INAP | CAMPUS VIRTUAL

Programación II

Ministerio de Modernización
Presidencia de la Nación

Módulo I

Programación II

Módulo I: Introducción a las funciones

Módulo I:

- **Conceptos basicos sobre funciones.**
- **Parametros de una función.**
- **Valor devuelto por una función.**
- **El valor void.**
- **Variables locales y variables globales.**
- **Funciones útiles.**
- **Recursividad.**

Introducción a las funciones

Diseño modular de programas: Descomposición modular

Hasta ahora hemos estado pensando los pasos que deberíamos dar para resolver un cierto problema, y hemos creado programas a partir de cada uno de esos pasos. Esto es razonable cuando los problemas son sencillos, pero puede no ser la mejor forma de actuar cuando se trata de algo más complicado.

A partir de ahora vamos a empezar a intentar descomponer los problemas en trozos más pequeños, que sean más fáciles de resolver. Esto nos puede suponer varias ventajas:

- Cada "trozo de programa" independiente será más fácil de programar, al realizar una función breve y concreta.
- El "programa principal" será más fácil de leer, porque no necesitará contener todos los detalles de cómo se hace cada cosa.
- Evitaremos mucho código repetitivo.
- Podremos repartir el trabajo, para que cada persona se encargue de realizar un "trozo de programa", y finalmente se integrará el trabajo individual de cada persona (aunque para proyectos realmente grandes, pronto veremos una alternativa que hace que el reparto y la integración sean más sencillos).

Esos "trozos" de programa son lo que suele llamar "subrutinas", "procedimientos" o "funciones". En el lenguaje C y sus derivados (entre los que está C#), el nombre que más se usa es el de **funciones**.

Conceptos básicos sobre funciones

En C#, al igual que en C y los demás lenguajes derivados de él, todos los "trozos de programa" son funciones, incluyendo el propio cuerpo de programa, **Main**. De hecho, la forma básica de **definir** una función será indicando su nombre seguido de unos paréntesis vacíos, como hacíamos con "Main", y precediéndolo por ciertas palabras reservadas, como "public static **void**", cuyo significado iremos viendo muy pronto. Después, entre llaves indicaremos todos los pasos que queremos que dé esa "porcion de programa".

Por ejemplo, podríamos crear una función llamada "Saludar", que escribiera varios mensajes en la pantalla:

```
public static void Saludar()
{
  Console.Write("Bienvenido al programa ");
  Console.WriteLine("de ejemplo");
  Console.WriteLine("Espero que estés bien");
}
```

Ahora desde dentro del cuerpo de nuestro programa, podríamos "**llamar**" a esa función:

```
public static void Main()
{
  Saludar();
  …
}
```

Así conseguimos que nuestro programa principal sea más fácil de leer.

Un detalle importante: tanto la función habitual "Main" como la nueva función "Saludar" serían parte de nuestra "class", es decir, el fuente completo sería así:

```
using System;

public class Ejemplo_05_02a
{

    public static void Saludar()
    {
        Console.Write("Bienvenido al programa ");
        Console.WriteLine("de ejemplo");
        Console.WriteLine("Espero que estés bien");
    }

    public static void Main()
    {
        Saludar();
        Console.WriteLine("Nada más por hoy...");
    }

}
```

Como ejemplo más detallado, la parte principal de una agenda o de una base de datos simple, podría ser simplemente:

```
LeerDatosDeFichero();

do {
    MostrarMenu():
    opcion = PedirOpcion();
    switch( opcion ) {
        case 1: BuscarDatos(); break;
        case 2: ModificarDatos(); break;
        case 3: AnadirDatos(); break;
        ...
```

Parámetros de una función

Es muy frecuente que nos interese indicarle a nuestra función ciertos datos con los que queremos que trabaje. Los llamaremos "parámetros" y los indicaremos dentro del paréntesis que sigue al nombre de la función, separados por comas. Para cada uno de ellos, deberemos indicar su tipo de datos (por ejemplo "int") y luego su nombre.

```
public static void EscribirNumeroReal( float n )
{
    Console.WriteLine( n.ToString("#.###") );
}
```

