2. Estructuras de control

Casi cualquier problema del mundo real que debamos resolver o tarea que deseemos automatizar supondrá tomar decisiones: dar una serie de pasos en función de si se cumplen o no se cumplen ciertas condiciones. En muchas ocasiones, además esos pasos deberán ser repetitivos. Vamos a ver cómo podemos comprobar si se cumplen condiciones y también cómo hacer que un bloque de un programa se repita.

2.1. Estructuras alternativas

2.1.1. if

La primera construcción que emplearemos para comprobar si se cumple una condición será **"si ... entonces ...**". Su formato es

```
if (condición) sentencia;
```

Es decir, debe empezar con la palabra "if", la condición se debe indicar entre paréntesis y a continuación se detallará la orden que hay que realizar en caso de cumplirse esa condición, terminando con un punto y coma.

Vamos a verlo con un ejemplo:

```
// Ejemplo_02_01_01a.cs
// Condiciones con if
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_01a
{
    static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero > 0) Console.WriteLine("El número es positivo.");
    }
}
```

Este programa pide un número al usuario. Si es positivo (mayor que 0), escribe en pantalla "El número es positivo."; si es negativo o cero, no hace nada.

Como se ve en el ejemplo, para comprobar si un valor numérico es mayor que otro, usamos el símbolo ">". Para ver si dos valores son iguales, usaremos dos símbolos de "igual": `if (numero == 0)`. Las demás posibilidades las veremos algo más adelante. En todos los casos, la condición que queremos comprobar deberá indicarse entre paréntesis.

Este programa comienza por un comentario que nos recuerda de qué se trata. Como nuestros fuentes irán siendo cada vez más complejos, a partir de ahora incluiremos comentarios que nos permitan recordar de un vistazo qué pretendíamos hacer.

A no ser que la orden "if" sea extremadamente corta, se suele partir en dos líneas para que resulte más legible, y en ese caso (frecuente), la "sentencia" se tabula un poco más a la derecha que el "if":

```
if (numero > 0)
    Console.WriteLine("El número es positivo.");
```

Un poco más adelante hablaremos del tamaño recomendado para esas tabulaciones.

Ejercicios propuestos:
(2.1.1.1) Crea un programa que pida al usuario un número entero y diga si es par (pista: habrá que comprobar si el resto que se obtiene al dividir entre dos es cero: if (x % 2 == 0) ...).
(2.1.1.2) Crea un programa que pida al usuario dos números enteros y diga cuál es el mayor de ellos.
(2.1.1.3) Crea un programa que pida al usuario dos números enteros y diga si el primero es múltiplo del segundo (pista: igual que en el ejercicio 2.1.1.1, habrá que ver si el resto de la división es cero: a % b == 0).

2.1.2. if y sentencias compuestas

Nuestro primer ejemplo de la orden "if" ejecutaba una única sentencia cuando se cumplía la condición. En un programa real, más complejo, es muy habitual que haya que dar varios pasos, no sólo uno. La forma de hacerlo es agrupar todos esos pasos entre llaves (**{** y **}**), formando lo que se conoce como una "sentencia **compuesta**":

```
// Ejemplo_02_01_02a.cs
// Condiciones con if (2): Sentencias compuestas
// Introducción a C#, por Nacho Cabanes
```

```
using System;

class Ejemplo_02_01_02a
{
    static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero > 0)
        {
            Console.WriteLine("El número es positivo.");
            Console.WriteLine("Recuerde que también puede usar negativos.");
        }  // Aquí acaba el "if"
    }      // Aquí acaba "Main"
}          // Aquí acaba "Ejemplo_02_01_02a
```

En este caso, si el número es positivo, se hacen dos cosas: escribir un texto y luego… ¡escribir otro! (no es gran cosa; más adelante iremos encontrando casos en lo que necesitemos hacer cosas "más reales" dentro de una sentencia compuesta).

Como se ve en este ejemplo, cada nuevo "bloque" se suele escribir un poco más a la derecha que los anteriores, para que sea fácil ver dónde comienza y termina cada sección de un programa. Por ejemplo, el contenido de "Ejemplo06" está tabulado un poco más a la derecha que la cabecera "class Ejemplo06", y el contenido de "Main" algo más a la derecha, y la sentencia compuesta que se debe realizar si se cumple la condición del "if" está aún más a la derecha. Este "**sangrado**" del texto se suele llamar "**escritura indentada**". Un tamaño habitual para el sangrado es de 4 espacios, aunque en este texto en algunas ocasiones puntuales usaremos sólo dos espacios, para que los fuentes más complejos quepan entre los márgenes del papel.

Las recomendaciones habituales para esos tamaños de tabulación son:

- No emplear el carácter de tabulación, sino **espacios** en blanco. Eso no significa que sea necesario pulsar varias veces la barra espaciadora, sino que se deberá configurar el editor que se está empleado (si, como es habitual, lo permite), para que cada pulsación de la tecla de tabulación escriba varios espacios.

