valor false, no será necesario evaluar q, en el caso AND. También si se evalúa p y da valor verdadero, no será necesario seguir calculando el valor de q, ya que puede asegurarse que la expresión en b) será verdadera. Lamentablemente en Pascal estándar no se considera esta propiedad, que lenguajes de diseño más reciente incorporan (Por ejemplo: Ada, C y Módula). Específicamente, el programador debe asegurarse que el segundo factor quede bien definido.

Un ejemplo de esto es el siguiente:

```
.... {búsqueda lineal}
   i:=0;
   repeat i:=i+1 until (i>n) or (a[i]=b);
   estaba:=i<=n;
```

.....

La repetición termina si se encuentra el elemento b en el arreglo a, o si se llega al final. Sin embargo, en caso que b no esté en el arreglo, el segundo factor de la condición de término queda indefinido, ya que a[n+1] no pertenece al arreglo. (Report página 21). Problema que no se presenta en lenguajes que "cortocircuitan" las expresiones booleanas.

Una forma de simular el cortocircuito del OR, en Pascal, es la siguiente:

```
....
i:=0;
repeat
  i:=i+1;
  if i>n then c:=true else c:=a[i]=b
until c;
estaba:=i<=n;
....
```

Para el caso del cortocircuito del AND:

```
c := p AND q;
```

Puede escribirse:

```
if p then c:=q else c:=false;
```

Es decir, sólo se evalúa q si p es verdadero.

Volviendo al tema de la búsqueda lineal, es conveniente simplificar la expresión de término, ya que esto reducirá el tiempo del algoritmo, sobre todo si n es elevado. Esto debido, a que la evaluación es necesario efectuarla n veces, en el peor caso. Una forma de lograr la simplificación es la técnica del centinela que se expone a continuación.

17.9.4.2. Búsqueda lineal con centinela.

Una variante consiste en disponer el arreglo con una componente adicional, al final. Esta posición se denomina centinela, y en ella se almacena el valor buscado.

La búsqueda lineal con centinela, simplifica la evaluación de la condición, acelerando la búsqueda.

Ahora:

```
var  a : array [1..n+1] of integer;
     estaba : boolean;
....
a[n+1]:=b;    {se pone el centinela}
i:=1;
while a[i]<>b do i:=i+1;
estaba:=i<n+1;
....
```

Si al salir i=n+1, ningún elemento tiene el valor b.

Nótese que en la declaración del tipo del índice de a, debe reemplazarse n+1 con su valor. Ya que la declaración de cota, en tipo por subrango, debe ser una constante.

17.9.4.3. Búsqueda binaria.

En los casos anteriores de búsqueda, se asume que las componentes están en cualquier orden. En el peor de los casos deben realizarse n operaciones de comparación.

En ciertos casos interesa realizar búsquedas frecuentes en un arreglo en que no cambian sus valores. Por ejemplo: busca si una palabra está o no en una lista de palabras reservadas. En este ambiente conviene efectuar la búsqueda en un arreglo previamente ordenado. Puede comprobarse que el número de comparaciones se reduce a logaritmo en base 2 de n. Efectuando una comparación de esta función con n, se podrá ver que, para valores grandes de n, la reducción de tiempo es notable.

Si el arreglo está ordenado en forma ascendente, es decir:

a[i-1] <= a[i] para todo i=2..N

Se compara si el valor buscado está en la mitad superior o inferior. Si está en una de las mitades, se sigue subdividiendo en mitades, hasta encontrar o no el elemento buscado. En cada comparación se descartan la mitad de los elementos en que aún se busca.

La variable booleana estaba será verdadera si el elemento b está en el arreglo ordenado.

```
....          {búsqueda binaria }
i:=1;         {apunta a la cabeza}
```

```
j:=n;          {apunta a la cola  }

repeat
  k:=(i+j) div 2;                {busca la mitad}
  if a[k]<=b then i:=k+1;   {sigue en mitad inferior}
  if a[k]>=b then j:=k-1;    {sigue en mitad superior}
until i>j;

estaba:=a[k]=b;
```

Si estaba es true, el elemento buscado ocupa la posición k. Tiene valor a[k].

