Introducción a la Programación usando Pascal como primer lenguaje

Capítulo 10
Estructuras dinámicas

Francisco J. Ballesteros

Estructuras dinámicas

10.1. Tipos de memoria

En general, las variables (y constantes) son de uno de estos dos tipos:

1. **Variables estáticas o globales**. Estas viven en la memoria durante todo el tiempo que dura la ejecución del programa. En este curso no se permite el uso de variables globales. Estas variables son peligrosas, ya que son accesibles desde cualquier subprograma, y no son recomendables para aprender a programar. Las constantes que hemos estado usando en casi todos nuestros programas son globales.
2. **Variables automáticas o locales**. Estas viven en la pila de memoria que se utiliza para controlar las llamadas a procedimiento. Cuando se produce la invocación de un procedimiento se añade a la cima de la pila espacio para las variables locales del procedimiento al que se invoca (además de para los parámetros y para otras cosas). Cuando la llamada a procedimiento termina se libera ese espacio de la cima de la pila. Dicho de otro modo, estas variables se crean y destruyen automáticamente cuando se invocan procedimientos y cuando las invocaciones terminan.

Disponer sólo de estos dos tipos de almacenamiento o variables hace que resulte necesario saber qué tamaño van a tener los objetos que va a manipular el programa antes de ejecutarlo. Por ejemplo, si vamos a manipular palabras, tenemos que fijar un límite para la palabra más larga que queramos manipular y utilizar un vector de caracteres de ese tamaño para todas y cada una de las palabras que use nuestro programa. Esto tiene el problema de que, por un lado, no podemos manejar palabras que superen ese límite y, por otro, gastamos memoria del ordenador de forma innecesaria para palabras mas pequeñas.

Además, por el momento, resulta imposible crear variables durante la ejecución del programa de tal forma que sobrevivan a llamadas a procedimiento.

Hay una solución para estos problemas. Para poder manipular **objetos de tamaño variable** y para poder **crear objetos nuevos** disponemos en todos los lenguajes de programación de un tercer tipo de variable: **variables dinámicas**.

Estas variables pueden crearse en tiempo de ejecución, mediante una sentencia, cuando es preciso y pueden destruirse cuando dejan de ser necesarias. A la primera operación se la suele denominar **asignación de memoria dinámica** y la segunda **liberación de memoria dinámica**.

10.2. Variables dinámicas

Para nosotros, el nombre de una variable es lo mismo que la memoria utilizada por

dicha variable. Una variable es un nombre para un valor guardado en la memoria del ordenador. Esto está bien para variables locales (y globales). Las variables dinámicas funcionan de otro modo: **las variables dinámicas no tienen nombre**.

Una variable dinámica es un nuevo trozo de memoria que podemos pedir durante la ejecución del programa (por eso se denomina "dinámica"; una variable estática se denomina así puesto que su gestión se puede decidir en tiempo de compilación). Este trozo de la memoria que pedimos en tiempo de ejecución no corresponde a la declaración de ninguna variable de las que tenemos en el programa. Del mismo modo que podemos pedir un nuevo trozo de memoria, también podemos devolverlo (allí a dónde lo hemos pedido) cuando deje de ser útil.

Consideremos un ejemplo de uso. Podemos declarar una variable que sea una cadena de caracteres (un *array* de **`char`**) de cierto tamaño. Cuando el programa ejecute, ese *array* siempre tendrá ese tamaño. Da igual cómo ejecute el programa, la memoria usada por el *array* siempre es la misma. Con variables dinámicas, la idea es que durante la ejecución del programa podemos ver cuánta memoria necesitamos para almacenar nuestro *array*, y pedir justo esa nueva cantidad de memoria para un nuevo *array*.

Cuando el *array* deja de sernos útil podemos devolver al sistema la memoria que hemos pedido, para que pueda utilizarse para otros propósitos. Piensa que en el primer caso (estático) el *array* siempre tiene el mismo tamaño; en el segundo caso (dinámico) tendrá uno u otro en función de cuánta memoria pidamos en **tiempo de ejecución** (mientras que el programa ejecuta). Piensa también que en el primer caso el *array* existe durante toda la vida del procedimiento donde está declarado; en el segundo sólo existe cuando pidamos su memoria, y podría sobrevivir a cuando el procedimiento que lo ha creado termine.