Y esta función se podría usar desde el cuerpo de nuestro programa así:

```
EscribirNumeroReal(2.3f);
```

(recordemos que el sufijo "f" sirve para indicar al compilador que trate ese número como un "float", porque de lo contrario, al ver que tiene cifras decimales, lo tomaría como "double", que permite mayor precisión... pero a cambio nosotros tendríamos un mensaje de error en nuestro programa, diciendo que estamos pasando un dato "double" a una función que espera un "float").

```
using System;

public class Ejemplo_05_03a
{

    public static void EscribirNumeroReal( float n )
    {
        Console.WriteLine( n.ToString("#.###") );
    }

    public static void Main()
    {
        float x;

        x= 5.1f;
        Console.WriteLine("El primer numero real es: ");
        EscribirNumeroReal(x);
        Console.WriteLine(" y otro distinto es: ");
        EscribirNumeroReal(2.3f);
    }

}
```

Como ya hemos anticipado, si hay más de un parámetro, deberemos indicar el tipo y el nombre para cada uno de ellos (incluso si todos son del mismo tipo), y separarlos entre comas:

```
public static void EscribirSuma( int a, int b )
{
    ...
}
```

De modo que un programa completo de ejemplo para una función con dos parámetros podría ser:

```
using System;

public class Ejemplo_05_03b
{

    public static void EscribirSuma( int a, int b )
    {
        Console.Write( a+b );
    }

    public static void Main()
    {
        Console.Write("La suma de 4 y 7 es: ");
        EscribirSuma(4, 7);
    }

}
```

Como se ve en estos ejemplos, se suele seguir un par de **convenios**:

- Ya que las funciones expresan acciones, en general su nombre será un verbo.
- En C# se recomienda que los elementos públicos se escriban comenzando por una letra mayúscula (y recordemos que, hasta que conozcamos las alternativas y el motivo para usarlas, nuestras funciones comienzan con la palabra "public"). Este criterio depende del lenguaje. Por ejemplo, en lenguaje Java es habitual seguir el convenio de que los nombres de las funciones deban comenzar con una letra minúscula.

Valor devuelto por una función. El valor "void"

Cuando queremos dejar claro que una función no tiene que devolver ningún valor, podemos hacerlo indicando al principio que el tipo de datos va a ser "void" (nulo), como hacíamos hasta ahora con "Main" y como hicimos con nuestra función "Saludar".

Pero eso no es lo que ocurre con las funciones matemáticas que estamos acostumbrados a manejar: sí devuelven un valor, que es el resultado de una operación (por ejemplo, la raíz cuadrada de un número tiene como resultado otro número).

De igual modo, para nosotros también será habitual crear funciones que realicen una serie de cálculos y nos "devuelvan" (return, en inglés) el resultado de esos cálculos, para poderlo usar desde cualquier otra parte de nuestro programa. Por ejemplo, podríamos crear una función para elevar un número entero al cuadrado así:

```
public static int Cuadrado ( int n )
{
    return n*n;
}
```

En este caso, nuestra función no es "void", sino "int", porque va a **devolver un número entero**. Eso sí, todas nuestras funciones seguirán siendo "public static" hasta que avancemos un poco más.

Podríamos usar el resultado de esa función como si se tratara de un número o de una variable, así:

```
resultado = Cuadrado( 5 );
```

En general, en las operaciones matemáticas, no será necesario que el nombre de la función sea un verbo. El programa debería ser suficientemente legible si el nombre expresa qué operación se va a realizar en la función.

