10. PUNTEROS Y VARIABLES DINÁMICAS

Conceptos: *Variables estáticas y dinámicas, Puntero, Apuntador, Dirección de memoria, Segmento de código, Segmento de datos, Segmento de pila o stack, Segmento de montículo o heap.*

Resumen: Este tema plantea la reserva dinámica de memoria y la referencia a variables a través de su dirección de memoria. Las técnicas de programación moderna exigen tratar con estructuras de datos dinámicas que, al contrario de las estáticas, se almacenan en áreas de la memoria de tamaño variable ya que su número crece o decrece en función de las necesidades del problema concreto. Se presentan las distintas partes en las que se organiza la memoria durante la ejecución de los programas: el segmento de código, el segmento de datos, el segmento de pila o stack y el segmento de montículo o heap. Es en este último segmento donde se reserva espacio en memoria para las variables dinámicas. La utilización de variables dinámicas se realiza a través de un tipo especial de variable denominado apuntador o puntero, que permite representar direcciones de memoria de un dato de un tipo determinado, y del procedimiento estándar new de TurboPascal. El tipo de variable dinámica especifica el espacio de memoria que se reserva para dicha variable al ejecutar la llamada al procedimiento new.

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

a) Definir el concepto de puntero o apuntador y de variable dinámica (*Conocimiento*)
b) Describir el mecanismo de funcionamiento de la memoria durante la ejecución de un programa (*Conocimiento*)
c) Escribir la declaración de tipos puntero y de variables de variables de cualquiera de los tipos de datos estructurados (*Comprensión*)
d) Interpretar el código fuente de un programa que utilice punteros y variables dinámicas (*Comprensión*)
e) Codificar una tarea sencilla convenientemente especificada, utilizando datos de tipo puntero y variables dinámicas (*Aplicación*)

10.1. INTRODUCCIÓN

Hasta ahora las variables que se han empleado en los programas, denominadas genéricamente ***estáticas***, reunen una serie de características comunes:

a) Todas las variables estáticas que utiliza un programa o una rutina se declaran explícitamente con la palabra reservada `var` en la zona de declaraciones.
b) Cada variable estática tiene un único nombre o identificador.
c) Las variables estáticas simples y estructuradas o estructuras de datos estáticas definen su tipo y tamaño durante la compilación, y en tiempo de ejecución, se reserva el espacio anteriormente definido en memoria, en el segmento de datos (variables globales) o en el segmento de pila o *stack* (variables locales).

El trabajar con variables estáticas estructuradas o estructuras de datos estáticas es muy sencillo pero su empleo implica una serie de inconvenientes:

a) Proporcionan una estructura **rígida** que apenas puede alterarse durante la ejecución del programa.
b) El espacio en memoria correspondiente tanto al segmento de datos como al segmento de pila es muy limitado.
c) No aprovechan de forma óptima la memoria disponible. Hay que reservar espacio en memoria para toda la estructura durante toda la ejecución independientemente de los requerimientos reales de los programas. En algunos casos se desperdiciará parte del espacio en memoria reservado y en otros no podrán almacenarse más datos de los previstos en un principio.

Estos inconvenientes pueden solventarse utilizando otro tipo de variables y las estructuras de datos a las que dan lugar. Este tipo de variables son las variables ***dinámicas***, que a diferencia de las estáticas:

a) No se declaran en la zona de declaraciones de variables del programa o de las rutinas.
b) Para almacenar una variable dinámica se reserva espacio en memoria en algún momento durante la ejecución del programa. Asimismo puede también liberarse ese espacio de memoria reservado para una variable dinámica durante la ejecución para otros usos. Es decir, la variable dinámica no tiene porqué reservar espacio en memoria durante toda la ejecución del programa.
c) Por consiguiente, el tamaño, y por lo tanto el espacio reservado en memoria, para las estructuras de datos dinámicas (formadas por un conjunto de variables dinámicas) puede variar durante la ejecución del programa según las necesidades del mismo.
d) Una misma variable dinámica puede tener varios identificadores.
e) La parte de la memoria utilizado para almacenar las variables dinámicas durante la ejecución de un programa es distinta del lugar de almacenamiento de las variables estáticas y se denomina el segmento de montículo o *heap*. Esto tiene como mayor consecuencia no tener la limitación del espacio máximo reservado para las variables estáticas (casi 64Kb para las variables globales y otros 64Kb para las variables locales) aumentándose el espacio utilizable para las variables dinámicas hasta el espacio máximo libre disponible en memoria.

