7. Gestión dinámica de memoria

7.1. ¿Por qué usar estructuras dinámicas?

Hasta ahora teníamos una serie de variables que declarábamos al principio del programa o de cada función. Estas variables, que reciben el nombre de **estáticas**, tienen un tamaño asignado desde el momento en que se crea el programa. Es el caso de los tipos de datos nativos del sistema (byte, int, double, etc), así como de algunos datos compuestos: los arrays y los struct.

Este tipo de variables son sencillas de usar y rápidas… en caso de sólo vayamos a manejar estructuras de datos que no cambien. A cambio, resultan poco eficientes si tenemos estructuras cuyo tamaño no sea siempre el mismo, por motivos que comentaremos a continuación.

Por ejemplo, si queremos crear una agenda, necesitaremos crear un array e ir añadiendo nuevos datos. Si reservamos espacio para un máximo de 10 fichas, no podremos llegar a añadir la número 11. Varios ejemplo de soluciones típicas a este problema, pero que no resultan óptimas podrían ser:

- Sobredimensionar: preparar una agenda contando con 1000 fichas, aunque supongamos que no vamos a pasar de 200. Esto tiene varios inconvenientes: se desperdicia memoria, obliga a conocer bien los datos con los que vamos a trabajar, sigue pudiendo verse sobrepasado, etc.
- Crear un nuevo array cada poco: comenzar por reservar un array de 10 datos y, en el momento en el que haya que añadir el dato 11, crear un nuevo array de 11 datos, volcar en él todos los datos anteriores y el nuevo dato y finalmente destruir el array original de 10 datos. Con el dato 12 y cada nuevo elemento se repetiría el mismo proceso. Esto resulta muy lento, así que esta alternativa se debería evitar, salvo que se tenga la certeza de que se van a realizar muchas lecturas pero que será muy poco frecuente añadir nuevos datos.
- Trabajar siempre con la información almacenada en el disco, leyendo cada ficha cuando sea necesario (en el próximo tema veremos cómo), pero esta alternativa es muchísimo más lenta: el acceso típico a los datos de un fichero en disco (mecánico) puede ser más del orden de 1000 veces más lento que el acceso en memoria.

La alternativa real es crear estructuras **dinámicas**, que puedan ir creciendo o disminuyendo según nos interese. En los lenguajes de programación "clásicos",

como C y Pascal, este tipo de estructuras se tienen que crear de forma básicamente artesanal, mientras que en lenguajes modernos como C#, Java o las últimas versiones de C++, existen esqueletos ya creados que podemos utilizar con facilidad. A lo largo de este tema, nos centraremos en usar las facilidades de los lenguajes modernos, y apenas comentaremos un poco cómo sería la forma artesanal de crear estas estructuras con un lenguaje "antiguo".

Algunos ejemplos de estructuras de este tipo son:

- Las **pilas**. Una "pila de datos" se comportará de forma similar a una pila de libros: podemos apilar cosas en la cima, o extraer de la cima. Se supone que no se puede tomar elementos de otro sitio que no sea la cima, ni dejarlos en otro sitio distinto. De igual modo, se supone que la pila no tiene un tamaño máximo definido, sino que puede crecer arbitrariamente.

- Las **colas**. Una "cola de datos" se comportará como las colas del cine (en teoría): la gente llega por un sitio (la cola) y sale por el opuesto (la cabeza). Al igual que antes, supondremos que un elemento no puede entrar ni salir en posiciones intermedias de la cola y que ésta puede crecer hasta un tamaño indefinido.

- Las **listas**, mas versátiles pero más complejas de programar, en las que se puede añadir elementos en cualquier posición y obtenerlos o borrarlos de cualquier posición.

Y existen otras estructuras dinámicas más complejas y que nosotros no trataremos, como los **árboles**, en los que cada elemento puede tener varios sucesores (se parte de un elemento "raíz", y la estructura se va ramificando), o los **grafos**, formados por una serie de nodos unidos por aristas.

Todas estas estructuras tienen en común que, si se programan correctamente, pueden ir creciendo o decreciendo según haga falta, al contrario que un array, que tiene su tamaño (máximo) prefijado. Además, ciertas operaciones, como las inserciones en posiciones intermedias o los borrados, pueden ser más rápidas que en un array, porque no será necesario mover todos los datos que estaban almacenados a partir de esa posición.

7.2. Una pila en C#

Para crear una pila, emplearemos la clase Stack. Una pila nos permitirá introducir un nuevo elemento en la cima ("apilar", en inglés "**push**") y quitar el elemento que hay en la cima ("desapilar", en inglés "**pop**").

Este tipo de estructuras se suele denotar también usando las siglas "**LIFO**" (Last In First Out: lo último en entrar es lo primero en salir).

Para utilizar la clase "Stack" y la mayoría de las que veremos en este tema, necesitamos incluir en nuestro programa una referencia a "System.Collections".