Usar variables dinámicas es fácil si se entienden bien los cuatro párrafos anteriores. Te sugerimos que los vuelvas a leer después, cuando veamos algunos ejemplos.

10.3. Punteros

Las variables dinámicas se utilizan mediante tipos de datos que son capaces de actuar como "flechas'' o "apuntadores'' y señalar o apuntar a cualquier posición en la memoria. A estos tipos se los denomina **punteros**. Un puntero es tan sólo una variable que guarda una dirección de memoria. Aunque esto es sencillo, suele resultar confuso la primera vez que se ve, y requiere algo de esfuerzo hacerse con ello.

Si tenemos un tipo de datos podemos construir un nuevo tipo que sea un puntero a dicho tipo de datos. Por ejemplo, si tenemos el tipo de datos **`integer`**, podemos declarar un tipo de datos que sea "puntero a *integer*''. Un puntero a **`integer`** podrá apuntar a cualquier zona de la memoria en la que tengamos un **`integer`**. La idea entonces es que, si en tiempo de ejecución queremos un **nuevo** entero, podemos pedir

nueva memoria para un **`integer`** y utilizar un puntero a **`integer`** para referirnos a ella.

Con un ejemplo se verá todo mas claro. En Pascal se declaran tipos de datos puntero usando el símbolo "`^`" antes del tipo apuntado. Se usa dicho símbolo porque recuerda a una flecha. Por ejemplo, esto declara un nuevo tipo de datos para representar punteros a enteros:

```
types:
     TipoPtrEntero = ^integer;
```

Igualmente, esto declara un tipo de datos para punteros a datos de tipo **`TipoPalabra`**:

```
TipoPtrPalabra = ^TipoPalabra;
```

Las variables de tipo **`TipoPtrEntero`** pueden o bien **apuntar** a una variable de tipo **`int`** o bien tener el valor especial **`nil`**. Este es un valor nulo que representa que el puntero no apunta a ningún sitio. También se le suele llamar (en otros lenguajes) *null*. Dicho valor es compatible con cualquier tipo que sea un puntero. Normalmente, corresponde a la dirección de memoria "**`0`**" que suele ser inválida.

Si tenemos un tipo para punteros a entero, **`TipoPtrEntero`**, podemos declarar una variable de este tipo como de costumbre:

```
var
    pentero: TipoPtrEntero;
```

Esto tiene como efecto lo que podemos ver en la figura 1.

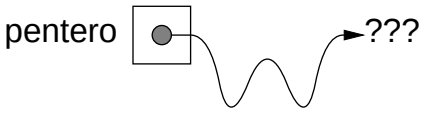

Fig. 1. Puntero recién declarado. No sabemos hacia dónde apunta.

Tenemos una variable de tipo puntero a entero, esto es, capaz de referirse o apuntar a enteros, pero no le hemos dado un valor inicial y no sabemos hacia donde apunta (qué dirección de memoria contiene la variable). Fíjate bien en que *no tenemos ningún entero*. Tenemos algo capaz de referirse a enteros, pero no tenemos entero alguno.

Antes de hacer nada podemos inicializar dicha variable asignando el literal **`nil`**, que hace que el puntero no apunte a ningún sitio.

```
pentero := nil;
```

El efecto de esto en el puntero lo podemos representar como muestra la figura 2 (utilizamos una simbología similar a poner una línea eléctrica a tierra para indicar que el

puntero no apunta a ningún sitio, tal y como se hace habitualmente).

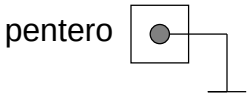

Fig. 2. Ejemplo de puntero a nil. No apunta a ningún sitio.

Pues bien, ¡Llegó el momento! Por primera vez vamos a crear nosotros una nueva variable (sin nombre alguno, eso sí). Hasta el momento siempre lo había hecho Pascal por nosotros. Pero ahora, teniendo un puntero a entero, podemos crear un nuevo entero y utilizar el puntero para referirnos a el. Esta sentencia

```
new(pentero);
```

crea una nueva variable de tipo **`integer`** (sin nombre alguno) y hace que su dirección se le asigne a **`pentero`**. El efecto es el que muestra la figura 3.