Es aconsejable utilizar variables dinámicas si hay que manejar grandes cantidades de datos (>64 KB) y si se desconoce el tamaño de los datos a emplear. Para utilizar variables dinámicas es preciso introducir previamente otro tipo de dato: el tipo apuntador o puntero. Aunque antes se va a comentar algo más acerca de la utilización de la memoria del ordenador durante la ejecución de un programa escrito en TurboPascal.

10.2. ASIGNACIÓN DE MEMORIA DURANTE LA EJECUCIÓN DE UN PROGRAMA

Durante **la ejecución** de un programa escrito en TurboPascal en el sistema operativo D.O.S., la memoria del ordenador queda dividida en varias partes o segmentos que cumplen distintas funciones (Tabla 23). Cada segmento tiene un nombre específico:

a) Segmento de código.
b) Segmento de datos (*Data Segment*).
c) Segmento de pila (*Stack segment*).
d) Segmento de montículo o *heap*.

10.2.1. Segmento de código

En el segmento de código se almacenan en zonas o módulos independientes el código del programa principal, el código de las unidades en orden inverso a su declaración y el código de la unidad `System`. El tamaño total de este segmento está sólo limitado por la memoria libre del ordenador pero el tamaño de cada módulo está limitado a 64 Kb.

10.2.2. Segmento de datos

En el segmento de datos se almacenan, durante la ejecución, las variables globales y las constantes con tipo del programa principal y de las distintas unidades empleadas. El tamaño de este segmento está limitado a 65520 bytes (algo menos de 64Kb).

10.2.3. Segmento de pila

En el segmento pila o *stack* se reserva espacio para las variables locales de las rutinas, cuando éstas se están ejecutando, los parámetros que se pasan a éstas y otras variables auxiliares. El tamaño reservado por defecto paa este segmento es de 16Kb en un entorno D.O.S., pero puede modificarse (reducirse ó aumentarse entre 1024 y 65520 *bytes*) mediante la directiva `{$M}` del compilador hasta un máximo de 64Kb con el primero de sus tres parámetros especificándolo en bytes.

La directiva `{$S}` se emplea para incluir la comprobación del desbordamiento de pila en el código. Por defecto está activada `{$S+}` y la consecuencia es que se genera código para comprobar el desbordamiento de la pila al comenzar a ejecutarse un procedimiento o función. Un desbordamiento de la pila puede generar un bloqueo del sistema, así que no se recomienda la desactivación de la directiva, a no ser que se está absolutamente seguro de que no va a ocurrir.

10.2.4. Segmento de montículo

En el segmento de almacenamiento dinámico, montículo o *heap*, se reserva espacio para las variables dinámicas. Su tamaño máximo también viene limitado por la memoria convencional (640Kb) libre del sistema, si bien puede reducirse mediante la directiva `{$M}` del compilador. El segundo y tercer parámetro de esta directiva especifican en bytes el tamaño mínimo y máximo reservado para el segmento montículo.

Tabla 23. Estructura de la memoria durante la ejecución de un programa.

Segmento	Contenido
Segmento de código	*Código del programa principal*
	Código Unidad n
	...
	Código Unidad 2
	Código Unidad 1
	Código Unidad System
Segmento de datos	*Constantes con tipo y variables globales*
Segmento pila	*Variables locales, parámetros por valor de rutinas y otras variables auxiliares*
Segmento montículo	*Variables dinámicas*

10.3. APUNTADORES O PUNTEROS

Un puntero es la dirección de "algo" en la memoria del ordenador. Una variable de tipo apuntador o puntero es una variable que almacena un valor que es la dirección de una posición de la memoria en la que se almacena otro dato de cualquiera de los tipos vistos hasta ahora excepto de tipo archivo (`file of ...`, `text` o `file`).

Una dirección de memoria en el sistema operativo DOS se determina mediante dos enteros de dos bytes, llamados **segmento** y **desplazamiento**, con lo que una variable puntero necesitará 4 bytes de espacio en memoria.

Las variables puntero se emplean generalmente para almacenar direcciones de memoria de variables dinámicas en el segmento de montículo. Es muy importante destacar que las variables puntero que se empleen al principio serán variables *estáticas*, pero más adelante se introducirán también variables *dinámicas* de tipo puntero.