Es conveniente revisar el algoritmo, en su condición de término. También ponerse en el caso que el elemento buscado no esté en el vector, y analizar el comportamiento del algoritmo.

Esto agota las operaciones básicas con arreglos. A continuación se verán algunas aplicaciones típicas.

17.10. Uso de Arreglos.

17.10.1. Aplicaciones numéricas.

Algunas situaciones quedan espontáneamente descritas en términos de arreglos. Por ejemplo: vectores y matrices. También es la forma natural de describir funciones discretas.

Si una situación puede describirse con una variable con subíndice de tipo entero, también podrá modelarse, sin dificultad, en términos de arreglos.

Programas con número de entradas variable, pero acotado; y en el cual se necesite memorizar todos los valores antes de producir resultados, también pueden representarse en base a arreglos.

17.10.1.1. Estadísticas.

Calcular el promedio de n enteros, donde n es variable pero menor que N; no necesita un arreglo.

$$\text{Promedio} = (\sum_{i=1}^{i=n} Xi) / n$$

Pero sí, para calcular la desviación, empleando la siguiente fórmula:

$$\text{Desviacion} = \sqrt{\frac{\sum_{i=1}^{i=n} (Xi - \text{Promedio})^2}{n}}$$

Para calcular la desviación primero debe calcularse el promedio y luego se necesitan los valores de cada Xi.

Pero si se emplea la fórmula:

$$(\text{Desviacion})^2 * n = \sum_{i=1}^{i=n} Xi^2 - 2*\text{Promedio}*\sum_{i=1}^{i=n} \text{Xi} + n*\text{Promedio}^2$$

que puede desarrollarse expandiendo el cuadrado del binomio. Puede comprobarse que no es necesario almacenar los diferentes Xi; y por lo tanto no es necesario el arreglo.

17.10.1.2. Cálculo de polinomios.

Existen numerosas técnicas de aproximación, que permiten representar una función mediante un polinomio. La utilización de sub-índices para describir los coeficientes, permite escribir en general:

$$f = c(n)*x^n + c(n-1)*x^{n-1} + ... + c(1)*x + c(0)*1$$

Agrupando:

$$f = ((..((c(n)*x+c(n-1))*x+c(n-2)))*x+...+c(1))*x+c(0)$$

Se logra la regla de evaluación de Horner:

El polinomio puede calcularse según:

```
....
f:=0;
for i:=n downto 0 do f := f*x + c[i]
....
```

Con una expresión simple para el algoritmo, al elegir un vector de coeficientes del polinomio. Cambiando los coeficientes pueden obtenerse aproximaciones para diferentes funciones. El ejemplo muestra que, en ciertos casos, al elegir un arreglo se logran descripciones algorítmicas simples y compactas.

17.10.1.3. Aritmética de precisión múltiple.

Se desea manipular números en notación posicional, y en cualquier base (menor o igual a 10).

Esto permitirá manipular enteros mayores que el máximo representable.

Un número se representa por un vector, en el cual cada componente representa un dígito en la base dada.

El siguiente programa ilustra la suma y resta de enteros de largo n, dando la respuesta en un entero de largo n y una reserva (o préstamo) en base b.

```
var  u,v : array [1..n] of integer;  {operandos}
     r   : array [0..n] of integer;      {resultado}
     j,k,t:integer;

....
begin
k:=0;  {reserva}
for j:=n downto 1 do
  begin
  t:=u[j]+v[j]+k;      {-v[j] para la resta}
  r[j]:=t mod b;
  k:=t div b
  end;
r[0]:=k
end;
....
```

Nótese que en cada posición sólo se almacena un dígito, y que la reserva final sólo puede ser 0 ó 1. El algoritmo ilustra cómo se estructuran operaciones aritméticas en base a operaciones más primitivas.