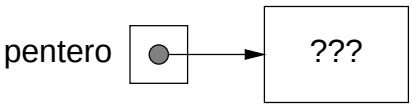

Fig. 3. Un puntero apuntando a un nuevo entero

Ahora el puntero apunta a un nuevo entero. Eso sí, como no hemos inicializado el entero no sabemos qué valor tiene el entero. Puede tener cualquier valor.

Nótese que *tenemos dos variables* en este momento. Una con nombre, el puntero, y una sin nombre, el entero. En realidad del puntero sólo nos interesa que nos sirve para utilizar el entero. Pero una cosa es el entero y otra muy distinta es el puntero.

En Pascal, para referirnos a la variable a la que apunta un puntero tenemos que añadir "^" tras el nombre del puntero. Esta sentencia inicializa el entero para que su valor sea 4:

```
pentero^ := 4;
```

El resultado puede verse en la figura 4. Nótese que esta asignación ha cambiado el entero. El puntero sigue teniendo el mismo valor, esto es, la dirección del entero al que apunta.

Al acto de referirse al elemento al que apunta un puntero se le denomina

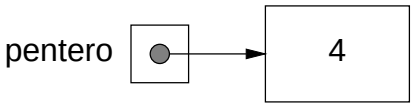

Fig. 4. Un puntero que apunta a nuestro entero ya inicializado.

vulgarmente *atravesar el puntero* y más técnicamente **aplicar una indirección** al puntero. De hecho, se suele decir que el puntero es en realidad una indirección para indicar que no es directamente el dato que nos interesa (y que hay que efectuar una indirección para obtenerlo).

Es un error atravesar un puntero que no está apuntando a un elemento. Por ejemplo, utilizar **`pentero^`** cuando el valor de **`pentero`** es **`nil`** provoca la detención del programa con una indicación de error.

Utilizar un puntero cuando no está inicializado es aún peor: no sabemos dónde apunta el puntero y podemos estar accediendo (¡al azar!) a cualquier parte de la memoria. El resultado es muy parecido a un poltergueist. Tienes que ser muy cuidadoso, estos errores son los más difíciles de arreglar.

Respecto al estilo, los nombres de tipos puntero deberían ser siempre identificadores que comiencen por **`TipoPtr`** para indicar que se trata de un tipo de puntero. Igualmente, los nombres de variables de tipo puntero deberían dejar claro que son un puntero; por ejemplo, pueden ser siempre identificadores cuyo nombre comience por **`p`**. Esto permite ver rápidamente cuándo tenemos un puntero entre manos y cuándo no, lo que evita muchos errores y sufrimiento innecesario.

Vamos a ver como ejemplo algunas declaraciones y sentencias que manipulan varios punteros. Tras cada una mostramos el resultado de su ejecución. Empecemos por declarar dos punteros:

```
var
    pentero1: ^integer; { es lo mismo que TipoPtrEntero }
    pentero2: TipoPtrEntero;
```

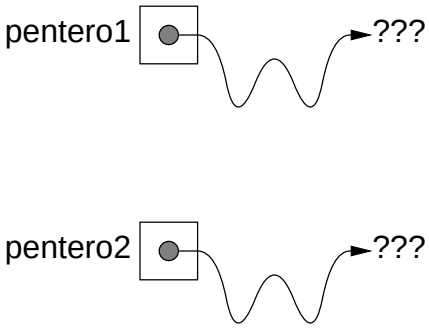

Fig. 5. Tras declarar dos punteros.

