7 ARRAYS

En capítulos anteriores se ve cómo emplear variables para guardar valores resultados intermedios en la ejecución de un programa. Una vez que se declara una variable de un determinado tipo (hasta ahora `int`, `long`, `float`, `double`, `bool`, `char`, `string`, `Stopwatch`, ...) a dicha variable se le puede asignar entonces un valor del mismo tipo el cual puede ser recuperado posteriormente cuando dicha variable se usa como un término parte de una expresión, cuando a su vez su valor se le asigna a otra variable o cuando su valor se le pasa directamente como parámetro a un método.

En principio no hay restricciones sobre la cantidad de variables que se pueden definir y usar en un programa. Pero ¿cómo conservar un conjunto de valores de un mismo tipo, sobre los cuales se quiere realizar luego un cierto procesamiento común, si no sabemos en el momento de escribir el código de cuántos valores estamos hablando en una ejecución real de dicho programa?

El siguiente fragmento de código calcula el promedio de 5 valores:

```
double a, b, c, d, e;
// Asignación de valores
// ...
double promedio = (a + b + c + d + e) / 5.0;
```

En este ejemplo se ha dado explícitamente un nombre a cada una de las 5 variables. Pero cómo hacerlo cuando la cantidad de valores no se conoce con anticipación porque depende de la propia ejecución de la aplicación, o aun cuando se conozca pero es una cantidad grande lo que hace engorroso y propenso a errores darle explícitamente un nombre a cada variable.

El código del Listado 7-1 lee interactivamente una secuencia de números enteros (hasta que el usuario de la aplicación decida terminar dando una línea en blanco). Pero imagine que además de calcular el promedio queremos listar cuáles son mayores que el promedio. ¿Dónde se han guardado esos valores? No puede pretenderse declarar una variable para cada uno porque no sabemos cuántos son. Tampoco se podría expresar el proceso repetitivo en un ciclo porque en cada iteración el nombre de la variable tendría que ser diferente.

```
float total = 0;
int cantidad = 0;

while (true)
{
    string linea = Console.ReadLine();
    if (string.IsNullOrEmpty(linea)) break;

    total += float.Parse(linea);
    cantidad += 1;
}
```

```
Console.WriteLine("El promedio es: {0:0.0}", total / cantidad);
```

Listado 7-1 Calcular el promedio de una secuancia de números

Una alternativa sería que una vez calculado el promedio la aplicación vuelva a leer de nuevo el conjunto de valores y vaya preguntando si cada uno es mayor que dicho promedio. Pero no es práctico pedirle al usuario que vuelva a teclear los valores, ni es sensato pretender que los vaya a volver a teclear de igual modo sin equivocarse. Pero, lo que es peor aún, los datos no siempre se producen por un usuario que interactivamente los teclea. Las fuentes de los datos que procesamos en una aplicación pueden ser muy diversas. Una secuencia de datos que se produjo hace un tiempo atrás no tiene que producirse de igual modo un tiempo después, imagine por ejemplo mediciones de temperatura realizadas por un sensor en un intervalo determinado de tiempo, si se vuelven a medir no tienen por qué dar igual.

Se necesita una forma en que una aplicación pueda guardar una cantidad no predeterminada de datos de modo de poderla procesar en otro momento de la misma aplicación. Para esto, casi todos los lenguajes de programación (C# entre ellos) cuentan con un recurso denominado array.[1]

7.1 Declaración de un array

Un array es una representación en memoria que permite almacenar un conjunto o secuencia de elementos de un mismo tipo. En este sentido puede considerarse como una colección de variables del mismo tipo, agrupadas bajo un único nombre común. En C#, el tipo de un array se define en base al tipo de los elementos que éste va a almacenar utilizando la siguiente sintaxis:

```
T[] nombreDelaVariable;
```

Donde `T` denota un tipo cualquiera (como los presentados en capítulos anteriores o como cualquiera de los que se verán en el resto de este libro). Los corchetes forman parte de la declaración del tipo, es decir, se dice que el tipo de la variable es `T[]` (léase array de `T`), mientras que el tipo de los elementos que el array contiene es simplemente `T`. Los siguientes son ejemplos de declaraciones de arrays:

```
int[] calificaciones;
float[] salarios;
string[] nombres;
Stopwatch[] cronos;
```

7.2 Creación de un array

El operador de creación new

Como un array corresponde a una sección continua de memoria, antes de poder almacenar elementos se debe indicar de qué tamaño se quiere sea el array, es decir

[1] A falta de consenso entre los diferentes países de habla hispana de cómo traducir array hemos preferido usar el término en inglés.

cuántos elementos puede almacenar. En C# esta operación se realiza con la siguiente sintaxis:

```
Tipo[] nombredelarray = new Tipo[longituddelarray];
```

El valor de `longituddelarray` debe ser un entero positivo (es decir puede ser una expresión de tipo `int`) y denota la cantidad de elementos que puede almacenar el array.

El tamaño físico real `que` tendrá el array en memoria depende a su vez del tamaño de los elementos almacenados lo que dependerá del tipo de estos.

Una vez creado un array, cada índice válido está asociado a un valor de tipo `Tipo`. Inicialmente, todos los índices almacenan el valor por defecto para el tipo correspondiente: 0 para los tipos numéricos, `false` para el tipo `bool`, y `null` para los tipos por referencia. En la Figura 7-1 se muestra una posible representación en memoria al crear un array para varios salarios. En esta representación se observa como la variable `salarios` consiste en una referencia que indica el lugar en memoria donde se encuentra el contenido el array. Como se puede apreciar, en la representación en memoria el array almacena no solo sus los elementos sino que también tiene información sobre su longitud y sobre el tipo de elementos que puede contener.

```
float[] salarios = new float[6];
```

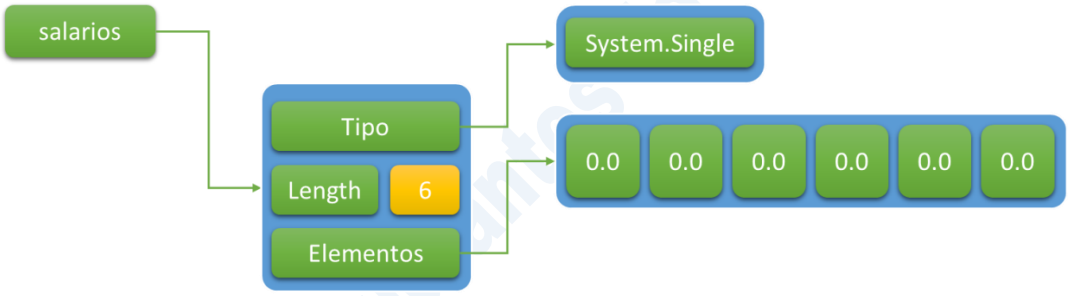

Figura 7-1 Representación en memoria de un *array* de `float` recién creado.

Por simplicidad en las ilustraciones el espacio de un elemento de un array se mostrará siempre como una celda (un rectángulo simple), lo que no debe confundirse con un byte de memoria. En este ejemplo si un valor de tipo `float` ocupa 4 bytes entonces el array `float[6]` ocupa 24 bytes

Construcción explícita de arrays

C# permite además la creación explícita de un array, indicando cada uno de los elementos que lo forman. Por ejemplo, para crear un array con las letras del alfabeto inglés se puede usar la siguiente sintaxis:

```
char[] alfabeto =
{
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
    'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
};
```

Figura 7-2 Representación en memoria del *array* `alfabeto`.

O también:

```
string[] los3Tenores = { "Luciano", "Plácido", "José" };
```

Para tipos más complejos también es posible crear un array de forma explícita si a su vez cada elemento se construye de la forma adecuada. Por ejemplo, si se cuenta con una clase `Punto` con el constructor correcto, el siguiente fragmento de código permitiría crear un array de puntos.

```
Punto[] vértices =
{
    new Punto(30, 30), new Punto(50, 40), new Punto(20, 10),
    new Punto(40, 50), new Punto(30, 50), new Punto(10, 40),
};
```

Evidentemente, esta forma de construcción explícita de un array solamente es útil cuando se conocen previamente todos los valores que se quieren guardar en el array.

7.3 Acceso a los elementos de un array

Si un array expresa de cierta forma a una "colección de variables", la gracia es que se pueda acceder y también modificar el valor contenido en cada una de esas "variables" que son parte del array. Cada una de estas variables se identifica con el nombre del array y entre corchetes [ ] un valor entero que se utiliza como un índice o posición dentro del array. Este índice comienza por `0`.

De modo que si `float[] salarios` es un array de tipo `float` entonces `salarios[0]` denota el salario guardado en la posición 0 del array y es un valor de tipo `float`, `salarios[1]` es el elemento en la segunda posición y así sucesivamente. La cantidad de elementos o longitud del array se puede conocer mediante la propiedad `Length`. De modo que `salarios.Length` da la longitud o tamaño del array `salarios`.

En C#, el compilador y el entorno de ejecución garantizan que se cumplan una serie de condiciones relacionadas con el intento de acceder a un elemento de un array.

Como mismo es ilegal asignar a una variable de tipo `int` un valor de tipo `string`, no se puede asignar a un elemento de un array un valor que no sea el del tipo de los elementos del array. Una asignación como `salarios[1] = 1500` es correcta porque el tipo de `salarios` es `float[]` y el valor `1500` puede verse como un `float`, pero es errónea `salarios[1] = "alto"` porque no se puede asignar un `string` a un `float`. Esto se controla por el propio compilador cuando se escribe el código.

Por otro lado durante la ejecución se controla que no se pueda acceder a ningún índice fuera de los "límites o rango" del array lanzándose una excepción de tipo `IndexOutOfRangeException` (índice fuera de rango). La expresión `salarios[-9]` provoca una excepción de ejecución. Un error común que cometen los principiantes es escribir `salarios[salarios.Length]` cuando se quieren referir al último elemento de un array, cuando realmente el último elemento del array está en la posición `salarios.Length-1`.

Acceso estructurado a los elementos de un array

La mayor utilidad de array consiste justamente en poder referirnos a los elementos del array no explícitamente con un valor entero constante que indique una posición concreta de un elemento del array explícito, sino con una variable (o en general una expresión) que tenga un valor `int`. Por ejemplo si `i` es de tipo `int` es válido escribir `salarios[i]` y `salarios[i+1]`.

Esta característica de los arrays es la que les da su mayor utilidad ya que un mismo código como `salarios[i]` puede estar haciendo referencia a los distintos elementos del array en dependencia del valor de `i`. De modo que variando adecuadamente el valor de `i` se puede acceder a diferentes elementos del array `salarios`

El problema anterior de leer una secuencia de salarios, calcular el salario promedio y luego listar aquellos salarios que son mayores que el promedio se puede ahora solucionar fácilmente con un código (Listado 7-2)

```
int cuantos = 0;
float[] salarios = new float[100];
float total = 0f;

for (int i = 0; i < salarios.Length; i++)
{
    string s = Console.ReadLine();
    if (string.IsNullOrEmpty(s)) break;

    salarios[i] = float.Parse(s);
    total += salarios[i];
    cuantos++;
}

float promedio = total/cuantos;

Console.WriteLine("El promedio es {0:0.00}", promedio);

for (int i = 0; i < cuantos; i++)
    if (salarios[i] > promedio)
        Console.WriteLine("El valor {0} en {1} es mayor que el promedio.",
                          salarios[i], i);
```

Listado 7-2 Cálculo de promedio de una secuencia números numéricos

Como se ve en este ejemplo se ha empleado un ciclo `for` para recorrer los elementos del array, la propia variable de control del ciclo `for` sirve de índice del array.

En este ejemplo como no se conoce la cantidad de valores que se entrarán, se ha creado el array con una longitud prevista (100 en este caso) pero a su vez se lleva la cuenta en la variable `cuantos` de cuál es la cantidad real de elementos leídos. Por lo tanto el ciclo `for` que recorre el array para determinar cuáles salarios son mayores que el promedio solo lo hace desde `0` hasta `cuantos-1` (la última posición realmente ocupada en el array.

El compilador controla que no se le asigne a un elemento del array un valor que no corresponda con el tipo del array, y el runtime controla en ejecución que no se trabaje con una posición inválida del array. Por otra parte al crear un array con `new` los

elementos del array se inicializan a un valor predeterminado según su tipo. Sin embargo no se puede controlar que usted no intente acceder al valor en una posición del array en la que previamente no haya guardado un valor porque ya eso depende de la lógica de uso que se haya hecho con dicho array.

Recorrer los elementos de un array

Un patrón frecuente de programación por el que se usan arrays es recorrer los elementos del array para hacer determinado procesamiento con ellos, como ha sido el ejemplo para determinar los que son mayores que el promedio.

El Listado 7-3 recibe un array de `float` y devuelve como resultado el promedio de todos los valores.

```
public static float Promedio(float[] a)
{
    float suma = 0;

    for (int i = 0; i < a.Length; i++)
        suma += a[i];

    return suma / a.Length;
}
```

Listado 7-3 Método para calcular el promedio de los valores en un array

Note cómo la variable de control del ciclo `for` comienza en el valor `0` (primer índice válido) y la iteración se mantiene mientras el índice actual sea menor que el valor `a.Length`. Es decir este método supone que se quieren procesar todos los elementos del array y por tanto el último índice válido es `a.Length - 1`. Es responsabilidad del código que llama al método asegurarse que todos los elementos del array se han ocupado con valores válidos. Si se quisiera disponer de un método al que se le indique la cantidad real de elementos ocupados en el array entonces podría escribirse éste como se muestra en el Listado 7-4

```
public static float Promedio(float[] a, int cuantos)
{
    float suma = 0;

    for (int i = 0; i < cuantos; i++)
        suma += a[i];

    return suma / cuantos;
}
```

Listado 7-4 Método para calcular el promedio de los valores en un array no completamente ocupado

Otro patrón común de trabajo con arrays es encontrar dentro de los elementos del array un valor que cumpla con cierta condición. Por ejemplo encontrar el máximo de un conjunto de valores. El método `Max` (Listado 7-5) devuelve el máximo de entre los valores del array. Se comienza suponiendo que el mayor valor es el que está en la posición `0` y lo ponemos en la variable `mayor`, luego se hace un ciclo desde la posición `1` hasta el último elemento del array, cada vez que nos encontremos un elemento que es mayor que

el hasta ahora calculado como mayor entonces se le asigna como nuevo valor a la variable `mayor`. Al terminar el ciclo el mayor de todos los elementos habrá quedado como valor de la variable `mayor`.

```
public static int Max(int[] a)
{
    int mayor = a[0];

    for (int i = 1; i < a.Length; i++)
        if (a[i] > mayor) mayor = a[i];

    return mayor;
}
```

Listado 7-5 Método que halla el valor mayor en un array

Note que si el array es de longitud 1, es decir tiene un solo elemento, el método funciona correctamente. El ciclo no se hace pero ese único elemento (que está en la posición `0`) es por lo tanto el mayor. Pero ¿qué pasa si el array es vacío, es decir si tiene longitud `0`?. En este caso sería incorrecto hacer `int mayor = a[0];` lo que provocaría la mencionada excepción de `IndexOutOfRangeException`. Lo cierto es que es incorrecto pretender calcular el máximo de un conjunto vacío y esto es responsabilidad del que ha invocado al método `Max`. El método `Max` podría protegerse de un tal error y dar una excepción más elocuente (Listado 7-6).

```
public static int Max(int[] a)
{
    if (a.Length == 0)
        throw new ArgumentException("Array vacío");

    int mayor = a[0];

    for (int i = 1; i < a.Length; i++)
        if (a[i] > mayor) mayor = a[i];

    return mayor;
}
```

Listado 7-6 Método Max mejorado

El compilador controla que los métodos sean invocados con valores que correspondan en el tipo y orden a los parámetros de la definición del método. Pero como este simple ejemplo demuestra, eso no basta para que un método sea invocado con los valores adecuados. Tal tipo de control solo puede hacerse en ejecución y es responsabilidad de quien escribe el código del método.

Accediendo a los elementos del array con índices no consecutivos

Aunque en muchos casos, como se ha visto en los ejemplos anteriores, los elementos del array son recorridos consecutivamente (es decir variando el índice de 1 en 1), puede ser útil trabajar con los arrays accediendo a sus elementos siguiendo otros criterios. Para ilustrar esto veamos un ejemplo que lista todos los números primos menores que cierto valor n.

El algoritmo más sencillo consistiría en iterar por todos los valores entre 2 y n, y comprobar en cada caso si dicho valor es primo (Listado 7-7).

```
static void ImprimirPrimos(int n)
{
    for (int i = 2; i < n; i++)
        if (EsPrimo(i))
            Console.WriteLine(i);
}
```

Listado 7-7 Listar números primos menores que n

Aunque este algoritmo es correcto, no es muy eficiente ya que por cada número se llama al método que averigua si éste es primo. La deficiencia fundamental radica en que para comprobar si un número k es primo no se aprovecha lo ya averiguado en la comprobación de si los menores que k son primos . Es decir que cada iteración se "olvida" todo lo que ha aprendido sobre el número anterior. Observe que una vez que se determina que un número k cualquiera es primo se pueden descartar como primos todos los múltiplos de k.