Un programa más detallado de ejemplo podría ser:

```
using System;

public class Ejemplo_05_04a
{

    public static int Cuadrado ( int n )
    {
        return n*n;
    }

    public static void Main()
    {
        int numero;
        int resultado;
        numero= 5;
        resultado = Cuadrado(numero):
        Console.WriteLine("El cuadrado del numero {0} es {1}",
            numero, resultado);
        Console.WriteLine(" y el de 3 es {0}", Cuadrado(3));
    }

}
```

Podremos devolver cualquier tipo de datos, no sólo números enteros. Como segundo ejemplo, podemos hacer una función que nos diga cuál es el mayor de dos números reales así:

```
public static float Mayor ( float n1, float n2 )
{
    if (n1 > n2)
        return n1;
    else
        return n2;
}
```

Como se ve en este ejemplo, una función puede tener más de un "return".

En cuanto se alcance un "return", se sale de la función por completo. Eso puede hacer que una función mal diseñada haga que el compilador nos dé un aviso de "código inalcanzable", como en el siguiente ejemplo:

```
public static string Inalcanzable()
{
    return "Aquí sí llegamos";

    string ejemplo = "Aquí no llegamos";
    return ejemplo;
}
```

Variables locales y variables globales

Hasta ahora, hemos declarado las variables dentro de "Main". Ahora nuestros programas tienen varios "bloques", así que las variables se comportarán de forma distinta según donde las declaremos.

Las variables se pueden declarar dentro de un bloque (una función), y entonces sólo ese bloque las conocerá, no se podrán usar desde ningún otro bloque del programa. Es lo que llamaremos "variables **locales**".

Por el contrario, si declaramos una variable al comienzo del programa, fuera de todos los "bloques" de programa, será una "**variable global**", a la que se podrá acceder desde cualquier parte. Por ahora, para nosotros, una variable global deberá llevar siempre la palabra "**static**" (dentro de poco veremos el motivo real y cuándo no será necesario).

En general, deberemos intentar que la **mayor cantidad de variables** posible sean **locales** (lo ideal sería que todas lo fueran). Así hacemos que cada parte del programa trabaje con sus propios datos, y ayudamos a evitar que un error en un trozo de programa pueda afectar al resto. La forma correcta de pasar datos entre distintos trozos de programa no es a través de variables globales, sino usando los parámetros de cada función y los valores devueltos, como en el anterior ejemplo. Aun así, esta restricción es menos grave en lenguajes modernos, como C#, que en otros lenguajes más antiguos, como C, porque, como veremos en el próximo tema, el hecho de descomponer un programa en varias clases minimiza los efectos negativos de esas variables que se comparten entre varias funciones, además de que muchas veces tendremos datos compartidos, que no serán realmente "variables globales" sino datos específicos del problema, que llamaremos "**atributos**"

Vamos a ver el uso de variables locales con un ejemplo. Crearemos una función que calcule la potencia de un número entero (un número elevado a otro), y el cuerpo del programa que la use. La forma de conseguir elevar un número a otro será a base de multiplicaciones, es decir:

```
3 elevado a 5   =   3 · 3 · 3 · 3 · 3
```

(multiplicamos 5 veces el 3 por sí mismo). En general, como nos pueden pedir cosas como "6 elevado a 100" (o en general números que pueden ser grandes), usaremos la orden "for" para multiplicar tantas veces como haga falta:

```
using System;

public class Ejemplo_05_05a
{

    public static int Potencia(int nBase, int nExponente)
    {
        int temporal = 1;        // Valor inicial que voy incrementando

        for(int i=1; i<=nExponente; i++)  // Multiplico "n" veces
            temporal *= nBase;            // Para aumentar el valor temporal

        return temporal; // Al final, obtengo el valor que buscaba
    }


    public static void Main()
    {
        int num1, num2;

        Console.WriteLine("Introduzca la base: ");
        num1 = Convert.ToInt32( Console.ReadLine() );

        Console.WriteLine("Introduzca el exponente: ");
        num2 = Convert.ToInt32( Console.ReadLine() );

        Console.WriteLine("{0} elevado a {1} vale {2}",
            num1, num2, Potencia(num1,num2));
    }

}
```

En este caso, las variables "temporal" e "i" son locales a la función "potencia": para "Main" no existen. Si en "Main" intentáramos hacer *i=5;* obtendríamos un mensaje de error. De igual modo, "num1" y "num2" son locales para "Main": desde la función "potencia" no podemos acceder a su valor (ni para leerlo ni para modificarlo), sólo desde "Main". Este ejemplo no contiene ninguna variable global.