Y ahora ejecutemos algunas sentencias...

```
pentero1 := nil;
```

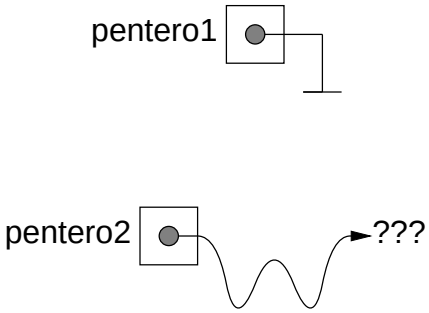

Fig. 6. puesto a nil.

```
pentero2 := pentero1;
```

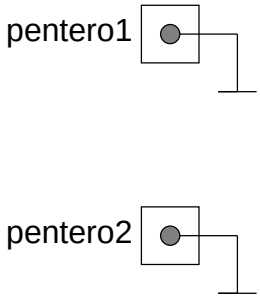

Fig. 7. Ya asignado

```
new(pentero1);
pentero1^ := 5;
```

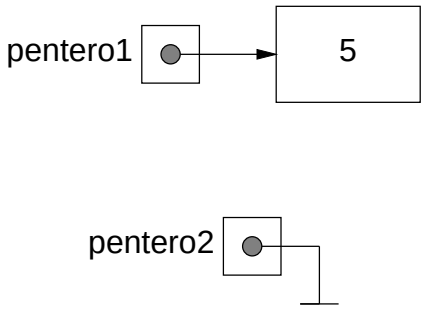

Fig. 8. Creado un entero.

```
pentero2 := pentero1;
```

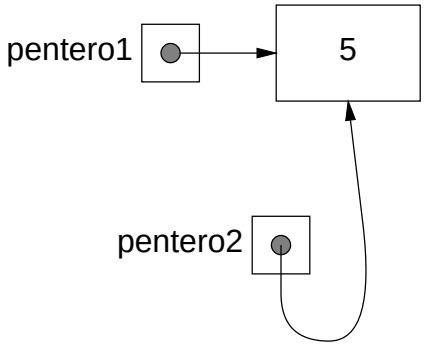

Fig. 9. Puntero asignado otra vez.

```
pentero2^ := 7;
```

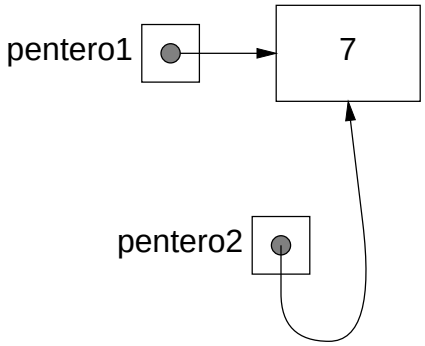

Fig. 10. Entero asignado.

Si en este punto ejecutamos

```
write(pentero1^);
```

veríamos que el valor de **pentero1^** es 7. Habrás visto que esta sentencia

```
pentero1^ := pentero2^;
```

copia el valor al que apunta **puntero2** a la variable a la que apunta **puntero1**, pero no modifica ningún puntero. En cambio

```
pentero1 := pentero2;
```

modifica **pentero1** y lo hace apuntar allí donde apunte **pentero2**.

La clave para entender cómo utilizar los punteros es distinguir muy claramente entre el puntero y el objeto al que apuntan. Se recomienda dibujar todas las operaciones con punteros que se programen, tal y como hemos estado haciendo aquí, hasta familiarizarse como los mismos.

Cuando deja de ser necesaria la memoria que se ha pedido, hay que liberarla. Si nunca liberamos memoria, al final se nos acabará. El programa tiene que liberar todos los recursos que ha reservado cuando ya no los necesita.

De igual modo que ejecutamos

```
new(pentero);
```

cuando queremos crear un nuevo entero, tenemos que destruir dicho entero cuando ya no nos resulte útil. La forma de hacer esto en Pascal es utilizar el procedimiento **dispose**. Este procedimiento, igual que el procedimiento **new**, admite como parámetro cualquier tipo que sea un puntero a otro tipo de datos.

Por ejemplo, este programa crea una variable dinámica para guardar un entero en ella, entonces inicializa la variable a 3, imprime el contenido de la variable y, por último, libera la memoria dinámica solicitada anteriormente.

```
 1 {
 2     Usar un puntero.
 3 }
 4
 5 program ptrint;
 6
 7 type
 8     TipoPtrInt = ^integer;
 9
10 var
11     pint: TipoPtrInt;
12 begin
13     new(pint);
14     pint^ := 3;
15     writeln(pint^);
16     dispose(pint);
17 end.
```

Antes de llamar a **dispose**, el estado de la memoria podría ser como se ve en la figura.