10.4. DECLARACIONES NECESARIAS

En general, primero se define el tipo de dato puntero y luego la variable o variables de ese tipo. Al definir un tipo de dato puntero se debe indicar explícitamente el tipo de dato que se va a almacenar en la posición de memoria referenciada. Esto es necesario porque datos de diferentes tipos requieren distintas cantidades de espacio en memoria.

10.4.1. Declaraciones de tipos de dato y variables puntero

Un tipo puntero se declara con el tipo de la variable a la que apunta precedida por el símbolo `^`. Sintaxis de la definición de tipos de dato puntero:

```
type tipo_puntero = ^tipo_dato;
                    { ^ : se lee como "apunta a" }
```

Ej.:
```
type  punteroaentero = ^integer;  { declara el tipo }
var   p : punteroaentero;       { declara la variable }
```

En este ejemplo, en primer lugar se ha declarado un tipo de dato puntero, `punteroaentero`, que va a representar la dirección de memoria de un dato de tipo `integer` y posteriormente, se ha declarado una variable (¡estática!) de este tipo `tipo_puntero`. Aunque en un programa también se puede incluir implícitamente la declaración del tipo en la declaración de la variable.

Ej.:
```
var   q : ^integer;
      r : ^char;
      s : ^string[20];
```

Si bien este último formato no es tan recomendable como el primero: es preferible declarar en primer lugar los tipos de dato puntero y luego las variables correspondientes.

Ej.:
```
type   pentero = ^integer;
       pchar = ^char;
       pcadena30 = ^string[20];
       dato = record;
         nombre : string[30];
         numero : integer
         end;
       elemento: ^dato;
var    q : pentero;
       r : pchar;
       s : pcadena30;
       t : elemento;
```
En resumen, una variable puntero almacena un dato que es la posición de otro dato. Si se define el tipo de dato `puntero` y la variable `p` de la siguiente manera:
```
type  puntero = ^integer;
var   p : puntero;
```
la declaración de la variable `p` reserva 4 bytes de espacio en el segmento de datos de la memoria para una **variable estática** que puede almacenar una dirección de memoria de una **variable dinámica** de tipo `integer`. En un principio, esta variable no almacena la dirección de memoria de ningún dato determinado

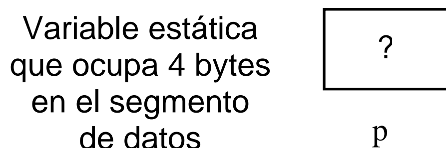

Figura 55. Consecuencia de la declaración de una variable `p` de tipo `puntero`

Como se verá en este capítulo, en un programa podrán existir variables puntero que apunten a datos estructurados: datos de tipo `array`, `record`, ... datos de tipo puntero como campos de un tipo `record` y además, también se podrán utilizar variables punteros como parámetros de funciones y procedimientos.

Aunque, quizás, todo lo dicho hasta ahora no diga mucho de la utilidad de este tipo de variables, como se verá más adelante, serán útiles para construir determinadas estructuras de datos en las que las variables puntero serán los lazos de unión entre los elementos que forman la estructura.

10.4.2. El tipo de dato pointer

Por otro lado, existe un tipo genérico de puntero predefinido en TurboPascal que apunta a un dato de un tipo no determinado. Es el tipo de dato `pointer`. Al ser un tipo de dato predefinido la declaración de una variable de este tipo no requiere de la declaración de tipo previa. Por ejemplo: `var p : pointer;`

10.5. OPERACIONES CON VARIABLES PUNTERO

Con las variables puntero pueden realizarse operaciones similares a las que se llevan a cabo con variables de otros tipos y, además, otras operaciones específicas.

10.5.1. Creación de una nueva variable dinámica

El procedimiento estándar de TurboPascal `new` crea una variable dinámica y establece que una variable puntero apunte a ella. Si la variable puntero es una variable estática debe

haberse declarado en la sección de declaración de variables correspondiente. Así, dada la variable puntero declarada en el ejemplo anterior, la sentencia de llamada al procedimiento `new(p);` puede encontrarse en el cuerpo de sentencias del programa:

Ej.:
```
type  puntero = ^integer;
var p:puntero;
begin
new(p);
```
la ejecución de `new(p);` produce el siguiente efecto:

a) En primer lugar, asigna o reserva en el segmento de montículo el espacio de memoria conveniente para almacenar el tipo de dato correspondiente. En este caso, un total de 2 bytes ya que la variable dinámica es de tipo `integer`