Así, un ejemplo básico que creara una pila, introdujera tres palabras y luego las volviera a mostrar sería:

```
// Ejemplo_7_02a.cs
// Ejemplo de clase "Stack" (Pila)
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

class Ejemplo_7_02a
{
    static void Main()
    {
        string palabra;

        Stack miPila = new Stack();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        for (byte i=0; i<3; i++)
        {
            palabra = (string) miPila.Pop();
            Console.WriteLine( palabra );
        }
    }
}
```

cuyo resultado sería:

```
yo
soy
Hola,
```

Como se puede ver en este ejemplo, no hemos indicado que sea una "pila de strings", sino simplemente "una pila". Por eso, los datos que extraemos son

"objetos", que deberemos convertir al tipo de datos que nos interese utilizando un "typecast" (conversión forzada de tipos), como en

```
palabra = (string) miPila.Pop();
```

En ocasiones puede ser interesante algo un poco más rígido, que esté adaptado a un único tipo de datos, y no necesite una conversión de tipos cada vez que extraigamos un dato.

Por ello, en la versión 2 de la "plataforma .Net" (del año 2005) se introdujeron los "generics", que definen estructuras de datos genéricas, que nosotros podemos particularizar en cada uso. Por ejemplo, una pila de strings se definiría con:

```
Stack<string> miPila = new Stack<string>();
```

Y necesitaríamos incluir un "using" distinto al principio del programa:

```
using System.Collections.Generic;
```

De modo que una segunda versión de nuestro contacto con las pilas sería:

```
// Ejemplo_7_02b.cs
// Ejemplo de clase "Stack" (Pila) + Generics
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Ejemplo_7_02b
{
    static void Main()
    {
        string palabra;

        Stack<string> miPila = new Stack<string>();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        for (byte i=0; i<3; i++)
        {
            palabra = miPila.Pop();
            Console.WriteLine( palabra );
        }
    }
}
```

En las estructuras dinámicas que incluye la plataforma .Net, es habitual que además podamos saber cuántos elementos hay almacenados, usando su propiedad "**.Count**" (en vez del .Length que existe para arrays). Así, una versión mejorada del programa anterior, que permita introducir cualquier cantidad de textos en una pila, podría ser:

```
// Ejemplo_7_02c.cs
// Segundo ejemplo de clase "Stack" (Pila)
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Ejemplo_7_02c
{
    static void Main()
    {
        string palabra;

        Stack<string> miPila = new Stack<string>();
        do
        {
            Console.Write( "Dime una palabra (Intro para terminar): ");
            palabra = Console.ReadLine();
            if (palabra != "")
                miPila.Push(palabra);
        }
        while (palabra != "");

        int cantidad = miPila.Count;
        for (int i=0; i<cantidad; i++)
        {
            palabra = miPila.Pop();
            Console.WriteLine( "{0} ", palabra );
        }
    }
}
```

Como se ve en el ejemplo anterior, hemos mirado el ".Count" antes de entrar al bucle "for", porque se trata de un valor que va cambiando a medida que se extraen datos de la pila. Por eso, será peligroso que .Count sea el límite de un bucle "for", o, de lo contrario, es probable que recorramos menos posiciones que el total existente en la pila. Como alternativa, se puede usar un "while", como muestra el siguiente ejemplo:

```
// Ejemplo_7_02d.cs
// Tercer ejemplo de clase "Stack" (Pila)
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Ejemplo_7_02d
```

```
{
    static void Main()
    {
        string palabra;

        Stack<string> miPila = new Stack<string>();
        do
        {
            Console.Write( "Dime una palabra (Intro para terminar): ");
            palabra = Console.ReadLine();
            if (palabra != "")
                miPila.Push(palabra);
        }
        while (palabra != "");

        while (miPila.Count > 0)
        {
            palabra = miPila.Pop();
            Console.WriteLine( "{0} ", palabra );
        }
    }
}
```

La implementación de una pila en C# es algo más avanzada que lo que podríamos esperar en una pila estándar: incorpora también métodos adicionales como:

- "Peek", que mira el valor que hay en la cima, pero sin extraerlo.
- "Clear", que borra todo el contenido de la pila.
- "Contains", que indica si un cierto elemento está en la pila.
- "ToArray", que devuelve toda la pila convertida a un array.

Ejercicios propuestos:

(7.2.1) Crea un programa que pida al usuario 5 números enteros y luego los muestre en orden contrario, utilizando una pila.

(7.2.2) Crea una clase que imite el comportamiento de una pila, pero usando internamente un array (si no lo consigues, no te preocupes; en un apartado posterior veremos una forma de hacerlo).