A la innumerable cantidad de situaciones en que pueden aplicarse arreglos de contenidos numéricos, se agregan los arreglos de caracteres (strings); y también los arreglos de booleanos (que pueden representar conjuntos, además de aplicaciones digitales convencionales).

17.10.2. Procesamiento de textos usando arreglos.

La unidad básica de un texto, es una línea. La cual puede representarse por un arreglo de caracteres.

Lo anterior puede describirse:

```
Type linea=array [1..largo] of char;
```

Donde largo es una constante usada para especificar el máximo largo de una línea; en caso de textos suele ser 80.

Una vez definida una estructura de datos es conveniente diseñar procedimientos que la manipulen.

Veremos a continuación algunas posibilidades de leer y escribir, una variable de tipo línea, desde y hacia un archivo.

Asumiendo las siguientes variables globales:

```
var l:linea;
    x,y:text;   {x abierto para lectura; y para escritura}
    cl:integer; {caracteres gráficos en la línea}
```

El procedimiento traigalinea llena la variable l, y deja en cl el número de caracteres en la línea. Se tendrá que cl retorna con cero si la línea no tiene caracteres; y -1 si es fin del archivo x.

La comunicación entre el procedimiento y el programa se efectúa a través de las variables globales. Más adelante, se verán diseños más eficientes de intercambio a través de los parámetros del procedimiento.

```
procedure traigalinea;
var i:integer;     {local}
begin
i:=0;
```

```
if not eof(x)
then
  begin
  while not eoln(x) do
    begin
    i:=i+1;read(x,l[i])
    end;
  readln(x);
  cl:=i
  end
else
  cl:=-1
end;
```

Puede también desarrollarse procedimientos para leer y escribir variables de tipo línea desde y hacia los archivos estándares de entrada y salida; se dejan como ejercicio, ya que son leves variantes de los anteriores.

Nótese que no pueden leerse ni escribirse, a través de read y write variables de tipo línea; pero si es posible leer y escribir char, que es lo que se efectúa en los procedimientos.

Otro aspecto que debe notarse, es el empleo de contadores locales (la variable i). Su uso da mayor velocidad; ya que siempre es más fácil el acceso a variables locales que a las globales en un programa Pascal.

17.10.3. Strings. Arreglos empaquetados.

En Pascal estándar se definen strings como secuencias de más de un caracter, encerrados entre apóstrofes; y son constantes de tipo:

```
packed array [1..n] of char
```

Por extensión puede definirse:

```
Type string = packed array [1..n] of char;
```

La palabra packed (empaquetado) es una indicación para el compilador, para que reserve espacio guardando varios caracteres en una palabra de memoria. De esta forma la representación empaquetada, de arreglos de caracteres, ocupa menor lugar que la no empaquetada.

Para el programador, además de disponer de todas las características de un arreglo de caracteres, se tienen las siguientes ventajas:

a) El procedimiento write puede escribir expresiones de tipo packed array.

b) Puede efectuarse la asignación total de la estructura:

Ejemplo:

```
var s:packed array [1..5] of char;
  a:      array [1..5] of char;
```

Pueden escribirse:

```
s:='12345';
write(s); write('abcde')
```

c) Pero no son válidas:

```
a:='12345';  a:=s;              {tipos incompatibles}
write(a); read(s); read(a); {argumento ilegal}
s:='123';                       {tipo incompatible}
```

Un string debe leerse, carácter a carácter, ya que el procedimiento read no convierte strings de representación externa a la correspondiente interna.

El tratamiento de strings que efectúa el Pascal estándar ha sido criticado fuertemente; ya que dificulta el procesamiento de la información alfanumérica, muy común en el ambiente de procesamiento de datos. Por esta razón casi todos los compiladores Pascal existentes traen extensiones para la manipulación de strings. En ellas suele disponerse como tipo estándar al string; además pueden escribirse y leerse expresiones y variables de ese tipo; y se dispone de funciones para su manipulación. Otro rasgo destacable es cierta liberalidad en el largo; permitiendo asignaciones de cualquier largo, siempre que no se exceda un máximo.

