11. Punteros y gestión dinámica de memoria

11.1. ¿Por qué usar estructuras dinámicas?

Hasta ahora teníamos una serie de variables que declaramos 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.

Este tipo de variables son sencillas de usar y rápidas... si sólo vamos a manejar estructuras de datos que no cambien, pero resultan poco eficientes si tenemos estructuras cuyo tamaño no sea siempre el mismo.

Es el caso de una agenda: tenemos una serie de fichas, e iremos añadiendo más. Si reservamos espacio para 10, no podremos llegar a añadir la número 11, estamos limitando el máximo. Una solución sería la de trabajar siempre en el disco: no tenemos límite en cuanto a número de fichas, pero es muchísimo más lento.

Lo ideal sería aprovechar mejor la memoria que tenemos en el ordenador, para guardar en ella todas las fichas o al menos todas aquellas que quepan en memoria.

Una solución "típica" (pero mala) es 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.
+
La solución suele ser 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.

Algunos ejemplos de estructuras de este tipo son:

- Las **pilas**. Como una pila de libros: vamos apilando cosas en la cima, o cogiendo 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**. Como las 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 a la cola ni salir de ella en posiciones intermedias y que la cola puede crecer hasta un tamaño indefinido.
- Las **listas**, en las que se puede añadir elementos en cualquier posición, y borrarlos de cualquier posición.

Y la cosa se va complicando: en los **árboles** cada elemento puede tener varios sucesores (se parte de un elemento "raíz", y la estructura se va ramificando), etc.

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 prefijado.

Veremos ejemplos de cómo crear estructuras dinámicas de estos tipos en C#, y después comentaremos los pasos para crear una estructura dinámica de forma "artesanal".

11.2. Una pila en C#

Para crear una pila tenemos preparada la clase Stack. Los métodos habituales que debería permitir una pila son 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 llamar también con 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 en C#            */
/*  pila1.cs                 */
/*                           */
/*  Ejemplo de clase "Stack" */
/*                           */
/*  Introduccion a C#,       */
/*    Nacho Cabanes          */
/*---------------------------*/

using System;
using System.Collections;

public class ejemploPila1  {

   public 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,
```

La implementación de una pila en C# es algo más avanzada: permite también métodos 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.
- "GetType", para saber de qué tipo son los elementos almacenados en la pila.
- "ToString", que devuelve el elemento actual convertido a un string.
- "ToArray", que devuelve toda la pila convertida a un array.
- "GetEnumerator", que permite usar "enumeradores" para recorrer la pila, una funcionalidad que veremos con algún detalle más adelante.
- También tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

Ejercicios propuestos:

- **(11.2.1)** 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.

11.3. 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 nombran 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 anterior, que creara una cola, introdujera tres palabras y luego las volviera a mostrar sería:

```
/*--------------------------*/
/*  Ejemplo en C#           */
/*  cola1.cs                */
/*                          */
/*  Ejemplo de clase "Queue" */
/*                          */
/*  Introduccion a C#,      */
/*    Nacho Cabanes         */
/*--------------------------*/

using System;
using System.Collections;
```

```
public class ejemploCola1  {

   public 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 ocurría con la pila, la implementación de una cola que incluye C# es más avanzada que eso, con métodos similares a los de antes:

- "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.
- "GetType", para saber de qué tipo son los elementos almacenados en la cola.
- "ToString", que devuelve el elemento actual convertido a un string.
- "ToArray", que devuelve toda la pila convertida a un array.
- "GetEnumerator", que permite usar "enumeradores" para recorrer la cola, una funcionalidad que veremos con algún detalle más adelante.
- Al igual que en la pila, también tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

Ejercicios propuestos:

- **(11.3.1)** Crea un programa que lea el contenido de un fichero de texto, lo almacene línea por línea en una cola, luego muestre este contenido en pantalla y finalmente lo vuelque a otro fichero de texto.

11.4. Las listas

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 introduzca automáticamente en la posición adecuada para que todos ellos queden en orden.