- La cantidad de espacios recomendada es de **4**. En general se considera que 8 espacios es un espacio demasiado grande, y que hace que se alcance con

demasiada facilidad el límite recomendado de **80 caracteres por línea**, mientras que 2 espacios hace que sea menos legible el fuente, al distinguirse con menos facilidad el principio y final de cada bloque.

En el "mundo real", es habitual **incluir siempre** las llaves después de un "if", como medida de seguridad, porque un **fallo frecuente** es escribir una única sentencia tras "if", sin llaves, luego añadir una segunda sentencia y olvidar las llaves... de modo que la segunda orden no se ejecutará si se cumple la condición, sino siempre, como en este ejemplo:

```
// Ejemplo_02_01_02b.cs
// Condiciones con if (2b): Sentencias compuestas incorrectas
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_02b
{
    static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero > 0)
            Console.WriteLine("El número es positivo.");
            Console.WriteLine("También puede usar negativos.");  // Error
    }
}
```

Ejercicios propuestos:

(2.1.2.1) Crea un programa que pida al usuario un número entero. Si es múltiplo de 10, informará al usuario y pedirá un segundo número, para decir a continuación si este segundo número también es múltiplo de 10.

2.1.3. Operadores relacionales: <, <=, >, >=, ==, !=

Hemos visto que el símbolo ">" es el que se usa para comprobar si un número es mayor que otro. El símbolo de "menor que" también es sencillo, pero los demás son un poco menos evidentes, así que vamos a verlos:

Operador	Operación
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual a
!=	No igual a (distinto de)

Los símbolos se deben escribir exactamente como aparecen en esa tabla. No se puede cambiar el orden de los que están formados por dos caracteres: es válido != pero no lo es =!, y, del mismo modo, es válido <= pero no =<.

Así, un ejemplo, que diga si un número no es cero sería:

```
// Ejemplo_02_01_03a.cs
// Condiciones con if (3): "distinto de"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_03a
{
    static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero != 0)
            Console.WriteLine("El número no es cero.");
    }
}
```

Y otro que probara todas las condiciones sería así:

```
// Ejemplo_02_01_03b.cs
// Condiciones con if (3b): operadores relacionales
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_03b
{
    static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());

        if (numero > 0)
            Console.WriteLine("El número es positivo.");
        if (numero < 0)
            Console.WriteLine("El número es negativo.");
        if (numero == 0)
            Console.WriteLine("El número es cero.");
        if (numero != 0)
            Console.WriteLine("El número no es cero.");
        if (numero >= 0)
            Console.WriteLine("El número es positivo o cero.");
        if (numero <= 0)
            Console.WriteLine("El número es negativo o cero.");
```

```
    }
}
```

Ejercicios propuestos:

(2.1.3.1) Crea un programa que multiplique dos números enteros de la siguiente forma: pedirá al usuario un primer número entero. Si el número que se que teclee es 0, escribirá en pantalla "El producto de 0 por cualquier número es 0". Si se ha tecleado un número distinto de cero, se pedirá al usuario un segundo número y se mostrará el producto de ambos.

(2.1.3.2) Crea un programa que pida al usuario dos números enteros. Si el segundo no es cero, mostrará el resultado de dividir el primero entre el segundo. Por el contrario, si el segundo número es cero, escribirá "Error: No se puede dividir entre cero".

2.1.4. if-else

Podemos indicar lo que queremos que ocurra en caso de que no se cumpla la condición, usando la orden "else" (en caso contrario), así:

```
// Ejemplo_02_01_04a.cs
// Condiciones con if (4): caso contrario ("else")
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_04a
{
    static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero > 0)
            Console.WriteLine("El número es positivo.");
        else
            Console.WriteLine("El número es cero o negativo.");
    }
}
```

Podríamos intentar evitar el uso de "else" si utilizamos un "if" a continuación de otro, así:

```
// Ejemplo_02_01_04b.cs
// Condiciones con if (5): caso contrario, sin "else"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_04b
```

```
{
    static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());

        if (numero > 0)
            Console.WriteLine("El número es positivo.");

        if (numero <= 0)
            Console.WriteLine("El número es cero o negativo.");
    }
}
```

Pero el comportamiento **no es el mismo**: en el primer caso (ejemplo 02_01_04a) se mira si el valor es positivo; si no lo es, se pasa a la segunda orden, pero si lo es, el programa ya ha terminado. En el segundo caso (ejemplo 02_01_04b), aunque el número sea positivo, se vuelve a realizar la segunda comprobación para ver si es negativo o cero, por lo que el programa es ligeramente más lento.

Podemos enlazar varios "if" usando "else", para decir "si no se cumple esta condición, mira a ver si se cumple esta otra":

```
// Ejemplo_02_01_04c.cs
// Condiciones con if (6): condiciones encadenadas
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_04c
{
    static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());

        if (numero > 0)
            Console.WriteLine("El número es positivo.");
        else
            if (numero < 0)
                Console.WriteLine("El número es negativo.");
            else
                Console.WriteLine("El número es cero.");
    }
}
```

Ejercicio propuesto:

(2.1.4.1) Mejora la solución al ejercicio 2.1.3.1, usando "else".