El empleo de las extensiones debe planearse cuidadosamente, si se espera compilar un mismo programa en diversas instalaciones, o bajo compiladores diferentes en un mismo sistema.

Operaciones. Pack, Unpack.

La siguiente discusión puede omitirse, ser perder continuidad.

El acceso a una componente de un arreglo empaquetado toma más tiempo que el correspondiente a una componente no empaquetada. Para reducir el tiempo de acceso, el lenguaje estándar provee dos procedimientos para empaquetar o desempaquetar todas las componentes en una sola operación.

Ejemplo: La copia de valores de s en a se logra:

```
unpack(s,a,1)
```

La operación inversa, la copia de valores de la estructura no empaquetada a la empaquetada se logra con:

```
pack(a,1,s)
```

Donde a y s han sido definidos antes; y deben tener igual largo.

La secuencia:

```
s:='12345'; unpack(s,a,1)
```

Es equivalente a:

```
a[1]:='1';a[2]:='2';a[3]:='3';a[4]:='4';a[5]:='5'
```

Cuando se desea optimizar el almacenamiento de strings, ya sea en memoria principal o secundaria, las variables se manejan empaquetadas. Todos los accesos ya sea de lectura o escritura se realizan de esta forma; y si se desea accesar a los caracteres en forma individual, deberá desempaquetarse, efectuar las operaciones pertinentes; y volver a empaquetar si es necesario.

Una limitación del lenguaje estándar es que no permite pasar como parámetro una variable empaquetada. En algunas extensiones ésto es posible. Si se desea comunicar variables de este tipo es obligatorio desempaquetar la estructura, previo a la invocación del procedimiento.

En general si:

```
var u : array [a..d] of T;
    p : packed array [b..c] of T;
```

Con a<=b<=c<=d, todas de algún tipo ordinal, se tiene que:

pack(u,i,p); para a<=i<=b-c+d

es equivalente a:

for j:=b to c do p[j]:=u[j+i-b]

Y también que:

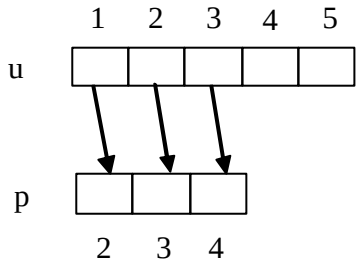
unpack(p,u,i); para a<=i<=b-c+d

es equivalente a:

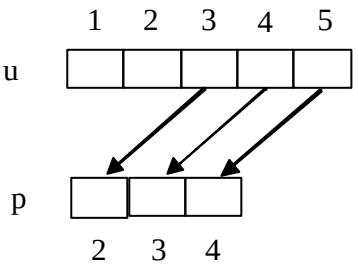
for j:=b to c do u[j+i-b]:=p[j]

Lo que se ilustra a continuación, en forma gráfica:

pack(u, 1, p) para 1<=i<=2-4+5=3 efectúa:



Y pack(u, 3, p):



Que muestran como puede extraerse parte (o la totalidad) de un arreglo no empaquetado u y llenar parte (o la totalidad) de uno empaquetado, y viceversa.

Comparaciones.

Otra ventaja de manipulación, es que los operadores de relación pueden aplicarse a operandos de tipo string de igual largo. El ordenamiento depende del código empleado para el conjunto de valores de tipo char.

```
Ejemplos:    'writeln'>'write  ';   {valor true}
             'aB'>'AB'         ;   {valor true}
             ' 1'>'1 '         ;   {valor false}
```

En un arreglo de caracteres las comparaciones deben efectuarse para cada una de los caracteres correspondientes. Si a y b son arreglos de caracteres, la relación de orden a<b puede implementarse averiguando si existe un valor de índice k tal que:

a[k] < b[k] y
a[i] = b[i] para todo i<k

Evidentemente, la operación anterior puede desarrollarse como un procedimiento.

