

## Static arrays

Bueno, aquí nos encontramos una vez más tratando de develar otro de los oscuros misterios de la programación. Por cierto, no se si servirá de ánimo para aquellos que empiezan, pero déjenme decirles que la programación no tiene nada de misterioso, simplemente hay que sentarse y esforzarse, como en cualquier materia de la secundaria o la universidad. Claro, es que por alguna estúpida razón los "genios", los "gurues" nos hacen pensar que todo es demasiado difícil y que sólo ellos son capaces de comprenderlo... imbéciles.

En fin, el día de hoy nos ha tocado hablar sobre los arrays. Un array es una colección indexada de elementos del mismo tipo. ¿Cómo? A ver, hasta el momento seguramente te has estado manejando con los tipos de datos comunes que vienen con Delphi, me refiero a integers, strings, chars, etc, etc. Si pensamos en una lista de varios integers o varios strings, etc. y a cada elemento le damos un índice, para luego poder hacer referencia a él fácilmente, entonces estamos pensando en un array.

En el array todos los elementos de la lista son del mismo tipo, por ejemplo, o son todos integers o son todos strings, pero no puede haber integers y strings dentro de un mismo array.

**Nota:** esto no es totalmente cierto, ya que como veremos más adelante también existen los arrays de tipo variant que nos permiten guardar datos de diferentes tipos dentro de un mismo array. Sin embargo, como regla general, los elementos de los arrays son todos del mismo tipo.

Veamos una declaración de 2 arrays de tipo integer:

```
var
  P: array [0..9] of integer;
  Q: array [10..19] of integer;
```

**Nota:** fíjate que lo que va entre corchetes es el rango de los índices y no la cantidad de elementos del array. Por ejemplo, [0..9] en vez de [10].

En la mayoría de los lenguajes el primer elemento de la lista tiene índice 0, el segundo 1, el tercero 2 y así hasta count - 1. Por ejemplo, el array de 10 elementos llamado P estará formado por P[0], P[1], ..., P[8], P[9]. Esto es lo "común" cuando hablamos de arrays y es lo que hemos hecho en el ejemplo anterior. Sin embargo, en el ejemplo anterior también vemos que hay otro array llamado Q, de 10 elementos, pero que empieza con índice 10. Como ves, en Delphi las cosas no son iguales que en el resto de los lenguajes, en este sentido hay más libertades. Sin embargo, no es recomendable crear arrays que comiencen con índices distintos de 0 fundamentalmente para no perder compatibilidad con los demás lenguajes.

Veamos ahora un ejemplo de un array en acción.

```
var
  P: array [0..9] of integer;
  i: integer;
begin
  for i := 0 to 9 do
    P[i] := i;
end;
```

Aquí se ve cómo llenar un array con números. Para leer o escribir cualquier elemento del array sólo debemos llamarlo por NombreArray[Índice]. Muy fácil, verdad.

Muy bien, hasta el momento hemos visto arrays de 1 dimensión, es decir, arrays que tienen elementos a los que podemos hacer referencia utilizando 1 sólo índice. Sin embargo, esto no siempre es así y algunas veces se torna necesario crear arrays más complejos, arrays multidimensionales (más de 1 dimensión).

Dentro de los arrays multidimensionales, el caso más común es el de los bidimensionales (2 índices) y de vez en cuando verás por ahí algún loco que se atrevió a usar uno de 3 dimensiones

(tridimensional). Cualquiera sea el caso, si no me equivoco Delphi permite crear arrays de hasta 12 dimensiones, algo de lo que por supuesto no haremos uso simplemente porque ya un array de 4 dimensiones es imposible de dibujar y por lo tanto muy difícil de imaginar.