El siguiente algoritmo, que se conoce como la criba de Eratóstenes[2], está precisamente basado en esta idea. La idea original de Eratóstenes consistía en dibujar todos los números entre 2 y n en una tablilla, y tachar consecutivamente todos los múltiplos del menor número primo. El algoritmo comienza por el número 2 y procede a tachar todos los números pares. Una vez completado este paso, se busca el primer número no tachado (3 en este caso) y se procede a tachar todos los múltiplos de éste. En cada paso, una vez tachados todos los múltiplos de un número cualquiera k, el primer número no tachado tiene que ser primo. Esto se debe a que dicho número no puede tener ningún divisor, o hubiera sido tachado anteriormente (Figura 7-3).

Eratóstenes (Cirene, 276 a. C.1 – Alejandría, 194 a. C.) fue un matemático, astrónomo y geógrafo griego, de origen cirenaico. Estudió en Alejandría y durante algún tiempo en Atenas. Fue discípulo de Aristón de Quíos, de Lisanias de Cirene y del poeta Calímaco y también gran amigo de Arquímedes. En el año 236 a. C., Ptolomeo III le llamó para que se hiciera cargo de la Biblioteca de Alejandría, puesto que ocupó hasta el fin de sus días. Se le debe un procedimiento, conocido como la Criba de Eratóstenes, para obtener de un modo rápido todos los números primos menores que un número dado. La versión algorítmica de este procedimiento se ha convertido con los años en un método estándar para caracterizar o comparar la eficacia de diferentes lenguajes de programación

[2].

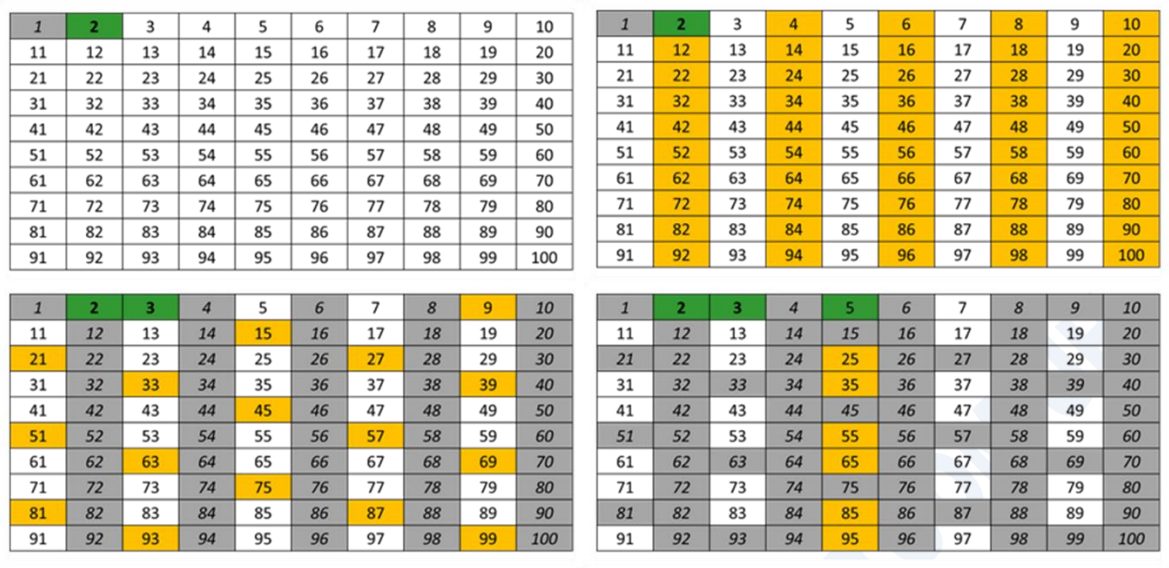

Figura 7-3 Recorrido con el método de Eratóstenes

Para hacer una implementación de este algoritmo utilizaremos un array de tipo `bool[]`. Aunque el algoritmo original funciona "marcando" los números primos como `true` y los no primos como `false`, para aprovechar que en C# todos los valores de un array de `bool` se inicializan a `false`, es conveniente marcar a los no primos como `true`; de tal modo que al finalizar el algoritmo si el valor del elemento en el índice `k` es `false` eso indique que el número `k` es primo y si es `true` eso indique que `k` no es primo (Listado 7-8).

```
static bool[] CribaDeEratostenes(int n)
{
    bool[] noprimos = new bool[n];

    noprimos[0] = true;
    noprimos[1] = true;

    for (int i = 2; i < n; i++)
        if (!noprimos[i])
            for (int mult = 2; mult*i < n; mult++)
                noprimos[mult*i] = true;

    return noprimos;
}
```

Listado 7-8 Cálculo de los primos menores que n por el método de Eratóstenes

Con este algoritmo implementado, enumerar los números primos se convierte en una tarea trivial (Listado 7-9).

```
static void ImprimirPrimos(int n)
{
    bool[] compuestos = CribaDeEratostenes(n);

    for (int i = 0; i < compuestos.Length; i++)
        if (!compuestos[i])
            Console.WriteLine(i);
}
```

Listado 7-9 Listar los primos menores que n

Usando el tipo `Stopwatch` presentado en los primeros ejemplos calcule el tiempo en ejecutar el método `ImprimirPrimos` del Listado y el del Listado que usa la `CribaDeEratostenes`. Para evitar la salida por consola, y solo medir el tiempo de cálculo, omita la instrucción `Console.WriteLine(i)`. Usted apreciará la diferencia en la medida que utilice valores grandes de *n*.

La razón que hace de la criba de Eratóstenes mucho más eficiente que una búsqueda exhaustiva radica en que aprovecha el conocimiento obtenido de las iteraciones anteriores para no hacer iteraciones innecesarias en la averiguación de si un número es primo. En última instancia, comprobar si un número es primo (inocente) o compuesto (culpable) consiste en intentar encontrar un testigo de su culpabilidad (encontrar un divisor, que lo convierte en compuesto), o fallar en el intento, aplicando la máxima de que todo número es inocente (primo) a menos que se demuestre lo contrario. Dado que la inmensa mayoría de los números son compuestos (es decir, culpables), es de esperar que tengan en común muchos divisores (compartan muchos testigos). El problema con la búsqueda exhaustiva es que una vez encontrado un testigo para cierto número compuesto, este testigo es inmediatamente olvidado, y tiene que ser encontrado nuevamente para cualquier otro número posterior del cual dicho testigo sea también divisor. Por este motivo el algoritmo exhaustivo está condenado a buscar a los mismos testigos una y otra vez.

La criba de Eratóstenes invierte el enfoque del problema. En vez de preguntar para un número, cuáles son los posibles testigos de su culpabilidad (los divisores de dicho número), averigua para cada testigo, cuáles números son los números contra los que puede testificar (los múltiplos de dicho testigo que por tanto serán compuestos). De esta forma descubre eficientemente todos los posibles números compuestos para cada divisor. Aunque el lector habrá apreciado que el método de la criba de Eratóstenes de todas formas "marca" varias veces a un mismo número compuesto, realmente desperdicia muchas menos operaciones que el algoritmo exhaustivo. Este tipo de cambio radical de enfoque para encontrar otra solución a un mismo problema ha probado ser efectivo en la solución de múltiples problemas (más sobre esto se estudia en los Capítulo 8 y 9).

7.4 Estructura en memoria de un array

Físicamente un array se representa en un segmento consecutivo de memoria RAM. En este sentido, los arrays son una estructura de datos que mimetiza la memoria física real. Por esta razón el acceso a una componente de un array a través de un índice (`a[i]`) demora el mismo tiempo independientemente del valor del índice, es decir demora lo mismo acceder al primer elemento `a[0]` que al último `a[a.Length-1]`. Decir el primero y el último es solo un eufemismo porque por lo general (aunque no es un requerimiento) cuando se recorren los elementos del array se empieza por el de la posición `0`. Esta característica de considerar que el tiempo de acceso a cualquier índice del array es esencialmente constante es clave para calcular el tiempo en que demoran muchos algoritmos que trabajan con arrays.

En C# los arrays en sí mismos se tratan por referencia pero el espacio que se separa para cada elemento del array depende de si el tipo de los elementos se trata por valor o por referencia.

Arrays de tipos por valor

Como se dice en el Capítulo 3 los tipos básicos como los numéricos, `bool` y `char` se tratan por valor. Un arrays de tipo por valor como `int[]` se representaría como se muestra en la Figura 7-4.

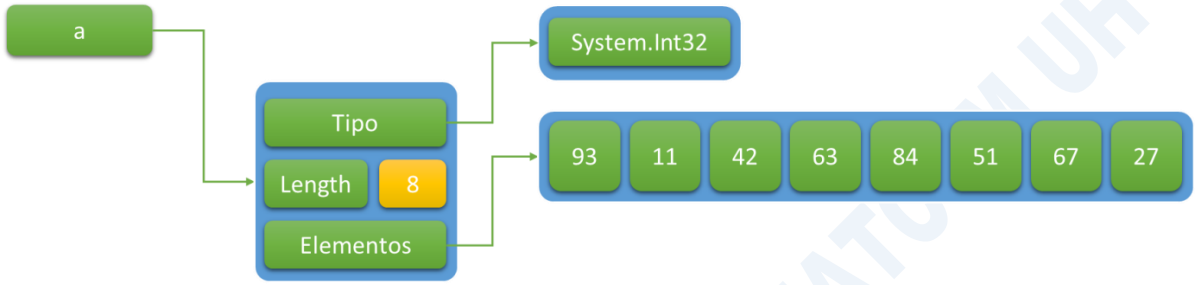

Figura 7-4 Representación en memoria de un *array* a de tipo int (por valor). En cada posición se encuentra almacenado el valor correspondiente.

El array como un todo se trata por referencia. Pero cada elemento del array se trata por valor. Por tal motivo, cuando se copia un elemento de un array a otro, se realiza una copia por valor (Figura 7-5).

```
int[] a = {1, 2, 3, 4};
int[] b = {5, 6, 7, 8, 9};

b[2] = a[3];
```

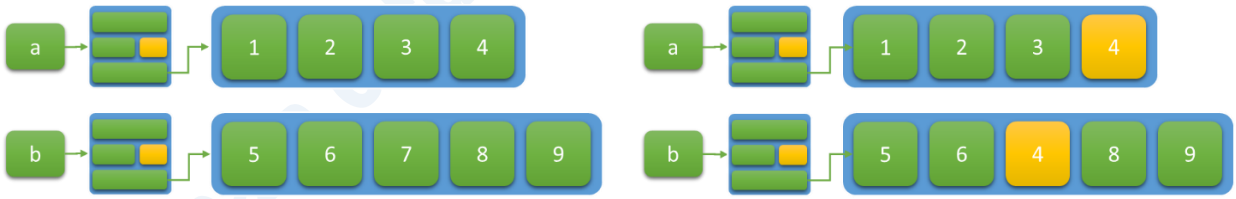

Figura 7-5: Representación simplificada en memoria de un *array* basado en tipos por valor. Al acceder a un índice particular del *array* `a` se obtiene una copia del valor correspondiente, que luego se pega en el *array* `b`. A partir de este punto ambos *arrays* tienen el mismo valor, pero son copias distintas. La modificación de uno de los valores no implica la modificación del otro. Por simplicidad se ha omitido en esta figura la representación de los metadatos del *array*.

Pero si ahora se hiciese `b = a;` se tendría en memoria la representación que se muestra en la Figura 7-6. En la variable `b` se ha asignado la referencia al valor de `a`. Es decir, no se copian todos los elementos del array uno a uno sino que solo se copian las referencias. De modo que en este caso luego de hacer la asignación anterior `a` y `b` se refieren al mismo array. Cualquier modificación de un elemento del array que se modifique a través de `a` se reflejará también a través de `b` (y viceversa).

Esta semántica de tratar predeterminadamente a los arrays por referencia es la más eficiente. Cuando se pasa un array como parámetro a un método no se copian todos los

elementos del array que se pasa como parámetro, sino que el método trabaja sobre los elementos del array original. De modo que el programador del método debe estar consciente de que cualquier modificación que haga de un elemento del array se reflejará en el array que se le pasó como parámetro. Aunque esto es la interpretación deseada para la mayoría de los problemas, también se dispone de un método `Copy` para copiar todo un array o un segmento de éste, elemento a elemento, pero en este caso lo debe indicar explícitamente el programador.

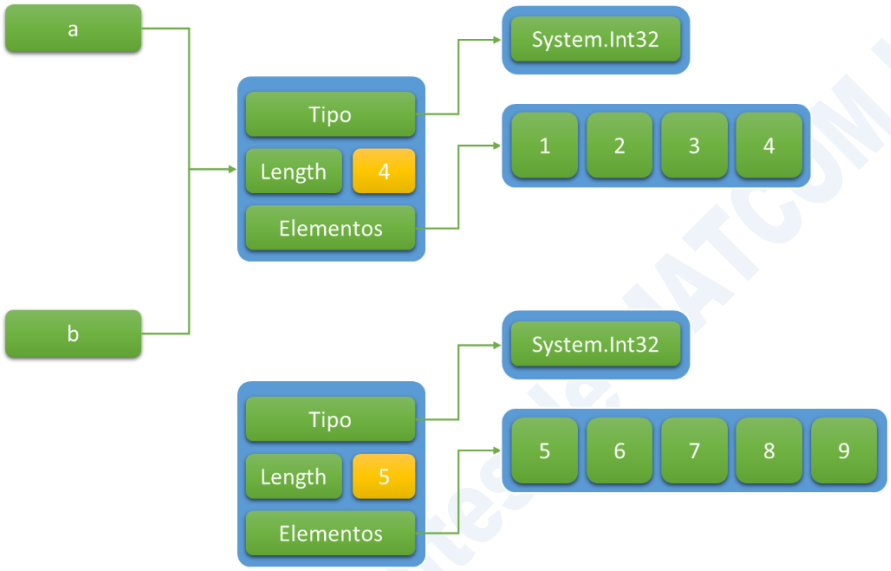

Figura 7-6 Al asignar a una referencia de un *array* b a otro variable del mismo tipo de *array* (a), realmente se realiza una copia de la referencia de a hacia b, efectivamente quedando ambas referencias ligadas al mismo *array* en memoria. El *array* que otrora era referenciado por b queda en memoria sin ninguna referencia que permita acceder a él, por lo que eventualmente será eliminado por el recolector de basura.

Arrays de tipos por referencia

Para los arrays de tipos por referencia (como los tipos `Stopwatch`, `Fecha`, `string` o en general cualquier tipo definido por una clase) cada elemento del array es a su vez tratado por referencia (Figura 7-7).

Debido a la semántica de copia por referencia, cuando se copia un elemento de un array de tipo por referencia a otro, realmente en ambos arrays quedará una referencia a la misma instancia (Figura 7-8). De tal modo que si se modifica dicho elemento desde uno de los arrays, se observará la modificación al ser accedido desde el otro array.

```
Fecha[] a = { new Fecha(25, "Enero", 2014), new Fecha(14, "Julio", 2015),
              new Fecha(8, "Mayo",2016), };
Fecha[] b = { a[2], new Fecha(7, "Marzo", 2099), };
```

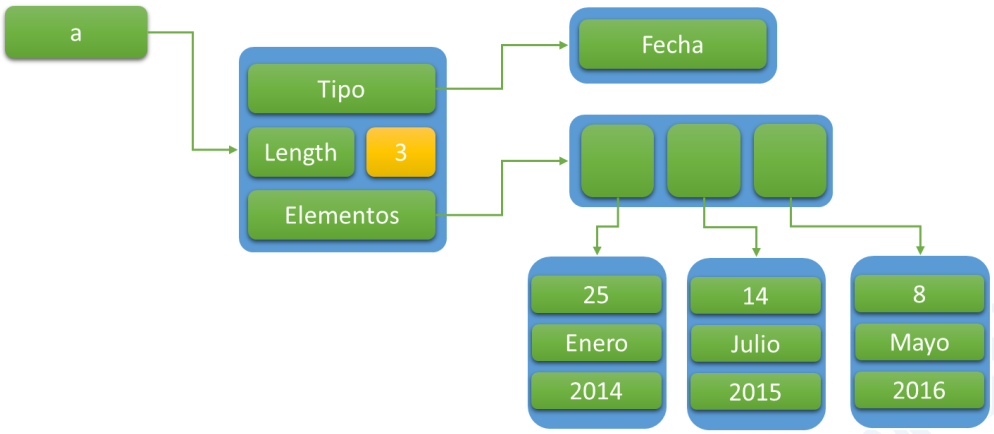

Figura 7-7: Representación en memoria de un *array* basado en tipos por referencia. Cada elemento del *array* contiene una referencia al lugar en memoria donde está el valor del elemento correspondiente.