(7.2.3) La "notación polaca inversa" es una forma de expresar operaciones que consiste en indicar los operandos antes del correspondiente operador. Por ejemplo, en vez de "3+4" se escribiría "3 4 +". Es una notación que no necesita paréntesis y que se puede resolver usando una pila: si se recibe un dato numérico, éste se guarda en la pila; si se recibe un operador, se obtienen los dos operandos que hay en la cima de la pila, se realiza la operación y se apila su resultado. El proceso termina cuando sólo hay un dato en la pila. Por ejemplo, "3 4 +" se convierte en: apilar 3, apilar 4, sacar dos datos y sumarlos, guardar 7, terminado. Impleméntalo y comprueba si el resultado de "3 4 6 5 - + * 6 +" es 21.

7.3. Una cola en C#

Podemos crear colas si nos apoyamos en la clase Queue. En una cola podremos introducir elementos por la cabeza ("**Enqueue**", encolar) y extraerlos por el extremo opuesto, el final de la cola ("**Dequeue**", desencolar). Este tipo de estructuras se denotan a veces también por las siglas **FIFO** (First In First Out, lo primero en entrar es lo primero en salir). Un ejemplo básico similar al de la pila, que creara una cola, introdujera tres palabras y luego las volviera a mostrar sería:

```
// Ejemplo_7_03a.cs
// Ejemplo de clase "Queue" (Cola)
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

class Ejemplo_7_03a
{
    static void Main()
    {
        string palabra;

        Queue miCola = new Queue();
        miCola.Enqueue("Hola,");
        miCola.Enqueue("soy");
        miCola.Enqueue("yo");

        for (byte i=0; i<3; i++)
        {
            palabra = (string) miCola.Dequeue();
            Console.WriteLine( palabra );
        }
    }
}
```

que mostraría:

```
Hola,
soy
yo
```

Al igual que en las pilas, podemos usar "Generics" para que la pila contenga elementos de un único tipo, y emplearemos ".Count" para saber la cantidad de datos almacenados, de modo que se podría pedir al usuario una cantidad indeterminada de datos numéricos, así:

```
// Ejemplo_7_03b.cs
// Segundo ejemplo de clase "Queue" (Cola), con Generics
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;
```

```
class Ejemplo_7_03b
{
    static void Main()
    {
        int n;

        Queue<int> miCola = new Queue<int>();
        do
        {
            Console.Write( "Dime un entero (0 para terminar): ");
            n = Convert.ToInt32( Console.ReadLine());
            if (n != 0)
                miCola.Enqueue(n);
        }
        while (n != 0);

        Console.Write( "Los datos eran: ");
        while (miCola.Count > 0)
            Console.Write( "{0} ", miCola.Dequeue() );
    }
}
```

Como también ocurría con la pila, la implementación de una cola que incluye C# es más avanzada que eso, con otros métodos como:

- "Peek", que mira el valor que hay en la cabeza de la cola, pero sin extraerlo.
- "Clear", que borra todo el contenido de la cola.
- "Contains", que indica si un cierto elemento está en la cola.
- "ToArray", que devuelve toda la cola convertida a un array.

Ejercicios propuestos:
(7.3.1) Crea un programa que pida al usuario 5 números reales de doble precisión, los guarde en una cola y luego los muestre en pantalla.
(7.3.2) Crea un programa que pida frases al usuario, hasta que introduzca una frase vacía. En ese momento, mostrará todas las frases que se habían introducido.

7.4. Los ArrayList

Una lista es una estructura dinámica en la que se puede añadir elementos sin tantas restricciones. Es habitual que se puedan introducir nuevos datos en ambos extremos, así como entre dos elementos existentes, o bien incluso de forma ordenada, de modo que cada nuevo dato se quede colocado automáticamente en la posición adecuada para que todos ellos se puedan recorrer en orden.

En el caso de C#, tenemos dos variantes especialmente útiles: una lista, a cuyos elementos se puede acceder de forma individual como a los de un array

("ArrayList"), y una lista ordenada ("SortedList"). En este apartado vamos a centranos en la primera.

En un ArrayList, podemos:

- Añadir datos en la última posición con "Add"
- Insertar en cualquier otra con "Insert"
- Recuperar un elemento de cualquier posición usando corchetes (igual que hacíamos con los "arrays")
- Recorrer toda la lista con "foreach"
- Incluso ordenarla con "Sort" (si los datos que hay guardados son "fáciles de comparar", pero no servirá "tal cual" si son datos más complejos, como un "struct")
- También podemos hacer una "búsqueda binaria" (como vimos en el apartado 4.7) si los datos están ordenados.