(Nota: el parámetro no se llama "base" sino "nBase" porque la palabra "base" es una palabra reservada en C#, que no podremos usar como nombre de variable).

Los conflictos de nombres en las variables

¿Qué ocurre si damos el mismo nombre a dos variables locales distintas? Vamos a comprobarlo con un ejemplo:

```
using System;

public class Ejemplo_05_06a
{
    public static void CambiaN()
    {
        int n = 7;
        n ++;
    }


    public static void Main()
    {
        int n = 5;
        Console.WriteLine("n vale {0}", n);
        CambiaN():
        Console.WriteLine("Ahora n vale {0}", n);
    }

}
```

El resultado de este programa es:

```
n vale 5
Ahora n vale 5
```

¿Por qué? Sencillo: tenemos una variable local dentro de "cambiaN" y otra dentro de "Main". El hecho de que las dos variables tengan el mismo nombre no afecta al funcionamiento del programa, siguen siendo distintas, porque cada una está en un bloque ("ámbito") distinto.

Si la variable es "global", declarada fuera de estas funciones, sí será accesible por todas ellas:

```
using System;

public class Ejemplo_05_06b
{

    static int n = 7;

    public static void CambiaN()
    {
        n ++;
    }

    public static void Main()
    {
        Console.WriteLine("n vale {0}", n);
        CambiaN();
        Console.WriteLine("Ahora n vale {0}", n);
    }

}
```

Modificando parámetros

Podemos modificar el valor de un dato que recibamos como parámetro, pero posiblemente el resultado no será el que esperamos. Vamos a verlo con un ejemplo:

```
using System;

public class Ejemplo_05_07a
{

    public static void Duplicar(int x)
    {
        Console.WriteLine("  El valor recibido vale {0}", x);
        x = x * 2;
        Console.WriteLine("  y ahora vale {0}", x);
    }

    public static void Main()
    {
        int n = 5;
        Console.WriteLine("n vale {0}", n);
        Duplicar(n);
        Console.WriteLine("Ahora n vale {0}", n);
    }

}
```

El resultado de este programa será:

```
n vale 5
  El valor recibido vale 5
  y ahora vale 10
Ahora n vale 5
```

Vemos que al salir de la función, **no se conservan los cambios** que hagamos a esa variable que se ha recibido como parámetro.

Esto se debe a que, si no indicamos otra cosa, los parámetros "**se pasan por valor**", es decir, la función no recibe los datos originales, sino una copia de ellos. Si modificamos algo, estamos cambiando una copia de los datos originales, no dichos datos.

Si queremos que las modificaciones se conserven, basta con hacer un pequeño cambio: indicar que la variable se va a pasar "**por referencia**", lo que se indica usando la palabra "ref", tanto en la declaración de la función como en la llamada, así:

```
using System;

public class Ejemplo_05_07b
{

    public static void Duplicar(ref int x)
    {
        Console.WriteLine("  El valor recibido vale {0}", x);
        x = x * 2;
        Console.WriteLine("  y ahora vale {0}", x);
    }

    public static void Main()
    {
        int n = 5;
        Console.WriteLine("n vale {0}", n);
        Duplicar(ref n);
        Console.WriteLine("Ahora n vale {0}", n);
    }

}
```

En este caso sí se modifica la variable n:

```
n vale 5
  El valor recibido vale 5
  y ahora vale 10
Ahora n vale 10
```