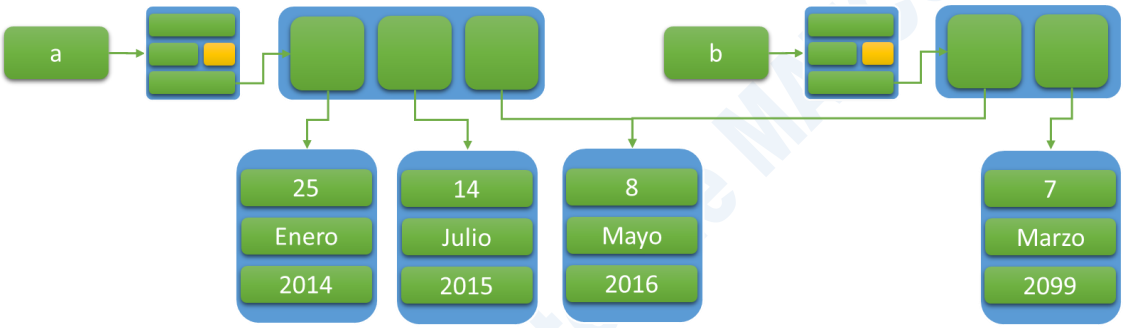

Figura 7-8: Representación simplificada en memoria de un *array* basado en tipos por referencia. El segundo *array* comparte una referencia con el primer *array*. Por tanto, si se modifica algún valor de la Fecha que hay en b[0], la modificación será apreciable al recuperar el objeto referenciado por a[0] (y viceversa). Por simplicidad se ha omitido en esta figura la representación de los metadatos del *array*.

7.5 Algunos de los métodos más usuales que trabajan con arrays

Con los arrays se pueden resolver ahora una gran cantidad de problemas que necesitan trabajar con colecciones de datos de un mismo tipo. En los siguientes epígrafes se verán ejemplos de soluciones a problemas donde se utilizan arrays.

Creación de un array con valores aleatorios

Con propósitos de hacer pruebas en muchos casos es conveniente crear arrays con valores aleatorios. El código del Listado 7-10 Creación de un array con valores aleatorios devuelve un array de longitud *n* con valores valores aleatorios de tipo `double` en el intervalo `[0, 1]`.

```
public static double[] ArrayAleatorio(int n)
{
    Random r = new Random();
    double[] a = new double[n];

    for (int i = 0; i < n; i++)
```

```
        a[i] = r.NextDouble();

    return a;
}
```

Listado 7-10 Creación de un array con valores aleatorios

Métodos que devuelven un valor

Entre las operaciones más comunes a realizar con un array, se encuentran los métodos que computan un valor a partir de los elementos del array. Tal fue el caso de los métodos `Promedio`, `Min` y `Max` que se vieron anteriormente.

Una de las operaciones frecuentes en la que participa un array es buscar un elemento dentro del array que cumpla con determinada condición.

El caso más elemental es averiguar si un determinado valor es igual a alguno de los que está en el array. El Listado 7-11 Determinar si un cierto valor está contenido en un array muestra una solución exhaustiva para hacer esto. El método recibe un array de enteros y un valor a buscar y con un ciclo `for` recorre todos los elementos del array retornando valor `true` si lo encuentra o valor `false` si llega hasta el final del array sin haberlo encontrado.

```
public static bool Contiene(int[] a, int x)
{
    for (int i = 0; i < a.Length; i++)
        if (a[i] == x)
            return true;

    //Si se termina el ciclo y se llega a esta instrucción
    // es porque el elemento no fue encontrado
    return false;
}
```

Listado 7-11 Determinar si un cierto valor está contenido en un array

Si además de saber si un valor se encuentra en el array nos interesa saber en la posición en que está, el método `Contiene` del Listado 7-12 Buscar la posición de la primera ocurrencia de un valor en un array puede cambiarse por el método `Pos` del Listado. El convenio adoptado para indicar que no fue encontrado es devolver `-1` ya que este no puede ser el valor de ninguna posición válida del array.

```
public static int Pos(int[] a, int x)
{
    for (int i = 0; i < a.Length; i++)
        if (a[i] == x)
            return i;

    return -1;
}
```

Listado 7-12 Buscar la posición de la primera ocurrencia de un valor en un array

Esta operación de búsqueda demora un tiempo proporcional a la cantidad de elementos que contiene el array, pues en el peor caso es necesario examinar todos los elementos del array. Es decir, si el tiempo que demora en ejecutarse cada iteración del ciclo es t_1,

entonces el tiempo que demora el método en su totalidad es aproximadamente $n \cdot t_1$, donde n es la cantidad de elementos del array. Esto se puede mejorar si se puede hablar de un orden entre los valores que hay en el array lo que se verá en la subsección siguiente.

Búsqueda Binaria

Si se puede garantizar que sobre el tipo de los elementos del array se puede hacer una comparación de orden, y que los valores que están en el array están ordenados (digamos que en orden creciente, es decir que si `i <= k` entonces se cumple que el valor en la posición `i` es `<=` que el valor en la posición `k`) entonces se puede realizar una implementación mucho más eficiente del método `Pos`, conocida como **búsqueda binaria**.

Aunque para todos los tipos se pueden aplicar las operaciones de comparación `==` y `!=`, por ahora solo sobre los valores de los tipos numéricos se pueden aplicar las operaciones de comparación de orden `<`, `>`, `<=` y `>=`. Pudiera aspirarse a comparar también valores de tipo `string` considerando una ordenación de tipo alfabética; sin embargo, los operadores anteriores no se aplican al tipo `string`. En el capítulo 11 se presenta cómo expresar una abstracción del concepto de **comparable** mediante las interfaces.

La idea del método de búsqueda binaria parte del supuesto de que el array está ordenado (digamos ascendentemente) y comienza por tomar el elemento que se encuentre en la posición mitad; si el elemento buscado es menor que el que está en la mitad entonces si se encuentra en el array tendrá que estar en la primera mitad; si es mayor que el que está en la mitad entonces de estar en el array tendrá que estar en la segunda mitad; y si no es ni menor ni mayor que el que está en la mitad entonces es porque es igual y habremos encontrado un valor igual al valor buscado. Ahora con el segmento mitad en cuestión aplicamos el mismo método hasta encontrar el elemento o hasta llegar a un segmento de un solo elemento que ya no se puede continuar dividiendo, lo que significará que el valor buscado no se encuentra en el array. El Listado 7-13 muestra una implementación de este método.

```
public static int PosPorBusquedaBinaria(int[] a, int x)
{
    // En cada iteración, inicio y fin determinan
    // el segmento de array donde se está realizando la búsqueda
    int inicio = 0;
    int fin = a.Length - 1;

    // Se repite mientras este segmento tenga algún elemento
    while (inicio <= fin)
    {
        int medio = (inicio + fin)/2; // Calcular la posición en la mitad
        if (x < a[medio])
        {
            // Buscar en los elementos de la mitad izquierda del segmento
            fin = medio - 1;
        }
        else if (x > a[medio])
        {
            // Buscar en los elementos de la mitad derecha del segmento
            inicio = medio + 1;
```

```
            }
            // El elemento es igual al del medio y por tanto se ha encontrado
            else return medio;
        }

        // Si se alcanza este punto es porque el valor x  no se encuentra en el
array
        return -1;
    }
```

Listado 7-13 Método de búsqueda binaria en un array

En la Figura 7-9 se muestra un ejemplo de una iteración del ciclo `while` en el método `PosPorBusquedaBinaria`. Como `x` es menor que `a[medio]`, y los elementos están ordenados, se puede garantizar que, en caso de existir, `x` tiene que encontrarse entre `inicio` y `medio - 1`. De esta forma realizando solamente una pregunta se puede eliminar toda una mitad del segmento analizado.

Figura 7-9: Ejemplo de una iteración del algoritmo de búsqueda binaria. Al comparar el valor a buscar (50) con el pivote (90) es posible desechar la mitad superior del *array*, pues, en caso de existir, este valor tiene que estar antes de la posición `mid`.

Intuitivamente observe que este método es más eficiente que el de buscar recorriendo todos los elementos del array. En cada iteración del ciclo o se ha encontrado el elemento y por tanto ya se termina, o el segmento se ha divido a la mitad para en la próxima iteración buscar solamente en una de esas dos mitades. Como en cada nueva iteración el segmento se ha divido a la mitad, en el caso peor en que el valor no se encuentre en el array el ciclo terminará cuando lleguemos a un segmento que ya no pueda dividirse (o sea cuando el segmento es de longitud 1, note que el criterio de repetición es `inicio <= fin`). De modo que si la longitud inicial del segmento es n (longitud del array o del segmento en que se quiere buscar) entonces la cantidad de veces que había que dividirlo a la mitad es $n/2/2/.../2$ hasta que sea igual a 1. Es decir se tendrán que hacer m iteraciones donde m tiene que ser tal que $n/2^m = 1$, por lo que $m = \log_2(n)$. Note que si n fuese por ejemplo `1,000,000` entonces m sería `20`, es decir que un máximo de un millón de iteraciones en la búsqueda secuencial ahora quedarían reducidos a solo un máximo de `20`. Es determinante en esta valoración el hecho de que en los arrays tiene el mismo costo en tiempo acceder a cualquiera de sus elementos sin importar la posición.

Las diferentes estrategias de solución de problemas pretenden, si fuese posible, reducir el costo de la aplicación de un método de solución en función de la cantidad de elementos a procesar. Aunque aquí se ha expuesto de forma intuitiva, este análisis del costo en tiempo es toda una disciplina formal de la computación que se conoce como **Complejidad Algorítmica**, cuyo contenido se sale del alcance de este libro, aunque se puede ver más sobre esto en los capítulos 8 y 9.

Métodos con array que devuelven un array

Como se ha dicho los arrays se trabajan por referencia de modo que cuando se hacen modificaciones sobre los elementos de un parámetro array esta modificación realmente recaerá en el array que se le pasó como parámetro concreto en la llamada. Esto es lo que ocurre predeterminadamente y por lo general es lo que se quiere. Sin embargo, hay situaciones en las que se desea no afectar el array original sino que el método devuelva un nuevo array. Por ejemplo, el método `SubArray` (Listado 7-14) devuelve un nuevo array formado por una copia de los valores array original indicados por el segmento que va de `inicio` a `fin`.

```
public static int[] SubArray(int[] origen, int inicio, int fin)
{
    int[] sub = new int[fin - inicio + 1];

    for (int i = 0; i < sub.Length; i++)
        sub[i] = origen[inicio + i];

    return sub;
}
```

Listado 7-14 Método subarray

Nótese que la inclusión del último índice (`fin`) implica que la cantidad de elementos del *sub*-array resultante es `fin - inicio + 1`. La iteración del ciclo `for` se realizar por los índices del array `sub`, y se calcula entonces cuál es el índice correspondiente en el array `origen`.

Otra operación común con arrays consiste en obtener un nuevo array con los mismos valores pero en orden inverso en las posiciones (Listado 7-15).

```
public static int[] Inverso(int[] array)
{
    int[] resultado = new int[array.Length];

    for (int i = 0; i < resultado.Length; i++)
        resultado[i] = array[array.Length - 1 - i];

    return resultado;
}
```

Listado 7-15 Método para invertir un array

Si el lector es observador se habrá percatado que tanto el método `SubArray` como el método `Inverso` se han programado para array de enteros; si se quisiera tener un método

`Inverso` para un array de `bool` habría que programar de nuevo un código similar (Listado 7-16).

```
public static bool[] Inverso(bool[] array)
{
    bool[] resultado = new bool[array.Length];

    for (int i = 0; i < resultado.Length; i++)
        resultado[i] = array[array.Length - 1 - i];

    return resultado;
}
```

Listado 7-16 Método para invertir un array de bool

Lo mismo habría que hacer si se quiere invertir un array de `string`, de `Fecha` o de cualquier otro tipo. Replicar código no es una buena práctica de programación y un buen lenguaje de programación debe ofrecer recursos para evitarlo. Los **métodos genéricos** para el trabajo con arrays que se ven en la sección a continuación son uno de estos recursos.

7.6 Métodos genéricos

Por lo general la lógica de funcionamiento de muchos de los métodos que trabajan con arrays no depende del tipo de sus elementos del array. Un problema con estos métodos es que no pueden ser reusado para arrays de otro tipo. Si se definió `Pos(int[] a, int x)` entonces si `colores` fuese de tipo `string[]` no se podría hacer la invocación con `Pos(colores, "rojo")`. El compilador daría error ya que a un parámetro a que es de tipo `int[]` no se le puede pasar un valor `colores` que es de tipo `string[]`, ni al parámetro `x` que es de tipo `int` se le puede pasar la cadena `"rojo"`.

Para propiciar la reusabilidad y evitar la replicación de código similar C# cuenta con un recurso lingüístico denominado **genericidad**, que permite definir clases y métodos reusables para múltiples tipos.

Una definición genérica de un método incluye uno o más **tipos genéricos**, que actúan como comodines en la definición del método y que luego tendrán que ser sustituidos en el código que use a dicho método. Así por ejemplo basándonos por los tipos reales en la genericidad el método `SubArray` se podría definir con independencia del tipo de los elementos del array (Listado 7-17).

```
public static T[] SubArray<T>(T[] origen, int inicio, int fin)
{
    T[] sub = new T[fin - inicio + 1];

    for (int i = 0; i < sub.Length; i++)
        sub[i] = origen[inicio + i];

    return sub;
}
```

Listado 7-17 Método Subarray genérico

Observe que donde antes en el método se usaba el tipo `int` haciendo que éste solo valiese para `int` ahora se está usando el nombre `T`. Este `T` es lo que se denomina un tipo genérico, y hay que expresarlo con la sintaxis `<T>` justo detrás del nombre del método. Esto le indica al compilador de C# que este método operará sobre cierto tipo `T`, cuyo tipo concreto tendrá que ser especificado en el código que necesite usar a dicho método. En este ejemplo al decir que el parámetro `origen` es de tipo `T[]` lo que se dice es que este método recibe un array de cierto tipo `T` (según el tipo que se especifique en lugar donde está la llamada), pero que devolverá entonces un array del mismo tipo `T`. De modo que la ejecución de un código como el siguiente garantiza en el valor devuelto en este caso es un array de tipo `int[]`.

```
int[] calificaciones = new int[20];
// Llenar el array
int[] sub = SubArray<int>(calificaciones, 4, 9);
```

De manera similar, puede invocarse el método con un array de tipo `float[]`, `Fecha[]`, o cualquier otro tipo, simplemente indicando en la invocación el tipo real que se quiere emplear.

A todos los efectos prácticos puede considerarse como si el compilador definiera textualmente un método por cada una de los tipos diferentes con los que se usa el método genérico. Sin embargo, la implementación es más eficiente y realmente el compilador evita, cuando es posible, la replicación del código generado por cada instanciación del método genérico.

Así mismo, el método `Pos` puede definirse también de forma genérica (Listado 7-18), pues en el proceso de búsqueda no interesa el tipo concreto del elemento que se desea buscar. La única operación que se aplica sobre el tipo genérico es la de comparación `==` de los elementos y esta operación se puede aplicar a valores de cualquier tipo.

Recuerde que la comparación `==` compara valores si los operandos son de un tipo por valor pero compara referencias si los operandos son de un tipo por referencia. El método genérico `Pos` (Listado 7-18) se basa en este criterio de comparación. Puede que usted desee que el método de buscar se base en un criterio de igualdad más personalizado en tal caso la implementación del método `Pos` debería basarse en una definición más genérica de igualdad (para ello existe el método `Equals` que se estudia con los conceptos de herencia y redefinición de métodos en el capítulo 10).

```
public static T Pos(T[] a, T x)
{
    for (int i = 0; i < a.Length; i++)
        if (a[i] == x)
            return i;
    return -1;
}
```

Listado 7-18 Método Pos genérico

Sería erróneo pretender definir un método genérico `PosPorBusquedaBinaria` para realizar la búsqueda binaria (Listado 7-19) ya que las operaciones `x < a[medio]` y `x > a[medio]` no se pueden aplicar a cualquiera que sea el tipo `T`.

```
public static int PosPorBusquedaBinaria<T>(T[] a, T x)
{
```

```
    // Este método es érroneo
    int inicio = 0;
    int fin = a.Length - 1;
    while (inicio <= fin)
    {
        int medio = (inicio + fin)/2;
        if (x < a[medio])
          // Esta operación no está definida para cualquier T
        {
            fin = medio - 1;
        }
        else if (x > a[medio])
          // Esta operación no está definida para cualquier T
        {
            inicio = medio + 1;
        }
        else return medio;
    }
    return -1;
}
```

Listado 7-19 Búsqueda binaria genérico

Por ejemplo es erróneo pretender usar este supuesto método genérico como se muestra a continuación porque para el tipo `string` no es aplicable la comparación `<` ni la comparación `>`.

```
string[] colores = { "azul", "blanco", "rojo" };
PosPorBusquedaBinaria<string>(colores, "negro");
```

Pero no se angustie el lector, esta "reusabilidad" en dependencia de cuál es el tipo con que se instancie el tipo genérico tiene solución y se verá más adelante en el Capítulo 11 con las interfaces.