Vamos a ver un ejemplo de la mayoría de sus posibilidades:

```
// Ejemplo_7_04a.cs
// Ejemplo de ArrayList
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

class Ejemplo_7_04a
{
    static void Main()
    {
        ArrayList miLista = new ArrayList();
        // Añadimos en orden
        miLista.Add("Hola,");
        miLista.Add("soy");
        miLista.Add("yo");

        // Mostramos lo que contiene
        Console.WriteLine( "Contenido actual:");
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Accedemos a una posición
        Console.WriteLine( "La segunda palabra es: {0}",
            miLista[1] );

        // Insertamos en la segunda posicion
        miLista.Insert(1, "Como estas?");

        // Mostramos de otra forma (con "for") lo que contiene
        Console.WriteLine( "Contenido tras insertar:");
        for (int i=0; i<miLista.Count; i++)
            Console.WriteLine( miLista[i] );
```

```
        // Buscamos un elemento
        Console.WriteLine( "La palabra \"yo\" está en la posición {0}",
            miLista.IndexOf("yo") );

        // Ordenamos
        miLista.Sort();

        // Mostramos lo que contiene tras ordenar
        Console.WriteLine( "Contenido tras ordenar");
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Buscamos con búsqueda binaria
        Console.WriteLine( "Ahora \"yo\" está en la posición {0}",
            miLista.BinarySearch("yo") );

        // Invertimos la lista
        miLista.Reverse();

        // Borramos el segundo dato y la palabra "yo"
        miLista.RemoveAt(1);
        miLista.Remove("yo");

        // Mostramos nuevamente lo que contiene
        Console.WriteLine("Contenido dar la vuelta y tras eliminar dos:");
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Ordenamos y vemos dónde iría un nuevo dato
        miLista.Sort();
        Console.WriteLine("La frase \"Hasta Luego\"...");
        int posicion = miLista.BinarySearch("Hasta Luego");
        if (posicion >= 0)
            Console.WriteLine("Está en la posición {0}", posicion);
        else
            Console.WriteLine("No está");
    }
}
```

El resultado de este programa es:

```
Contenido actual:
Hola,
soy
yo
La segunda palabra es: soy
Contenido tras insertar:
Hola,
Como estas?
soy
yo
La palabra "yo" está en la posición 3
Contenido tras ordenar
Como estas?
Hola,
soy
```

```
yo
Ahora "yo" está en la posición 3
Contenido dar la vuelta y tras eliminar dos:
Hola,
Como estas?
La frase "Hasta Luego"...
No está.
```

Casi todo debería resultar fácil de entender. El único detalle importante que hemos omitido es que "BinarySearch" nos devolverá un número negativo en caso de que el dato no exista. Vamos a practicarlo con los ejercicios.

En el caso de los "ArrayList", al emplear "Generics" cambia la nomenclatura y no tendremos un ArrayList<string>, sino un List<string>:

```
// Ejemplo_7_04b.cs
// Ejemplo de Lista de strings
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Ejemplo_7_04b
{
    static void Main()
    {
        List<string> miLista = new List<string>();

        miLista.Add("Hola,");
        miLista.Add("soy");
        miLista.Add("yo");

        foreach (string frase in miLista)
            Console.Write(frase + " ");
    }
}
```

Aun así, la verdadera potencia de usar "Generics" aparece cuando se usan listas de datos que son de tipos básicos, sino de tipos compuestos, como structs o incluso objetos. Por ejemplo, el siguiente programa muestra cómo se podría crear una lista de struct (así como una forma compacta de darle valores a los campos de un struct, que no habíamos visto):

```
// Ejemplo_7_04c.cs
// Ejemplo de Lista de structs
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

struct Persona
{
```

```
    public string Nombre;
    public string Apellidos;
}

class Ejemplo_7_04c
{

    static void Main()
    {
        List<Persona> miLista = new List<Persona>();

        miLista.Add(new Persona { Nombre = "Uno", Apellidos = "Dos" } );
        miLista.Add(new Persona { Nombre = "Lope", Apellidos = "López" });
        miLista.Add(new Persona { Nombre = "Juan", Apellidos = "Juanes" });

        foreach (Persona p in miLista)
            Console.WriteLine(p.Nombre + " " + p.Apellidos);
    }
}
```

Y una variante aún más avanzada, que emplee clases e implemente la interfaz "IComparable" para permitir ordenar los datos, podría ser:

```
// Ejemplo_7_04d.cs
// Ejemplo de Lista de "objetos"
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Persona : IComparable<Persona>
{
    public string Nombre {get; set;}
    public string Apellidos { get; set; }

    public override string ToString()
    {
        return Nombre + " " + Apellidos;
    }

    public int CompareTo(Persona otro)
    {
        return Apellidos.CompareTo(otro.Apellidos);
    }
}

class Ejemplo_7_04d
{

    static void Main()
    {
        List<Persona> miLista = new List<Persona>();

        miLista.Add(new Persona { Nombre = "Uno", Apellidos = "Dos" } );
        miLista.Add(new Persona { Nombre = "Lope", Apellidos = "López" });
        miLista.Add(new Persona { Nombre = "Juan", Apellidos = "Juanes" });

        miLista.Sort();
```

```
        foreach (Persona p in miLista)
            Console.WriteLine(p);
    }
}
```

Ejercicios propuestos:

(7.4.1) Crea un programa que pida frases al usuario (hasta que introduzca una línea vacía), las almacene en un ArrayList (o una lista de strings) y luego pregunte de forma repetitiva al usuario qué línea desea ver. Terminará cuando el usuario introduzca "-1".