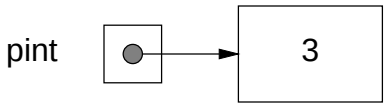

Fig. 11. Antes de liberar.

La llamada

```
dispose(pint);
```

libera la memoria a la que apunta el puntero. *A partir de este momento no puede utilizarse el puntero*. Téngase en cuenta que la memoria a la que apuntaba ahora podría utilizarse para cualquier otra cosa, dado que hemos dicho que ya no la queremos utilizar más para guardar el elemento al que apuntábamos. En este momento la situación ha pasado a ser:

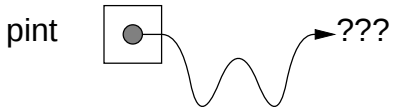

Fig. 12. Tras liberar.

por lo que es más seguro hacer que el puntero no apunte a ningún sitio a partir de este momento, para evitar atravesarlo por error.

```
pentero := nil;
```

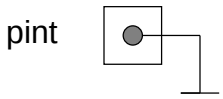

Fig. 13. Puesto a nil.

Aplicar una indirección a un puntero (esto es, utilizar el valor al que apunta) cuando se ha liberado la memoria a la que apuntaba es un error de consecuencias catastróficas.

Si modificamos la variable a la que apunta el puntero, alguna otra variable

cambia de modo mágico (¡De magia nada! la memoria a la que apuntaba el puntero ahora la está utilizando otra variable, y al modificarla hemos cambiado otra variable). En ocasiones el lenguaje (como Pascal) y/o el sistema operativo pueden detectar que el elemento al otro lado del puntero o no se encuentra en una zona de memoria válida o no tiene un valor dentro del rango para el tipo de datos al que corresponde y se producirá un error en tiempo de ejecución. Pero en la mayoría de los casos, **utilizar memoria ya liberada es un poltergueist**.

Como depurar estos errores cuando se comenten es extremadamente difícil, se suele intentar evitarlos a base de buenas costumbres y mucho cuidado. Por ejemplo, asignando **`nil`** inmediatamente después de liberar un puntero para que el sistema pueda detectar si intentamos atravesarlo y detener la ejecución del programa, en lugar de dejarlo continuar y alterar memoria que no debería alterar.

Si se ha dejado memoria sin liberar, se dice que tenemos un **memory leak** (en inglés, gotera) de memoria. ¡O que perdemos memoria! Esto es, que nos hemos dejado memoria dinámica sin liberar. En este curso es obligatorio liberar toda la memoria antes de acabar.

Algunos lenguajes, entre los que **no** se encuentra Pascal, inicializan casi todos los punteros (aquellos que no son parámetros) a **`nil`** y también se encargan de liberar la memoria dinámica cuando no quedan punteros apuntando hacia ella, sin que tengamos que llamar a **`dispose`** o ninguna otra función. A esta última habilidad se la llama **recolección de basura**. Muchos lenguajes notables, por ejemplo Pascal, C y C++, no tienen esta habilidad. Otros la tienen sólo a medias, por lo que no puedes confiar en que el lenguaje de programación solucione estos problemas.

10.4. Listas enlazadas

Un buen ejemplo de cómo utilizar punteros es ver la implementación de una estructura de datos que se utiliza siempre que se quieren tener **colecciones de datos de tamaño variable**. Esto es, algo que (igual que un *array*) permite tener una secuencia ordenada de elementos pero que (al contrario que un *array*) puede crecer de tamaño cuando son precisos más elementos en la secuencia.

Intenta prestar atención a cómo se utilizan los punteros. Cómo sea una lista es algo de lo que aún no deberías preocuparte (aunque te hará falta en el futuro) . Una vez domines la técnica de la programación podrás aprender más sobre algoritmos y sobre estructuras de datos. Pero hay que aprender a andar antes de empezar a correr.