El hecho de poder modificar valores que se reciban como parámetros abre una posibilidad que no se podría conseguir de otra forma: con "return" sólo se puede devolver un valor de una función, pero con parámetros pasados por referencia podríamos **devolver más de un dato**. Por ejemplo, podríamos crear una función que intercambiara los valores de dos variables:

```
public static void Intercambia(ref int x, ref int y)
```

La posibilidad de pasar parámetros por valor y por referencia existe en la mayoría de lenguajes de programación. En el caso de C# existe alguna posibilidad adicional que no existe en otros lenguajes, como los "**parámetros de salida**". No podemos llamar a una función que tenga parámetros por referencia si los parámetros no tienen valor inicial. Por ejemplo, una función que devuelva la primera y segunda letra de una frase sería así:

```
using System;

public class Ejemplo_05_07c
{

    public static void DosPrimerasLetras(string cadena,
        out char l1, out char l2)
    {
        l1 = cadena[0];
        l2 = cadena[1];
    }

    public static void Main()
    {
        char letra1, letra2;
        DosPrimerasLetras("Nacho". out letra1. out letra2):
        Console.WriteLine("Las dos primeras letras son {0} y {1}",
            letra1, letra2);
    }

}
```

Si pruebas este ejemplo, verás que no compila si cambias "out" por "ref", a no ser que des valores iniciales a "letra1" y "letra2".

El orden no importa

En algunos lenguajes, una función debe estar declarada antes de usarse. Esto no es necesario en C#. Por ejemplo, podríamos rescribir el ejemplo 05_07b, de modo que "Main" aparezca en primer lugar y "Duplicar" aparezca después, y seguiría compilando y funcionando igual:

```
using System;

public class Ejemplo_05_08a
{

    public static void Main()
    {
        int n = 5:
        Console.WriteLine("n vale {0}", n);
        Duplicar(ref n);
        Console.WriteLine("Ahora n vale {0}", n);
    }

    public static void Duplicar(ref int x)
    {
        Console.WriteLine("  El valor recibido vale {0}", x);
        x = x * 2:
        Console.WriteLine("  y ahora vale {0}", x);
    }

}
```

Algunas funciones útiles

Números aleatorios

En un programa de gestión o una utilidad que nos ayuda a administrar un sistema, no es habitual que podamos permitir que las cosas ocurran al azar. Pero los juegos se encuentran muchas veces entre los ejercicios de programación más completos, y para un juego sí suele ser conveniente que haya algo de azar, para que una partida no sea exactamente igual a la anterior.

Generar números al azar ("números aleatorios") usando C# no es difícil: debemos crear un objeto de tipo "Random" (una única vez), y luego llamaremos a "Next" cada vez que queramos obtener valores entre dos extremos:

```
// Creamos un objeto Random
Random generador = new Random();

// Generamos un número entre dos valores dados
// (el segundo límite no está incluido)
int aleatorio = generador.Next(1, 101);
```

También, una forma simple de obtener un único número "casi al azar" entre 0 y 999 es tomar las milésimas de segundo de la hora actual:

```
int falsoAleatorio = DateTime.Now.Millisecond;
```

Pero esta forma simplificada no sirve si necesitamos obtener dos números aleatorios a la vez, porque los dos se obtendrían en el mismo milisegundo y tendrían el mismo valor; en ese caso, no habría más remedio que utilizar "Random" y llamar dos veces a "Next".