(7.4.2) Crea un programa que pida frases al usuario (hasta que introduzca una línea vacía), las almacene en un ArrayList (o una lista de strings) y luego pregunte de forma repetitiva al usuario qué texto desea buscar y muestre las líneas que contienen ese texto. Terminará cuando el usuario introduzca una cadena vacía.

(7.4.3) Crea un programa que pida frases al usuario (hasta que introduzca una línea vacía), las almacene en un ArrayList (o una lista de strings), lo ordene y lo muestre ordenado en pantalla.

(7.4.4) Crea un programa que pida frases al usuario (hasta que introduzca una línea vacía), las almacene en un ArrayList luego muestre en pantalla las líneas impares (primera, tercera, etc.) y finalmente muestre las líneas pares (segunda, cuarta, etc.).

(7.4.5) Crea una nueva versión de la "base de datos de ficheros" (ejemplo 04_06a), usando ArrayList (no una lista de datos con tipo) en vez de un array convencional.

(7.4.6) Crea una nueva versión de la "base de datos de ficheros" (ejemplo 04_06a), usando una lista de datos con tipo en vez de un array convencional.

7.5. Diccionarios (1: SortedList)

En un "diccionario", los elementos están formados por una pareja de datos: una clave y un valor (como en un diccionario: la palabra y su definición). Se puede añadir elementos con "Add", o acceder a los elementos mediante su clave (con corchetes), como en este ejemplo:

```
miDiccionario.Add("hola", "hello");
Console.WriteLine( miDiccionario["hola"] );
```

En C# existen dos implementaciones distintas de un diccionario:

- Las "SortedList" buscan ocupar poco espacio y que los datos se puedan obtener en orden.

- Las "Hashtable" buscan la máxima velocidad de acceso, pero a cambio ocupan mucho más, y los datos no se pueden recuperar en orden.

Vamos a ver un primer ejemplo del uso de "SortedList"

```
// Ejemplo_7_05a.cs
// Ejemplo de SortedList: Diccionario esp-ing
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

class Ejemplo_7_05a
{
    static void Main()
    {
        // Creamos e insertamos datos
        SortedList miDiccio = new SortedList();
        miDiccio.Add("hola", "hello");
        miDiccio.Add("adiós", "good bye");
        miDiccio.Add("hasta luego", "see you later");

        // Mostramos los datos
        Console.WriteLine( "Cantidad de palabras en el diccionario: {0}",
            miDiccio.Count );
        Console.WriteLine( "Lista de palabras y su significado:" );
        for (int i=0; i<miDiccio.Count; i++)
        {
            Console.WriteLine( "{0} = {1}",
            miDiccio.GetKey(i), miDiccio.GetByIndex(i) );
        }
        Console.WriteLine( "Traducción de \"hola\": {0}",
            miDiccio.GetByIndex( miDiccio.IndexOfKey("hola") ));
        Console.WriteLine( "Que también se puede obtener con corchetes: {0}",
            miDiccio["hola"]);
    }
}
```

Su resultado sería

```
Cantidad de palabras en el diccionario: 3
Lista de palabras y su significado:
adiós = good bye
hasta luego = see you later
hola = hello
Traducción de "hola": hello
```

Otras posibilidades de la clase SortedList son:

- "Contains", para ver si la lista contiene una cierta clave.
- "ContainsValue", para ver si la lista contiene un cierto valor.

- "Remove", para eliminar un elemento a partir de su clave.
- "RemoveAt", para eliminar un elemento a partir de su posición.
- "SetByIndex", para cambiar el valor que hay en una cierta posición.

Si preferimos usar "Generics", cambian algunos detalles:

- No usaremos SortedList, sino SortedList<string,string> (en caso de que la clave sea una cadena de texto y el valor almacenado sea otra cadena de texto, que es lo más habitual.
- No podremos usar "Contains" para ver si aparece una cierta clave, sino ContainsKey.
- En vez de mirar GetKey(i), veremos qué hay en Keys[i]; de la misma forma, podríamos ver qué valor hay en una cierta posición con Values[i].
- Desparecen SetByIndex y GetByIndex, se debe usar la notación de corchetes.