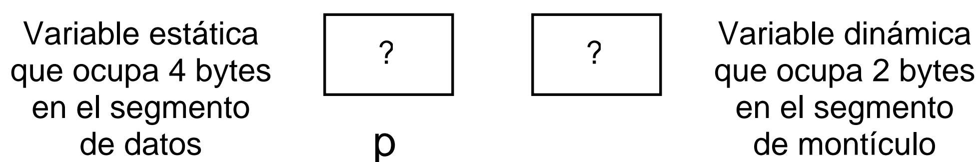

Figura 56. Reserva de espacio en el segmento de montículo para la variable dinámica

b) La dirección de esta posición de la memoria se almacena en la variable puntero `p` creándose el vínculo entre ésta y la variable dinámica, que toma ahora como identificador el de la variable puntero seguida del caracter `^`. La variable dinámica almacenada en la posición de memoria indicada en la variable puntero `p` se representa por `p^`.

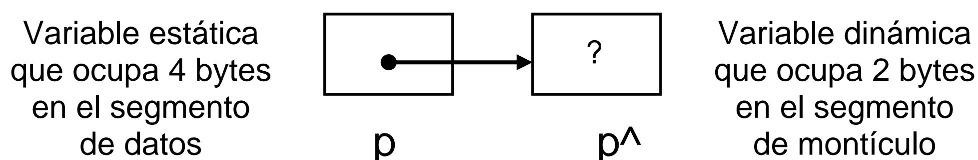

Figura 57. Almacenamiento en la variable puntero de la dirección de memoria de la variable dinámica

10.5.2. Identificación de una variable dinámica

En general, para la asignación de valores a *variables dinámicas* ya disponibles se utilizan las mismas sentencias de asignación, funciones, procedimientos, etcétera, empleados para las *variables estáticas* del mismo tipo de dato.

Por ejemplo, para asignar un valor a la variable dinámica anterior y visualizar el dato almacenado por pantalla se emplearían respectivamente las sentencias:

```
type  puntero = ^integer;
var   p : puntero;
begin
new(p);
p^:=50;            { se almacena 50 en la variable dinámica }
```

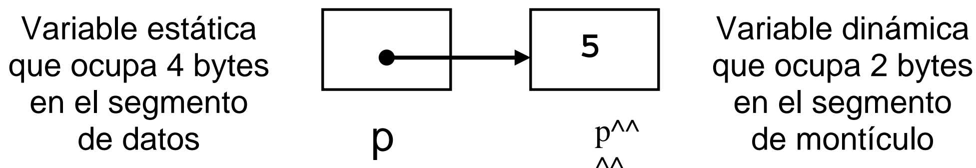

Figura 58. Asignación de un valor a la variable dinámica

```
write(p^);          { se visualiza 50 en pantalla }
p^:=60;             { se almacena 60 en la variable dinámica }
```

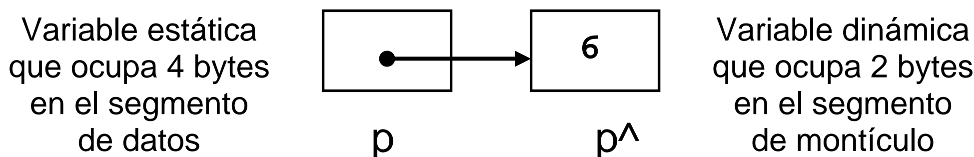

Figura 59. Asignación de un nuevo valor a la variable dinámica

```
write(p^+20);       { se visualiza 80 en pantalla }
```

¡Cuidado!: La sentencia de asignación **`p:=50;`** intentaría cambiar la posición a la que apunta la variable puntero y el compilador de TurboPascal 7.0 no lo permitiría (error de compilación). Por lo tanto, hay que recordar que:

`p` identifica la posición de la variable apuntada y

`p^` identifica el contenido de la variable apuntada.

10.5.3. Asignaciones entre variables puntero

Si se declaran dos variables puntero que apunten a variables dinámicas del mismo tipo, pueden realizarse asignaciones entre dichas variables puntero:

```
type  puntero = ^integer;
var p,q:puntero;
begin
new(p);
q:=p;
{ ambas variables puntero apuntan
  a la misma dirección de memoria }
```

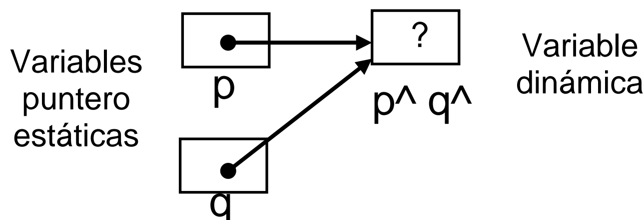

Figura 60. Consecuencia de la asignación entre dos variables puntero

Como ambas variables puntero almacenan la dirección de memoria de la misma variable dinámica, a ésta se le asocian dos identificadores: `p^` y `q^`.