Vamos a ver un ejemplo, que muestre en pantalla dos números al azar:

```
using System;

public class Ejemplo_05_09_01a
{
    public static void Main()
    {
        Random r = new Random();
        int aleatorio = r.Next(1, 11);
        Console.WriteLine("Un número entre 1 y 10: {0}", aleatorio);
        int aleatorio2 = r.Next(10, 21);
        Console.WriteLine("Otro entre 10 y 20: {0}", aleatorio2);
    }
}
```

Funciones matemáticas

En C# tenemos muchas funciones matemáticas predefinidas, como:

- Abs(x): Valor absoluto
- Acos(x): Arco coseno
- Asin(x): Arco seno
- Atan(x): Arco tangente
- Atan2(y,x): Arco tangente de y/x (por si x o y son 0)
- Ceiling(x): El valor entero superior a x y más cercano a él
- Cos(x): Coseno
- Cosh(x): Coseno hiperbólico
- Exp(x): Exponencial de x (e elevado a x)
- Floor(x): El mayor valor entero que es menor que x
- Log(x): Logaritmo natural (o neperiano, en base "e")
- Log10(x): Logaritmo en base 10
- Pow(x,y): x elevado a y
- Round(x, cifras): Redondea un número
- Sin(x): Seno
- Sinh(x): Seno hiperbólico
- Sqrt(x): Raíz cuadrada
- Tan(x): Tangente
- Tanh(x): Tangente hiperbólica

(casi todos ellos usan parámetros X e Y de tipo "double"; en el caso de las funciones trigonométricas, el ángulo se debe indicar en radianes, no en grados)

y también tenemos una serie de constantes como

- E, el número "e", con un valor de 2.71828...
- PI, el número "Pi", 3.14159...

Todas ellas se usan **precedidas por "Math."**

La mayoría de ellas son específicas para ciertos problemas matemáticos, especialmente si interviene la trigonometría o si hay que usar logaritmos o exponenciales. Pero vamos a destacar las que sí pueden resultar útiles en situaciones más variadas, que casi cualquier programador pueda necesitar:

- La raíz cuadrada de 4 se calcularía haciendo x = Math.Sqrt(4);
- La potencia: para elevar 2 al cubo haríamos y = Math.Pow(2, 3);
- El valor absoluto: para trabajar sólo con números positivos usaríamos n = Math.Abs(x);

Un ejemplo más avanzado, usando funciones trigonométricas, que calculase el "coseno de 45 grados" podría ser:

```
using System;

public class Ejemplo_05_09_02a
{
    public static void Main()
    {
        double anguloGrados = 45;
        double anguloRadianes = anguloGrados * Math.PI / 180.0;

        Console.WriteLine("El coseno de 45 grados es: {0}",
            Math.Cos(anguloRadianes));
    }
}
```

Otras funciones

En C# hay muchas más funciones de lo que parece. De hecho, salvo algunas palabras reservadas (int, float, string, if, switch, for, do, while...), gran parte de lo que hasta ahora hemos llamado "órdenes", son realmente "funciones", como Console.ReadLine (que devuelve la cadena que se ha introducido por teclado) o Console.WriteLine (que es "void", no devuelve nada). Nos iremos encontrando con otras muchas funciones a medida que avancemos.

Recursividad

La recursividad consiste en resolver un problema a partir de casos más simples del mismo problema. Una función recursiva es aquella que se "llama a ella misma", reduciendo la complejidad paso a paso hasta llegar a un caso trivial.

Dentro de las matemáticas tenemos varios ejemplos de funciones recursivas. Uno clásico es el "factorial de un número":

El factorial de 1 es 1:

```
1! = 1
```

Y el factorial de un número arbitrario es el producto de ese número por los que le siguen, hasta llegar a uno:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

(por ejemplo, el factorial de 4 es $4 \cdot 3 \cdot 2 \cdot 1 = 24$)

Si pensamos que el factorial de n-1 es

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

Entonces podemos escribir el factorial de un número a partir del factorial del siguiente número:

$$n! = n \cdot (n-1)!$$

Esta es la definición recursiva del factorial, ni más ni menos. Esto se podría programar así:

```
using System;

public class Ejemplo_05_10a
{

    public static long Factorial(int n)
    {
        if (n==1)               // Aseguramos que termine (caso base)
            return 1;
        return n * Factorial(n-1);  // Si no es 1, sigue la recursión
    }

    public static void Main()
    {
        int num;
        Console.WriteLine("Introduzca un número entero: ");
        num = Convert.ToInt32(System.Console.ReadLine());
        Console.WriteLine("Su factorial es: {0}", Factorial(num));
    }

}
```