Métodos genéricos que devuelven otro array

En ocasiones es necesario modificar la estructura misma del array para lograr el efecto de insertar, quitar o mover de lugar elementos del array.

Los arrays se crean con un tamaño y este no puede cambiar, de modo que el efecto de añadir, insertar o quitar elementos debe lograrse creando un nuevo array con el tamaño deseado y copiando hacia el nuevo array aquellos elementos del viejo array que se desea estén también en el nuevo array. Por ejemplo si se desea insertar un nuevo elemento en el array, sin sobrescribir ninguno de los elementos existentes, es necesario "hacer crecer" al array, para dar espacio al nuevo elemento. Igualmente, eliminar un elemento del array requiere "hacer decrecer" al array para “borrar” el espacio que dicho elemento ocupaba. Para el caso de insertar un elemento, es necesario crear un nuevo array con una longitud 1 más que el array original. Luego se copian hacia el nuevo array todos los elementos anteriores al índice donde se desea insertar, se ubica el nuevo elemento en dicho índice, y por último se copian desde el array original todos los elementos desde la posición del índice en adelante (Listado 7-20). La Figura 7-10 ilustra este proceso

```
public static T[] Insertar<T>(T[] origen, T x, int pos)
{
```

```
    T[] copia = new T[origen.Length + 1];

    for (int i = 0; i < pos; i++)
        copia[i] = origen[i];

    copia[pos] = x;

    for (int i = pos; i < origen.Length; i++)
        copia[i + 1] = origen[i];

    return copia;
}
```

Listado 7-20 Insertar en un array

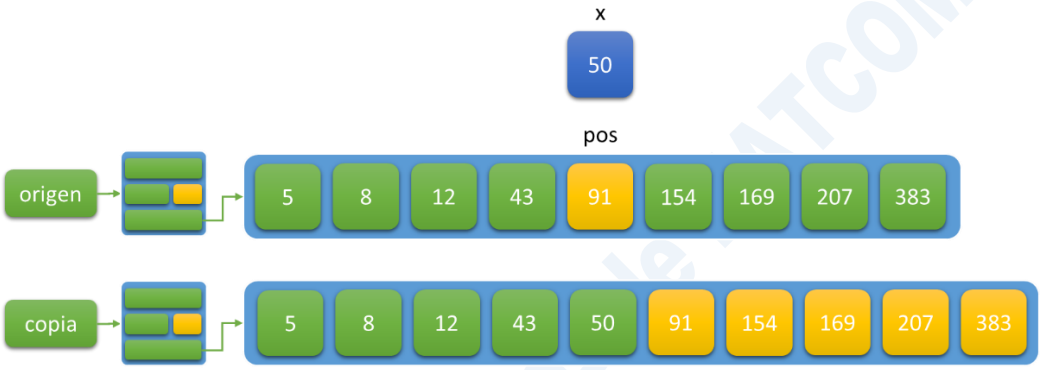

Figura 7-10: Inserción de un elemento en un *array*. Todos los elementos posteriores a la posición pos deben ser desplazados un índice hacia la derecha (final del *array*).

Para el caso de quitar un elemento, se procede de forma similar, primero copiar los elementos anteriores al índice cuyo elemento se desea quitar, y luego copiar los elementos posteriores. El elemento ubicado particularmente en el índice a eliminar no se copia (Listado 7-21).

```
public static T[] Eliminar<T>(T[] origen, int pos)
{
    T[] copia = new T[origen.Length - 1];

    for (int i = 0; i < pos; i++)
        copia[i] = origen[i];

    for (int i = pos; i < copia.Length; i++)
        copia[i] = origen[i + 1];

    return copia;
}
```

Listado 7-21 Eliminar un elemento de un array

Métodos genéricos que modifican el propio array

Otra categoría de métodos generales con arrays son aquellos que modifican el estado del array, es decir, cambian los valores de los elementos de un array, en el propio array sin

crear una copia. Note que por lo tanto estos métodos no devuelven un tipo `T[]` sino `void` para indicar que no devuelven directamente un nuevo valor.

Entre estos métodos están los que cambian de posición algunos valores dentro del array. El más elemental de estos métodos es el que intercambia los valores de dos posiciones (Listado 7-22).

```
public static void Intercambiar<T>(T[] a, int i, int j)
{
    T temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Listado 7-22 Intercambiar dos valores dentro de un array

Anteriormente se implementó el método `Inverso` devolviendo una copia del array con los elementos cambiados de orden. Una variante consiste en invertir el orden en que se encuentran los elementos dentro del propio array, empleando la operación de `Intercambiar` (Listado 7-23).

```
public static void Invertir<T>(T[] a)
{
    for (int i = 0; i < a.Length / 2; i++)
        Intercambiar(a, i, a.Length - i - 1);
}
```

Listado 7-23 Método Invertir usando Intercambiar

7.7 Ordenar un array

Posiblemente una de las operaciones más utilizadas que modifican un array es ordenar sus elementos según un criterio de comparación en el tipo de sus elementos. La "búsqueda en" y la "ordenación de" un array son componentes fundamentales de un sinnúmero de algoritmos más complejos.

Uno de los métodos más sencillos de comprender e implementar es el llamado **ordenación por mínimos sucesivos**. La idea fundamental de este método consiste en hacer un ciclo que compare el primer elemento con todos los restantes para ir dejando en el primero el menor de todos. Luego se repite el proceso con el elemento en la segunda posición y así sucesivamente hasta el elemento en la última posición que ya no tendrá que intercambiarse con nadie porque no hay ningún segmento del array posterior a él, de modo que al finalizar el array habrá quedado ordenado (Listado 7-24)

Al igual que en la búsqueda binaria a modo de ilustración este método se ha definido para un array de enteros int[] para poder aplicar la operación de comparación <. Este método podrá implementarse de forma genérica cuando se vea en el Capítulo 11 el tipo IComparable.

```
public static void Ordenar(int[] a)
{
    for (int k = 0; k < a.Length; k++)
        Intercambiar(a, k, PosDelMenor(a, k));
```

```
}

private static int PosDelMenor(int[] a, int inicio)
{
    int p = inicio;

    for (int i = inicio + 1; i < a.Length; i++)
        if (a[i] < a[p])
            p = i;

    return p;
}
```

Listado 7-24 Ordenar un array por mínimos sucesivos

Una demostración más formal del por qué este algoritmo funciona se puede ver en la sección a continuación mediante el concepto de **invariante de ciclo**.

Invariantes de ciclo

Una invariante de ciclo es una propiedad que se cumple antes y después de cada iteración de un ciclo. Si antes de comenzar una iteración `i`, se garantiza que la propiedad se cumple para `i-1`, y se demuestra que al terminar la iteración la propiedad se cumple también para el índice `i`, entonces por inducción completa se demuestra que la propiedad se cumple para todo `i`. Para aplicar esto en la demostración de que el método `Ordenar` trabaja correctamente lo que hace falta decidir cuál será el tal invariante de ciclo.

Consideremos como invariante que en cada iteración del ciclo `for` en el método `Ordenar` se cumple que los *k* menores valores del array se encuentran en las primeras *k* posiciones, y que ese *sub*-array entre `0` y *k* está ordenado. Si se demostrara que esta invariante se cumple entonces se puede asegurar que al terminar el ciclo `for` el array estaría ordenado por completo (pues el valor *k* sería igual a la longitud del array).

Para `pos` igual a `0` la propiedad se cumple trivialmente, pues el *sub*-array anterior a la posición `0` no existe (segmento vacío) y por tanto se cumple cualquier propiedad sobre sus elementos. Suponga entonces que en una iteración `i` cualquiera, se cumple que *sub*-array de `a[0]` a `a[i-1]` cumple con la invariante de ciclo. Para lograr que la invariante se cumpla entonces luego de la iteración `i` es necesario encontrar en el array el menor de los valores restantes, y ubicarlo en la posición `i`. Si asumimos que la invocación `PosDelMenor(a, k)` nos da la posición del menor de los restantes elementos, entonces la acción `Intercambiar(a, k, PosDelMenor(a, k))` lo intercambia con el de la posición `i` y por tanto se cumple el invariante para `i`. Claro aquí se está asumiendo que el método `PosDelMenor` es correcto, proposición cuya demostración le dejamos al lector.

El análisis de invariantes de ciclo es una herramienta formal poderosa para demostrar o refutar la corrección de muchos algoritmos iterativos. Para simplificar, en este texto no se pretende demostrar por esta vía la corrección de los ejemplos que se presentan, lo que por lo general se deja a la interpretación intuitiva del lector

7.8 La palabra reservada params

Hay métodos que quisiéramos aplicarlos a una cantidad variable de parámetros. Por ejemplo un método `Max` que devuelva el máximo entre varios valore enteros quisiéramos poder invocarlo en la forma:

```
Max(a,b), Max(a,b,c), Max(a, 100, b, c)
```

¿Cuántos métodos `Max` definir para tal fin si queremos que la cantidad de valores para hallar el mayor pueda ser cualquiera?

Una solución podría ser definir el método como se definió en el Listado X pero esto obligaría a hacer las invocaciones anteriores de la forma

```
Max(new int[]{a,b}), Max(new int[]{a,b,c}), Max(new int[]{a,100,b,c}),)
```

C# ofrece una mejor notación para lograr esto permitiendo asociar la palabra reservada `params` a un parámetro que sea de tipo array. El método `Max` se escribiría entonces con el encabezado:

```
public static int Max(params int[] a)
```

En este caso sí se puede escribir:

```
Max(a,b), Max(a,b,c), Max(a, 100, b, c)
```

Siendo el compilador quien construye de forma transparente un array con los parámetros concretos de cada llamada y es dicho array lo que realmente se le pasa al parámetro formal especificado con `params`.

Este tipo de recursos suele denominarse **azúcar sintáctica**, porque aunque no añade nada que no pueda lograrse con otros recursos disponibles, ofrece sin embargo una notación más simple, más elegante, más expresiva para lograr algo.

En particular, solamente puede haber un modificador `params` en la signatura de un método, y éste tiene que estar asociado al último parámetro del método. Es decir, una definición como la siguiente es inválida:

```
static void F(params int[] a1, params int[] a2)
```

Por supuesto, en este caso es comprensible la restricción, pues en un llamado particular a este método al compilador le sería imposible determinar qué valores asociar a cuál array. Una llamada como `F(1, 2, 3, 4)` sería entonces ambigua de interpretar, ¿cuáles de los parámetros `1, 2, 3` y `4` se le pasan como array a `a1` y cuáles a `a2?;`

Sin embargo, la siguiente signatura, aunque no es ambigua, también es inválida:

```
static void F(params int[] array1, params string[] array2)
```

Aunque en este caso el compilador pudiera decidir ante cualquier posible llamada, cómo repartir los argumentos[3].

A pesar de las restricciones a su uso, el modificador `params` es una construcción sintáctica muy útil en un número elevado de casos comunes y es recomendable su

[3] Aparentemente los diseñadores de C# decidieron que los posibles beneficios de permitir esto no se justificaban con la complejidad de implementarlo.

empleo siempre que sea posible, y la semántica del método implementado lo justifique (por ejemplo métodos que se aplican a una cantidad de variable de valores de un mismo tipo como es el caso del ejemplo `Max`).

7.9 El tipo System.String

En capítulos anteriores se ha trabajado con el tipo básico `string` cuyos valores son cadenas de texto. De cierta forma se puede ver una cadena (string) como un array de tipo `char[]` porque se puede iterar por ella y consultar sus elementos a través de un índice de la misma forma que con un array, pero con la particularidad de que a diferencia de un array, a una cadena no se le pueden modificar sus elementos, es decir, que si `s` es de tipo `string` es un error escribir `s[i] = 'a'`. Por eso en el Capítulo 3 se dice que los valores de tipo `string` son **inmutables**, es decir, una vez creados no se pueden modificar.

La palabra clave `string` no es más que una forma abreviada de C# para denotar al tipo `System.String`. Este tipo tiene métodos para pasar una cadena a un array de caracteres (`char[]`) y viceversa. El constructor

```
public String(char[] value);
```

crea un objeto de tipo `string` con la misma longitud y caracteres iguales a los valores del array que recibe como parámetro. Y el método

```
public char[] ToCharArray();
```

devuelve un array de char con los caracteres iguales a los del string.

Puesto que las cadenas son inmutables, la clase `System.String` tiene fundamentalmente dos tipos de métodos: los que hacen consultas a una cadena devolviendo algún valor y los que devuelven una nueva cadena.

Un ejemplo de método que devuelve un valor a partir de valores de tipo `string` es el método `int IndexOf(string value)`, que devuelve el índice de la primera ocurrencia de la cadena `value` dentro de la cadena a la que se le aplica el método. Como ilustración del acceso a las componentes de un valor `string` se muestra una posible implementación de este método (

Listado 7-25).

Como no se dispone del código fuente para modificar a la clase `String` a modo de ilustración el código del método `IndexOf` se ha definido como un método estático.

```
private static int IndexOf(string s, string value)
{
    for (int i = 0; i < s.Length; i++)
        if (IsSubstring(s, value, i))
            return i;

    return -1;
}
```

Listado 7-25 Posible implementación de IndexOf

```
private static bool IsSubstring(string s, string value, int from)
{
    if (value.Length > s.Length - from)
        return false;

    for (int k = 0; k < value.Length; k++)
        if (s[from + k] != value[k])
            return false;

    return true;
}
```

Listado 7-26 Método IsSubstring

Un ejemplo de método sobre string que devuelve un string pero sin modificar el string original es el método `string Substring(int startIndex, int length)`. El Listado 7-27 nos muestra una posible implementación de este método. Note cómo para ir creando la nueva cadena se ha utilizado un array `char[]` para poder asignar caracteres a sus elementos y luego a partir de este array se crea un objeto de tipo `string` que es el que finalmente se devuelve.

```
private static string Substring(string s, int startIndex, int length)
{
    if (startIndex + length > s.Length)
        throw new ArgumentOutOfRangeException("Longitud fuera de rango");

    char[] result = new char[length];

    for (int k = 0; k < length; k++)
        result[k] = s[startIndex + k];

    return new string(result);
}
```

Listado 7-27 Posible implementación de Substring

Esta operación podría haberse implementado utilizando solamente valores de tipo `string` y usando en este caso la operación `Concat`, pero hubiese sido más ineficiente. Invitamos al lector a hacer esta implementación.

7.10 El tipo System.Array

En la biblioteca estándar de .NET, en el espacio de nombres `System`, existe una clase denominada `Array` que contiene un conjunto de métodos útiles para trabajar con arrays. En esta sección se muestran algunos de los métodos más importantes de esta clase.

El método `Array.Copy` permite copiar una sección de un array hacia otro array. Hay varios métodos `Copy`, el más general tiene la forma:

```
static void Copy(Array source, int srcIndex, Array destination, int destIndex,
                 int count)
```

La semántica de este método consiste en copiar una cantidad `count` de elementos del array `source` (a partir de la posición `srcIndex`) y) hacia el array `destination` (a partir de la posición `destIndex`).

El método `Array.Sort` permite ordenar los elementos de un array a partir de diferentes criterios de ordenación. Este método cuenta con varias sobrecargas, que permiten especificar el criterio de ordenación en disímiles formatos.

Una de las versiones del método `Sort` es

```
static void Sort(Array array, int index, int length)
```

Estos métodos ordenan los elementos de un *sub*-array por el criterio de ordenación definido en el tipo correspondiente al array (en el Capítulo 11 sobre interfaces se verán los convenios adoptados para dotar a un tipo de un criterio de comparación y por tanto ser susceptible de utilizar en una ordenación).

Los parámetros `index` y `length` determinan respectivamente el inicio y la cantidad de elementos del *sub*-array que se quiere ordenar.

Otra versión del método Sort es

```
static void Sort(Array keys, Array items, int index, int length)
```

Lo interesante de este método es que permite ordenar el array `items` según el criterio de ordenación que se aplique al array `keys` cuyos valores actúan como "llaves". Es decir, se ordena el array `keys` y correspondientemente se cambian las posiciones de los elementos en el array `items` según como quedaron las del array `keys`. Por ejemplo, si se tiene un array de instancias de una clase `Persona`, la cual tiene una propiedad `Nombre` de tipo string, una manera de lograr ordenar el array `estudiantes` de tipo `Persona[]` se muestra en el Listado NNN, de este modo se está aprovechando la capacidad de emplear el método `Array.Sort` sobre el array `nombres` que es de tipo `string[]`.