(2.1.4.2) Mejora la solución al ejercicio 2.1.3.2, usando "else".

2.1.5. Operadores lógicos: &&, ||, !

Las condiciones se puede **encadenar** con "y", "o", "no". Por ejemplo, una partida de un juego puede acabar si nos quedamos sin vidas **o** si superamos el último nivel. Y podemos avanzar al nivel siguiente si hemos llegado hasta la puerta **y** hemos encontrado la llave. O deberemos volver a pedir una contraseña si **no** es correcta **y no** hemos agotado los intentos.

Esos operadores se indican de la siguiente forma

Operador	*Significado*
&&	Y
||	O
!	No

De modo que, ya con la sintaxis de C#, podremos escribir cosas como

```
if ((opcion == 1) && (usuario == 2))  ...
if ((opcion == 1) || (opcion == 3))   ...
if ((!(opcion == opcCorrecta)) || (tecla == ESC))   ...
```

Así, un programa que dijera si dos números introducidos por el usuario son positivos, podría ser:

```
// Ejemplo_02_01_05a.cs
// Condiciones con if enlazadas con &&
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_05a
{
    static void Main()
    {
        int n1, n2;

        Console.Write("Introduce un número: ");
        n1 = Convert.ToInt32(Console.ReadLine());
        Console.Write("Introduce otro número: ");
        n2 = Convert.ToInt32(Console.ReadLine());

        if ((n1 > 0) && (n2 > 0))
            Console.WriteLine("Ambos números son positivos.");
        else
            Console.WriteLine("Al menos uno no es positivo.");
    }
}
```

Es frecuente analizar el comportamiento de estos conectores lógicos usando "**tablas de verdad**", que detallan cómo se van a comportar según si cada uno de

los valores comparados es verdadero o es falso. Por ejemplo, si unimos condiciones con "**y**", el resultado será "verdadero" sólo en el caso de que ambas lo sean (por ejemplo, "es par y positivo" sólo será verdad si el número es par y además es positivo):

a	*b*	*a && b*
falso	falso	falso
falso	verdadero	falso
verdadero	falso	falso
verdadero	verdadero	verdadero

De forma similar, si enlazamos dos condiciones con "**o**", el resultado será "verdadero" en cuanto una de ellas lo sea (por ejemplo, "si x es par o positivo" se cumplirá si x es par, pero también si es positivo, o si ocurren ambas cosas a la vez):

a	*b*	*a || b*
falso	falso	falso
falso	verdadero	verdadero
verdadero	falso	verdadero
verdadero	verdadero	verdadero

Y la tabla de verdad del conector "**no**" es la más sencilla de todas: lo contrario de algo verdadero será algo falso, y viceversa (por ejemplo, "si x no es par" se cumplirá cuando no ocurra que x sea par):

a	*! a*
falso	verdadero
verdadero	falso

Una curiosidad: en C# (y en algún otro lenguaje de programación), la evaluación de dos condiciones que estén enlazadas con "Y" se hace "**en cortocircuito**": si la primera de las condiciones no se cumple, ni siquiera se llega a comprobar la segunda, porque se sabe de antemano que la condición formada por ambas no podrá ser cierta. Eso supone que en el primer ejemplo anterior, `if ((opcion==1) && (usuario==2))`, si "opcion" no vale 1, el compilador no se molesta en ver cuál es el valor de "usuario", porque, sea el que sea, no podrá hacer que sea "verdadera" toda la expresión. Lo mismo ocurriría si hay dos condiciones enlazadas con "o", y la primera de ellas es "verdadera": no será necesario comprobar la segunda, porque ya se sabe que la expresión global será "verdadera".

Como la mejor forma de entender este tipo de expresiones es practicándolas, vamos a ver unos cuantos ejercicios propuestos...

Ejercicios propuestos:

(2.1.5.1) Crea un programa que pida al usuario un número entero y responda si es múltiplo de 2 o de 3.

(2.1.5.2) Crea un programa que pida al usuario un número entero y responda si es múltiplo de 2 y de 3 simultáneamente.

(2.1.5.3) Crea un programa que pida al usuario un número entero y responda si es múltiplo de 2 pero no de 3.

(2.1.5.4) Crea un programa que pida al usuario un número entero y responda si no es múltiplo de 2 ni de 3.

(2.1.5.5) Crea un programa que pida al usuario dos números enteros y diga si ambos son pares.

(2.1.5.6) Crea un programa que pida al usuario dos números enteros y diga si (al menos) uno es par.

(2.1.5.7) Crea un programa que pida al usuario dos números enteros y diga si uno y sólo uno es par.

(2.1.5.8) Crea un programa que pida al usuario dos números enteros y diga "Uno de los números es positivo", "Los dos números son positivos" o bien "Ninguno de los números es positivo", según corresponda.

(2.1.5.9) Crea un programa que pida al usuario tres números y muestre cuál es el mayor de los tres.

(2.1.5.10) Crea un programa que pida al usuario dos números enteros y diga si son iguales o, en caso contrario, cuál es el mayor de ellos.