Dos consideraciones importantes:

- Atención a la primera parte de la función recursiva: es **muy importante** comprobar que hay salida de la función, para que nuestro programa no se quede dando vueltas todo el tiempo y deje el ordenador (o la tarea actual) "colgado". Debemos encontrar un "caso trivial" que alcanzar, y un modo de disminuir la complejidad del problema acercándolo a ese caso.

- Los factoriales **crecen rápidamente**, así que no conviene poner números grandes: el factorial de 16 es 2.004.189.184, luego a partir de 17 podemos obtener resultados erróneos, si usamos números enteros "normales".

¿Qué utilidad tiene esto? Más de la que parece: muchos problemas complicados se pueden expresar a partir de otro más sencillo. En muchos de esos casos, ese problema se podrá expresar de forma recursiva. Los ejercicios propuestos te ayudarán a descubrir otros ejemplos de situaciones en las que se puede aplicar la recursividad.

Parámetros y valor de retorno de "Main"

Es muy frecuente que un programa llamado desde la "línea de comandos" tenga ciertas opciones que le indicamos como argumentos. Por ejemplo, bajo Linux o cualquier otro sistema operativo de la familia Unix, podemos ver la lista detallada de ficheros que terminan en .cs haciendo

```
ls -l *.cs
```

En este caso, la orden sería "ls", y las dos opciones (argumentos o parámetros) que le indicamos son "-l" y "*.cs".

La orden equivalente en MsDos y en el intérprete de comandos de Windows sería

```
dir *.cs
```

Ahora la orden sería "dir", y el parámetro es "*.cs".

Pues bien, estas opciones que se le pasan al programa se pueden leer desde C#. Se hace indicando un parámetro especial en Main, un array de strings:

```
static void Main(string[] args)
```

Para conocer esos parámetros lo haríamos de la misma forma que se recorre habitualmente un array cuyo tamaño no conocemos: con un "for" que termine en la longitud ("Length") del array:

```
for (int i = 0; i < args.Length; i++)
{
    System.Console.WriteLine("El parametro {0} es: {1}",
      i, args[i]);
}
```

Por otra parte, si queremos que nuestro programa **se interrumpa** en un cierto punto, podemos usar la orden "Environment.Exit". Su manejo habitual es algo como

```
Environment.Exit(1);
```

Es decir, entre paréntesis indicamos un cierto código, que suele ser (por convenio) un 0 si no ha habido ningún error, u otro código distinto en caso de que sí exista algún error.

Este valor se podría comprobar desde el sistema operativo. Por ejemplo, en MsDos y Windows se puede leer desde un fichero BAT o CMD usando "IF ERRORLEVEL", así:

```
IF ERRORLEVEL 1 ECHO Ha habido un error en el programa
```

Una forma alternativa de que "Main" indique errores al sistema operativo es no declarándolo como "void", sino como "int", y empleando entonces la orden "return" cuando nos interese (igual que antes, por convenio, devolviendo 0 si todo ha funcionado correctamente u otro código en caso contrario):

```
public static int Main(string[] args)
{
  ...
  return 0;
}
```

Un ejemplo que pusiera todo esto a prueba podría ser:

```
using System;

public class Ejemplo_05_11a
{
    public static int Main(string[] args)
    {
        Console.WriteLine("Parámetros: {0}", args.Length);

        for (int i = 0; i < args.Length; i++)
        {
            Console.WriteLine("El parámetro {0} es: {1}",
                i, args[i]);
        }

        if (args.Length == 0)
        {
            Console.WriteLine("No ha indicado ningún parámetro!");
            Environment.Exit(1);
        }

        return 0;
    }
}
```