Por otro lado, también pueden realizarse asignaciones entre variables dinámicas siempre que se respete la compatibilidad entre tipos de dato:

```
type  puntero = ^integer;
var p,q:puntero;
```

```
begin
new(p);
p^:=50;
new(q);
q^:=p^;
{ en ambas direcciones de memoria / variables dinamicas
  se almacena el mismo valor }
```

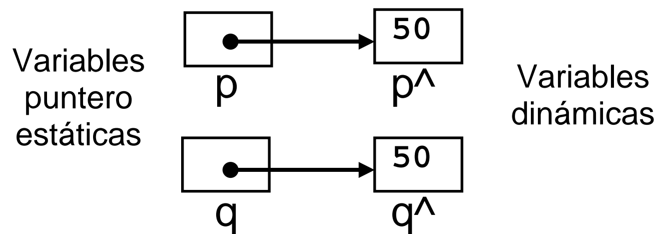

Figura 61. Consecuencia de la asignación entre variables dinámicas

Se debe evitar la situación de tener variables dinámicas cuya dirección de memoria no esté almacenada en ninguna variable puntero, ya que en ese mismo momento, la variable dinámica pierde todo identificador y queda inaccesible. Por ejemplo:

```
type  puntero = ^integer;
var p,q:puntero;
begin
new(p);
p^:=50;
new(q);
q^:=p^;      { en ambas direcciones de memoria / variables dinamicas
               se almacena el mismo valor }
q:=p;         { ambas variables puntero apuntan
                a la misma dirección de memoria }
```

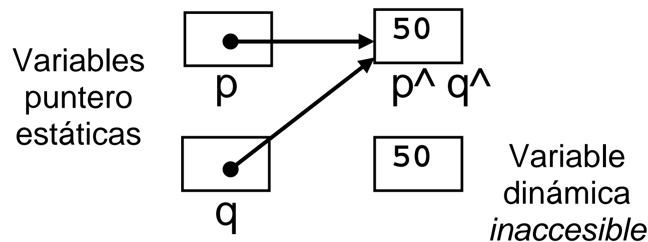

Figura 62. Problema: generación de una variable dinámica no accesible al *perder* su dirección de memoria

10.5.4. La constante nil

La constante `nil` es una constante predefinida de TurboPascal cuyo valor es una posición de memoria determinada que hace las veces de *tierra* (en términos eléctricos). Un ejemplo de uso de esta constante es el siguiente:

```
type  puntero = ^integer;
var p:puntero;
begin
p:=nil;
```

Con la asignación `p:=nil;` la variable puntero `p` toma el valor de la constante `nil`, o dicho de forma, se deja a la variable puntero apuntando a una posición determinada que no corresponde a ningún dato concreto.

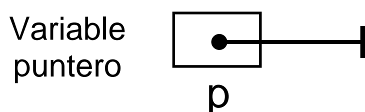

Figura 63. Representación gráfica de la asignación de la constante `nil` a una variable puntero

La constante `nil` se suele emplear para determinar si una variable puntero apunta o no a una variable dinámica en un instante determinado durante la ejecución de un programa. La constante `nil` es una dirección de memoria compatible con (es decir, puede asignarse a) una variable puntero de cualquier tipo.

10.5.5. Liberación del espacio en memoria de una variable dinámica

Una vez que se termine de trabajar con una variable dinámica en un programa, el procedimiento estándar de TurboPascal `DISPOSE` libera el espacio de memoria ocupado por esa variable dinámica. La ejecución de la llamada al procedimiento `dispose(p);` puede incluirse en el cuerpo del programa una vez se deje de necesitar la variable dinámica:

```
type  puntero = ^integer;
var   p : puntero;
begin
new(p);
p^:=50;
{ Operaciones varias .. }
dispose(p);
```

la ejecución de la llamada a `dispose(p);` tiene estrictamente como consecuencia que:

a) Se libere el espacio de memoria (del segmento de montículo) cuya dirección se almacenaba en la variable puntero `p` dejando la variable dinámica `p^` de existir *formalmente*.

c) Cualquier reserva posterior de memoria dinámica podrá realizarse en este espacio de la memoria que queda libre.