```
Persona[] estudiantes = new Persona[n];
// ... Llenar el array de personas

// Formar un array nombres con los nombres de cada persona
string[] nombres = new string[alumnos.Length];

for (int i = 0; i < nombres.Length; i++)
    nombres[i] = alumnos[i].Nombre;

// Ordenar el array de estudiantes según su nombre
Array.Sort(nombres, estudiantes);
```

La clase `Array` también contiene métodos para buscar elementos en un array (usando tanto con el algoritmo de búsqueda lineal como con el algoritmo de búsqueda binaria), para invertir el orden de un array, y para aumentar el tamaño de un array, entre otros. Se recomienda al lector ver la documentación de Visual Studio y familiarizarse con esta clase, que puede servir de ayuda para la implementación de otros métodos de gran utilidad.

7.11 Tipos que usan arrays en su implementación

En la programación moderna muchas veces los arrays no se utilizan directamente en la solución directa de un problema, sino que forman parte de la implementación de otros tipos más abstractos cercanos al dominio del problema que se quiere solucionar. De hecho, el propio tipo `string`, que ya viene integrado en C# y otros lenguajes, es un ejemplo que evita que muchas aplicaciones tengan que trabajar con arrays de `char`. En esta sección se verán algunos ejemplos que ilustran el uso de arrays en la implementación de otros tipos.

El tipo Polinomio

En términos matemáticos, un polinomio es una función construida como suma finita de monomios, que a su vez constituyen expresiones de la forma:

$$a_i \cdot x^i$$

Un polinomio se dice de grado n si n es la potencia mayor de los términos que tienen coeficiente diferente de 0.

$$P_n(x) = \sum_{i}^{n} a_i \cdot x^i = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_n \cdot x^n$$

La clase `Polinomio` del Listado 7-28 representa a polinomios cuyos términos tienen coeficientes de tipo `double`. El array `coeficientes` se ha utilizado para representar los coeficientes de modo que el elemento en la posición `0` se interpretará como el a_0, es decir el coeficiente del término x^0 y el elemento en la posición m será el coeficiente del término x^m. El grado del polinomio sería entonces la longitud de dicho array menos 1 (propiedad `Grado`).

```
public class Polinomio
{
    private double[] coeficientes;

    public Polinomio(params double[] coeficientes)
    {
        this.coeficientes = coeficientes;
    }

  public Grado
  {
    get { return coeficientes.Length-1; }
    }
    //...
}
```

Listado 7-28 Clase Polinomio

Un polinomio nos representa una curva en el plano. La operación fundamental a realizar con un polinomio es la de evaluar el polinomio para un cierto valor x de modo que el resultado sea la coordenada y de la curva en el plano.

Una implementación sencilla de este método consiste en iterar por todos los coeficientes, multiplicar cada uno por la potencia correspondiente e ir sumando esos resultados (Listado 7-29).

```
public double Evaluar(double x)
{
    double total = 0;

    for (int i = 0; i < coeficientes.Length; i++)
        total += coeficientes[i] * Math.Pow(x, i);

    return total;
}
```

Listado 7-29 Método para evaluar un polinomio en un valor

Se ha utilizado aquí la función `Pow` de la clase `Math`, que eleva un número `x` a una potencia `i`. Aunque esta implementación es correcta, esto se puede implementar de modo más eficiente. La deficiencia de esta versión consiste en que nuevamente estamos ante una situación en la que en cada iteración se "olvida" todo lo aprendido para calcular la potencia en la iteración anterior. La operación `Math.Pow(x, i)` para calcular x^i implica multiplicar i veces el valor de x. Sin embargo, en la iteración anterior ya se realizó una operación similar, para calcular x^{i-1}. En total, este método realiza una cantidad de multiplicaciones proporcional a n^2, cuando bastaría n multiplicaciones para conseguir el mismo resultado si se utilizase el código del Listado 7-30.

```
public double Evaluar(double x)
{
    double total = 0;
    double potencia = 1;

    for (int i = 0; i < coeficientes.Length; i++)
    {
        total += coeficientes[i] * potencia;
        potencia *= x;
    }
    return total;
}
```

Listado 7-30 Método Evaluar mejorado

Otra operación sencilla sobre polinomios es la suma de polinomios (Listado 7-31). La resta se implementaría de modo similar.

```
public Polinomio Sumar(Polinomio otro)
{
    double[] suma = new double[Math.Max(coeficientes.Length,
                                        otro.coeficientes.Length)];

    for (int i = 0; i < coeficientes.Length; i++)
        suma[i] = coeficientes[i];
```

```
    for (int i = 0; i < otro.coeficientes.Length; i++)
        suma[i] += otro.coeficientes[i];

    return new Polinomio(suma);
}
```

Listado 7-31 Suma de polinomios

Sin embargo, hay un detalle importante a tener en cuenta y es cuál es el grado del polinomio resultante, porque en la suma se podrían anular coeficientes. Incluso todos los coeficientes podrían quedar en `0` si por ejemplo se resta un polinomio consigo mismo.

Aunque este problema puede ser resuelto directamente en los métodos `Sumar` o `Restar`, es más conveniente resolverlo una sola vez en el constructor de la clase. De esta forma no solo se simplifica el código, sino que se garantiza que un polinomio construido potencialmente con mayor grado que el que en realidad tiene (debido a que los mayores coeficientes sean `0`) se almacene con el grado correcto. La modificación al constructor consiste en realizar un ciclo para determinar el mayor multiplicador distinto de 0, y solamente copiar al array interno los coeficientes necesarios. El código del constructor del Listado 7-28 tendría que sustituirse por el que se muestra en el Listado 7-32.

```
public Polinomio(params double[] coeficientes)
{
    int grado = 0;

    for (int i = coeficientes.Length - 1; i >= 0; i--)
        if (coeficientes[i] != 0)
        {
            grado = i;
            break;
        }

    this.coeficientes = new double[grado + 1];

    Array.Copy(coeficientes, 0, this.coeficientes, 0, grado + 1);
}
```

Listado 7-32 Nuevo constructor para la clase Polinomio

Finalmente, considere la operación de multiplicación entre dos polinomios. Los términos a_i y b_j, al ser multiplicados, generan un término $a_i \cdot b_j$ de grado $k = i + j$. Existen múltiples pares de términos que aportan al término de grado k en el polinomio resultante, a saber, todos los pares de términos cuya suma de grados sea k. El algoritmo consiste en iterar por todos los pares de términos en ambos polinomios e ir acumulando en el polinomio resultante el término correspondiente (Listado 7-33).

```
public Polinomio Multiplicar(Polinomio otro)
{
    double[] producto = new double[coeficientes.Length +
                                   otro.coeficientes.Length];

    for (int i = 0; i < coeficientes.Length; i++)
```

```
            for (int j = 0; j < otro.coeficientes.Length; j++)
                producto[i + j] += coeficientes[i] * otro.coeficientes[j];

        return new Polinomio(producto);
    }
```

Listado 7-33 Multiplicación de polinomios

Afortunadamente, si la clase `Polinomio` incluye un constructor como el del Listado 7-32 entonces no hay que controlar aquí que algunos términos se hayan cancelado al hacer la multiplicación. Se propone al lector implementar la división entre polinomios y la derivada de un polinomio.

El tipo Polígono

Un tipo que se trabaja mucho en las aplicaciones de gráficos son los polígonos. Un polígono en términos geométricos está definido por una secuencia de vértices, que forman una figura cerrada. En términos computacionales un polígono podría representarse usando un array de vértices (instancias de tipo `Punto`), de forma que las aristas van del vértice en la posición *i* del array al vértice siguiente (el de la posición *i+1*) y finalmente del último al primero (al de la posición 0) (Listado 7-34).

```
public class Polígono
{
    private Punto[] vertices;

    public Poligono(params Punto[] puntos)
    {
        this.vertices = puntos;
    }
}
```

Listado 7-34 Clase Polígono

Note que al estar el parámetro del constructor definido de la forma `params Punto[] puntos`, esta definición permitiría construir polígonos con 0, 1 ó 2 vértices, lo que en términos geométricos es un error.

Esto podría controlarse verificando en ejecución que el array `puntos` tenga longitud mayor o igual que 3 (Listado 7-35).

```
public Poligono(params Punto[] puntos)
{
    if (puntos == null || puntos.Length < 3)
        throw new ArgumentOutOfRangeException("Se necesitan al menos 3
puntos.");

    this.vertices = puntos;
}
```

Listado 7-35 Constructor para Poligono

Pero una debilidad de esta forma de garantizar la consistencia radica en que el chequeo se hace en tiempo de ejecución. Es posible forzar sintácticamente a que un polígono se tenga que construir aportando al menos 3 puntos (Listado 7-36) de modo que cualquier violación pueda ser detectada por el compilador.

```
public Poligono(Punto x1, Punto x2, Punto x3, params Punto[] puntos)
{
    vertices = new Punto[puntos.Length + 3];
    vertices[0] = x1;
    vertices[1] = x2;
    vertices[2] = x3;

    for (int i = 0; i < puntos.Length; i++)
        vertices[i + 3] = puntos[i];
}
```

Listado 7-36 Otro constructor para Polígono

La operación más sencilla de interés en un polígono es calcular su perímetro. La implementación de una propiedad `Perimetro` consiste en sumar las distancias entre todos los pares de vértices consecutivos, sin olvidar la arista que va del último al primer vértice (Listado 7-37). Si se asume la existencia de un método `Distancia` en la clase `Punto` que calcula la distancia euclidiana entre dos puntos del plano R^2.

```
public double Perimetro
{
    get
    {
        double p = puntos[puntos.Length - 1].Distancia(puntos[0]);

        for (int i = 0; i < puntos.Length - 1; i++)
            p += puntos[i].Distancia(puntos[i + 1]);

        return p;
    }
}
```

Listado 7-37 Método Perimetro

Calcular el área interna del polígono ya es algo más complicado y desviaría la atención del lector del tema que nos ocupa en este capítulo que son los arrays. Sin embargo, si se asume que el polígono es convexo existe un algoritmo sencillo basado en una triangulación del polígono (Figura 7-11).

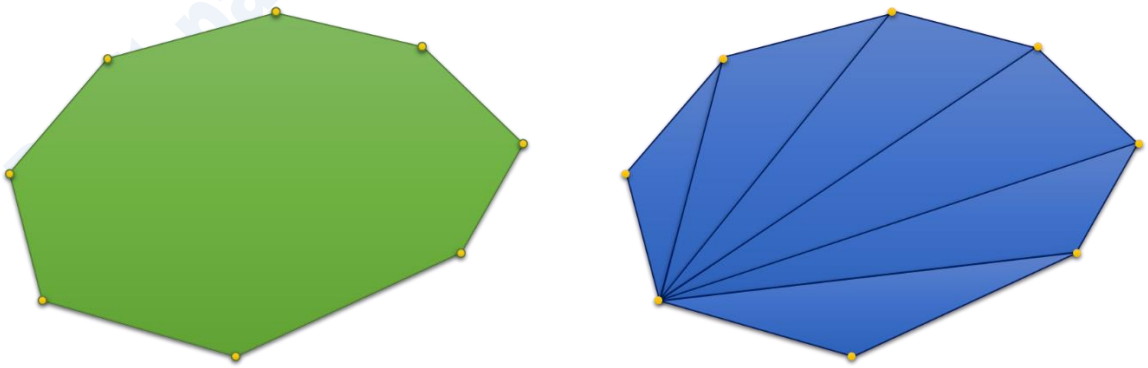

Figura 7-11: Una posible triangulación de un polígono convexo, que muestra como calcular el área interna mediante la suma de las áreas de los triángulos que se forman.

El algoritmo consiste en construir triángulos entre tríos de vértices sucesivos, que cubran toda el área del polígono. Se fija un vértice inicial que sea común a todos los triángulos, y de ahí en adelante se va acumulando la suma del área de cada triángulo (Listado 7-38 Área de un polígono). El área de cada triángulo puede calcularse empleando la fórmula de Herón:

$$A_{\Delta} = \sqrt{\frac{p}{2} \cdot (p - d_{1,2}) \cdot (p - d_{2,3}) \cdot (p - d_{3,1})}$$

Donde p representa el perímetro del triángulo, y $d_{i,j}$ la distancia entre el punto x_i y el punto x_j respectivamente.

```
public double Area
{
    get
    {
        double a = 0;

        for (int i = 0; i < puntos.Length - 3; i++)
            a += AreaTriangulo(puntos[0], puntos[i + 1], puntos[i + 2]);

        return a;
    }
}

private double AreaTriangulo(Punto p1, Punto p2, Punto p3)
{
    double d12 = p1.Distancia(p2);
    double d23 = p2.Distancia(p3);
    double d31 = p3.Distancia(p1);
    double p = d12 + d23 + d31;

    return Math.Sqrt((p/2)*(p - d12)*(p - d23)*(p - d31));
}
```

Listado 7-38 Área de un polígono

Realmente no es estrictamente necesario que el polígono sea convexo. Basta con que al realizar la triangulación, las aristas de todos los triángulos queden dentro del polígono. Aunque es posible concebir polígonos no convexos que cumplan esta condición, determinar que el algoritmo es correcto para este tipo de polígonos depende de cuál vértice se escoja para comenzar la triangulación. Por motivos de simplicidad, para el capítulo se ha restringido el algoritmo a polígonos estrictamente convexos.

Listas

Como se ha dicho, los arrays son estructuras básicas cuya mejor cualidad es que representan a una zona de memoria consecutiva para almacenar valores de un mismo tipo que pueden ser accedidos de manera directa mediante un índice y que el costo de este acceso no depende del valor del índice. Como se ha visto en la sección anterior, los

arrays se utilizan también para implementar otros tipos más complejos orientados a la solución de un determinado problema.

Algunas de las limitaciones más comunes que enfrentan los programadores cuando usan directamente arrays para la solución de muchos problemas son:

- Hay que prever un tamaño a la hora de crear el array. Los arrays no tienen operaciones para crecer ni decrecer en la cantidad de elementos. Si se crea con un tamaño muy grande se puede estar desperdiciando memoria, si se crea con un tamaño pequeño ésta puede resultar insuficiente.
- Los arrays no llevan un control de cuántos elementos del array están realmente ocupados según la lógica en que están siendo utilizados. Si se interpreta el array como un contenedor para "guardar" valores, la propiedad `Length` da la longitud con que fue creado, pero no los valores de interés que hayan sido guardados en las diferentes posiciones del array. Para compensar esta insuficiencia, al construirse un array con la operación new se inicializan los elementos del array con un valor por defecto según el tipo (aunque éste no puede ser otra cosa que `null` cuando el tipo es por referencia).

Para la solución de muchos problemas es conveniente contar con una capa de abstracción mayor que sin perder la cualidad esencial de los arrays, que es el acceso directo a sus elementos a través de un índice entero, incluya nuevas funcionalidades que permitan insertar, quitar o añadir elementos de forma simple, así como llevar la cuenta de los elementos reales que la estructura contiene. Esta estructura de datos se suele denominar **lista** (aunque el Capítulo 12 está dedicado a estudiar algunas de las estructuras de datos más relevantes, por la utilidad de las listas, y por su similitud de uso con los arrays, se presentarán a continuación).

Una lista es una secuencia de elementos que se pueden acceder con índices enteros consecutivos, al igual que un array, pero con la diferencia de que permite insertar o eliminar elementos en cualquier posición, encargándose de forma automática de gestionar el espacio físico que éstos ocupan y de lidiar con el manejo de los índices y con el posible desplazamiento de los elementos en memoria para intentar garantizar que el acceso a los elementos a través del índice sea simple y se logre en un tiempo constante.

Las listas permiten, al igual que los arrays, el acceso a los elementos a través de un índice, preferentemente con una sintaxis similar a la que se usa para los arrays (como es el caso de C#), es decir en la forma `nombre[indice]`. Si un array controla que el índice no se salga del rango según la longitud del array, en una lista se controla que el índice no se salga de la cantidad de elementos que hay en la lista.

Las dos operaciones fundamentales que diferencian a una lista de un array es la existencia de un método para **añadir** un elemento (`Add` en el caso del tipo `List` de C#) y de una propiedad que nos dice cuántos elementos hay en la lista (`Count` en el caso del tipo `List` de C#). Los conceptos longitud de array o del tamaño físico de memoria ocupada quedan ocultos dentro de la implementación de la lista y de forma general no tienen por qué ser asunto del programador que las use.

Las listas suelen tener muchas otras operaciones como determinar si un valor está en la lista, quitar un valor, insertar un valor en una posición y otras muchas.

El tipo genérico List<T>

Los arrays se dice son estructuras genéricas porque su sintaxis nos permite escribir:

```
T[] a = new T[n];
```

Se dice que el array es genérico porque esta notación vale para cualquiera que sea el nombre `T` de un tipo. El compilador controla entonces que sobre los elementos `a[i]` solo se puedan hacer operaciones legales para el tipo `T` y que a los elementos del array solo se le puedan asignar valores de tipo `T`.

De forma similar a los métodos genéricos, una clase genérica consiste en una definición de una clase que se hace en base a uno o varios tipos genéricos. Para poder crear un objeto del tipo de tal clase genérica, el código en donde esto se haga deberá indicar los tipos "concretos" que se le asocian a los tipos genéricos. El Listado 7-39 nos muestra un esbozo de los métodos más importantes de la clase `List<T>` de C#. Más adelante en esta sección se mostrarán posibles implementaciones de estos métodos.