17.10.4. Arreglos de booleanos.

Como se verá un arreglo de booleanos posee iguales propiedades que la estructura conjunto (set). En las implementaciones de Pascal, el número máximo de elementos de un conjunto está limitado; cuando no se puedan emplear conjuntos, resultará conveniente la representación mediante arreglos de booleanos.

En estas situaciones cada componente del arreglo está asociada a un elemento del conjunto. Para incluir un elemento basta hacer true la componente correspondiente.

Ejemplo: var conjunto:array [1..100] of boolean;

Las asignaciones:

```
conjunto[i]:=true;  {incorpora elemento i al conjunto}
conjunto[j]:=false; {extrae el elemento j del conjunto}
```

La condición:

conjunto[k]

permite probar si el elemento k pertenece o no al conjunto.

La estructura arreglo de booleanos permite representar la arquitectura de máquinas digitales. Por ejemplo:

Una visualización de la memoria de un computador puede efectuarse con la siguiente estructura de datos.

```
const largopalabra = 16;
      capacidad = 65536;  { 2^(largopalabra) }
type  palabra: array [1..largopalabra] of boolean;
      memoria: array [1..capacidad] of palabra;
```

17.10.5. Ejercicios.

17.10.5.1. Dado un arreglo "a" de "n" enteros, determinar si existen elementos iguales.

Se dan cuatro posibles soluciones, determinar si son correctas, y cuál es más eficiente.

En caso de existir error, corregirlo.

Solución a)

```
hayiguales:=false;
for i:=1 to n-1 do
    for j:=i+1 to n do
        hayiguales:=a[i] = a[j];
```

Solución b)

```
hayiguales:=false;
i:=1;
repeat
    j:=1;
    repeat
        j:=j+1;
        hayiguales:=a[i]=a[j]
    until (j=n) or hayiguales;
    i:=i+1
until (i=n) or hayiguales;
```

Solución c)

```
for i:=1 to n-1 do
for j:=i+1 to n do
    if a[i]=a[j]
    then begin
        hayiguales:=true;
        goto 1
    end;
hayiguales:=false;
1:   ...
```

Solución d)

```
conjunto:=[];
hayiguales:=false;
i:=1;
while (i<n+1) and (not hayiguales) do
begin
    if a[i] in conjunto
        then hayiguales:=true
        else conjunto:=conjunto+a[i];
    i:=i+1
end;
```

17.10.5.2. Arreglo de enteros diferentes.

Se desea almacenar nmax valores enteros diferentes. Los números enteros se digitan en el teclado, pueden entrarse números iguales, pero sólo debe aceptarse la primera ocurrencia.

Diseño.

La idea es almacenar nmax enteros, lo cual requiere nmax variables enteras. Será engorroso definir y tratar los nombres de las variables, salvo cuando nmax sea pequeño.

Consideremos, el siguiente espacio:

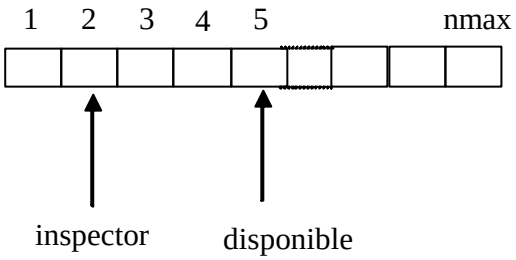
```
var  almacen : array[1..nmax] of integer
```

La tarea consiste en llenar el almacén con valores diferentes, lo cual podría refinarse según :

while el almacén no esté lleno do
lea número entero;
if el numero no está en el almacén
then ponga el número en el almacén.