2.1.6. El peligro de la asignación en un "if"

Cuidado con el comparador de **igualdad**: hay que recordar que el formato es "`if (a==b) ...`". Si no nos acordamos y escribimos "`if (a=b)`", estamos intentando asignar a "a" el valor de "b".

En algunos compiladores de lenguaje C, esto podría ser un problema serio, porque se considera válido hacer una asignación dentro de un "if". La mayoría de compiladores modernos al menos nos avisarían de que quizá estemos asignando un valor sin pretenderlo, pero no se trata de un "error" que invalide la compilación, sino de un "aviso", lo que permite que se genere un ejecutable, y podríamos pasar por alto el aviso, dando lugar a un funcionamiento incorrecto de nuestro programa.

En el caso del lenguaje C#, este riesgo no existe, porque la "condición" debe ser algo cuyo resultado sea "verdadero" o "falso" (lo que pronto llamaremos un dato

de tipo "booleano"), de modo que obtendríamos un error de compilación "Cannot implicitly convert type 'int' to 'bool'" (*no se puede convertir implícitamente el tipo "int" en "bool"*). Es el caso del siguiente programa:

```
// Ejemplo_02_01_06a.cs
// Condiciones con if: comparación incorrecta
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_06a
{
    static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero = 0)
            Console.WriteLine("El número es cero.");
        else
            if (numero < 0)
                Console.WriteLine("El número es negativo.");
            else
                Console.WriteLine("El número es positivo.");
    }
}
```

Nota: en lenguajes como C y C++, en los que sí existe este riesgo de asignar un valor en vez de comparar, hay autores que recomiendan plantear la comparación al revés, colocando el número en el lado izquierdo, de modo que si olvidamos el doble signo de "=", obtendríamos una asignación no válida y el programa no compilaría:

```
if (0 == numero)  ...
```

Aun así, la mayoría de entornos de desarrollo modernos en C y C++ detectan las asignaciones dentro de un "if" y muestran un mensaje de **aviso** (pero, insisto, no un "error fatal": la compilación continúa y se crea un ejecutable; por eso, en estos lenguajes es vital ser muy cuidadoso y prestar atención a los mensajes de aviso, no sólo a los errores).

Ejercicios propuestos:

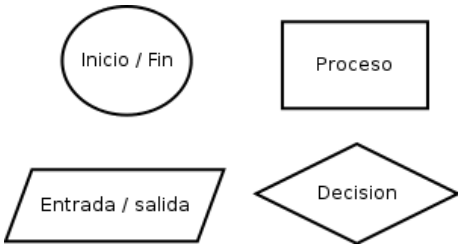
(2.1.6.1) Crea una variante del ejemplo 02_01_06a, en la que la comparación de igualdad sea correcta y en la que las variables aparezcan en el lado derecho de la comparación y los números en el lado izquierdo.

2.1.7. Introducción a los diagramas de flujo

A veces puede resultar difícil ver claro donde usar un "else" o qué instrucciones de las que siguen a un "if" deben ir entre llaves y cuales no. Generalmente la dificultad está en el hecho de intentar teclear directamente un programa en C#, en vez de pensar en el problema que se pretende resolver.

Para ayudarnos a centrarnos en el problema, existen notaciones gráficas, como los diagramas de flujo, que nos permiten ver mejor qué se debe hacer y cuando.