```
public class List<T>
{
    private T[] items;

    public int Capacity { get { return items.Length; } }
    public int Count { get; private set; }
    public T this[int index] {}
    public List() {}
    public List(int capacity) {}
    public void Add(T item) {}
    public bool Contains(T item) {}
    public int IndexOf(T item) {}
    public void Insert(int index, T item) {}
    public bool Remove(T item) {}
    public void RemoveAt(int index) {}
    public T[] ToArray() {}
    // ... otros métodos
}
```

Listado 7-39 Esbozo de la clase List

Si tenemos las siguientes definiciones

```
List<int> calificaciones = new List<int>();
List<string> nombres = new List<string>();
List<Persona> estudiantes = new List<Persona>();
```

El compilador permite entonces escribir:

```
nombres.Add("Juan"); calificaciones.Add(5);
```

Porque el tipo `T` de los elementos de la lista `nombres` es `string`; y el de la lista `calificaciones` es `int`. Pero daría error escribir:

```
nombres.Add(4)
```

Porque 4 no es un valor de tipo `string`.

Del mismo modo el Intellisense aprovecha esta característica, como se muestra en la Figura 7-12. El compilador sabe que `estudiantes` es de tipo `List<Persona>`, de modo que `estudiantes[i]` es de tipo `Persona` y por tanto despliega los recursos del tipo `Persona`.

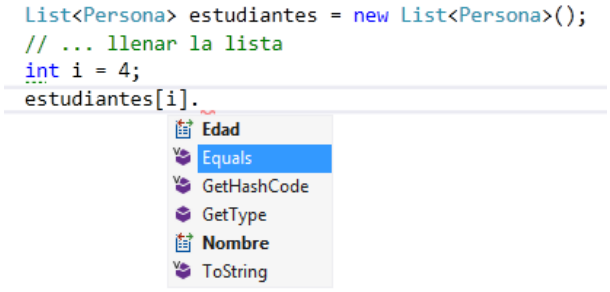

Figura 7-12: La genericidad permite al compilador efectuar inferencia de tipos y brindar ayuda al programador.

Añadir un elemento a una lista

Una de las operaciones más distintivas de las listas para representar colecciones de valores es la capacidad de que se le pueden añadir elementos. En este sentido, funcionan como los arrays, pues permiten "almacenar" elementos lo cual van haciendo secuencialmente, es decir, si se añade un elemento y éste queda en una posición *i* el siguiente elemento que se añada quedará en la posición *i+1*. Sin embargo, a diferencia de los arrays las listas no están limitadas por una longitud y siempre se pueden añadir elementos (claro, la memoria total de un sistema siempre está acotada, de manera que esto de añadir no puede ser infinito). Recuerde que para calcular los salarios mayores que el promedio era necesario prever con antelación la máxima cantidad posible de salarios a almacenar. Empleando una lista es posible resolver este problema sin necesidad de conocer de antemano cuántos elementos serán añadidos, ni tener que lidiar con el índice concreto donde cada elemento será almacenado porque la propia lista lleva el control de cuántos elementos se le añaden (Listado 7-40).

```
List<float> salarios = new List<float>();
float total = 0f;

while (true)
{
    string s = Console.ReadLine();
    if (string.IsNullOrEmpty((s))) break;

    float salario = float.Parse(s);
    salarios.Add(salario);
}

// ...
```

Listado 7-40 Adicionando elementos a una lista para calcular el valor promedio

Cantidad de elementos en la lista

Si una lista permite añadir cualquier cantidad de elementos, es necesario que entonces también nos pueda decir en cualquier momento la cantidad de elementos realmente hay

almacenados en la lista. Para esto hay una propiedad `Count`. Siguiendo con el ejemplo de calcular los salarios mayores que el promedio, ahora podemos hacerlo usando la propiedad `Count`, sin necesidad de tener que contar explícitamente la cantidad de elementos que se han añadido (Listado 7-41).

```
List<float> salarios = new List<float>();
float total = 0f;

while (true)
{
    string s = Console.ReadLine();
    if (string.IsNullOrEmpty((s))) break;

    float salario = float.Parse(s);
    salarios.Add(salario);
}

// Calculando el promedio
float promedio = total/salarios.Count;
Console.WriteLine("El promedio es {0:0.00}", promedio);
// ...
```

Listado 7-41 Cálculo del promedio de los elementos en una lista

Acceso a un elemento a través de un índice

Si a la lista se le pueden añadir elementos, y también lleva la cuenta de la cantidad de elementos añadidos, para completar su utilidad debe ofrecernos una forma de acceder a estos elementos (en general en la vida de qué vale "guardar" si no podemos luego de alguna forma acceder a lo "guardado").

La definición de `List` tiene un indizador (*indexer*)

```
public T this[int index]
{
    get { ... }
    set { ... }
}
```

Esto permite acceder a los elementos de la lista `salarios` con la misma notación `salarios[i]` con la que accedemos a los elementos de un array.

De modo que el primer elemento que se añade a una lista estará en la posición `0` y así consecutivamente; el último elemento que se ha añadido estará en la posición `Count-1`.

Volviendo al problema de calcular los salarios mayores que el promedio, empleando el indizador (*indexer*) de la lista ya es posible completar la implementación (Listado 7-42).

```
List<float> salarios = new List<float>();
float total = 0f;

while (true)
{
    string s = Console.ReadLine();
    if (string.IsNullOrEmpty((s))) break;
```

```
    float salario = float.Parse(s);
    salarios.Add(salario);
}

// Calculando el promedio
float promedio = total/salarios.Count;
Console.WriteLine("El promedio es {0:0.00}", promedio);

// Detectar los salarios mayores que el promedio
for (int i = 0; i < salarios.Count; i++)
  if (salarios[i] > promedio)
    Console.WriteLine("El valor {0} en pos {1} es mayor que el promedio.",
                      salarios[i], i);
```

Listado 7-42 Escribir los mayores que el promedio usando una lista

Al igual que en un array, referirse a un elemento de la lista en la forma `salarios[i]` exige que el valor de `i` esté en un rango válido, es decir `0<=i<Count`. Al igual que un array, la lista permite cambiar el valor en una posición haciendo una asignación `salarios[i] = … ;` pero note que no hay modo de trabajar con una posición que no tenga un valor realmente guardado en ella.

Otras operaciones que también cambian la cantidad de elementos de una lista son:

```
public void Insert(int index, T item) {}
public bool Remove(T item) {}
public void RemoveAt(int index) {}
```

[redacted] Implementando una lista

Para implementar el tipo `List` utilizaremos internamente un array `items` (

Listado 7-43). Una de las ventajas de usar listas en lugar de directamente arrays consiste en que internamente la lista será capaz de ajustar su capacidad de almacenamiento cambiando, de forma transparente al código que usa la lista, el tamaño del array interno que usa para el almacenamiento.

La forma más sencilla de implementar esta funcionalidad es chequear el tamaño del array interno antes de hacer cada operación que provoque añadir un elemento a la lista (como son el caso de `Add` e `Insert`) y hacerlo "crecer" en caso necesario (Listado 7-44).

```
public void Add(T item)
{
    if (Count == items.Length)
        Grow();

    items[Count++] = item;
}
```

Listado 7-43 Uso de items en la implementación de List

```
private void Grow()
{
```

```
    T[] temp = new T[items.Length + 1];

    for (int i = 0; i < Capacity; i++)
        temp[i] = items[i];

    items = temp;
}
```

Listado 7-44 Método para hacer crecer el array items

La operación de crecer cada vez que se llena el array interno crea un nuevo array con mayor capacidad, y se copian todos los elementos ya almacenados en el viejo array hacia el nuevo. En la implementación mostrada aquí la capacidad se aumenta exactamente en 1, lo cual es una estrategia muy ineficiente, pues para cada nuevo elemento será necesario volver a copiar todos los elementos ya almacenados. Se pueden aplicar distintas estrategias sobre en cuánto hacer crecer la capacidad de este array.

Una estrategia que ha demostrado tener valor práctico es hacer la crecer el array en una cantidad proporcional a la cantidad de elementos en el array lleno, por ejemplo, duplicando la capacidad (listado NNN). Esto presupone que en la construcción de la lista original se pueda indicar la capacidad inicial que se quiera para este array.

Esto puede significar que cuando la lista hubiese alcanzado una gran cantidad de elementos pudiera desperdiciarse mucho espacio al duplicar éste. Una opción es hacer crecer la capacidad en factor más pequeño.

Escoger un factor de crecimiento adecuado puede ser un problema dependiente del contexto en que vaya a ser utilizada la lista, con qué frecuencia serán añadidos nuevos elementos, y en general es necesario experimentar con distintos valores para encontrar el balance ideal entre eficiencia y consumo de memoria. Éste es un tema activo de investigación para el cual no existe una respuesta correcta en todos los casos.

Aunque puede ser conveniente, en general en las implementaciones de lista no se considera el problema de hacer decrecer el array cuando se quiten elementos de la lista y llegue a quedar mucho espacio subutilizado en el array. Esto se debe a que en la mayoría de los casos las operaciones de adición e inserción son mucho más comunes que las operaciones de eliminación. De todas formas, se sugiere al lector que como entrenamiento implemente una estrategia de decrecimiento recíproca a la de crecimiento cuando se detecte mucho espacio ocioso.

Una estrategia que ha demostrado tener valor práctico es hacer la crecer el array en una cantidad proporcional a la cantidad de elementos en el array lleno, por ejemplo, duplicando la capacidad. Esto presupone que en la construcción de la lista original se pueda indicar la capacidad inicial que se quiera para este array.

Esto puede significar que cuando la lista hubiese alcanzado una gran cantidad de elementos pudiera desperdiciarse mucho espacio al duplicar éste. Una opción es hacer crecer la capacidad en factor más pequeño.

Escoger un factor de crecimiento adecuado puede ser un problema dependiente del contexto en que vaya a ser utilizada la lista, con qué frecuencia serán añadidos nuevos elementos, y en general es necesario experimentar con distintos valores para encontrar el

balance ideal entre eficiencia y consumo de memoria. Éste es un tema activo de investigación para el cual no existe una respuesta correcta en todos los casos.

Aunque puede ser conveniente, en general en las implementaciones de lista no se considera el problema de hacer decrecer el array cuando se quiten elementos de la lista y llegue a quedar mucho espacio subutilizado en el array. Esto se debe a que en la mayoría de los casos las operaciones de adición e inserción son mucho más comunes que las operaciones de eliminación. De todas formas, se sugiere al lector que como entrenamiento implemente una estrategia de decrecimiento recíproca a la de crecimiento cuando se detecte mucho espacio ocioso.

Implementación de un indizador

Según lo dicho anteriormente, es necesario poder acceder y modificar los elementos que se han guardado en la lista con una notación de índice similar a la que se usa en los arrays y a la vez controlar si el índice es correcto no por la longitud del array que se use internamente en la implementación, sino por la cantidad real de elementos que se han guardado en la lista. Para mantener las ventajas del encapsulamiento y permitir una sintaxis cómoda, C# brinda un recurso lingüístico denominado **indizador** (*indexer*). Un indizador se define como una propiedad con el nombre especial `this`, y las respectivas secciones `get` y `set`, con la diferencia de que se adiciona en la definición un parámetro para indicar el índice (Listado 7-45).

```
public T this[int index]
{
    get
    {
        if (index < 0 || index >= Count)
            throw new ArgumentOutOfRangeException("index");

        return items[index];
    }
    set
    {
        if (index < 0 || index >= Count)
            throw new ArgumentOutOfRangeException("index");

        items[index] = value;
    }
}
```

Listado 7-45 Indizador en una lista

Una vez definido el indizador es posible accede a los elementos de la lista usando la misma sintaxis que para los arrays, con la ventaja de que realmente a la implementación del indizador responde un método que puede realizar todos los chequeos y validaciones necesarios.

Pertenencia de un elemento a una lista

Considerando a una lista como un conjunto de elementos, es deseable disponer también de una operación que determine si un valor "pertenece" o no a la lista. Esta operación se

implementa en un método a menudo denominado `Contains`, muy similar al método de búsqueda lineal para arrays (Listado 7-46).

```
public bool Contains(T item)
{
    for (int i = 0; i < Count; i++)
        if (Object.Equals(item, this[i]))
            return true;

    return false;
}
```

Listado 7-46 Método Contains en una lista

Como la lista, además de ser un conjunto, "indiza" los elementos por posición, puede tener utilidad una operación de pertenencia que devuelva la posición en la que se encuentra un determinado valor. Esto es lo que ofrece el método `IndexOf`, que devuelve el índice donde se encuentre el primer elemento que se igual al buscado o `-1` en caso de no aparecer (Listado 7-47).

```
public int IndexOf(T item)
{
    for (int i = 0; i < Count; i++)
        if (Object.Equals(item, this[i]))
            return i;

    return -1;
}
```

Listado 7-47 Método IndexOf en lista

Otras operaciones que modifican la cantidad de elementos

La operación `Add` para adicionar un elemento no es la única que puede provocar cambios en la cantidad de elementos y por tanto tener que hacer crecer o decrecer el array interno.

El método `Insert` permite insertar un elemento a la lista en una posición específica provocando un "desplazamiento" de la posición de los restantes elementos que estaban en una posición posterior. (Listado 7-48).

```
public void Insert(int index, T item)
{
    if (index < 0 || index > Count)
        throw new ArgumentOutOfRangeException("index");

    if (Count == items.Length)
        Grow();

    if (index < Count)
        Array.Copy(items, index, items, index + 1, Count - index);

    items[index] = item;
    Count++;
}
```

Listado 7-48 Método Insert en List

El método `RemoveAt` elimina un elemento indicando una posición en la lista (Listado 7-49).

```
public void RemoveAt(int index)
{
    if (index < 0 || index >= Count)
        throw new ArgumentOutOfRangeException("index");

    Count--;

    if (index < Count)
        Array.Copy(items, index + 1, items, index, Count - index);


}
```

Listado 7-49 Método Remove en List

Otro método para eliminar elementos de la lista es `Remove`, que quita el primer elemento que sea igual al valor que se indique como parámetro. En lugar de lanzar una excepción si el elemento a eliminar no se encuentra se retorna `false` y `true` cuando se encuentra y se ha quitado (Listado 7-50).

```
public bool Remove(T item)
{
    int index = IndexOf(item);
    if (index < 0) return false;
    RemoveAt(index);
    return true;
}
```

Listado 7-50 Método Remove

Para poder trabajar con una lista en aquellos métodos codificados para arrays se dispone del método `ToArray` cuya implementación puede ser como la del Listado 7-51.

```
public T[] ToArray()
{
    T[] objArray = new T[Count];
    Array.Copy(items, 0, objArray, 0, Count);
    return objArray;
}
```

Listado 7-51 Método ToArray

Este método además garantiza que se devuelve un array exactamente del tamaño necesario para almacenar todos los elementos ya que como array no necesita de la estrategia de crecimiento de la lista para poderle adicionar elementos.

7.12 Arrays multidimensionales

Los arrays estudiados hasta ahora se consideran unidimensionales: se pueden considerar como una secuencia de valores consecutivos en memoria los cuales se acceden con un solo índice. En C# también es posible crear arrays de dos, tres o más dimensiones, cuyos elementos se acceden entonces con varios índices según su dimensión.

Los arrays de más de una dimensión más utilizados en la práctica son los arrays de dos dimensiones, denominados **bidimensionales**, y se crean indicando la longitud de cada dimensión, por ejemplo

```
string[,] tablero = new string[8, 8];
```

Los elementos de este array se acceden entonces mediante dos índices, que deben respetar los rangos correspondientes a cada dimensión; por ejemplo, es correcto hacer:

```
string pieza = tablero[3, 5];
valores[7,7] = "reina negra";
```

Pero el intento de hacer `tablero[10, 2]` daría un error de índice fuera de rango.

En analogía con la matemática, un array de dos dimensiones puede utilizarse para representar una matriz o tabla. El convenio adoptado en la práctica es que la primera dimensión represente a las filas y la segunda dimensión a las columnas (aunque acogerse a este convenio facilita la escritura de código y la lectura de código ajeno, realmente eso no es una exigencia; lo importante es que las operaciones que usen el array sean consistentes con el convenio adoptado). De modo que el array siguiente podemos imaginarlo como la tabla de la Figura 7-13: Posible representación de un array bidimensional visto como una matriz. En cada celda se muestran los índices correspondientes, si se asumen las filas como la primera dimensión (0) y las columnas como la segunda dimensión (1)..