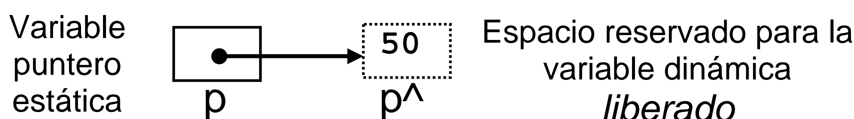

Figura 64. Liberación del espacio reservado para una variable dinámica con `dispose`

En la práctica, la ejecución de la llamada al procedimiento `dispose(p);` no borra ni la dirección de memoria almacenada en la variable puntero `p`, ni el valor almacenado en la variable dinámica `p^`.

La ejecución puede asemejarse a dejar disponible una cinta de vídeo donde se ha grabado previamente algún programa de televisión y que, en un momento dado, deja de interesar. De forma que no se destruye ni la cinta ni su contenido, sino que simplemente se deja en el cajón de cintas disponibles por si se desea darle otro uso en el futuro. No se debe confiar en que el contenido de la cinta sea el mismo dentro de unos días, porque el componente más joven de la familia puede grabar el programa de dibujos animados sin previo aviso.

El compilador no prohibe ni restringe el uso de variable dinámicas *liberadas*, así que, el buen programador simplemente debe olvidarse, a todos los efectos, de la variable dinámica una vez liberada mediante el procedimiento `dispose`.

Con la posibilidad de empleo de los procedimientos estándar `NEW` y `DISPOSE` se puede optimizar el uso de la memoria durante la ejecución de un programa, reservando o liberando el espacio en la memoria conforme se vaya o no necesitando.

10.5.6. Operadores de relación

Sólo pueden compararse variables punteros en expresiones de igualdad (operador =) o desigualdad (operador <>) y si apuntan a datos del mismo tipo. Por ejemplo, pueden encontrarse las siguientes expresiones en un programa:

```
type  puntero = ^integer;
var p,q:puntero;
begin
... if p=q then ...
... while p<>nil do ...
```

Independientemente de éstas, podemos hacer otras comparaciones (<, <=, >=, >) y operaciones, en general, con las variables almacenadas en sus direcciones de memoria si los tipos de dato lo permiten:

```
type  puntero = ^integer;
var p,q:puntero;
begin
... if p^<q^ then ...
... while p^<>90 do ...
```

10.6. MÁS EJEMPLOS DE UTILIZACIÓN DE PUNTEROS

En esta sección se incluyen ejemplos de utilización de variables puntero que apuntan a variables dinámicas de tipos de dato estructurados y de una función que devuelve un valor de tipo puntero.

10.6.1. Puntero que apunta a datos de tipo `array`.

En el siguiente programa se trabaja con una variable puntero que almacena la dirección de memoria de una variable dinámica de tipo `array`:

```
type  vector = array[0..10] of integer;
      punt = ^vector;
var a:punt;
    i:integer;
begin
new(a);
for i:=0 to 10 do a^[i]:=2*i+1;
for i:=0 to 10 do writeln(a^[i]:8);
dispose(a)
end.
```

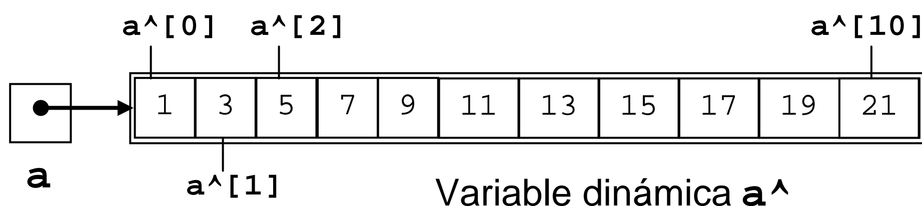

Figura 65. Variable puntero que apunta a una variable dinámica de tipo `array`

10.6.2. Puntero que apunta a datos de tipo `record`.

En el siguiente programa se trabaja con una variable puntero que almacena la dirección de memoria de una variable dinámica de tipo `record`:

```
type  persona = record
                nombre : string[20];
                dato : integer
```

```
               end;
      ptr = ^persona;
var p:ptr;
begin
new(p);
write('Introduce nombre: ');
readln(p^.nombre);
write('Introduce dato: ');
readln(p^.dato);
with p^ do writeln('Individuo: ',nombre,'; Edad: ',dato);
dispose(p)
end.
```