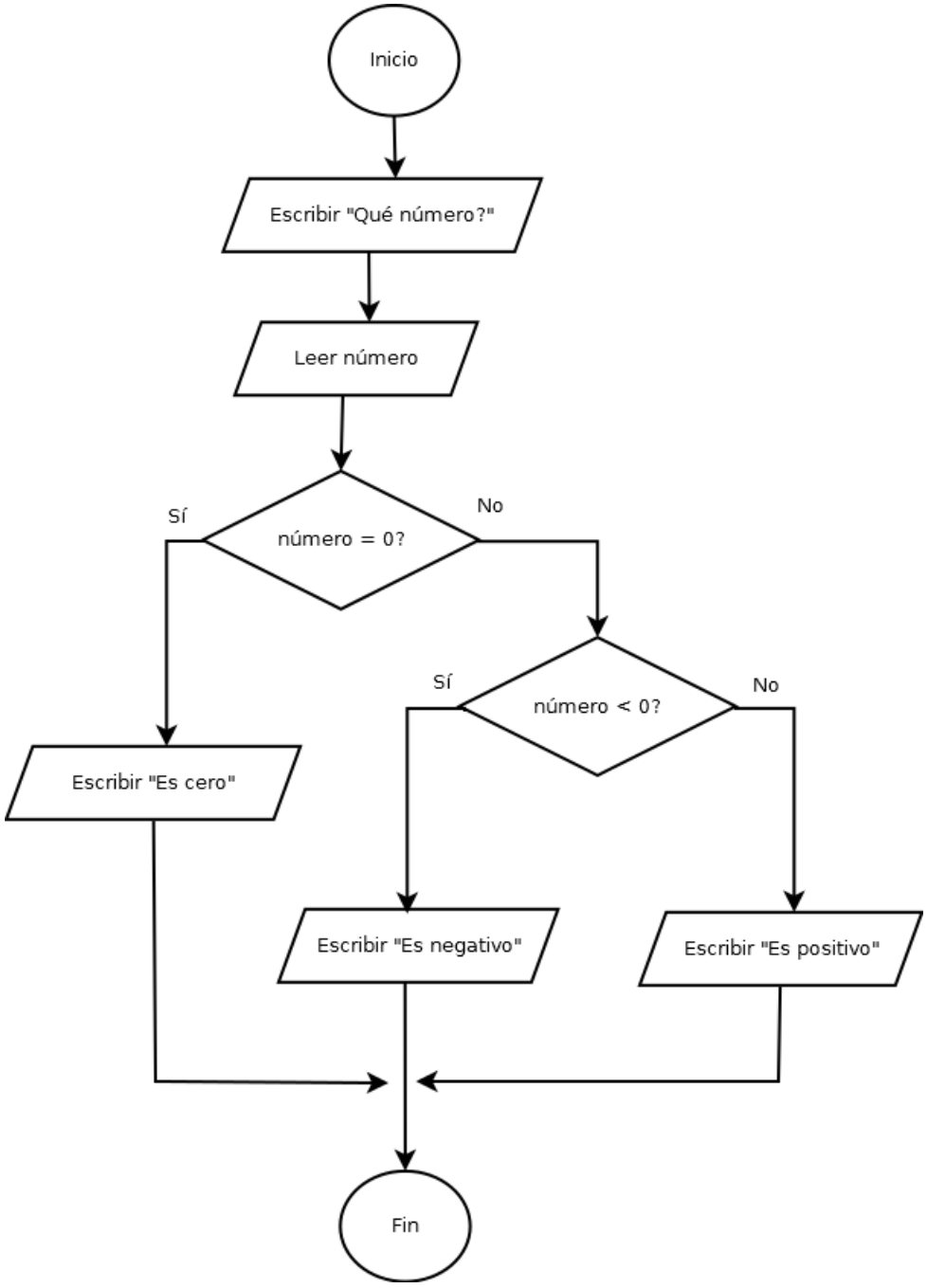
En primer lugar, vamos a ver los 4 elementos básicos de un diagrama de flujo, y luego los aplicaremos a un caso concreto.



Es decir:

- El inicio o el final del programa se indica dentro de un círculo.
- Los procesos internos, como realizar operaciones aritméticas (sumas, restas, multiplicaciones, etc.), se encuadran en un rectángulo.
- Las entradas y salidas (escrituras en pantalla y lecturas de teclado) se indican con un paralelogramo que tenga su lados superior e inferior horizontales, pero inclinados los otros dos.
- Las decisiones se indican dentro de un rombo, desde el que saldrán dos flechas. Cada una de ellas corresponderá a la secuencia de pasos a dar si se cumple una de las dos opciones posibles.

Vamos a aplicarlo al ejemplo de un programa que pida un número al usuario y diga si es positivo, negativo o cero:



El paso de aquí al correspondiente programa en lenguaje C# (el que vimos en el ejemplo 11) debe ser casi inmediato: sabemos cómo leer de teclado, como escribir en pantalla, y las decisiones serán un "if", que si se cumple ejecutará la sentencia que aparece en su salida "si" y si no se cumple ("else") ejecutará lo que aparezca en su salida "no".

Eso sí, hay que tener en cuenta que ésta es una **notación anticuada**, y que no permite representar de forma fiable las estructuras repetitivas que veremos dentro de poco, por lo que su uso actual es muy limitado. Hoy en día se usa para

especificar criterios de diseño a alto nivel, al igual que el pseudocódigo, pero no como representación directa de las órdenes de un programa.

Ejercicios propuestos:

(2.1.7.1) Crea el diagrama de flujo para el programa que pide dos números al usuario y dice cuál es el mayor de los dos.

(2.1.7.2) Crea el diagrama de flujo para el programa que pide al usuario dos números y dice si uno de ellos es positivo, si lo son los dos o si no lo es ninguno.

(2.1.7.3) Crea el diagrama de flujo para el programa que pide tres números al usuario y dice cuál es el mayor de los tres.

(2.1.7.4) Crea el diagrama de flujo y la versión en C# de un programa que dé al usuario tres oportunidades para adivinar un número del 1 al 10, usando 3 "if" encadenados.

2.1.8. Operador condicional: ?

En C#, al igual que en la mayoría de lenguajes que derivan de C, hay otra forma de asignar un valor según se cumpla una condición o no. Es un formato más compacto pero también más difícil de leer. Se trata del "**operador condicional**" **?** : (también conocido como "**operador ternario**"), que se emplea así:

```
nombreVariable = condicion ?  valor1 : valor2;
```

y equivale a decir "si se cumple la condición, la variable *nombreVariable* debe tomar el valor *valor1*; si no, tomará el valor *valor2*". Un ejemplo de cómo podríamos usarlo sería para calcular el mayor de dos números:

```
numeroMayor = a > b ?  a : b;
```

esto equivale a la siguiente orden "if":

```
if ( a > b )
    numeroMayor = a;
else
    numeroMayor = b;
```

Al igual que en este ejemplo, podremos usar el operador condicional en tros muchos problemas reales, cuando queramos optar entre **dos valores posibles para una misma variable**, dependiendo de si se cumple o no una condición.

