5. Introducción a las funciones

5.1. 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 subproblemas, 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 más breve y más 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, que será descomponer el proyecto en varias "clases" que cooperan entre ellas).

Estos "trozos" de programa se suelen llamar "funciones", "procedimientos" o "subrutinas", según el lenguaje del que se trate. En el caso de C y sus derivados (entre los que está C#), el nombre que más se emplea es el de **funciones**.

5.2. 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 "static **void**", cuyo significado iremos viendo muy pronto. Después, entre llaves indicaremos todos los pasos que queremos que realice ese "trozo de programa".

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

```
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:

```
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", de modo que el fuente completo sería así:

```
// Ejemplo_05_02a.cs
// Función "Saludar"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_02a
{

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

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

}
```

El flujo de un programa que contenga funciones ya no será tan claramente "de arriba a abajo", porque cada "llamada" a una función interrumpirá ese orden, y en

ese punto del programa se "saltará" a esa funcione y se seguirán los pasos que dicha función indique. Por ejemplo, el fuente anterior se comportaría igual que éste:

```
// Ejemplo_05_02b.cs
// Eliminando la función "Saludar"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_02b
{
    static void Main()
    {
        Console.WriteLine("Empezamos...");
        Console.Write("Bienvenido al programa ");
        Console.WriteLine("de ejemplo");
        Console.WriteLine("Espero que estés bien");
        Console.WriteLine("Nada más por hoy...");
    }
}
```

Como ejemplo (incompleto) más real, la parte principal de una agenda o de una base de datos simple como las que hicimos en el tema anterior, podría ser simplemente:

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

Ejercicios propuestos:

(5.2.1) Crea una función llamada "BorrarPantalla", que borre la pantalla dibujando 25 líneas en blanco. Crea también un "Main" que permita probarla.

(5.2.2) Crea una función llamada "DibujarCuadrado3x3", que dibuje un cuadrado formato por 3 filas con 3 asteriscos cada una. Incluye un "Main" para probarla.

(5.2.3) Descompón en funciones la base de datos de ficheros (ejemplo 04_06a), de modo que el "Main" sea breve y más legible (Pista: las variables que se compartan entre varias funciones deberán estar fuera de todas ellas, y deberán estar precedidas por la palabra "static").

5.3. 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**.

Por ejemplo, si escribimos en pantalla números reales con frecuencia, nos puede resultar útil crear una función auxiliar que nos los muestre con el formato que nos interese (que podría ser con exactamente 3 decimales). Lo podríamos hacer así:

```
    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 obtendríamos un mensaje de error en nuestro programa, diciendo que estamos pasando un dato "double" a una función que espera un "float").

El programa completo podría quedar así:

```
// Ejemplo_05_03a.cs
// Función "EscribirNumeroReal"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_03a
{

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

    static void Main()
    {
        float x;

        x= 5.1f;
        Console.WriteLine("El primer número 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:

```
// Ejemplo_05_03b.cs
// Función "EscribirSuma"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_03b
{

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

    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**.

- También por convenio, en C# los nombres de las funciones se suelen escribir comenzando con una letra **mayúscula** (especialmente si son públicas, detalle que veremos dentro de poco). 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.

Ejercicios propuestos:

(5.3.1) Crea una función "DibujarCuadrado" que dibuje en pantalla un cuadrado de asteriscos del ancho (y alto) que se indique como parámetro. Completa el programa con un Main que permita probarla.

(5.3.2) Crea una función "DibujarRectangulo" que dibuje en pantalla un rectángulo del ancho y alto que se indiquen como parámetros, en ese orden. Incluye un Main para probarla.

(5.3.3) Crea una función "DibujarRectanguloHueco" que dibuje en pantalla un rectángulo hueco del ancho y alto que se indiquen como parámetros, formado por una letra que también se indique como parámetro. Completa el programa con un Main que pida esos datos al usuario y dibuje el rectángulo.

(5.3.4) Crea una función "EscribirRepetido", que reciba un carácter y un número, y escriba ese carácter tantas veces como indique ese número (todas ellas en la misma línea).

(5.3.5) Crea una nueva versión de la función "DibujarRectangulo", que se apoye en la "EscribirRepetido" que acabas de crear.

5.4. Valor devuelto por una función. El valor "void"

Hasta ahora hemos creado funciones que escribían textos en pantalla y que no "devolvían" ningún resultado. Por eso, antes del nombre de la función escribíamos la palabra "void" (nulo), como ocurría hasta ahora con "Main" y como hemos hecho con nuestra función "Saludar".

Pero eso no es lo que sucede con las funciones matemáticas que estamos acostumbrados a manejar, como la raíz cuadrada: sí devuelven un valor, que es el resultado de una operación.

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:

```
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 "static" hasta

que profundicemos un poco más y veamos el significado de esa palabra, en el próximo tema.

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:

```
// Ejemplo_05_04a.cs
// Función "Cuadrado"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_04a
{

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

    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 permita saber cuál es el mayor de dos números reales así:

```
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", si bien puede resultar desaconsejable en funciones más grandes que esta, porque hará que sea más difícil seguir el flujo de ese fragmento de programa. Por eso, la función anterior se podría escribir también así, con un solo "return", alternativa que generalmente resultará más legible:

```
static float Mayor ( float n1, float n2 )
{
    float mayor = n1;
    if (n2 > n1)
        mayor = n2;
    return mayor;
}
```

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:

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

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

Ejercicios propuestos:

(5.4.1) Crea una función "Cubo" que calcule el cubo de un número real (float) que se indique como parámetro. El resultado deberá ser otro número real. Prueba esta función para calcular el cubo de 3.2 y el de 5.

(5.4.2) Crea una función "Menor" que calcule el menor de dos números enteros que recibirá como parámetros. El resultado será otro número entero.

(5.4.3) Crea una función llamada "Signo", que reciba un número real, y devuelva un número entero con el valor: -1 si el número es negativo, 1 si es positivo o 0 si es cero.

(5.4.4) Crea una función "Inicial", que devuelva la primera letra de una cadena de texto. Prueba esta función para calcular la primera letra de la frase "Hola".

(5.4.5) Crea una función "UltimaLetra", que devuelva la última letra de una cadena de texto. Prueba esta función para calcular la última letra de la frase "Hola".

(5.4.6) Crea una función "MostrarPerimSuperfCuadrado" que reciba un número entero y calcule y muestre en pantalla el valor del perímetro y de la superficie de

un cuadrado que tenga como lado el número que se ha indicado como parámetro, sin devolver ningún valor.

5.5. 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 en ese caso 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, se tratará una "**variable global**", a la que se podrá acceder desde cualquier parte. Por ahora, para nosotros, una variable global deberá ser precedida por 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 conseguimos que cada parte del programa trabaje con sus propios datos, y ayudamos a evitar que un error en un fragmento 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 (05_04). 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**", como también veremos en el próximo tema.

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

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

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

```
// Ejemplo_05_05a.cs
// Ejemplo de función con variables locales
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_05a
{

    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
    }


    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).

Ejercicios propuestos:

(5.5.1) Crea una función "PedirEntero", que reciba como parámetros el texto que se debe mostrar en pantalla, el valor mínimo aceptable y el valor máximo aceptable. Esta función deberá pedir al usuario que introduzca el valor, tantas veces como sea necesario, deberá volvérselo a pedir en caso de error, y deberá devolver un valor correcto. Pruébalo con un programa que pida al usuario un año entre 1800 y 2100.

(5.5.2) Crea una función "EscribirTablaMultiplicar", que reciba como parámetro un número entero, y escriba la tabla de multiplicar de ese número (por ejemplo, para el 3 deberá llegar desde "3x0=0" hasta "3x10=30").

(5.5.3) Crea una función "EsPrimo", que reciba un número y devuelva el valor booleano "true" si es un número primo o "false" en caso contrario.

(5.5.4) Crea una función "ContarLetra", que reciba una cadena y una letra, y devuelva la cantidad de veces que dicha letra aparece en la cadena. Por ejemplo, si la cadena es "Barcelona" y la letra es 'a', debería devolver 2 (porque la "a" aparece 2 veces).

(5.5.5) Crea una función "SumarCifras" que reciba un numero cualquiera y que devuelva como resultado la suma de sus dígitos. Por ejemplo, si el número fuera 123, la suma sería 6.

(5.5.6) Crea una función "DibujarTriángulo" que reciba una letra y un número, y escriba un "triángulo" formado por esa letra, que tenga como anchura inicial la que se ha indicado. Por ejemplo, si la letra es * y la anchura es 4, debería escribir

****

***

**

*

Adaptar un programa realizado anteriormente (y que tenga una cierta complejidad) para **dividirlo en funciones** supondrá decidir qué variables se deben compartir entre distintas funciones (y serán variables globales, precedidas por "static") y qué variables no es necesario compartir (que serán variables locales). Por ejemplo, el "ejemplo_04_06a" se podría convertir en algo como esto:

```
// Ejemplo_05_05b.cs
// Versión del Ejemplo_04_06a.cs ampliada usando funciones
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_05b
{

    struct tipoFicha
    {
        public string nombreFich;   // Nombre del fichero
```

```
    public long tamanyo;        // El tamaño en KB
}

// Variables "globales"
static tipoFicha[] fichas;
static int numeroFichas;

static void MostrarMenu()
{
    Console.WriteLine();
    Console.WriteLine("Escoja una opción:");
    Console.WriteLine("1.- Añadir datos de un nuevo fichero");
    Console.WriteLine("2.- Mostrar los nombres de todos los ficheros");
    Console.WriteLine("3.- Mostrar ficheros por encima de un cierto tamaño");
    Console.WriteLine("4.- Ver datos de un fichero");
    Console.WriteLine("5.- Salir");
}

static int PedirOpcion()
{
    return Convert.ToInt32(Console.ReadLine());
}

static void Anyadir()
{
    if (numeroFichas < 1000)
    {  // Si queda hueco
        Console.WriteLine("Introduce el nombre del fichero: ");
        fichas[numeroFichas].nombreFich = Console.ReadLine();
        Console.WriteLine("Introduce el tamaño en KB: ");
        fichas[numeroFichas].tamanyo = Convert.ToInt32(
           Console.ReadLine());
        // Y ya tenemos una ficha más
        numeroFichas++;
    }
    else   // Si no hay hueco para más fichas, avisamos
        Console.WriteLine("Máximo de fichas alcanzado (1000)!");
}

static void MostrarTodos()
{
    for (int i = 0; i < numeroFichas; i++)
        Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
          fichas[i].nombreFich, fichas[i].tamanyo);
}

static void MostrarPorTamanyo()
{
    Console.WriteLine("¿A partir de que tamaño quieres ver?");
    long tamanyoBuscar = Convert.ToInt64(Console.ReadLine());
    for (int i = 0; i < numeroFichas; i++)
        if (fichas[i].tamanyo >= tamanyoBuscar)
            Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
              fichas[i].nombreFich, fichas[i].tamanyo);
}

static void MostrarUnDato()
{
    Console.WriteLine("¿De qué fichero quieres ver todos los datos?");
    string textoBuscar = Console.ReadLine();
    for (int i = 0; i < numeroFichas; i++)
        if (fichas[i].nombreFich == textoBuscar)
            Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
              fichas[i].nombreFich, fichas[i].tamanyo);
}
```

```
    static void AvisarFin()
    {
        Console.WriteLine("Fin del programa");
    }

    static void AvisarError()
    {
        Console.WriteLine("Opción desconocida!");
    }

    static void Main()
    {
        // Variables locales
        int opcion;