En el caso de C#, no tenemos ninguna clase "List" que represente una lista genérica, pero sí dos variantes especialmente útiles: una lista ordenada ("SortedList") y una lista a cuyos elementos se puede acceder como a los de un array ("ArrayList").

11.4.1. ArrayList

En un ArrayList, podemos añadir datos en la última posición con "Add", insertar en cualquier otra con "Insert", recuperar cualquier elemento usando corchetes, o bien ordenar toda la lista con "Sort". Vamos a ver un ejemplo de la mayoría de sus posibilidades:

```
/*---------------------------*/
/*  Ejemplo en C#            */
/*  arrayList1.cs            */
/*                           */
/*  Ejemplo de ArrayList     */
/*                           */
/*  Introduccion a C#,       */
/*    Nacho Cabanes          */
/*---------------------------*/

using System;
using System.Collections;

public class ejemploArrayList1  {

   public 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 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
      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 dato inmediatamente mayor es el {0}: {1}",
        ~posicion, miLista[~posicion] );

   }

}
```

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. El dato inmediatamente mayor es el 1: Hola,
```

Casi todo debería resultar fácil de entender, salvo quizá el símbolo ~. Esto se debe a que BinarySearch devuelve un número negativo cuando el texto que buscamos no aparece, pero ese número negativo tiene un significado: es el "valor complementario" de la posición del dato inmediatamente mayor (es decir, el dato cambiando los bits 0 por 1 y viceversa). En el ejemplo anterior, "posición" vale -2, lo que quiere decir que el dato no existe, y que el dato inmediatamente mayor está en la posición 1 (que es el "complemento a 2" del número -2, que es lo que indica la expresión "~posición"). En el apéndice 3 de este texto hablaremos de cómo se representan internamente los números enteros, tanto positivos como negativos, y entonces se verá con detalle en qué consiste el "complemento a 2".

A efectos prácticos, lo que nos interesa es que si quisiéramos insertar la frase "Hasta Luego", su posición correcta para que todo el ArrayList permaneciera ordenado sería la 1, que viene indicada por "~posicion".

Veremos los operadores a nivel de bits, como ~, en el tema 13, que estará dedicado a otras características avanzadas de C#.

Ejercicios propuestos:

- **(11.4.1.1)** Crea un programa que lea el contenido de un fichero de texto, lo almacene línea por línea en un ArrayList, luego muestre en pantalla las líneas impares (primera, tercera, etc.) y finalmente vuelque a otro fichero de texto las líneas pares (segunda, cuarta, etc.).
- **(11.4.1.2)** Crea una nueva versión de la "bases de datos de ficheros" (ejemplo 46), pero usando ArrayList en vez de un array convencional.

11.4.2. SortedList

En un SortedList, los elementos están formados por una pareja: 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 índice numérico (con "GetKey") o mediante su clave (sabiendo en qué posición se encuentra una clave con "IndexOfKey"), como en este ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C#           */
/*  sortedList1.cs          */
```

```
/*                          */
/*  Ejemplo de SortedList:  */
/*  Diccionario esp-ing     */
/*                          */
/*  Introduccion a C#,      */
/*    Nacho Cabanes         */
/*--------------------------*/

using System;
using System.Collections;
public class ejemploSortedList  {

   public 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.

Ejercicios propuestos:

- **(11.4.2.1)** 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.

11.5. 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, de forma que no se pueden recorrer secuencialmente, pero a cambio el acceso a partir de la clave es **muy rápido**, más que si hacemos una búsqueda secuencial (como en un array) o binaria (como en un ArrayList ordenado).

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