Con estos cambios, el ejemplo anterior quedaría:

```
// Ejemplo_7_05b.cs
// Ejemplo de SortedList con tipo base: Diccionario esp-ing
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Ejemplo_7_05b
{
    static void Main()
    {
        // Creamos e insertamos datos
        SortedList<string,string> miDiccio =
            new SortedList<string, string>();
        miDiccio.Add("hola", "hello");
        miDiccio.Add("adiós", "good bye");
        miDiccio.Add("hasta luego", "see you later");

        // Mostramos los datos
        Console.WriteLine("Cantidad de palabras en el diccionario: {0}",
            miDiccio.Count);
        Console.WriteLine("Lista de palabras y su significado:");
        for (int i = 0; i < miDiccio.Count; i++)
        {
            Console.WriteLine("{0} = {1}",
            miDiccio.Keys[i], miDiccio.Values[i]);
        }
        Console.WriteLine("Traducción de \"hola\": {0}",
            miDiccio["hola"]);
    }
}
```

Ejercicios propuestos:

(7.5.1) Crea un programa que, cuando el usuario introduzca el nombre de un número del 1 al 10 en inglés (por ejemplo, "two"), diga su traducción en español (por ejemplo, "dos").

(7.5.2) Crea un programa que, cuando el usuario introduzca el nombre de un mes en español (por ejemplo, "abril"), muestre su traducción en inglés (por ejemplo, "april").

(7.5.3) Crea un traductor básico de C# a Pascal, que tenga las traducciones almacenadas en una SortedList (por ejemplo, "{" se convertirá a "begin", "}" se convertirá a "begin", "WriteLine" se convertirá a "WriteLn", "ReadLine" se convertirá a "ReadLn", "void" se convertirá a "procedure" y "Console." se convertirá a una cadena vacía. Úsalo para que el usuario teclee (o "copie y pegue") un fuente en C# y mostrar su equivalente (que no será perfecto) en Pascal.

7.6. Diccionarios (2: las "tablas hash")

En una "tabla hash", los elementos están formados por una pareja: una clave y un valor, como en un SortedList, pero la diferencia está en la forma en que se manejan internamente estos datos: la "tabla hash" usa una "función de dispersión" para colocar los elementos dentro de una estructura de datos muy grande (típicamente, un array con ciertas "mejoras añadidas" en las que no entraremos). Eso supone que no se pueden recorrer secuencialmente (los datos no se guardan ordenados) y que ocupan más espacio, pero a cambio el acceso a partir de la clave es **muy rápido**, mucho más que si hacemos una búsqueda secuencial (como haríamos si los datos estuvieran en un array convencional) o incluso una búsqueda binaria (como haríamos en un array o ArrayList ordenado).

Un ejemplo de diccionario, parecido al anterior (y cuya ejecución es más rápida para consultar para un dato concreto, pero que no se puede recorrer en orden), podría ser:

```
// Ejemplo_7_06a.cs
// Ejemplo de Tabla Hash: Diccionario de informática
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

class Ejemplo_7_06a
{
    static void Main()
    {
        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
```

```
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos algún dato
        Console.WriteLine( "Cantidad de palabras en el diccionario: {0}",
            miDiccio.Count );
        try
        {
            Console.WriteLine( "El significado de PC es: {0}",
            miDiccio["pc"]);
        }
        catch (Exception)
        {
            Console.WriteLine( "No existe esa palabra!");
        }
    }
}
```

que escribiría en pantalla:

```
Cantidad de palabras en el diccionario: 3
El significado de PC es: personal computer
```

Si un elemento que se busca no existe, se lanzaría una excepción, por lo que deberíamos controlarlo con un bloque try-catch. Lo mismo ocurre si intentamos introducir un dato que ya existe. Una alternativa a usar try-catch es comprobar antes si el dato ya existe, con el método "Contains" (o su sinónimo "ContainsKey"), como en este ejemplo:

```
// Ejemplo_7_06b.cs
// Ejemplo de Tabla Hash: Diccionario de informática
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

class Ejemplo_7_06b
{
    static void Main()
    {
        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos algún dato
        Console.WriteLine( "Cantidad de palabras en el diccionario: {0}",
            miDiccio.Count );
        if (miDiccio.Contains("pc"))
            Console.WriteLine( "El significado de PC es: {0}",
                miDiccio["pc"]);
        else
            Console.WriteLine( "No existe la palabra PC");
    }
```

```
}
```

Otras posibilidades son: borrar un elemento ("Remove"), vaciar toda la tabla ("Clear"), o ver si contiene un cierto valor ("ContainsValue", mucho más lento que buscar entre las claves con "Contains").