De todas formas, los arrays bidimensionales sí son muy comunes, en verdad seguramente los has estudiado en la escuela con el nombre de "tabla de doble entrada". Bueno, pero basta de cháchara y veamos como usar un array de estas características.

```
var
  P: array [0..9, 0..9] of integer;
  i, x: integer;
begin
  for i := 0 to 9 do
    for x := 0 to 9 do
      P[i,x] := i;
    end;
  end;
```

Bien, el ejemplo anterior declara al array bidimensional (10 x 10) P de tipo integer. De esta forma tendremos 10 x 10 = 100 casilleros para llenar con integers. Si corremos el ejemplo anterior veremos que el primer for navega por las filas y el segundo por las columnas de nuestro array, o mejor dicho (para que tengan una mejor idea en sus cabezas) de nuestra tabla de doble entrada. De esta forma i representa los nros. de fila y x los nros. de columna.

Filas(abajo)\Columnas (derecha)	1	2	3	4	5
1	1,1	1,2	1,3	1,4	1,5
2	2,1	2,2	2,3	2,4	2,5
3	3,1	3,2	3,3	3,4	3,5

La tabla anterior es un ejemplo perfecto de un array bidimensional (3 x 5) y nos enseña cuál es la forma de hacer referencia a los elementos del array, me estoy refiriendo a cuales deben ser los índices y en qué orden van.

### Ventajas y desventajas

Antes de continuar hagamos un breve resumen de cuales son las ventajas y las desventajas de esta estructura de datos llamada arrays. ¿Por qué es importante conocer las ventajas y desventajas de los arrays? La respuesta es simple. Básicamente porque pueden darnos una gran ayuda a la hora de decidir qué estructura de datos se adecúa mejor a nuestras necesidades.

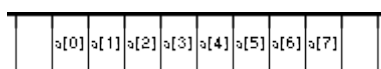
**Nota:** los arrays, al igual que los records, los árboles binarios, etc. son *estructuras de datos*, lo que implica que a diferencia de los tipos de datos comunes en Delphi (integers, strings, bool, etc.) estos almacenan *varios* datos; guardan no solamente 1 valor, sino muchos.

### VENTAJAS

Una de las grandes ventajas de los arrays es que al almacenar todos sus elementos en forma de bloque en la memoria, acceder a cualquiera de ellos se transforma en un simple cálculo matemático muy fácil de resolver y sobre todo muy rápido.

$\text{DirecciónElemN} = \text{DirecciónArray} + (\text{N} * \text{tamaño(TipoElem)})$

Donde DirecciónElemN es la dirección en memoria del elemento N del array, DirecciónArray es la dirección en memoria del array, N es el índice del elemento, y tamaño(TipoElem) es el tamaño del tipo del array (en caso de un array de integers el tamaño de integer = 4 bytes).



El esquema anterior nos muestra cómo se almacenan los elementos del array en memoria, uno pegado al otro, en bloque. Es esta característica la que los hace muy veloces a la hora de manejar muchos elementos.

Una breve aclaración antes de continuar. Esto es en caso de que el array comience con índice 0, de

lo contrario la fórmula varía un poquito:

$\text{DirecciónElemN} = \text{DirecciónArray} + ((N - X) * \text{tamaño}(\text{TipoElem}))$

Sin embargo, como ves, la velocidad varía muy poco, lo que hace que sin importar si empieza con índice 0 o no (aunque empezar con índice 0 como te habrás dado cuenta es aún más veloz -requiere menos cuentas-) los array son muy rápidos en comparación con otras estructuras de datos que si tenemos suerte analizaremos en detalle en futuros documentos.

## DESVENTAJAS

Una de las mayores desventajas que tienen los arrays tiene que ver con borrar elementos de la lista. Supongamos un array de 4 elementos y llenémoslo con números:

Indice	Llenamos	Borramos	Reordenamos
0	55	55	55
1	27	- (lo borré)	45
2	45	45	12
3	12	12	-

Al borrar el elemento de índice 1 todos los que quedan abajo (2, 3) hay que subirlos una posición. Esto que parece una estupidez es una de las grandes desventajas de los arrays, sobre todo cuando pensamos en arrays muy grandes.

Cuidado, aquí no es necesario llamar a ningún procedimiento Delete, ni Free ni nada que se le parezca, borrar un elemento se reduce simplemente a tomar todos los elementos que le siguen y "subirlos un escalon". Por ejemplo, en un array de 300 elementos borrar el segundo elemento implica mover desde el tercero hasta el 300 todos uno para arriba. De esta forma el tercero reemplaza al segundo, el cuarto al tercero y así.