```
/*----------------------------*/
/*  Ejemplo en C#             */
/*  HashTable1.cs             */
/*                            */
/*  Ejemplo de HashTable:     */
/*  Diccionario de inform.    */
/*                            */
/*  Introduccion a C#,        */
/*    Nacho Cabanes           */
/*----------------------------*/

using System;
using System.Collections;
public class ejemploHashTable  {

   public 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 e) {
        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 si el dato ya existe, con el método "Contains" (o su sinónimo "ContainsKey"), como en este ejemplo:

```
/*----------------------------*/
/*  Ejemplo en C#             */
/*  HashTable2.cs             */
/*                            */
/*  Ejemplo de HashTable 2:   */
/*  Diccionario de inform.    */
/*                            */
/*  Introduccion a C#,        */
/*    Nacho Cabanes           */
/*----------------------------*/

using System;
using System.Collections;
public class ejemploHashTable2  {

   public 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);
```

Ejercicios propuestos:

- **(11.5.1)** Crea una versión alternativa del ejercicio 11.4.2.1, pero que tenga las traducciones almacenadas en una tabla Hash.

11.6. Los "enumeradores"

Un enumerador es una estructura auxiliar que permite recorrer las estructuras dinámicas de forma secuencial. Casi todas ellas contienen un método GetEnumerator, que permite obtener un enumerador para recorrer todos sus elementos. Por ejemplo, en una tabla hash podríamos hacer:

```
/*----------------------------*/
/*  Ejemplo en C#             */
/*  HashTable3.cs             */
/*                            */
/*  Ejemplo de HashTable      */
/*  y enumerador              */
/*                            */
/*  Introduccion a C#,        */
/*    Nacho Cabanes           */
/*----------------------------*/

using System;
using System.Collections;
public class ejemploHashTable3  {

   public 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 se puede ver, los enumeradores tendrán un método "MoveNext", que intenta moverse al siguiente elemento y devuelve "false" si no lo consigue. En el caso de las tablas hash, que tiene dos campos (clave y valor), el enumerador a usar será un "enumerador de diccionario" (IDictionaryEnumerator), que contiene los campos Key y Value.

Como se ve en el ejemplo, es habitual que 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.

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:

```
/*--------------------------*/
/*  Ejemplo en C#           */
/*  pila2.cs                */
/*                          */
/*  Ejemplo de clase "Stack" */
/*  y enumerador            */
/*                          */
/*  Introduccion a C#,      */
/*    Nacho Cabanes         */
/*--------------------------*/

using System;
using System.Collections;

public class ejemploPila1  {

   public 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,
```

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

Se puede saber más sobre las estructuras dinámicas que hay disponibles en la plataforma .Net consultando la referencia en línea de MSDN (mucha de la cual está sin traducir al español):

http://msdn.microsoft.com/es-es/library/system.collections(en-us,VS.71).aspx#

11.7. 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 en C#           */
/*  pilaEstatica.cs         */
/*                          */
/*  Ejemplo de clase "Pila"  */
/*  basada en un array      */
/*                          */
/*  Introduccion a C#,      */
/*    Nacho Cabanes         */
/*--------------------------*/

using System;
using System.Collections;

public class PilaString  {

   string[] datosPila;
   int posicionPila;
   const int MAXPILA = 200;

   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 );
      }

    }

    // 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;
    }

}  // Fin de la clase
```

Ejercicios propuestos:

- **(11.7.1)** Usando esta misma estructura de programa, crear 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".
- **(11.7.2)** Crear una clase "ListaOrdenada", que almacene un único dato (no un par clave-valor como los SortedList). Debe contener un método "Insertar", que añadirá un nuevo dato en orden en el array, y un "Extraer(n)", que obtenga un elemento de la lista (el número "n"). Deberá almacenar "strings".
- **(11.7.3)** Crea una pila de "doubles", usando internamente un ArrayList en vez de un array.
- **(11.7.4)** Crea una cola que almacene un bloque de datos (struct, con los campos que tú elijas) usando un ArrayList.
- **(11.7.5)** Crea una lista ordenada (de "strings") usando un ArrayList.

11.8. Introducción a los "generics"

Una ventaja, pero también a la vez un inconveniente, de las estructuras dinámicas que hemos visto, es que permiten guardar datos de cualquier tipo, incluso datos de distinto tipo en una