Aplicado a un programa sencillo, podría ser

```
// Ejemplo_02_01_08a.cs
// El operador condicional
```

```
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_08a
{
    static void Main()
    {
        int a, b, mayor;

        Console.Write("Escriba un número: ");
        a = Convert.ToInt32(Console.ReadLine());
        Console.Write("Escriba otro: ");
        b = Convert.ToInt32(Console.ReadLine());

        mayor = a > b ?  a : b;

        Console.WriteLine("El mayor de los números es {0}.", mayor);
    }
}
```

Realmente, no es necesario guardar en una variable el resultado de la condición. Se podría usar directamente, por ejemplo en un "WriteLine", así:

```
// Ejemplo_02_01_08b.cs
// El operador condicional (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_08b
{
    static void Main()
    {
        int a, b;

        Console.Write("Escriba un número: ");
        a = Convert.ToInt32(Console.ReadLine());
        Console.Write("Escriba otro: ");
        b = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("El mayor de los números es {0}.",
            a > b ?  a : b);
    }
}
```

Un tercer ejemplo, que sume o reste dos números según la opción que se escoja, sería:

```
// Ejemplo_02_01_08c.cs
// El operador condicional (3)
// Introducción a C#, por Nacho Cabanes

using System;
```

```
class Ejemplo_02_01_08c
{
    static void Main()
    {
        int a, b, operacion, resultado;

        Console.Write("Escriba un número: ");
        a = Convert.ToInt32(Console.ReadLine());

        Console.Write("Escriba otro: ");
        b = Convert.ToInt32(Console.ReadLine());

        Console.Write("Escriba una operación (1 = resta; otro = suma): ");
        operacion = Convert.ToInt32(Console.ReadLine());

        resultado =  operacion == 1 ?  a-b : a+b;
        Console.WriteLine("El resultado es {0}.", resultado);
    }
}
```

Ejercicios propuestos:

(2.1.8.1) Crea un programa que use el operador condicional para mostrar el valor absoluto de un número de la siguiente forma: si el número es positivo, se mostrará tal cual; si es negativo, se mostrará cambiado de signo.

(2.1.8.2) Usa el operador condicional para calcular el menor de dos números.

2.1.9. switch

Si queremos analizar **varios posibles valores** de una misma variable, puede resultar muy pesado tener que hacerlo con muchos "if" seguidos o encadenados:

```
// Ejemplo_02_01_09a.cs
// Acercamiento a "switch"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_09a
{
    static void Main()
    {
        int n;

        Console.Write("Introduce un número del 1 al 5: ");
        n = Convert.ToInt32(Console.ReadLine());

        if (n==1)
            Console.WriteLine("Uno");
        else if (n==2)
            Console.WriteLine("Dos");
        else if (n==3)
            Console.WriteLine("Tres");
```

```
        else if (n==4)
            Console.WriteLine("Cuatro");
        else if (n==5)
            Console.WriteLine("Cinco");
        else
            Console.WriteLine("No es del uno al cinco");
    }
}
```

La alternativa es emplear la orden "switch", cuya sintaxis es

```
switch (expresión)
{
    case valor1:
            sentencia1;
            break;
    case valor2:
            sentencia2;
            sentencia2b;
            break;
    case valor3:
            goto case valor1;
     ...
    case valorN:
            sentenciaN;
            break;
    default:
            otraSentencia;
            break;
}
```

Es decir:

- Tras la palabra "**switch**" se escribe la expresión a analizar, entre paréntesis (habitualmente será el nombre de una variable).
- Después, tras varias órdenes "**case**" se indica cada uno de los valores posibles, seguidos por un símbolo de "dos puntos".
- A continuación de cada "case" se indican los pasos (porque pueden ser varios) que se deben dar si la expresión tiene ese valor concreto, terminando con "**break**".
- Si es necesario hacer algo en caso de que no se cumpla ninguna de las condiciones (por ejemplo, responder con un mensaje de error), se detalla después de la palabra "**default**".
- Si dos casos tienen que hacer lo mismo, se añade "**goto case**" a uno de ellos para indicarlo.