**Nota:** es importante notar que todo esto se debe a que los arrays no pueden cambiar de tamaño en tiempo de ejecución, de lo contrario sí se podría realmente borrar el elemento de la lista. Lo que hemos hecho nosotros, en cambio es mover todos los elementos que le seguían uno para arriba y dejamos el último elemento en blanco. Esto puede considerarse una "borrada" pero no lo es realmente porque el array sigue siendo del mismo tamaño.

```
procedure TForm1.Button1Click(Sender: TObject);
var Q: array [0..99] of byte;
    i: integer;
begin
  {llenamos el array con todos 1}
  for i := 0 to 99 do
    Q[i] := 1;
  {borramos el nro 54}
  for i := 55 to 99 do
    Q[i-1] := Q[i];
  {aquí debería ir código para llenar el último elemento con
  basura o un valor que no tenga sentido para lo que estamos
  haciendo en nuestro programa. No olvides que Q[98] = Q[99]
  copia el valor del ultimo elemento en el anteúltimo,
  pero nos quedó pendiente borrar el valor que hay en Q[99]}
end;
```

Como ves, con el borrado, los arrays son, disculpen la palabra "un dolor de huevos". Pero las desventajas no terminan allí, hay algunas más serias. Claro, porque la anterior desventaja, como bien dijimos se debe nada más y nada menos a que los arrays no pueden cambiar de tamaño en tiempo de ejecución. Esto significa que en tiempo de diseño nosotros le damos un tamaño y así quedará por siempre. Si te fijas detenidamente en todo el código que hemos venido escribiendo verás que esto es así. Por ejemplo, en el último caso Q tiene 100 elementos, así lo hemos determinado en tiempo de diseño y así morirá.

Otra de las grandes desventajas de los arrays es la que también hemos dicho algo es que todos sus elementos deben ser del mismo tipo. O son todos integers, o son todos strings, o son todos bools, etc. Como es evidente, no siempre nosotros nos vamos a encontrar con la necesidad de crear listas de elementos del mismo tipo, quizá necesitemos una lista de strings, integers, bools, etc. todo mezclado. Bueno, aquí los arrays no pueden ayudarnos.

Sin embargo, a no desesperar, los arrays no son tan malos después de todo y además se han encontrado soluciones bastante elegantes para todos y cada uno de estos "problemas".

### Dynamic arrays

Hasta el momento hemos visto arrays estáticos, es decir, arrays a los que nosotros le dabamos un tamaño fijo, sin importar si realmente se iban a ocupar todos los espacios, o peor aún, siempre corriendo el riesgo de que el tamaño que nosotros le hemos dado al array sea insuficiente para la cantidad de datos a almacenar. Como ves las desventajas de los arrays estáticos son muchas, sobre todo por su rigidez.

Esta rigidez no viene de la nada, ocurre que la computadora debe saber cuanta memoria debe almacenar para ese array y por lo tanto debe conocer cuantos elementos tiene y de que tipo son. ¿Por qué? Bueno, para calcular el espacio de memoria que debe almacenar toma el nro. de elementos y lo multiplica por el tamaño del tipo del array, supongamos integer (4 bytes). Por ejemplo, para un array de 10 integers se reservarán 40 bytes.

Sin embargo, no siempre conocemos de antemano la cantidad de elementos que tendrá nuestro array y es aquí donde entran en escena los dynamic array. A partir de Delphi 4, se introdujeron en Object Pascal estos nuevos amiguillos muy útiles. La idea es seguir utilizando arrays pero con la ventaja de que su tamaño sea variable, es decir, que pueda cambiar en tiempo de ejecución según lo necesitemos, en definitiva, que fueran dinámicos. Para ir directo al grano, veamos un ejemplo de un dynamic array en acción:

```
var
  P: array of integer;
  i: integer;
begin
  SetLength(p, 5);
  for i := Low(p) to High(p) do
    p[i] := 3;
end;
```