        // Valores iniciales a las variables compartidas
        fichas = new tipoFicha[1000];
        numeroFichas = 0;

        do
        {
            MostrarMenu();
            opcion = PedirOpcion();
            switch (opcion )
            {
                case 1: Anyadir(); break;
                case 2: MostrarTodos(); break;
                case 3: MostrarPorTamanyo(); break;
                case 4: MostrarUnDato(); break;
                case 5: AvisarFin(); break;
                default: AvisarError(); break;
            }
        } while (opcion != 5);
    }
}
```

En un ejemplo como éste, si aparece la palabra "**public**" delante de nuestras funciones, también deberíamos añadir esa palabra delante de nuestro "struct", para evitar un error de compilación que diga que se está utilizando un dato que no es público desde funciones que sí lo son. Si, como en nuestro caso, no hemos utilizado "public", ese problema no debería existir.

Como se ve en este ejemplo, es posible que en los tipos de datos definidos por nosotros (como ese "**struct**") debamos añadir también el especificador "public", para evitar un error de compilación que diga que se está utilizando un dato que no es público desde funciones que sí lo son.

Las variables globales se deben evitar tanto como sea posible. Por ejemplo, una variable "i" que controle bucles siempre debería ser local, porque no es un dato que se deba compartir entre funciones. Lo mismo ocurre en el ejemplo anterior con variables como "textoBuscar" o "tamanyoBuscar".

En lenguajes más antiguos, como C, se tendía a intentar que **nada fuera global**, para evitar errores difíciles de detectar, especialmente en programas formados

por varios fuentes. Ese tipo de problemas son menos frecuentes en lenguajes más modernos y más modulares, como C#, pero aun así se podría crear una versión alternativa del fuente anterior en la que ninguna variable fuera global, sino que desde Main se llamara a las demás funciones pasándoles como parámetros los datos con los que deben trabajar, y, según el caso, recibiendo esos valores devueltos si han sido modificados, por ejemplo así:

```
// Ejemplo_05_05c.cs
// Versión del Ejemplo_05_05b.cs sin variables globales
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_05c
{

    struct tipoFicha
    {
        public string nombreFich;   // Nombre del fichero
        public long tamanyo;        // El tamaño en KB
    }

    static void MostrarMenu()
    {
        Console.WriteLine();
        Console.WriteLine("Escoja una opción:");
        Console.WriteLine("1.- Añadir datos de un nuevo fichero");
        Console.WriteLine("2.- Mostrar los nombres de todos los ficheros");
        Console.WriteLine("3.- Mostrar ficheros por encima de un cierto tamaño");
        Console.WriteLine("4.- Ver datos de un fichero");
        Console.WriteLine("5.- Salir");
    }

    static int PedirOpcion()
    {
        return Convert.ToInt32(Console.ReadLine());
    }

    static int Anyadir(tipoFicha[] fichas, int numeroFichas)
    {
        if (numeroFichas < 1000)
        {  // Si queda hueco
            Console.WriteLine("Introduce el nombre del fichero: ");
            fichas[numeroFichas].nombreFich = Console.ReadLine();
            Console.WriteLine("Introduce el tamaño en KB: ");
            fichas[numeroFichas].tamanyo = Convert.ToInt32(
               Console.ReadLine());
            // Y ya tenemos una ficha más
            numeroFichas++;
        }
        else   // Si no hay hueco para más fichas, avisamos
            Console.WriteLine("Máximo de fichas alcanzado (1000)!");

        return numeroFichas;
    }

    static void MostrarTodos(tipoFicha[] fichas, int numeroFichas)
    {
        for (int i = 0; i < numeroFichas; i++)
            Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
              fichas[i].nombreFich, fichas[i].tamanyo);
    }
```

```
    static void MostrarPorTamanyo(tipoFicha[] fichas, int numeroFichas)
    {
        Console.WriteLine("¿A partir de que tamaño quieres ver?");
        long tamanyoBuscar = Convert.ToInt64(Console.ReadLine());
        for (int i = 0; i < numeroFichas; i++)
            if (fichas[i].tamanyo >= tamanyoBuscar)
                Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
                  fichas[i].nombreFich, fichas[i].tamanyo);
    }

    static void MostrarUnDato(tipoFicha[] fichas, int numeroFichas)
    {
        Console.WriteLine("¿De qué fichero quieres ver todos los datos?");
        string textoBuscar = Console.ReadLine();
        for (int i = 0; i < numeroFichas; i++)
            if (fichas[i].nombreFich == textoBuscar)
                Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
                  fichas[i].nombreFich, fichas[i].tamanyo);
    }