Una lista se define como algo que es una de estas dos cosas:

1. Una lista vacía. Esto es, nada.
2. Un elemento de un tipo de datos determinado y el resto de la lista (que es también una lista). A esta pareja se la suele llamar **nodo** de la lista.

Supongamos que tenemos una lista de enteros. Esto sería una lista vacía:

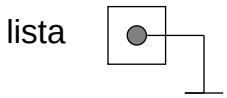

Fig. 14. Lista vacía

Y esto una lista con dos elementos (esto es, con un nodo que contiene el número 3 y otro que contiene el 5).

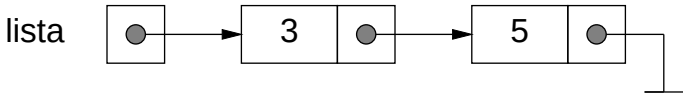

Fig. 15. Lista con dos elementos.

El tipo de datos en Pascal para una lista podría ser como sigue:

```
type
    TipoListaInt = ^TipoNodoInt;
    TipoNodoInt = record
        n: integer;        { valor del elemento }
        sig: TipoListaInt; { siguiente nodo en la lista }
    end;
```

Una lista es un puntero a un nodo. Un nodo contiene un valor y el resto de la lista, que a su vez es una lista. El valor lo declaramos como un campo del registro y el puntero al siguiente elemento de la lista (o el resto de la lista) como un campo del tipo de la lista.

Aquí cada tipo de datos (lista y nodo) hace referencia al otro: es una definición circular. Pascal nos permite hacer referencia a un tipo todavía no definido para romper la circularidad, pero sólo para tipos que son punteros. Posteriormente hay que declarar dicho tipo, claro está.

Nosotros vamos a utilizar las listas para implementar palabras dinámicas. Esto es, cadenas de caracteres de cualquier longitud.

10.5. Palabras dinámicas

Primero empecemos por declarar el tipo de datos para nuestras palabras. La idea es que tengamos una lista de caracteres, con los caracteres de la palabra.

```
type
    TipoStr = ^TipoNodoStr;
    TipoNodoStr = record
        c: char;
        sig: TipoStr;
    end;
```

Crear una lista vacía es tan simple como inicializar a **`nil`** una variable de tipo **`TipoStr`**. Este procedimiento hace el trabajo.

```
procedure vaciarstr(var str: TipoStr);
begin
    str := nil;
end;
```

Si tenemos una variable

```
var
    str: TipoStr;
```

y la inicializamos así

```
    vaciarstr(str);
```

entonces esta variable quedará como se ve en la figura.

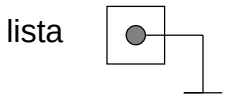

Fig. 16. Tras vaciar nuestra palabra.

Para ver si una palabra está vacía podemos implementar una función como esta:

```
function stresvacia(str: TipoStr): boolean;
begin
    result := str = nil;
end;
```

Insertar y extraer elementos de una lista suele hacerse de un modo u otro

según se quieran incluir al principio o al final. Si se insertan y extraen por el principio tenemos una pila. Si se insertan por el principio y extraen por el final tenemos una cola, también conocida como **LIFO** (o *Last In is First Out*, el último en entrar es el primero en salir) entonces se suelen insertar los elementos por el comienzo de la lista (la parte izquierda de la lista según la dibujamos) y se suelen extraer elementos también del mismo extremo de la lista.

Nosotros vamos a dar un procedimiento para incluir un nuevo carácter al final de la lista que implementa nuestra palabra. Así que vamos a insertar por el final. Primero veamos el procedimiento entero y luego volveremos a verlo poco a poco.

```
procedure appcar(var str: TipoStr; c: char);
var
    nodo, p: ^TipoNodoStr;
begin
    new(nodo);
    nodo^.c := c;
    nodo^.sig := nil;
    if str = nil then begin
        str := nodo;
    end
    else begin
        p := str;
        while p^.sig <> nil do begin
            p := p^.sig;
        end;
        p^.sig := nodo;
    end;
end;
```

Este procedimiento hace un “append’’, o añade, un carácter. Por eso su nombre. Lo primero que hace es crear un nuevo nodo con la línea

```
        new(nodo);
        nodo^.c := c;
        nodo^.sig := nil;
```