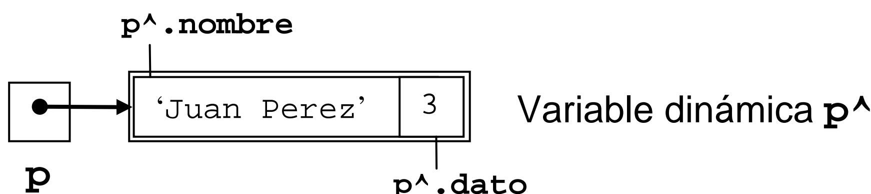

Figura 66. Variable puntero que apunta a una variable dinámica de tipo `record`

10.6.3. Puntero que apunta a datos de otro tipo puntero

En el siguiente programa se trabaja con una variable puntero que almacena la dirección de memoria de una variable dinámica de otro tipo puntero:

```
type  puntero1 = ^integer;
      puntero2 = ^puntero1;
var p:puntero2;
begin
new(p);
new(p^);
p^^:=25;
writeln(p^^+30);
dispose(p^);
dispose(p)
end.
```

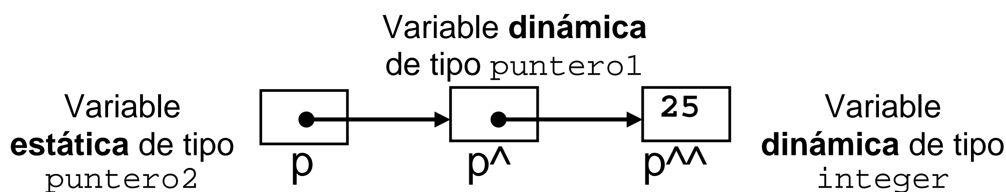

Figura 67. Variable puntero que apunta a una variable dinámica también de tipo puntero

10.6.4. Función que devuelve un puntero

Una función también puede devolver la dirección de memoria de una variable dinámica como se muestra en el siguiente programa:

```
type  pentero = ^integer;
var p,q:pentero;
function f1(n:integer):pentero;
  var t:pentero;
  begin
  new(t);
  t^:=n;
  f1:=t
  end;
begin
p:=f1(25);
```

```
q:=f1(60);
writeln(p^+q^);
dispose(p);
dispose(q)
end.
```

10.7. PUNTEROS COMO PARÁMETROS DE RUTINAS

Los parámetros de una subrutina pueden ser de tipo puntero. A simple vista, puede parecer que estos parámetros se comportan de manera distinta a los vistos hasta ahora. Se aconseja estudiar detenidamente el siguiente ejemplo que aclara el comportamiento de estos parámetros en diversos casos:

```
type  puntero = ^integer;
var a : puntero;
procedure pro1(b:puntero);
  begin
        b^:=12
        end;
procedure pro2(b:puntero);
        begin
        new(b);
        b^:=12
        end;
procedure pro3(var b:puntero);
        begin
        b^:=12
        end;
procedure pro4(var b:puntero);
        begin
        new(b);
        b^:=12
        end;
begin
new(a);
a^:=1;
{ llamada de ejecucion de uno de los
  procedimientos anteriores }
```

....

a) la ejecución del procedimiento `pro1(a)` deja almacenada en la variable dinámica a la que apunta la variable apuntador `a` el valor entero 12.

b) la ejecución de `pro2(a)` deja almacenada en la variable dinámica `a^` el valor 1.

c) la ejecución de `pro3(a)` deja almacenada en la variable dinámica `a^` el valor 12.

d) la ejecución de `pro4(a)` almacena en la *nueva* variable dinámica a la que apunta `a` el valor 12.

Como consecuencia de esto, lo más importante a destacar es que si se tiene en una rutina un parámetro por valor de tipo apuntador sólo se protege la dirección de memoria almacenada en la variable puntero de la llamada pero no lo almacenado en esa posición de memoria.

10.8. IMPLEMENTACIÓN DE ESTRUCTURAS DE DATOS CON PUNTEROS

Precisamente, la potencia en la utilización de punteros deriva de la posibilidad de introducirlos como campos de datos de tipo `record` que apunten a otros datos del mismo tipo como se verá en la siguiente sección.