Es fácil notar que hay varias diferencias. En primer lugar, notemos que NO HAY RANGO DE INDICES cuando declaramos el array, simplemente le decimos que es un array y especificamos el tipo, en este caso integer. Segundo, antes de usar el array utilizamos el procedure SetLength que hace el trabajo sucio por nosotros. Todo lo que debemos hacer es pasarle como parámetros el array (p) y el tamaño que deseamos que tenga. Nada más fácil, el resto es igual, a excepción de que en vez de utilizar los rangos en el for, utilizamos dos funciones Low y High a las que le pasamos como parámetro el array (p) y que en este caso nos devuelve 0 y 4 respectivamente.

**Nota:** Low y High no solamente pueden utilizarse con arrays. Si en vez del array p le hubiéramos pasado el integer i a Low, este nos devolvería -2147483648 y High devolvería 2147483647. En el caso de SetLength necesita como parámetro cualquier variable dinámica... en este caso nos tocó utilizarlo con un array pero podríamos haberlo utilizado pasándole un long string, que en definitiva no es otra cosa que un array dinámico de chars.

Hemos visto con los arrays estáticos que los había unidimensionales y multidimensionales, aquí ocurre lo mismo. Veamos un ejemplo.

```

var
  P: array of array of integer;
  i, x: integer;
begin
  SetLength(p, 5, 3);
  for i := Low(p) to High(p) do
    for x := Low(p[i]) to High(p[i]) do
      p[i, x] := 3;
    end;
  end;
end;

```

Las modificaciones son muy simples. En primer lugar una tabla de doble entrada, es decir un array bidimensional no es más que un array de un array. En el ejemplo anterior hay 5 filas y 3 columnas, para cada fila hay 3 valores, eso es un array de un array. Esto es exactamente lo que ponemos en la declaración del array, ya no es array of integer, sino array of array of integer.

Las demás modificaciones son más sencillas aún. En primer lugar agregamos un parámetro más a SetLength. Luego viene algo importante y en donde muchos suelen pisar el palito. Para el for de x, no debemos utilizar Low/High(p) sino más bien Low/High(p[i]), de esta forma le decimos a la computadora que recorra de la primera a la última columna. Esto tiene mucho sentido, con Low/High(p) le decimos recorré de la primera a la última fila y con Low/High(p[i]) le decimos ahora recorré de la primera a la última columna de la fila i.

### Casos especiales

Como se vió claramente, los dynamic arrays dejaron atrás todo ese "dolor de huevo" del que veníamos hablando y mató 2 pájaros de 1 tiro: el problema del borrado y el problema de poder cambiar de tamaño en tiempo de ejecución. Esto fue posible simplemente porque un problema (borrado) era consecuencia del otro (no poder cambiar de tamaño dinámicamente).

Sin embargo, nos han quedado algunas cosillas por resolver. Nuestros arrays, me refiero a los que hemos visto hasta ahora siguen necesitando elementos del mismo tipo. Veamos cual es el "work-around" de este problema.

### Variant arrays

Como no estoy seguro de que todos conozcan el tipo variant hagamos una breve introducción. En Delphi, así como hay strings, integer y demás también está el tipo variant que nos permite almacenar cualquiera de todos estos valores. Entonces, en vez de integer, string o cualquier otro, cada uno de los elementos de nuestro nuevo array será de tipo variant. De esta forma podremos almacenar datos de diferentes tipos dentro de un mismo array.

Para declarar un variant array no es posible hacerlo de la forma tradicional (o sea, var v: array [0..9] of variant), sino que debes utilizar alguna de las siguientes 2 funciones: VarArrayCreate o VarArrayOf. Por ejemplo,

```

var
  A: Variant;
begin
  A := VarArrayCreate([0, 4], varVariant);
  A[0] := 1;
  A[1] := 1234.5678;
  A[2] := 'Hello world';
  A[3] := True;
  A[4] := VarArrayOf([1, 10, 100, 1000]);
  WriteLn(A[2]); { Hello world }
  WriteLn(A[4][2]); { 100 }
end;

```

Los elementos del array pueden ahora ser accedidos utilizando los índices como lo veníamos haciendo hasta ahora. El primer parámetro que le pasamos a VarArrayCreate es el rango de los índices, eso es fácil de darse cuenta, y el segundo es el tipo base del array. Para obtener una lista de estos códigos debes buscar VarType en la ayuda de Delphi.