    static void AvisarFin()
    {
        Console.WriteLine("Fin del programa");
    }

    static void AvisarError()
    {
        Console.WriteLine("Opción desconocida!");
    }

    static void Main()
    {
        // Variables locales
        tipoFicha[] fichas = new tipoFicha[1000];
        int numeroFichas = 0;
        int opcion;

        do
        {
            MostrarMenu();
            opcion = PedirOpcion();
            switch (opcion )
            {
                case 1:
                    numeroFichas = Anyadir(fichas, numeroFichas);
                    break;
                case 2: MostrarTodos(fichas, numeroFichas); break;
                case 3: MostrarPorTamanyo(fichas, numeroFichas); break;
                case 4: MostrarUnDato(fichas, numeroFichas); break;
                case 5: AvisarFin(); break;
                default: AvisarError(); break;
            }
        } while (opcion != 5);
    }
}
```

Como se puede observar, este fuente se parece en general mucho al anterior, salvo por el detalle de que se pasan parámetros a las funciones y que "Anyadir" cambia ligeramente su comportamiento, porque puede alterar o no el valor de "numeroFichas", así que ese valor se devuelve. Dentro de poco veremos otras formas de modificar valores de datos que se pasen como parámetro.

Ejercicios propuestos:

- **(5.5.7)** Crea una nueva versión del ejercicio 4.6.3 (datos de 50 personas), en la que cada una de las funcionalidades que permite el programa esté desglosada en su propia función independiente de Main (PedirDatos, MostrarTodos, MostrarPorEdad, MostrarPorInicial):

Un concepto relacionado con el uso de variables globales y locales es el de "**transparencia referencial**": es deseable que una función devuelva siempre los mismos resultados cuando se llama con los mismos parámetros, sin depender de los valores de variables globales o de otros datos externos a la función.

5.6. Los conflictos de nombres en las variables

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

```
// Ejemplo_05_06a.cs
// Dos variables locales con el mismo nombre
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_06a
{

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


    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 y sí se propagarán los cambios:

```
// Ejemplo_05_06b.cs
// Una variable global
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_06b
{

    static int n = 7;

    static void CambiaN()
    {
        n ++;
    }

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

}
```

5.7. Modificando parámetros

5.7.1. Paso de parámetros por valor

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:

```
// Ejemplo_05_07_01a.cs
// Modificar una variable recibida como parámetro - acercamiento
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_07_01a
{

    static void Duplicar(int x)
    {
```

```
        Console.WriteLine("  El valor recibido vale {0}", x);
        x = x * 2;
        Console.WriteLine("  y ahora vale {0}", x);
    }

    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.

5.7.2. Paso de parámetros por referencia

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:

```
// Ejemplo_05_07_02a.cs
// Modificar una variable recibida como parámetro - correcto
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_07_02a
{

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

```
        Console.WriteLine("  y ahora vale {0}", x);
    }

    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:

```
static void Intercambiar(ref int x, ref int y)
```

Ejercicios propuestos:

(5.7.2.1) Crea una función "Intercambiar", que intercambie el valor de los dos números enteros que se le indiquen como parámetro. Crea también un programa que la pruebe.

(5.7.2.2) Crea una función "Iniciales", que reciba una cadena como "Nacho Cabanes" y devuelva las letras N y C (primera letra, y letra situada tras el primer espacio), usando parámetros por referencia. Crea un "Main" que te permita comprobar que funciona correctamente.

(5.7.2.3) Crea una función "MaxMinArray", que reciba un array de reales de doble precisión y devuelva el mayor valor almacenado en ese array y el menor, utilizando parámetros por referencia. Pruébala con un "Main" adecuado.

5.7.3. Parámetros de salida

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:

```
// Ejemplo_05_07_03a.cs
// Parámetros "out"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_07_03a
{

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

    static void Main()
    {
        char letra1, letra2;
        ObtenerDosPrimerasLetras("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 asignes valores iniciales a "letra1" y "letra2".