Una lista dinámica es una estructura secuencial de variables dinámicas en la que para acceder a una de ellas es preciso pasar por todas las situadas previamente. Un ejemplo de programa que construye una lista dinámica de elementos es el siguiente:

```
program lista;
{ Definición del elemento con el que se construye la lista }
type  puntero = ^elemento;
      elemento = record
               dato : integer;
               sig : puntero  { El segundo campo es de tipo puntero }
         end;
{ Se utilizaran dos variables puntero para construir la lista }
var p,q : puntero;
    a  : integer;
begin
{ Comienzo de la construccion de la lista }
p:=nil;
writeln('----- Construccion de una lista dinamica -----');
write('Introduce un valor entero (0 para lista vacia): ');
readln(a);
{ Se enlazaran elementos hasta que se introduzca el valor 0 }
while a<>0 do
  begin
  new(q);
  q^.dato:=a;
  q^.sig:=p;
  p:=q;
  write('Introduce un valor entero (0 para terminar): ');
  readln(a)
  end;
{ Fin de la construccion de la lista }
{ El ultimo elemento introducido es el primero de la lista }
{ El primer elemento introducido es el ultimo de la lista :
  su campo sig almacena la constante nil }
{ Visualizacion de los elementos de la lista }
write('Elementos de la lista: ');
while q<>nil do
  begin
  write(q^.dato:3);
  q:=q^.sig
  end
{ Fin de la visualizacion de los elementos de la lista }
{ Liberacion del espacio en memoria de toda la estructura }
q:=p;
while p<>nil do
  begin
  q:=p^.sig;
  dispose(p);
  p:=q
  end;
end.
```

```
Interfaz de comandos

C:\lista
----- Construccion de una lista dinamica -----
Introduce un valor entero (0 para lista vacia): 1
Introduce un valor entero (0 para terminar): 2
Introduce un valor entero (0 para terminar): 3
Introduce un valor entero (0 para terminar): 4
Introduce un valor entero (0 para terminar): 0
Elementos de la lista:    4  3  2  1
C:\
```

Figura 68. Ejemplo de ejecución del programa `lista` en el terminal de DOS

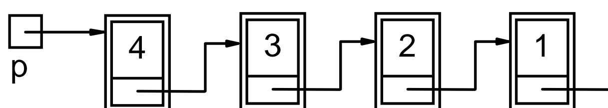

Figura 69. Estructura de variables dinámicas generada con el programa `lista` al introducir vía teclado los valores `1, 2, 3, 4 y 0.`

10.9. MÁS FUNCIONES Y PROCEDIMIENTOS ESTÁNDAR QUE TRABAJAN CON PUNTEROS

Para asignar, liberar y manipular memoria dinámica también pueden emplearse las siguientes rutinas estándar de TurboPascal que se comentan a continuación:

10.9.1. MARK

La ejecución de la llamada al procedimiento `mark(q);` almacena en el puntero genérico `q` la dirección actual (primera posición libre) del segmento montículo, que es dónde se almacenan las variables dinámicas. Se utiliza en combinación con el procedimiento estándar `release` (no con `dispose` o `freemem`).

10.9.2. RELEASE

La ejecución del procedimiento `release(q);` libera la memoria ocupada por todas las variables dinámicas creadas desde que se realizó la última llamada al procedimiento `mark`. Esto permite liberar el espacio ocupado por varias variables dinámicas de una vez. No debe emplearse junto con `dispose` o `freemem`.

10.9.3. GETMEM

El procedimiento `getmem(q,t);` reserva espacio en memoria para una nueva variable dinámica de un tamaño específico (`t` bytes) y almacena la dirección del bloque en la variable puntero genérica `q`.

10.9.4. FREEMEM

El procedimiento `freemem(q);` libera el espacio de la memoria reservado por la variable dinámica `q^`.

10.9.5. ADDR

La función `addr(x):pointer;` devuelve la dirección de una variable de cualquier tipo en formato de puntero genérico.

10.9.6. MAXMAVAIL

La función `maxavail` devuelve el tamaño en bytes del máximo bloque disponible en el segmento montículo.

10.9.7. MEMAVAIL

La función `memavail` devuelve (en bytes) y la suma de todos los bloques libres en el segmento montículo.

Bibliografía básica

- **García-Beltrán, A., Martínez, R.** y **Jaén, J.A.** *Métodos Informáticos en TurboPascal*, Ed. Bellisco, 2ª edición, Madrid, 2002
- Borland Pascal with Objects - Language Guide, Editorial Borland, 1992
- Borland Pascal with Objects - Programmer's Reference, Editorial Borland, 1992
- **Joyanes, L.** *Fundamentos de programación, Algoritmos y Estructuras de Datos*, McGraw-Hill, Segunda edición, 1996