## Arrays dinámicos II, el regreso

Ya vimos lo que son los arrays dinámicos, arrays que pueden cambiar de tamaño en tiempo de ejecución. Sin embargo, para aquellos que no tienen Delphi 4 o posterior utilizar arrays dinámicos también es posible, sólo que estos arrays son "caseros" y no venían con Delphi.

**Nota:** todo lo que aquí hablemos es en carácter más bien de anécdota, ya que no creo que hoy alguien no disponga de Delphi 4 o posterior. Sin embargo, nunca está de más saber algo más.

Una implementación de los arrays dinámicos que proviene de los días anteriores aún a la programación orientada a objetos es crear un puntero a un array de 1 elemento. Luego, agrandamos o achicamos el array a nuestro gusto, de la misma forma como lo hicimos siempre que trabajamos con punteros. Veamos cómo hacerlo:

```
type
  PMyArray: ^TMyArray
  TMyArray: array [0..0] of TMyType;
...
var
  MyArray: PMyArray;
begin
  GetMem(MyArray, 42*sizeof(TMyType));
  {... usamos el array ...}
  FreeMem(MyArray, 42*sizeof(TMyType));
end;
```

Fácil. Simplemente quise agregarlo como una curiosidad que puede llegar a ser de alguna utilidad.

## Open arrays

Muy bien, hasta aquí hemos visto todos los tipos de arrays que existen. Ahora bien, ¿qué pasa si yo quiero pasar como parámetro de un procedimiento o una función un array? ¿se puede? Claro, ¿por qué no se va a poder? Sin embargo, debemos utilizar una sintaxis especial, veamos:

```
function Buscar(A: array of Char): Integer;
```

declara una función llamada Buscar que lleva como parámetro un array de tipo Char DE CUALQUIER TAMAÑO que devuelve un integer.

La sintaxis del ejemplo anterior y en general la de los open arrays es parecida a la de los dynamic arrays, pero definitivamente no significan lo mismo. El ejemplo anterior crea una función que puede llevar un array de cualquier tamaño de tipo char, incluyendo (pero no limitado a) los dynamic arrays. Para declarar parámetros que deben ser arrays dinámicos, debes especificar un identificador, por ejemplo:

```
type TDynamicCharArray = array of Char;
function Buscar(A: TDynamicCharArray): Integer;
```

Los open arrays siempre comienzan con índice 0, el segundo 1 y así. Las funciones Low y High, entonces, devuelven 0 y Length - 1 respectivamente. La función SizeOf devuelve el tamaño del array que fue pasado a la rutina.

Cuando pasas como parámetro por valor (sin el var adelante) un open array el compilador crea una copia del array dentro de la porción de la pila (stack) reservada para la rutina, por lo tanto, debes tener mucho cuidado de no pasar open arrays muy grandes para no crear un desbordamiento de pila, más conocido como stack overflow.

Lo recomendable cuando utilices open arrays es utilizar var delante, para que de esta forma no se cree ninguna copia y cualquier modificación que le hagas al array lo haga directamente en el original, y no en una copia. Veamos un ejemplo:

```
procedure Borrar(var A: array of Real);  
var  
    I: Integer;  
begin  
    for I := 0 to High(A) do A[I] := 0;  
end;
```

### Ultimas palabras

Bueno, eso ha sido todo. Espero que hayan disfrutado de este documento al igual que yo. Debo admitir que yo mismo he aprendido al escribirlo, eso es lo fantástico de esto, realmente siento que aprendo al explicar. Saber y saberlo explicar es saber 2 veces dice el proverbio. Bueno, espero que ahora uds. también puedan explicárselos a sus amigos y compañeros de ese vicio llamado Delphi.

Cualquier duda, sugerencia o comentario no duden en ponerse en [contacto](#).