Debe notarse que la búsqueda en el almacén requiere inspeccionar los valores ya almacenados, para esto usaremos una variable llamada inspector. Además la inspección debe realizarse desde el inicio del almacén hasta la última posición en que se ha cargado un número; esta posición también es variable (su valor cambia a medida que se ingresan nuevos valores). La última posición también queda definida por el próximo hueco en el arreglo; a esta posición la llamaremos disponible.
Gráficamente:

almacén



Los programadores suelen llamar punteros a las variables inspector y disponible, ya que permiten apuntar a componentes específicas. En Pascal se emplea el concepto de puntero para referirse a variables que apuntan a variables dinámicas. Como se sabe el arreglo es una estructura estática, y es preferible denominar cursores a las variables anteriores.

Reflexionando sobre la situación, puede que la acción consiste en repetir un número determinado de veces la introducción de un número no presente en las componentes anteriores, lo cual sugiere que el primer verbo sea un for. Además puede emplearse la posición disponible para efectuar una búsqueda con centinela.

Entonces, escribiendo de nuevo la idea básica :

for disponible:=1 to nmax do
repetir
poner centinela;
busqueda lineal
hasta que la busqueda falle.

Poner el centinela, puede traducirse :

```
read(numero); almacen[disponible]:=numero;
```

Y la búsqueda lineal por:

```
inspector:=1;
while almacen[inspector] <> numero do
  inspector:=inspector+1;
```

La condición de que el número no esté incluido en las componentes anteriores, es que la búsqueda falle; es decir :

```
inspector = disponible
```

Implementación.

```
program llenaralmacen(input,output);
    const   nmax=10;
    var
      numero : integer;
      almacen : array[1..nmax] of integer;
      inspector,disponible : 1..nmax;
    begin
    for disponible:=1 to nmax do
      repeat
        read(numero); almacen[disponible]:=numero;
        inspector:=1;
        while almacen[inspector] <> numero do
            inspector:=inspector+1
      until inspector=disponible;
    for inspector:=1 to nmax do writeln(almacen[inspector])
    end.
```

Nótese que los cursores se han definido por subrango, para asegurarse que no se intentarán accesos a posiciones inexistentes del arreglo. Esta debería ser una práctica usual.

Resta agregar mensajes y comentarios.

Modificar el algoritmo para que almacene un número sólo si éste es mayor que todos los números ya almacenados.

17.10.5.3. Palíndromos y cuadrados.

Se desea encontrar números cuyos cuadrados sean palíndromos.

Un palíndromo es una palabra (o frase) que puede leerse de derecha a izquierda o al revés, con igual significado que al leerla de izquierda a derecha. Un ejemplo es: radar. En el caso de números: 1, 121, 232 son números palíndromos; además 1 y 121 son cuadrados.

En forma más precisa de desea generar una lista de enteros, entre 1 y N, cuyos cuadrados sean palíndromos.
Solución.
La primera versión es:

```
..
n:=0;
repeat
  n:=n+1;
  generar cuadrado de n;
  if el cuadrado es palíndromo
  then writeln(n)
until n=N;
..
```

Es necesario disponer un espacio para almacenar las cifras de un número. Emplearemos el arreglo b.

```
var     b : array [1..l] of integer;
```

El largo de b, debe permitir representar el número de cifras necesarias para escribir sqr(N).

En este espacio podemos representar las cifras decimales del cuadrado, usando notación decimal posicional, mediante:

$$n*n = \sum_{j=1}^{j=l} b[j]*10^{j-1}$$

Donde L es el número de cifras de n*n; debe ser:

L <= l

Un número es palíndromo si:

b[j] = b[L-(j-1)] para todo j tal que :

1 <= j < (L+1) div 2

Donde (L+1) div 2 es el índice al elemento central del arreglo b, que contiene las cifras decimales.

La generación de las cifras puede lograrse :

```
....{generación de cifras del cuadrado de n}
cuad:=n*n; L:=0;
repeat
  L:=L+1;
  b[L]:=cuad mod 10;
  cuad:=cuad div 10
until cuad=0;
....
```

La condición para ver si es palíndromo, puede obtenerse en una variable booleana espal.