Ahora guarda el carácter en ese nodo y deja el puntero al siguiente de dicho nodo a un valor nulo.

```
        nodo^.c := c;
        nodo^.sig := nil;
```

Si la lista estaba vacía, entonces basta hacer que sea el puntero al nodo que acabamos de crear y hemos terminado.

```
if str = nil then begin
    str := nodo;
end
```

En otro caso, necesitamos guardar el puntero al nuevo nodo en el último puntero al siguiente de la lista. Esto es, en el puntero al siguiente del último nodo de la lista. Esto es lo que hace este código:

```
p := str;
while pˆ.sig <> nil do begin
    p := pˆ.sig;
end;
pˆ.sig := nodo;
```

Se utiliza un puntero **p** para no perder el puntero de la lista **str** y poder avanzar nodo a nodo. La sentencia

```
p := pˆ.sig;
```

hace que el puntero avance al siguiente nodo. Y tenemos que seguir avanzando hasta que el siguiente del nodo al que apuntamos sea "**nil**" (en cuyo caso ya tenemos el puntero al último nodo).

Para liberar la memoria que utiliza nuestra palabra, necesitamos una nueva operación como la que sigue:

```
procedure disposestr(var str: TipoStr);
var
    p: ˆTipoNodoStr;
begin
    while str <> nil do begin
        p := str;
        str := strˆ.sig;
        dispose(p);
    end;
end;
```

Esta se ocupa de llamar a **dispose** para cada uno de los nodos. Teniendo cuidado, eso si, de guardar el puntero al siguiente nodo antes de liberar el nodo.

El programa que sigue es el mismo que vimos en un tema anterior y escribía los enteros presentes en la entrada. Lo hace leyendo palabras de forma controlada y convirtiendo éstas a enteros si es posible. Esta vez, el programa utiliza las

palabras dinámicas que acambos de describir.

Es posible que tengas que leer el código varias veces y que tengas que simular su ejecución con lápiz y papel para una entrada que imagines. Es perfectamente normal, y muy recomendable.

```
 1
 2 {
 3     Palabras dinamicas
 4
 5 }
 6
 7 program paldin;
 8
 9 const
10     Tab: char = '    ';
11 type
12     TipoStr = ^TipoNodoStr;
13     TipoNodoStr = record
14         c: char;
15         sig: TipoStr;
16     end;
17
18 procedure fatal(msg: string);
19 begin
20     writeln('error fatal: ', msg);
21     halt(1);
22 end;
23
24 procedure vaciarstr(var str: TipoStr);
25 begin
26     str := nil;
27 end;
28
29 procedure appcar(var str: TipoStr; c: char);
30 var
31     nodo, p: ^TipoNodoStr;
32 begin
33     new(nodo);
34     nodo^.c := c;
35     nodo^.sig := nil;
36     if str = nil then begin
37         str := nodo;
38     end
39     else begin
40         p := str;
41         while p^.sig <> nil do begin
42             p := p^.sig;
```

```
43          end;
44          pˆ.sig := nodo;
45      end;
46  end;
47
48  procedure disposestr(var str: TipoStr);
49  var
50      p: ˆTipoNodoStr;
51  begin
52      while str <> nil do begin
53          p := str;
54          str := strˆ.sig;
55          dispose(p);
56      end;
57  end;
58
59  function longstr(str: TipoStr): integer;
60  var
61      n: integer;
62  begin
63      n := 0;
64      while str <> nil do begin
65          str := strˆ.sig;
66          n := n+1;
67      end;
68      result := n;
69  end;
70
71
72  function esblanco(c: char): boolean;
73  begin
74      result := (c = ' ') or (c = Tab);
75  end;
76
77  procedure leerstr(var entrada: Text; var str: TipoStr);
78  var
79      c: char;
80      haypalabra: boolean;
81  begin
82      vaciarstr(str);
83      haypalabra := false;
84      while not eof(entrada) and not haypalabra do begin
85          if eoln(entrada) then begin
86              readln(entrada);
87          end
88          else begin
89              read(entrada, c);
90              haypalabra := not esblanco(c);
```

```
91          end;
92      end;
93      while haypalabra do begin
94          appcar(str, c);
95          if eof(entrada) or eoln(entrada) then begin
96              haypalabra := false;
97          end
98          else begin
99              read(entrada, c);
100              haypalabra := not esblanco(c);
101          end;
102      end;
103 end;
104 
105 procedure escrstr(str: TipoStr);
106 begin
107     while str <> nil do begin
108         write(strˆ.c);
109         str := strˆ.sig;
110     end;
111 end;
112 
113 procedure escrstrln(str: TipoStr);
114 begin
115     escrstr(str);
116     writeln();
117 end;
118 
119 function strtostring(str: TipoStr): string;
120 var
121     n: integer;
122     s: string;
123 begin
124     n := 0;
125     setlength(s, n);
126     while str <> nil do begin
127         n := n+1;
128         setlength(s, n);
129         s[n] := strˆ.c;
130         str := strˆ.sig;
131     end;
132     result := s;
133 end;
134 
135 procedure leerstring(var entrada: text; var s: string);
136     var str: TipoStr;
137 begin
138     leerstr(entrada, str);
```

```
139     s := strtostring(str);
140     disposestr(str);
141 end;
142 
143 procedure leerint(var entrada: text;
144                   var n: integer; var ok: boolean);
145 var
146     s: string;
147     pos: integer;
148 begin
149     ok := not eof();
150     if ok then begin
151         leerstring(entrada, s);
152         val(s, n, pos);
153         ok := pos = 0;
154     end;
155 end;
156 
157 var
158     n: integer;
159     ok: boolean;
160 begin
161     repeat
162         leerint(input, n, ok);
163         if ok then begin
164             writeln(n);
165         end;
166     until eof();
167 end.
```