Así, una versión alternativa del programa anterior podría ser:

```
// Ejemplo_02_01_09b.cs
// Contacto con "switch"
// Introducción a C#, por Nacho Cabanes
```

```
using System;

class Ejemplo_02_01_09b
{
    static void Main()
    {
        int n;

        Console.Write("Introduce un número del 1 al 5: ");
        n = Convert.ToInt32(Console.ReadLine());

        switch(n)
        {
            case 1: Console.WriteLine("Uno"); break;
            case 2: Console.WriteLine("Dos"); break;
            case 3: Console.WriteLine("Tres"); break;
            case 4: Console.WriteLine("Cuatro"); break;
            case 5: Console.WriteLine("Cinco"); break;
            default: Console.WriteLine("No es del uno al cinco"); break;
        }
    }
}
```

Vamos a ver otro ejemplo, que diga si el símbolo que introduce el usuario es una cifra numérica, un espacio u otro símbolo. Para ello usaremos un dato de tipo "**char**" (carácter), que veremos con más detalle en el próximo tema. De momento nos basta saber que deberemos usar Convert.ToChar si lo leemos desde teclado con ReadLine, y que le podemos dar un valor (o compararlo) usando comillas simples:

```
// Ejemplo_02_01_09c.cs
// La orden "switch" y caracteres
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_09c
{
    static void Main()
    {
        char letra;

        Console.WriteLine("Introduce una letra");
        letra = Convert.ToChar( Console.ReadLine() );

        switch (letra)
        {
            case ' ': Console.WriteLine("Espacio.");
                      break;
            case '1': goto case '0';
            case '2': goto case '0';
            case '3': goto case '0';
            case '4': goto case '0';
            case '5': goto case '0';
            case '6': goto case '0';
```

```
            case '7': goto case '0';
            case '8': goto case '0';
            case '9': goto case '0';
            case '0': Console.WriteLine("Dígito.");
                      break;
            default:  Console.WriteLine("Ni espacio ni dígito.");
                      break;
        }
    }
}
```

Cuidado quien venga del lenguaje C: en C se puede dejar que un caso sea manejado por el siguiente, lo que se consigue si no se usa "break", mientras que C# siempre obliga a usar "break" o "goto" al final de cada caso (para evitar errores provocados por una "break" olvidado) con la **única excepción** de que un caso no haga **absolutamente nada** excepto dejar pasar el control al siguiente caso, y en ese caso se puede dejar totalmente vacío:

```
// Ejemplo_02_01_09d.cs
// La orden "switch"  (variante sin break)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_09d
{
    static void Main()
    {
        char letra;

        Console.WriteLine("Introduce una letra");
        letra = Convert.ToChar( Console.ReadLine() );

        switch (letra)
        {
            case ' ': Console.WriteLine("Espacio.");
                      break;
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
            case '0': Console.WriteLine("Dígito.");
                      break;
            default:  Console.WriteLine("Ni espacio ni dígito.");
                      break;
        }
    }
}
```

En el lenguaje C, que es más antiguo, sólo se podía usar "switch" para comprobar valores de variables "simples" (numéricas y caracteres); en C#, que es un lenguaje más evolucionado, se puede usar también para comprobar valores de cadenas de texto ("strings").

Una cadena de texto, como veremos con más detalle en el próximo tema, se declara con la palabra "**string**", se puede leer de teclado con ReadLine (sin necesidad de convertir) y se le puede dar un valor desde programa si se indica entre comillas dobles. Por ejemplo, un programa que nos salude de forma personalizada si somos "Juan" o "Pedro" podría ser:

```
// Ejemplo_02_01_09e.cs
// La orden "switch" con cadenas de texto
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_09e
{
    static void Main()
    {
        string nombre;

        Console.WriteLine("Introduce tu nombre");
        nombre = Console.ReadLine();

        switch (nombre)
        {
            case "Juan":
                Console.WriteLine("Bienvenido, Juan.");
                break;
            case "Pedro":
                Console.WriteLine("Que tal estas, Pedro.");
                break;
            default:
                Console.WriteLine("Hola, desconocido.");
                break;
        }
    }
}
```

Ejercicios propuestos:

(2.1.9.1) Crea un programa que pida un número del 1 al 10 al usuario, y escriba el nombre de ese número, usando "switch" (por ejemplo, si introduce "1", el programa escribirá "uno").

(2.1.9.2) Crea un programa que lea una letra tecleada por el usuario y diga si se trata de un signo de puntuación (. , ; :), una cifra numérica (del 0 al 9) o algún otro carácter, usando "switch" (pista: necesitarás que usar un dato de tipo "char").

(2.1.9.3) Crea un programa que lea una letra tecleada por el usuario y diga si se trata de una vocal, una cifra numérica o una consonante, usando "switch".

(2.1.9.4) Repite el ejercicio 2.1.9.1, empleando "if" en lugar de "switch".
(2.1.9.5) Repite el ejercicio 2.1.9.2, empleando "if" en lugar de "switch" (pista: como las cifras numéricas del 0 al 9 están ordenadas, no hace falta comprobar los 10 valores, sino que se puede hacer con "if ((simbolo >= '0') && (simbolo <='9'))").
(2.1.9.6) Repite el ejercicio 2.1.9.3, empleando "if" en lugar de "switch".
(2.1.9.7) Pide al usuario el número de un día de la semana y escribe su nombre (por ejemplo, si escribe 2, la respuesta debería ser "Martes"). Hazlo primero con "if" y después con "switch".
(2.1.9.8) Pide al usuario el un número de mes y escribe su nombre (por ejemplo, si escribe 3, la respuesta debería ser "Marzo"), usando "switch".