```
string[,] matrix = new string[4,5];

for (int i = 0; i < 4; i++)
    for (int j = 0; j < 5; j++)
        matrix[i, j] = i + "," + j;
```

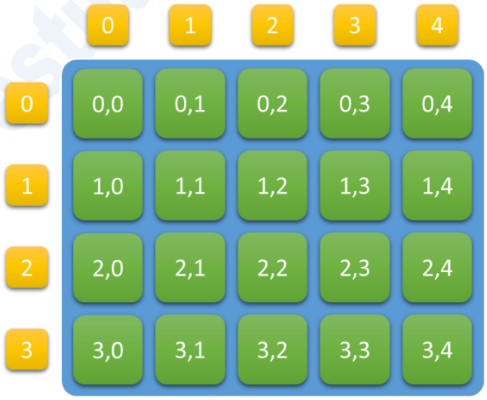

Figura 7-13: Posible representación de un array bidimensional visto como una matriz. En cada celda se muestran los índices correspondientes, si se asumen las filas como la primera dimensión (0) y las columnas como la segunda dimensión (1).

De manera similar, es posible definir y emplear arrays de tres o de más dimensiones.

Acceso a las dimensiones de un array multidimensional

Para arrays multidimensionales, la longitud de cada dimensión puede ser distinta. La propiedad `Length` en este caso devuelve la cantidad total de elementos del array. La

propiedad `Rank` devuelve la cantidad de dimensiones del array. En todos los casos vistos anteriormente, de arrays de una sola dimensión, `array.Rank` sería igual a `1`. La longitud de cada dimensión se obtiene mediante el método de instancia `GetLength`.

```
int[,,] array = new int[5,9,3];

Console.WriteLine(array.Length);        // 135
Console.WriteLine(array.GetLength(0));  // 5
Console.WriteLine(array.GetLength(1));  // 9
Console.WriteLine(array.GetLength(2));  // 3
```

La forma más sencilla y habitual de recorrer todos los elementos del array accediendo directamente a los índices consiste en una secuencia de ciclos anidados, donde cada ciclo recorre una de las dimensiones empezando por la de más a la izquierda en la definición (Listado 7-52 Recorriendo un array bidimensional), lo que en el caso de un array de dos dimensiones se interpreta como primero las filas y luego las columnas en cada fila.

```
for (int i = 0; i < array.GetLength(0); i++)
    for (int j = 0; j < array.GetLength(1); j++)
        for (int k = 0; k < array.GetLength(2); k++)
            Console.WriteLine(array[i, j, k]);
```

Listado 7-52 Recorriendo un array bidimensional

Representación interna de los arrays multidimensionales

Internamente un array multidimensional se representa con elementos consecutivos de memoria de forma similar a un array de una dimensión. La sintaxis mostrada anteriormente para acceder a los elementos no es más que una abstracción que provee C# para simplificar el trabajo con los índices y acercarse a la notación usada con tablas y matrices.

Por ejemplo, el código del Listado 7-53 nos muestra cómo se puede usar un array unidimensional para simular el uso de un array de dos dimensiones (el lector puede generalizar este ejemplo para cualquier cantidad de dimensiones).

```
int[] array = new int[n * m];

for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        Console.WriteLine(array[i * m + j]);
```

Listado 7-53 Usando un array unidimensional para representar uno bidimensional

Pero es evidente que pasar al desarrollador la responsabilidad de llevar la cuenta imaginaria de los índices enrarece la claridad del código y es una fuente de posibles errores.

Definiendo un tipo Matriz

Las matrices en términos matemáticos son útiles en la solución de problemas de optimización y para representar transformaciones afines en espacios vectoriales. Desde un punto de vista computacional, una representación muy conveniente para una matriz es empleando un array bidimensional de tipo `double[,]` y sus operaciones, que pudieran encapsularse (Listado 7-54).

```
public class Matriz
{
    private double[,] elementos;

    public Matriz(double[,] elementos)
    {
        this.elementos = elementos;
    }

    public Matriz(int filas, int columnas)
    {
        this.elementos = new double[filas, columnas];
    }

    public int Filas { get { return elementos.GetLength(0); } }
    public int Columnas { get { return elementos.GetLength(1); } }

    //...
}
```

Listado 7-54 Representación para una matriz

Para interactuar con los valores de la matriz es conveniente implementar un indizador bidimensional, que permita acceder y modificar los valores del array interno (Listado 7-55).

```
public double this[int i, int j]
{
    get
    {
        if (i < 0 || i >= Filas) throw new ArgumentOutOfRangeException("i");
        if (j < 0 || j >= Columnas) throw new ArgumentOutOfRangeException("j");

        return elementos[i, j];
    }
    set
    {
        if (i < 0 || i >= Filas) throw new ArgumentOutOfRangeException("i");
        if (j < 0 || j >= Columnas) throw new ArgumentOutOfRangeException("j");

        elementos[i, j] = value;
    }
}
```

Listado 7-55 Indizador bidimensional

Una de las operaciones aritméticas más simples definidas para las matrices es la suma. La matriz resultante tiene como valor en cada celda la suma de los valores en las celdas correspondientes en las matrices operandos. Se requiere para ello que ambas matrices coincidan en la cantidad de filas y columnas (Listado 7-56).

```
public Matriz Sumar(Matriz otra)
{
    if (otra == null) throw new ArgumentNullException("otra");

    if (otra.Filas != this.Filas || otra.Columnas != this.Columnas)
```

```
        throw new InvalidOperationException("Necesitan coincidir en ambas
dimensiones.");

    double[,] suma = new double[Filas, Columnas];

    for (int i = 0; i < Filas; i++)
        for (int j = 0; j < Columnas; j++)
            suma[i, j] = this[i, j] + otra[i, j];

    return new Matriz(suma);
}
```

Listado 7-56 Sumar dos matrices

Obviamente, el lector puede implementar el método `Resta`. Más interesante es la implementación de multiplicación de matrices.

Si $A_{m,n}$ y $B_{n,p}$ son matrices, entonces su multiplicación es una matriz $C_{m,p}$ tal que:

$$C_{i,j} = \sum_{k} A_{i,k} \cdot B_{k,j}$$

Si A y B son las matrices que representan ciertas transformaciones afines (por ejemplo, traslación, rotación o escala en 3 dimensiones), entonces la matriz producto $C = A \cdot B$ representa la composición de dichas transformaciones. Esto permite computar y almacenar de manera muy eficiente un conjunto arbitrariamente grande de transformaciones, lo que hace del producto de matrices una de las operaciones más importante en la disciplina de Gráficos por Computadora

Para poder multiplicar dos matrices, la primera matriz debe tener tantas columna como filas tiene la segunda matriz. La matriz resultante tendrá tantas filas como la primera matriz y tantas columnas como la segunda. Una implementación de esta operación en C# consiste justamente en recorrer primero por filas (ciclo más externo del Listado 7-57) y luego por columnas (ciclo intermedio) cada posible posición de la matriz resultante para ir asignándole el valor que le corresponde, resultante del producto escalar de la fila correspondiente de la primera matriz por la columna de la segunda matriz (lo que se hace en el ciclo más interno).

```
public Matriz Multiplicar(Matriz otra)
{
    if (otra == null) throw new ArgumentNullException("otra");
    if (otra.Filas != this.Columnas) throw new InvalidOperationException();

    Matriz producto = new Matriz(this.Filas, otra.Columnas);

    for (int i = 0; i < producto.Filas; i++)
        for (int j = 0; j < producto.Columnas; j++)
            for (int k = 0; k < this.Columnas; k++)
                producto[i, j] += this[i, k]*otra[k, j];

    return producto;
}
```

Listado 7-57 Multiplicación de Matrices

7.2 Arrays bidimensionales para representar tableros

Tal vez el uso más popular de los arrays bidimensionales es para representar tableros. Muchos juegos de mesa (ajedrez, damas, sudoku, tic-tac-toe, etc.) se basan en el concepto de tablero, sobre el que se mueven las piezas y se hacen las jugadas según las reglas de cada juego. En estos casos, para la solución computacional de problemas relacionados a este tipo de juegos, es muy útil representar los tableros como arrays bidimensionales en los que cada índice corresponde a una celda o posición en el tablero.

Por ejemplo, un tablero de ajedrez podría representarse con un array de tipo `string[,]` donde cada valor `string` indique el tipo de cada pieza (con un valor `null` indicando las celdas vacías). Supongamos que dado un tablero se quiere determinar si el rey blanco está amenazado por un caballo negro, una solución sencilla consiste en encontrar la posición del rey blanco, y partir de ahí en alguna de las celdas alcanzables con el movimiento de un caballo se encuentra un caballo negro (Listado 7-58).

```
public static bool EstaAmenazado(string[,] tablero)
{
    int iPos = 0;
    int jPos = 0;

    // Buscar al rey blanco
    for (int i = 0; i < 8; i++)
        for (int j = 0; j < 8; j++)
            if (tablero[i, j] == "RB") // Rey blanco
            {
                iPos = i;
                jPos = j;
                break;
            }

    // Chequear todas las posibles celdas alcanzables por un caballo
    // Hacia arriba dos celdas
    int ii = iPos - 2;
    if (ii >= 0)
    {
        // Arriba a la derecha
        int jj = jPos + 1;
        if (jj < 8 && tablero[ii, jj] == "CN") // Caballo negro
            return true;

        // Arriba a la izquierda
        jj = jPos - 1;
```

```
            if (jj >= 0 && tablero[ii, jj] == "CN")
                return true;
        }
        // Hacia arriba una celda
        ii = iPos - 1;
        if (ii >= 0) {
            // Arriba a la derecha
            int jj = jPos + 2;
            if (jj < 8 && tablero[ii, jj] == "CN")
                return true;

            // Arriba a la izquierda
            jj = jPos - 2;
            if (jj >= 0 && tablero[ii, jj] == "CN")
                return true;
        }
        // Hacia abajo dos celdas
        ii = iPos + 2;
        if (ii < 8) {
            // Abajo a la derecha
            int jj = jPos + 1;
            if (jj < 8 && tablero[ii, jj] == "CN")
                return true;

            // Abajo a la izquierda
            jj = jPos - 1;
            if (jj >= 0 && tablero[ii, jj] == "CN")
                return true;
        }
        // Hacia abajo una celda
        ii = iPos + 1;
        if (ii < 8) {
            // Abajo a la derecha
            int jj = jPos + 2;
            if (jj < 8 && tablero[ii, jj] == "CN")
                return true;

            // Abajo a la izquierda
            jj = jPos - 2;
            if (jj >= 0 && tablero[ii, jj] == "CN")
                return true;
        }

        return false;
    }
```

Listado 7-58 Determinación de si rey blanco está amenazado por un caballo

Es decir, un caballo se puede mover en 8 direcciones diferentes; esto hace engorroso y propenso a errores lidiar con todas las posibles combinaciones de movimientos. En la sección siguiente se mostrará cómo se puede simplificar este código empleando lo que se conoce como **arrays de direcciones**.

Arrays de direcciones

En la solución de problemas asociados con tableros es común la realización de recorridos que siguen una dirección particular, por ejemplo: hacia arriba, hacia la derecha, o diagonal sureste (Figura 7-14).

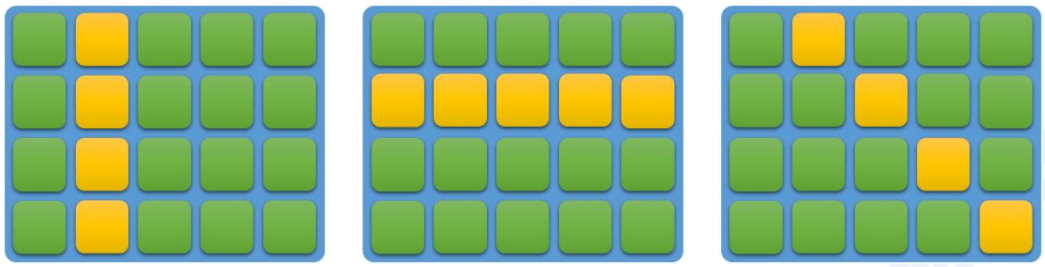

Figura 7-14: Posibles recorridos de interés en un tablero: vertical horizontal y diagonal.

Aunque es posible ejecutar este tipo de recorridos empleando ciclos `for` de manera explícita como se hizo para los movimientos del caballo en el Listado 7-58, es engorroso escribir en términos de índices la dirección en la que se realiza un recorrido. Por ejemplo, para recorrer un tablero de forma diagonal (como lo hace la reina o el alfil en el ajedrez), empezando en la esquina inferior izquierda en dirección a la esquina superior derecha, es necesario incrementar el índice asociado a las columnas mientras se decrementa el índice asociada a las filas (Listado 7-59).

```
string[,] tablero = new string[8, 8];

for (int i = tablero.GetLength(0) - 1; i >= 0; i--)
    for (int j = 0; j < tablero.GetLength(1); j++)
        //...Hacer algo con la casilla (i,j)
```

Listado 7-59 Recorrido en diagonal de esquina inferior izquierda hacia esquina superior derecha (suroeste a noreste)

Todos los posibles movimientos de la reina en el ajedrez (que se mueve en las 8 direcciones cardinales norte, sur, este, oeste, noroeste, suroeste, noreste y sureste), se pueden expresar con el siguiente par de arrays:

```
//           N   S   E   W   NW  SW  NE  SE
int[] df = {-1,  1,  0,  0,  -1,  1, -1,  1};
int[] dc = { 0,  0,  1, -1,  -1, -1,  1,  1};
```

El array `df` determina en cuánto debe incrementarse el índice de las filas, y el array `dc` en cuánto deben incrementarse las columnas, En la Figura 7-15 se muestran las ocho posibles direcciones, y cómo cambia el valor de la fila `(F)` y la columna `(C)` actual al desplazarse hacia cada dirección. El valor de este cambio (`+1`, `-1` o `0`) es precisamente el número que se almacena en cada array de direcciones.

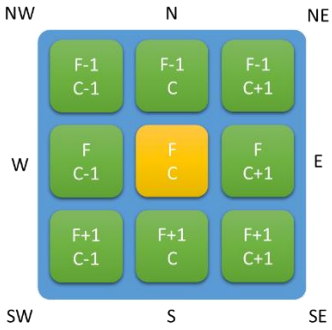

Figura 7-15: Direcciones de la rosa náutica, junto a los valores que es necesario adicionar a la fila y columna actual para llegar a la casilla correspondiente.

Por ejemplo, para desplazarse hacia la dirección `NE` un paso, es necesario restar 1 a las filas, y sumar 1 a las columnas. Note que en los arrays de direcciones, en la posición asociada a la dirección `NE`, éstos son justamente los valores almacenados: -1 en el array `df`, y 1 en el array `dc`. Si se desea calcular la casilla correspondiente a caminar en cierta dirección una cantidad k de pasos, entonces solamente es necesario multiplicar k por el valor correspondiente en cada array de direcciones.

El Listado 7-60 muestra como empleando arrays de direcciones es posible detectar si el rey blanco se encuentra amenazado por la reina negra. Para simplificar, se asume que al método se le pasan los parámetros `filaReina` y `columnaReina` que nos dan la posición de la reina negra en el tablero.