Un detalle importante que el lector astuto habrá notado es que, si a un procedimiento se le pasa como parámetro por valor una lista, dicho procedimiento puede siempre alterar el contenido de la lista. La lista se podrá pasar por valor o por referencia, pero los elementos que están en la lista siempre se pasan por referencia. Al fin y al cabo una lista es una referencia a un nodo.

Y sí, ¡Si Señor!, el paso de parámetros por referencia no es otra cosa que pedirle al lenguaje que use punteros internamente para referirse a los argumentos. Por eso se llama “por referencia”: el lenguaje utiliza un puntero al argumento y ese puntero lo utiliza el código del procedimiento para referirse al argumento. Claro, todo esto lo hace el lenguaje sin que tengamos que hacerlo nosotros (salvo en lenguajes como C y C++, en que tenemos que hacerlo nosotros).

10.6. Problemas

Como de costumbre, algunos enunciados corresponden a problemas hechos con anterioridad. Sugerimos que los vuelvas a hacer sin mirar en absoluto las soluciones. Esta vez sería deseable que compares luego tus soluciones con las mostradas en el texto.

1. Imprimir los números de la entrada estándar al revés, sin límite en el número de números que se pueden leer.
2. Leer una palabra de la entrada estándar y escribirla en mayúsculas, sin límite en la longitud de dicha palabra.
3. Implementar un conjunto de enteros con operaciones necesarias para manipularlo y sin límite en el número de enteros que puede contener.
4. Imprimir un histograma que indique cuántas veces se repite cada palabra en la entrada estándar, sin límite en el número de palabras que puede haber.
5. Imprimir un histograma que indique cuántas veces se repite cada palabra en la entrada estándar, sin límite en el número de palabras que puede haber y sin límite en cuanto a la longitud de la palabra.
6. Implementar un pila utilizando una lista enlazada. Utilizarla para ver si la entrada estándar es un palíndromo.
7. Implementar operaciones aritméticas simples (suma y resta) para números de longitud indeterminada.
8. Implementar un tipo de datos y operaciones para manipular *strings* de longitud variable de tal forma que cada *string* esté representado por una lista de *arrays*. La idea es que cada nodo en la lista contiene un número razonable de caracteres (por ejemplo 50). De esta forma un *string* pequeño usa en realidad un *array* (el del primer nodo) pero uno más largo utiliza una lista y puede crecer.

Índice