Entonces :

```
....{test palíndromo}
j:=1; k:=L;
repeat
  espal:=b[j]=b[k];
  j:=j+1; k:=k-1
until (j>=k) or (not espal);
....
```

17.10.5.4. Potencias de dos. (Uso arreglos. Precisión ilimitada)

a) Potencias positivas

Se desea obtener : p = 2^i

El mayor valor exacto de p está limitado por el máximo entero representable; éste suele ser, precisamente, una potencia de dos. Si se emplean 16 bits para un entero, el máximo p representable será 32.768; para un exponente 15. Debe recordarse que un bit se emplea para el signo, en la representación interna.

Para representar, en sistema decimal, un número p, se requiere un número de cifras dado por :

cifras = trunc((log p) + 1)

El logaritmo es en base 10.

Si p es una potencia de dos, el máximo exponente N produce un número que requiere :

cifras = trunc(log(2^N)+1) = trunc(N*log2+1)

Ejemplo. Si el máximo exponente es 32, se necesitan:

cifras = trunc(32*0,30103+1) = 10

Para producir potencias de dos con tantos dígitos como se desee, puede emplearse un arreglo de dígitos para representar la potencia. El tipo del arreglo puede ser entero y su dimensión debe ser tal que permita representar las cifras necesarias; esto dependerá del máximo exponente de la potencia que se desee generar.

Entonces:

```
var  p: array[0..cifras] of integer;
```

Un número de k cifras se representa por la secuencia:

p[k-1] p[k-2] ... p[0]

El siguiente segmento escribe un número como una secuencia de caracteres, justificado por la derecha en un ancho dado por la constante cifras:

```
... {escriba}
i:=cifras;
```

```
repeat i:=i-1; write(' ') until i=k;
repeat i:=i-1; write(chr(p[i]+ord('0'))) until i=0;
...
```

Nótese que cada p[i] debe mantenerse entre 0 y 9.

La parte esencial del algoritmo consiste en obtener la siguiente potencia de dos, a partir de la anterior. Si se asume que la potencia actual ocupa k cifras, puede plantearse :

```
... {multiplica}
for i:=0 to k-1 do
 begin
   t:=2*p[i]+reserva;
   if t>=10
   then
     begin p[i]:=t-10; reserva:=1 end
   else
     begin  p[i]:=t;   reserva:=0 end
 end;
if reserva>0
 then begin p[k]:=1; k:=k+1 end;
...
```

Se multiplican todas las cifras por dos, manteniendo éstas entre 0 y 9. Para esto se emplea la variable reserva. Si existe reserva final, el número crece en una cifra.

Las siguientes líneas generan una tabla de potencias de dos, hasta lograr la potencia de exponente N.

```
...
p[0]:=1; k:=1;
for j:=1 to N do
 begin
   reserva:=0;
   multiplica;
   escriba;
   writeln(j:6)
 end;
...
```

b) Potencias negativas.

En caso de potencias negativas, éstas tienen parte entera igual a cero, y un número de cifras decimales igual al módulo del exponente. Además, la última cifra decimal siempre es 5.

Sea:

```
var  p : array[1..cifras] of integer;
```

Para k cifras decimales, puede generarse la siguiente, mediante:

```
... {divide}
resto:=0;
for i:=1 to k-1 do
  begin
    resto:=10*resto+p[i]; p[i]:=resto div 2;
    resto:=resto mod 2
  end;
p[k]:=5;
...
```

Y la escritura, se logra:

```
... {escriba}
write(k:6); write('   0,');
for i:=1 to k do write(chr(p[i]+ord('0')));
writeln;
...
```

Para generar una tabla:

```
for k:=1 to N do
  begin  divide; escriba  end;
```

17.10.5.5. Ejercicio.

Desarrollar un programa que inicie un arreglo de char con los caracteres gráficos del código ascii en orden descendente. Luego debe escribirse el arreglo generado en la pantalla. Después hay que ordenar los caracteres en forma ascendente. Finalmente debe volver a escribirse el arreglo ordenado, en la pantalla.