Una tabla hash tiene una cierta capacidad inicial, que se amplía automáticamente cuando es necesario. Como la tabla hash es mucho más rápida cuando está bastante vacía que cuando está casi llena, podemos usar un constructor alternativo, en el que se le indica la capacidad inicial que queremos, si tenemos una idea aproximada de cuántos datos vamos a guardar:

```
Hashtable miDiccio = new Hashtable(500);
```

De manera parecida a como ocurría con los ArrayList, la nomenclatura cambia si deseamos usar "Generics": no emplearemos "Hashtable<string,string>" sino "Dictionary<string,string>". Además, en ese caso, y al igual que pasaba con las SortedList, deberemos emplear "ContainsKey" en vez de "Contains":

```
// Ejemplo_7_06c.cs
// Ejemplo de Tabla Hash con tipos base: Diccionario de informática
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Ejemplo_7_06c
{
    static void Main()
    {
        // Creamos e insertamos datos
        Dictionary<string,string> miDiccio =
            new Dictionary<string, string>();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos algún dato
        Console.WriteLine("Cantidad de palabras en el diccionario: {0}",
            miDiccio.Count);
        if (miDiccio.ContainsKey("pc"))
            Console.WriteLine("El significado de PC es: {0}",
                miDiccio["pc"]);
        else
            Console.WriteLine("No existe la palabra PC");
    }
}
```

Y si quisiéramos especificar una capacidad inicial en este caso, haríamos:

```
Dictionary<string,string> miDiccio = new Dictionary<string, string>(500);
```

Ejercicios propuestos:

(7.6.1) Crea una versión alternativa del ejercicio 7.5.1, pero que tenga las traducciones almacenadas en una tabla Hash.

(7.6.2) Crea una versión alternativa del ejercicio 7.5.2, pero que tenga las traducciones almacenadas en una tabla Hash.

7.7. Los "enumeradores"

Un enumerador es una estructura auxiliar que permite recorrer las estructuras dinámicas de forma secuencial. Casi todas estas estructuras contienen un método GetEnumerator, que permite obtener un enumerador para recorrer todos sus elementos.

Para las colecciones "normales", como las pilas y las colas, el tipo de Enumerador a usar será un "IEnumerator", con un campo "Current" para saber el valor actual y un método "MoveNext" para pasar al siguiente (que es un booleano y devolverá *false* si no existe un siguiente dato al que desplazarse):

```
// Ejemplo_7_07a.cs
// Ejemplo de Enumeradores en una pila (Stack)
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

class Ejemplo_7_07a
{
    static void Main()
    {
        Stack miPila = new Stack();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IEnumerator miEnumerador = miPila.GetEnumerator();
        while ( miEnumerador.MoveNext() )
            Console.WriteLine("{0}", miEnumerador.Current);
    }
}
```

que escribiría

```
Contenido:
yo
```

```
soy
Hola,
```

Como curiosidad, al recorrer la pila con un enumerador, al contrario que si empleamos el método "Pop", los datos no se extraen de la pila. Por eso, si al final de este programa decimos que nos muestre el valor de *miPila.Pop()*, no obtendríamos una excepción sino que se nos mostraría la palabra "yo".

En la versión con "Generics", no sólo pasaremos a tener Stack<string> sino también IEnumerator<string>:

```
// Ejemplo_7_07b.cs
// Ejemplo de Enumeradores en una pila de strings
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Ejemplo_7_07b
{
    static void Main()
    {
        Stack<string> miPila = new Stack<string>();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IEnumerator<string> miEnumerador = miPila.GetEnumerator();
        while (miEnumerador.MoveNext())
            Console.WriteLine("{0}", miEnumerador.Current);
    }
}
```

Aun así, en C# no es imprescindible usar enumeradores, puede bastar con "foreach" para recorrer esas estructuras dinámicas, y, al igual que con un enumerador, los datos no se eliminarían si se trata de una estructura con lectura destructiva, como una pila o una cola:

```
// Ejemplo_7_07c.cs
// "foreach" en vez de enumerador para una pila
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Ejemplo_7_07c
{
    static void Main()
    {
        Stack<string> miPila = new Stack<string>();
```

```
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        foreach(string s in miPila)
            Console.WriteLine(s);

        // Y vemos que el primero sigue ahí
        Console.WriteLine(miPila.Pop());
    }
}
```

Si usamos un enumerador para recorrer estructuras formadas por pares (clave, valor), como una tabla hash, no tendremos un ".Current" sino ".Key" y ".Value":

```
// Ejemplo_7_07d.cs
// Ejemplo de Enumeradores en una Tabla Hash
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

class Ejemplo_7_07d
{
    static void Main()
    {
        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IDictionaryEnumerator miEnumerador = miDiccio.GetEnumerator();
        while ( miEnumerador.MoveNext() )
            Console.WriteLine("{0} = {1}",
                miEnumerador.Key, miEnumerador.Value);
    }
}
```

cuyo resultado es

```
Contenido:
pc = personal computer
byte = 8 bits
kilobyte = 1024 bytes
```

Como puede apreciar en la salida correspondiente a ese ejemplo, es habitual que en una tabla Hash no obtengamos la lista de elementos en el mismo orden en el que los introdujimos, debido a que se colocan siguiendo la función de dispersión.