(Como se ve en este ejemplo, es habitual que si una función devuelve más de un valor, lo haga mediante parámetros “ref” o “out” y la función pase a ser “**void**”. No se suele considerar buena política devolver un parámetro mediante “return” y otros mediante “ref”, salvo quizá en el caso de que mediante “return” se devuelva un código de error para comprobar si el funcionamiento ha sido correcto).

Como curiosidad adicional, en versiones muy recientes de C# se puede declarar la variable directamente en la llamada a la función, como en el siguiente ejemplo (que funciona en Visual Studio 2017 y posteriores, pero no en Visual Studio 2015 ni en el compilador de línea de comandos de .Net que incluye –actualmente- Widows 10):

```
// Ejemplo_05_07_03b.cs
// Parámetros "out" declarados en la llamada
// Introducción a C#, por Nacho Cabanes

using System;
```

```
class Ejemplo_05_07_03a
{

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

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

Ejercicios propuestos:

(5.7.3.1) Crea una función "Iniciales", similar a la del ejercicio 5.7.2.2, que reciba una cadena como "Nacho Cabanes" y devuelva las letras N y C (primera letra, y letra situada tras el primer espacio), pero esta vez usando parámetros de salida. Crea un "Main" que te permita comprobar que funciona correctamente.

(5.7.3.2) Crea una función "MaxMinArray", parecida a la del ejercicio 5.7.2.3, que reciba un array de reales de doble precisión y devuelva el mayor valor almacenado en ese array y el menor, utilizando parámetros de salida. Pruébala con un "Main" adecuado.

5.7.4. Una aplicación de los parámetros de salida: pedir números sin try-catch (TryParse)

Existen algunas funciones ya incorporadas en la biblioteca estándar de C# que emplean parámetros de salida. Una de ellas, que puede resultar útil en ciertas circunstancias, es Int32.TryParse, que puede ser una alternativa cómoda a encerrar dentro de un bloque try-catch una conversión de cadena a número (por ejemplo, con Convert.ToInt32) para evitar que se interrumpa el programa en caso de introducir un dato incorrecto.

El siguiente ejemplo compara ambas formas de trabajar:

```
// Ejemplo_05_07_04a.cs
// Uso de "TryParse"
// Introducción a C#, por Nacho Cabanes

using System;

class ParseNums
{
    static void Main()
```

```
    {
        int a;

        // Planteamiento con try-catch
        Console.Write("Introduce un entero: ");
        try
        {
            a = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Su valor es {0}", a);
        }
        catch (Exception)
        {
            Console.WriteLine("No es un número válido");
        }

        // Alternativa con TryParse
        Console.Write("Introduce otro entero: ");
        if (Int32.TryParse(Console.ReadLine(), out a))
            Console.WriteLine("Su valor es {0}", a);
        else
            Console.WriteLine("No es un número válido");
    }
}
```

El formato de TryParse no sólo es más compacto, sino que además resulta más fácil de llevar a condiciones como un do-while, por lo que puede ser una alternativa más cómoda que try-catch para comprobaciones sencillas.

Ejercicios propuestos

(5.7.4.1) Crea un programa que te pida tu edad tantas veces como sea necesario, hasta que introduzcas un valor numérico aceptable.

5.8. 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:

```
// Ejemplo_05_08a.cs
// Función tras Main
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_08a
{

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

```
        Duplicar(ref n);
        Console.WriteLine("Ahora n vale {0}", n);
    }

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

}
```

Ejercicios propuestos:

(5.8.1) Crea una nueva versión del ejercicio 5.7.1, en el que la función "Intercambiar" esté declarada después de "Main".