```
private static bool ReinaAmenazaRey(string[,] tablero,
                                    int filaReina, int columnaReina)
{
    // Array de direcciones para los posibles movimientos de la reina
    //           N   S   E   W   NW  SW  NE  SE
    int[] df = {-1,  1,  0,  0, -1,  1, -1,  1};
    int[] dc = { 0,  0,  1, -1, -1, -1,  1,  1};

    // Escoger una dirección
    for (int d = 0; d < df.Length; d++)
    {
        // Cantidad de pasos en esa dirección
        int k = 1;

        while (true)
        {
            int nf = filaReina + df[d]*k; // Calcular la nueva fila
            int nc = columnaReina + dc[d]*k; // Calcular la nueva columna

            if (!PosicionValida(tablero.GetLength(0),
                                tablero.GetLength(1), nf, nc))
                break;
            if (tablero[nf, nc] == "RB") // Rey blanco
                return true;
            if (tablero[nf, nc] != null) // Cualquier otra pieza
                break;

            k++; // Incrementar el paso
```

```
            }
        }

        return false;
    }

    private static bool PosicionValida(int m, int n, int f, int c)
    {
        return f >= 0 && f < m && c >= 0 && c < n;
    }
```

Listado 7-60 Determinar si reina negra amenaza a rey blanco

El método `PosicionValida` es un método adicional que nos ayuda a controlar no salirnos de los límites del tablero. El algoritmo consiste en iterar por todas las posibles direcciones, y para cada dirección, avanzar tantos pasos como sea posible. Si en una dirección particular se llega a una posición inválida el ciclo se aborta con un `break`, es decir, se detiene el movimiento en esa dirección. Al llegar a una casilla válida, se verifica si el rey se encuentra en dicha casilla. En caso de encontrarse cualquier otra pieza en la casilla correspondiente también se aborta el movimiento en la dirección correspondiente. Este código es fácilmente generalizable para realizar en cada casilla válida una acción particular al problema en consideración.

Una gran ventaja de los arrays de direcciones es que permiten de forma transparente modificar las reglas de movimiento sin necesidad de cambiar el resto del algoritmo. Por ejemplo, si se elimina la posibilidad de moverse en diagonales (para representar los movimientos de las torres), o se elimina la posibilidad de moverse horizontal y verticalmente (para representar los movimientos de los alfiles), solamente hay que modificar los arrays de direcciones para quitar las direcciones correspondientes a los movimientos no permitidos. En el Listado 7-60 los arrays de direcciones se han incluido dentro del propio método `ReinaAmenazaRey`, sin embargo para hacer más general aún el método estos arrays se podrían haber definido también como parámetros del método para que fuesen aportados por quien decide ejecutar dicho método. De esta forma se podría implementar un método `EstaAmenazado` que sirva para cualquier tipo de pieza, simplemente variando los arrays de direcciones. Se sugiere al lector que piense en cómo describir los posibles movimientos de cada una de las piezas del ajedrez en términos de arrays de direcciones.

Algoritmo de Lee

Un tipo frecuente de problema en los juegos con tableros es la búsqueda del camino más corto entre un par de casillas según las piezas y las reglas de movimiento (que pueden tener restricciones como celdas con obstáculos por las cuales no se puede pasar o en las que no se puede ubicar determinada pieza).

Para simplificar el código y no tener que incluir explícitamente todas las variantes de movimiento en el tablero se suele usar con frecuencia un array bidimensional de tipo `bool[,]` para indicar (según si el valor de una celda sea true o false) si existe o no un obstáculo en la celda correspondiente. En la Figura 7-16 se muestra un ejemplo de este tipo de problemas, donde las casillas marcadas como false representan celdas inválidas, y

el problema consiste en encontrar un camino de longitud mínima de una esquina a la otra del tablero que solo pase por celdas válidas.

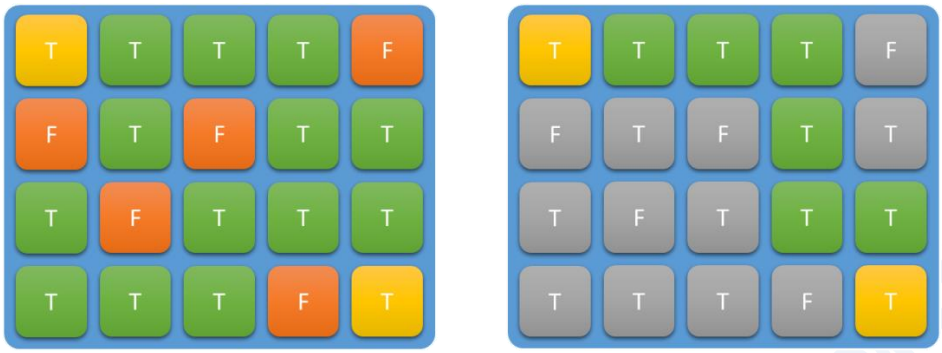

Figura 7-16: Representación de un mapa mediante un array de valores bool. La imagen de la derecha muestra un camino óptimo de la celda inicial (esquina superior izquierda) a la celda final (esquina inferior derecha). Las celdas F son impasables.

Un algoritmo muy utilizado para implementar el encontrar el camino óptimo en un tablero es el conocido como **algoritmo de Lee**. Este algoritmo comienza a partir de la celda inicial, realizando un proceso iterativo que en cada paso intenta alcanzar todas las celdas posibles a partir de las celdas ya alcanzadas. Cada vez que se llega a una celda nueva, se pone en dicha celda la cantidad de pasos que tomó llegar a la misma. Al terminar el algoritmo se tendrá en cada celda alcanzable un valor que indica la longitud del menor camino que lleva del origen a esa celda (en caso de existir algún camino). De esta información es posible obtener cuáles son las celdas alcanzables, a qué distancia se encuentran y además reconstruir la secuencia de pasos que permite llegar del origen hasta cada una.

La idea fundamental del algoritmo radica en cómo actualizar la información de cuáles son las celdas alcanzables a partir de una celda de partida (parámetros `filaInicial` y `columnaInicial`). Para esto, se almacena en un array bidimensional de tipo `int[,]` la cantidad de pasos en los que se ha llegado a cada celda. Un valor igual a 0 indica que la celda correspondiente aún no ha sido alcanzada. Una iteración consiste en analizar para cada celda todas sus celdas vecinas. Si alguna de estas celdas vecinas ya ha sido alcanzada, con una cantidad k de pasos, entonces se puede afirmar que la celda actual es alcanzable en al menos $k + 1$ pasos (Figura 7-17). El algoritmo termina cuando en una iteración no se puede alcanzar ninguna celda nueva (Listado 7-61).

```
static int[,] Lee(bool[,] tablero, int filaInicial, int columnaInicial)
{
  // Array de costos
  int totalFilas = tablero.GetLength(0);
  int totalColumnas = tablero.GetLength(1);
  int[,] distancias = new int[totalFilas, totalColumnas];

  // Marcar la celda inicial
  distancias[filaInicial, columnaInicial] = 1;

  //          N   S   E   W   NW  SW  NE  SE
  int[] df = {-1, 1, 0, 0, -1, 1, -1, 1};
  int[] dc = {0, 0, 1, -1, -1, -1, 1, 1};
```

```
  bool huboCambio;

  do
  {
    huboCambio = false;

    // Para cada posible celda del tablero
    for (int f = 0; f < totalFilas; f++)
    {
      for (int c = 0; c < totalColumnas; c++)
      {
        // Saltarse las celdas no marcadas
        if (distancias[f, c] == 0) continue;
        // Saltarse las celdas inválidas
        if (!tablero[f, c]) continue; // Hay obstáculo en la celda

        // Inspeccionar celdas vecinas a la celda [f, c]
        for (int d = 0; d < df.Length; d++)
        {
                    int vf = f + df[d];
                    int vc = c + dc[d];

                    // Determinar si es un vecino válido y no ha sido marcado
                    if (PosicionValida(totalFilas, totalColumnas, vf, vc)
                        && distancias[vf, vc] == 0 && tablero[vf, vc])
                    {
                        // Actualizar esta celda
                        distancias[vf, vc] = distancias[f, c] + 1;
                        huboCambio = true;

           break;

                    }
                }
             }
        }
    } while (huboCambio);

    return distancias;
}
```

Listado 7-61 Recorrido usando algoritmo de Lee

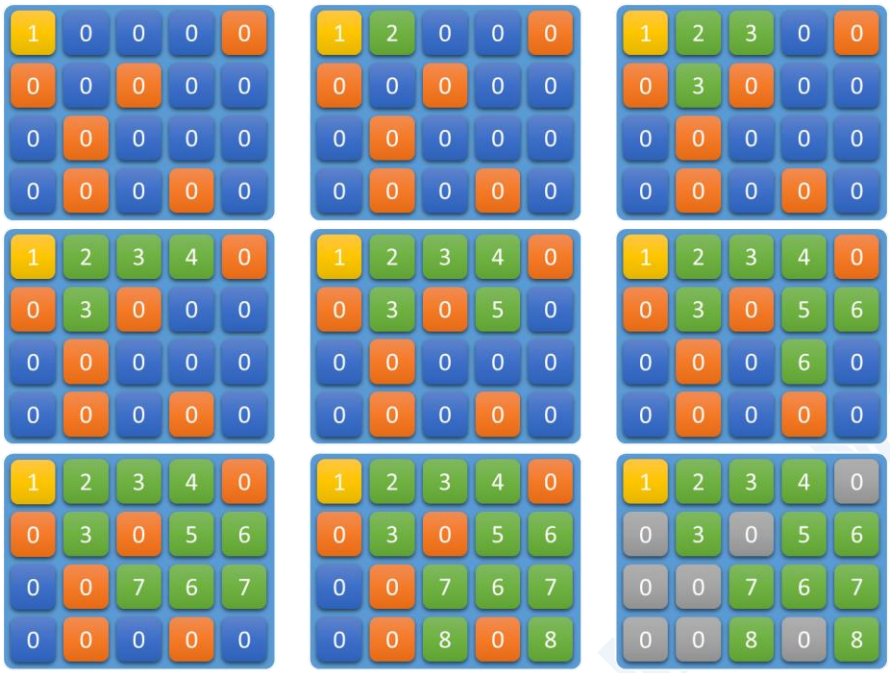

Figura 7-17: Ejemplo de ejecución del algoritmo de Lee partiendo de la celda 0,0 de un tablero de 4x4 en el que no es posible hacer movimientos en diagonal. Las celdas coloreadas en naranja son las que corresponden a celdas con obstáculos. En cada iteración el algoritmo marca todas las celdas alcanzables con la cantidad de pasos correspondiente. Al terminar, todas las celdas con distancia 0 son inalcanzables.

Para determinar si el algoritmo de Lee es correcto es conveniente volver al concepto de invariante de ciclo. Considere la siguiente invariante: en la i-ésima iteración el algoritmo detecta todas las celdas alcanzables a distancia i del origen. Para el caso inicial la invariante se cumple trivialmente, pues la única celda a distancia 1 es la propia celda origen. Para una iteración cualquiera intermedia, note que si existe una celda alcanzable a distancia i, dicha celda tiene que ser vecina de otra celda alcanzable a distancia $i-1$. Pero todas las celdas a distancia $i-1$ fueron detectadas en la iteración anterior, y el algoritmo revisa para cada celda todos sus vecinos. Por tanto, si existe una celda alcanzable a distancia i el algoritmo la detecta. Por otro lado, si el algoritmo marca una celda es porque algún vecino de esta celda era alcanzable, lo cual convierte en alcanzable la celda marcada. Además, dicha celda no puede ser alcanzable en una distancia menor que i porque habría sido detectada antes. Por lo tanto, en la i-ésima iteración el algoritmo detecta y marca exactamente todas las celdas alcanzables a distancia i. Como la distancia máxima hacia cualquier celda alcanzable es finita, el algoritmo está obligado a terminar en una cantidad finita de pasos.

El algoritmo de Lee, aunque correcto, es bastante ineficiente, porque cuando se hizo algún cambio se vuelven a recorrer de nuevo todas las celdas (los dos ciclos `for` mas externos) debido a que explora las mismas celdas una y otra vez. En general, su costo temporal es proporcional a $n^2 \cdot d$, donde d es la máxima distancia a la celda más lejana del origen. Este valor puede ser tan grande como n. Empleando estructuras de datos más complejas es posible implementar algoritmos mucho más eficientes. De todas formas, el algoritmo de Lee es una variante sencilla de implementar y fácil de comprender para problemas de cálculo de caminos en tableros bidimensionales.

7.3 Arrays de arrays

Los casos anteriores de arrays multidimensionales se denominan también rectangulares por su analogía, en el caso bidimensional, a un rectángulo de alto (cantidad de filas) y ancho (cantidad de columnas). Esta característica es la que los hace representables en una zona de memoria consecutiva y por tanto de costo constante el acceso a cualquiera de sus elementos. Sin embargo, si consideramos que realmente un array `T[]` es un tipo, entonces nada debería impedir a su vez tener un array cuyos elementos sean del tal tipo, es decir `T[][]`. El código siguiente declara a un array unidimensional de elementos que a su vez son arrays de enteros:

```
int[][] arrayDeArrays;
```

A diferencia de los arrays rectangulares, no se puede indicar en la construcción una cantidad de filas y columnas, porque en cada celda de `arrayDeArrays` puede haber a su vez un array de tamaño diferente. De modo que en la construcción del array con la operación new solo se puede indicar la cantidad de celdas del array principal. La ejecución del siguiente código:

```
int[][] arrayDeArrays = new int[5][];
```
[4]

Nos formaría un array como el que se muestra en la Figura 7-18.

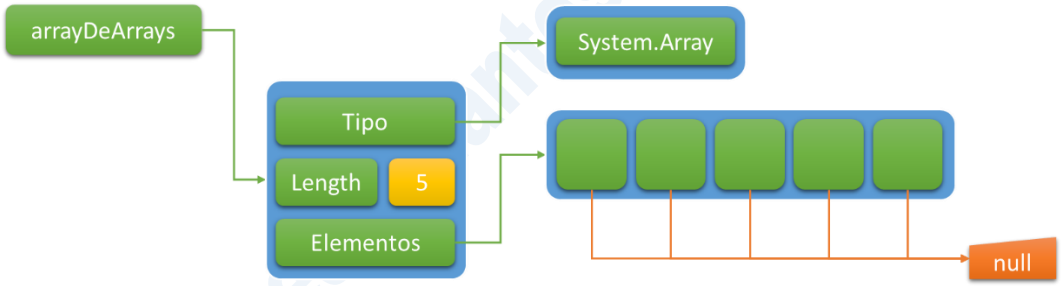

Figura 7-18 Representación en memoria de un array de arrays recién inicializado. El tipo interno del array es a su vez un array, por lo que cada elemento comienza inicializado con un valor null (valor por defecto de los tipos por referencia, y por tanto, de los arrays.

Como el `new` hace que cada elemento del array celda se inicialice con su valor por defecto, y los arrays son tipos por referencia, entonces cada elemento tendrá un valor `null`. Es responsabilidad del programador crear e inicializar en cada posición un array con el tamaño y los valores que desee. La ejecución de un código como:

```
for (int i = 0; i < 5; i++)
    arrayDeArrays[i] = new int[i + 1];
```

Nos dejaría en memoria una estructura como la que se muestra en Figura 7-19.

[4] Note que para acceder a un array de arrays los índices se indican de izquierda a derecha. Es decir, `arrayDeArrays[2][3]` indica la posición 2 del array "más externo", y luego la posición 3 del array "más interno", como es de esperar. Sin embargo, para ser consistentes con esta sintaxis, en la inicialización también se indican los índices en el mismo orden, por lo que la inicialización se hace `arrayDeArrays = new int[5][]`.

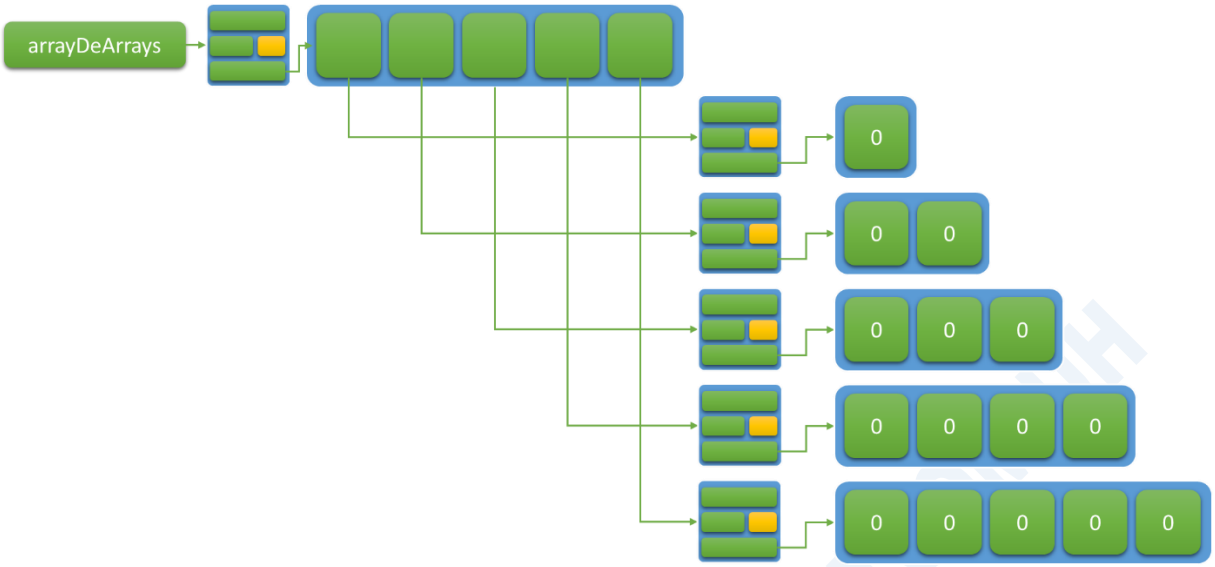

Figura 7-19: Representación simplificada de un array de arrays de enteros. Cada índice del array más externo contiene una referencia a un array de enteros. Por simplicidad se ha omitido en esta figura la representación de los metadatos de cada array.