Si usamos un tabla hash con un tipo, aun así podemos conservar el "using System.Collections" para emplear el enumerador de diccionario:

```
// Ejemplo_7_07e.cs
// Ejemplo de Enumeradores en una Tabla Hash con tipo
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;
using System.Collections.Generic;

class Ejemplo_7_07e
{
    static void Main()
    {
        // Creamos e insertamos datos
        Dictionary<string,string> miDiccio =
            new Dictionary<string, string>();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IDictionaryEnumerator miEnumerador = miDiccio.GetEnumerator();
        while (miEnumerador.MoveNext())
            Console.WriteLine("{0} = {1}",
                miEnumerador.Key, miEnumerador.Value);
    }
}
```

Pero, en general, será más sencillo emplear un "foreach", teniendo en cuenta que los datos que obtendremos de un diccionario serán "pares clave valor" (KeyValuePair)

```
// Ejemplo_7_07f.cs
// Ejemplo de foreach en una Tabla Hash con tipo
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

class Ejemplo_7_07f
{
    static void Main()
    {
        // Creamos e insertamos datos
        Dictionary<string,string> miDiccio =
            new Dictionary<string, string>();
        miDiccio.Add("byte", "8 bits");
```

```
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        foreach(KeyValuePair<string,string> dato in miDiccio)
            Console.WriteLine("{0} = {1}",
                dato.Key, dato.Value);
    }
}
```

Nota: los "enumeradores" existen también en otras plataformas, como Java, aunque allí reciben el nombre de "iteradores".

7.8. Ejemplo: Cómo "imitar" una pila usando "arrays"

Las estructuras dinámicas se pueden imitar usando estructuras estáticas sobredimensionadas, y esto puede ser un ejercicio de programación interesante. Por ejemplo, podríamos imitar una pila dando los siguientes pasos:

- Utilizamos internamente un array más grande que la cantidad de datos que esperemos que vaya a almacenar la pila.
- Creamos una función "Apilar", que añade en la primera posición libre del array (inicialmente la 0) y después incrementa esa posición, para que el siguiente dato se introduzca a continuación.
- Creamos también una función "Desapilar", que devuelve el dato que hay en la última posición, y que disminuye el contador que indica la posición, de modo que el siguiente dato que se obtuviera sería el que se introdujo con anterioridad a éste.

El fuente podría ser así:

```
// Ejemplo_7_08a.cs
// Ejemplo de clase "Pila" basada en un array
// Introducción a C#, por Nacho Cabanes

using System;

class PilaString
{
    string[] datosPila;
    int posicionPila;
    const int MAXPILA = 200;

    // Constructor
    public PilaString()
    {
        posicionPila = 0;
```

```
        datosPila = new string[MAXPILA];
    }

    // Añadir a la pila: Apilar
    public void Apilar(string nuevoDato)
    {
        if (posicionPila == MAXPILA)
            Console.WriteLine("Pila llena!");
        else
        {
            datosPila[posicionPila] = nuevoDato;
            posicionPila ++;
        }
    }

    // Extraer de la pila: Desapilar
    public string Desapilar()
    {
        if (posicionPila < 0)
            Console.WriteLine("Pila vacia!");
        else
        {
            posicionPila --;
            return datosPila[posicionPila];
        }
        return null;
    }


    public static void Main()
    {
        string palabra;

        PilaString miPila = new PilaString();
        miPila.Apilar("Hola,");
        miPila.Apilar("soy");
        miPila.Apilar("yo");

        for (byte i=0; i<3; i++)
        {
            palabra = (string) miPila.Desapilar();
            Console.WriteLine( palabra );
        }
    }  // Fin del Main de prueba

}  // Fin de la clase
```

Ejercicios propuestos:

(7.8.1) Usando esta misma estructura de programa, crea una clase "Cola", que permita introducir datos (números enteros) y obtenerlos en modo FIFO (el primer dato que se introduzca debe ser el primero que se obtenga). Debe tener un método "Encolar" y otro "Desencolar".

(7.8.2) Crea una clase "ListaOrdenada" de "strings", que almacene un único dato (no un par clave-valor como los SortedList). Debe utilizar internamente un array, contener un método "Insertar", que añadirá un nuevo dato en orden en el array, un "Obtener(n)", que obtenga un elemento de la lista (el número "n") sin borrarlo de la lista, y un "Borrar(n)", que elimine el n-ésimo elemento. Deberá almacenar "strings".
(7.8.3) Crea una pila de "doubles", usando internamente un ArrayList (o una lista con tipo) en vez de un array.
(7.8.4) Crea una cola que almacene un bloque de datos (struct, con los campos que tú elijas) usando una lista con tipo.
(7.8.5) Crea una lista ordenada de "strings", similar a la del ejercicio 7.8.2, pero usando internamente un ArrayList.