5.9. 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:

```
// Ejemplo_05_10a.cs
// Funciones recursivas: factorial
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_10a
{

    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
    }

    static void Main()
    {
        int num;
        Console.WriteLine("Introduzca un número entero: ");
        num = Convert.ToInt32(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 y, en muchos de esos casos, ese problema se podrá expresar de forma recursiva. La recursividad resulta muy útil también en la programación de cierto tipo de juegos: la búsqueda de soluciones para muchos juegos “por turnos” se puede plantear como una función recursiva que explore posibles soluciones a partir del estado actual del juego.

Los ejercicios propuestos te ayudarán a descubrir otros ejemplos de situaciones relativamente sencillas en las que se puede aplicar la recursividad.

Ejercicios propuestos:

(5.9.1) Crea una función que calcule el valor de elevar un número entero a otro número entero (por ejemplo, 5 elevado a 3 = 5^3 = 5 ·5 ·5 = 125). Esta función se debe crear de forma recursiva. Piensa cuál será el caso base (qué potencia se puede calcular de forma trivial) y cómo pasar del caso "n-1" al caso "n" (por ejemplo, si sabes el valor de 5^4, cómo hallarías el de 5^5 a partir de él).

(5.9.2) Como alternativa, crea una función que calcule el valor de elevar un número entero a otro número entero de forma NO recursiva (lo que llamaremos "de forma iterativa"), usando la orden "for".

(5.9.3) Crea un programa que emplee recursividad para calcular un número de la serie Fibonacci (en la que los dos primeros elementos valen 1, y para los restantes, cada elemento es la suma de los dos anteriores).

(5.9.4) Crea un programa que emplee recursividad para calcular la suma de los elementos de un vector de números enteros, desde su posición inicial a la final, usando una función recursiva que tendrá la apariencia: SumaVector(v, desde, hasta). Nuevamente, piensa cuál será el caso base (cuántos elementos podrías sumar para que dicha suma sea trivial) y cómo pasar del caso "n-1" al caso "n" (por ejemplo, si conoces la suma de los 6 primeros elementos y el valor del séptimo elemento, cómo podrías emplear esta información para conocer la suma de los 7 primeros).

(5.9.5) Crea un programa que emplee recursividad para calcular el mayor de los elementos de un vector. El planteamiento será muy similar al del ejercicio anterior.

(5.9.6) Crea un programa que emplee recursividad para dar la vuelta a una cadena de caracteres (por ejemplo, a partir de "Hola" devolvería "aloH"). La función recursiva se llamará "Invertir(cadena)". Como siempre, analiza cuál será el caso base (qué longitud debería tener una cadena para que sea trivial darle la vuelta) y cómo pasar del caso "n-1" al caso "n" (por ejemplo, si ya has invertido las 5 primeras letras, que ocurriría con la letra de la sexta posición).

(5.9.7) Crea, tanto de forma recursiva como de forma iterativa, una función diga si una cadena de caracteres es simétrica (un palíndromo). Por ejemplo, "OSO", "RADAR" y "DABALEARROZALAZORRAELABAD" son palíndromos.

(5.9.8) Crea un programa que encuentre el máximo común divisor de dos números usando el algoritmo de Euclides: Dados dos números enteros positivos m y n, tal que m > n, para encontrar su máximo común divisor, es decir, el mayor entero positivo que divide a ambos, se puede: 1) Dividir m por n para obtener el resto r ($0 \leq r < n$) ; 2) Si r = 0, el MCD es n; 3) Si el resto no es cero, el máximo común divisor es MCD(n,r).

(5.9.9) Crea dos funciones que sirvan para saber si un cierto texto es subcadena de una cadena. No puedes usar "Contains" ni "IndexOf", sino que debes analizar letra a letra. Una función debe ser iterativa y la otra debe ser recursiva.

(5.9.10) Crea una función que reciba una cadena de texto, y una subcadena, y devuelva cuántas veces aparece la subcadena en la cadena, como subsecuencia formada a partir de sus letras en orden. Por ejemplo, si recibes la palabra "Hhoola" y la subcadena "hola", la respuesta sería 4, porque se podría tomar la primera H con la primera O (y con la L y con la A), la primera H con la segunda O, la segunda H con la primera O, o bien la segunda H con la segunda O. Si recibes "hobla", la respuesta sería 1. Si recibes "ohla", la respuesta sería 0, porque tras la H no hay ninguna O que permita completar la secuencia en orden.
(5.9.11) El algoritmo de ordenación conocido como "Quicksort", parte de la siguiente idea: para ordenar un array entre dos posiciones "i" y "j", se comienza por tomar un elemento del array, llamado "pivote" (por ejemplo, el punto medio); luego se recoloca el array de modo que todos los elementos menores que el pivote queden a su izquierda y los mayores a su derecha; finalmente, se llama de forma recursiva a Quicksort para cada una de las dos mitades. El caso base de la función recursiva es cuando se llega a un array de tamaño 0 ó 1. Implementa una función que ordene un array usando este método.

5.10. 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 en línea de comandos se pueden leer desde C#. Se consigue indicando un parámetro especial en Main, un array de strings:

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

Para conocer el valor de 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, o bien con “foreach”:

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

Por otra parte, si queremos que nuestro programa **se interrumpa** en un cierto punto (algo poco recomendable en general, porque dificulta el seguimiento del flujo del programa), 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):

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

Un ejemplo que pusiera todo esto a prueba podría ser:

```
// Ejemplo_05_10a.cs
// Parámetros y valor de retorno de "Main" (1)
// Introducción a C#, por Nacho Cabanes

using System;
```

```
class Ejemplo_05_10a
{
    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;
    }
}
```

O, de forma alternativa::

```
// Ejemplo_05_10b.cs
// Parámetros y valor de retorno de "Main" (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_10b
{
    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!");
            return 1;
        }
        else
            return 0;
    }
}
```

Ejercicios propuestos:

(5.10.1) Crea un programa llamado "suma", que calcule (y muestre) la suma de dos números que se le indiquen como parámetros en línea de comandos. Por ejemplo,

si se teclea "suma 2 3" deberá responder "5", si se teclea "suma 2" responderá "2" y si se teclea únicamente "suma" deberá responder "No hay suficientes datos" y devolver un código de error 1.

(5.10.2) Crea una calculadora básica, llamada "calcula", que deberá sumar, restar, multiplicar o dividir los dos números que se le indiquen como parámetros. Ejemplos de su uso sería "calcula 2 + 3" o "calcula 5 * 60".

(5.10.3) Crea una variante del ejercicio 5.10.2, en la que Main devuelva el código 1 si la operación indicada no es válida o 0 cuando sí sea una operación aceptable.

(5.10.4) Crea una variante del ejercicio 5.10.3, en la que Main devuelva también el código 2 si alguno de los dos números con los que se quiere operar no tiene un valor numérico válido.

5.11. Parámetros con valores por defecto y con nombre

Cada vez más lenguajes permiten indicar parámetros con "valor por defecto", para los que es posible indicar un valor en la llamada o bien no detallarlo y utilizar el valor previsto en la declaración de la función. En el caso de C# es posible usarlos desde la versión 4, del año 2010. Ese tipo de parámetros, si los hay, deberán ser los últimos de la lista:

```
// Ejemplo_05_11a.cs
// Parámetros por defecto
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_11a
{

    static string Linea(int veces, char letra='*')
    {
        return new string(letra, veces);
    }

    static void Main()
    {
        Console.WriteLine( Linea(10, '-' ) );
        Console.WriteLine( Linea(12) );
    }
}
```

Otra posibilidad avanzada (y reciente, y que existe en pocos lenguajes) es la de indicar los parámetros en un orden distinto al que se definieron. Para eso, en C# se debe preceder cada parámetro por su nombre, usando un símbolo de "dos puntos" como separador:

```
// Ejemplo_05_11b.cs
// Parámetros con nombre
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_11b
{

    static string Linea(int veces, char letra='*')
    {
        return new string(letra, veces);
    }

    static void Main()
    {
        Console.WriteLine( Linea(letra: '=', veces: 8) );
    }
}
```

Ejercicios propuestos:

(5.11.1) Crea una función "DibujarRecuadro", que muestre en pantalla un recuadro del ancho y alto que indique el usuario, y relleno con el carácter que también se indique como tercer parámetro. El carácter será opcional, y se usarán "almohadillas" si no se indicar otro distinto. El alto también será opcional, con un valor por defecto de 3. Prúebalo desde Main.

(5.11.2) Crea un segundo programa que use la función "DibujarRecuadro", en esta ocasión llamando con los parámetros en orden inverso (primero el carácter, luego el alto y finalmente el ancho).