2.2. Estructuras repetitivas

Hemos visto cómo comprobar condiciones, pero no cómo hacer que una cierta parte de un programa se repita un cierto número de veces o mientras se cumpla una condición (lo que llamaremos un "**bucle**"). Tenemos varias formas de conseguirlo, según si queremos comprobar la condición antes de repetir, o después de cada repetición, o realizar la tarea una cierta cantidad de veces.

2.2.1. while

2.2.1.1. Estructura básica de un bucle "while"

Si queremos hacer que una sección de nuestro programa se repita mientras se cumpla una cierta condición, usaremos la orden "while". Esta orden tiene dos formatos distintos, según comprobemos la condición al principio o al final del bloque repetitivo.

En el primer caso, su sintaxis es

```
while (condición)
  sentencia;
```

Es decir, la sentencia se repetirá **mientras** la condición sea cierta. Si la condición es falsa ya desde un principio, la sentencia no se ejecuta nunca. Si queremos que se repita más de una sentencia, basta agruparlas entre llaves: { y }. Como ocurría con "if", puede ser **recomendable** incluir siempre las llaves, aunque se trate de una única sentencia, para evitar errores posteriores difíciles de localizar.

Un ejemplo que nos diga si cada número que tecleemos es positivo o negativo, y que termine cuando tecleemos 0, podría ser:

```
// Ejemplo_02_02_01_01a.cs
// La orden "while": mientras...
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_01_01a
{
    static void Main()
    {
        int numero;

        Console.Write("Teclea un número (0 para salir): ");
        numero = Convert.ToInt32(Console.ReadLine());

        while (numero != 0)
```

```
        {
            if (numero > 0)
                Console.WriteLine("Es positivo");
            else
                Console.WriteLine("Es negativo");

            Console.WriteLine("Teclea otro número (0 para salir): ");
            numero = Convert.ToInt32(Console.ReadLine());
        }
    }
}
```

En este ejemplo, si se introduce 0 la primera vez, la condición es falsa y ni siquiera se entra al bloque del "while", terminando el programa inmediatamente.

Ejercicios propuestos:

(2.2.1.1.1) Crea un programa que pida al usuario su contraseña (numérica). Deberá terminar cuando introduzca como contraseña el número 1111, pero volvérsela a pedir tantas veces como sea necesario.

(2.2.1.1.2) Crea un "calculador de cuadrados": pedirá al usuario un número y mostrará su cuadrado. Se repetirá mientras el número introducido no sea cero (usa "while" para conseguirlo).

(2.2.1.1.3) Crea un programa que pida de forma repetitiva pares de números al usuario. Tras introducir cada par de números, responderá si el primero es múltiplo del segundo. Se repetirá mientras los dos números sean distintos de cero (terminará cuando uno de ellos sea cero).

(2.2.1.1.4) Crea una versión mejorada del programa anterior, que, tras introducir cada par de números, responderá si el primero es múltiplo del segundo, o el segundo es múltiplo del primero, o ninguno de ellos es múltiplo del otro.

2.2.1.2. Contadores usando un bucle "while"

Ahora que sabemos "repetir" cosas, podemos utilizarlo también para **contar**. Por ejemplo, si queremos contar del 1 al 5, usaríamos una variable que empezase en 1, que aumentaría una unidad en cada repetición y se repetiría hasta llegar al valor 5, así:

```
// Ejemplo_02_02_01_02a.cs
// Contar con "while"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_01_02a
{
```

```
    static void Main()
    {
        int n = 1;

        while (n < 6)
        {
            Console.WriteLine(n);
            n = n + 1;
        }
    }
}
```

Y esta misma estructura se puede emplear también para hacer algo varias veces, aunque no se muestre el valor del contador en pantalla, como en este ejemplo, que escribe 5 letras X:

```
// Ejemplo_02_02_01_02b.cs
// Contar con "while" (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_01_02b
{
    static void Main()
    {
        int n = 1;

        while (n < 6)
        {
            Console.Write("X");
            n = n + 1;
        }
        Console.WriteLine();  // Para avanzar de línea al final
    }
}
```

Ejercicios propuestos:

(2.2.1.2.1) Crea un programa que escriba en pantalla los números del 1 al 10, usando "while".

(2.2.1.2.2) Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "while".

(2.2.1.2.3) Crea un programa calcule cuantas cifras tiene un número entero positivo (pista: se puede hacer dividiendo varias veces entre 10).

(2.2.1.2.4) Haz un programa que muestre tantos asteriscos (en la misma línea) como indique el usuario.

2.2.2. do ... while

Este es el otro formato que puede tener la orden "while": en este caso, la condición se comprueba **al final**, de modo que siempre se dará al menos una pasada por la zona repetitiva (se podría traducir como "repetir...mientras"). El punto en que comienza la parte repetitiva se indica con la orden "do", así:

```
do
  sentencia;
while (condición);
```

Al igual que en el caso anterior, si queremos que se repitan varias órdenes (es lo habitual), deberemos encerrarlas entre llaves.

Como ejemplo, vamos a ver cómo sería el típico programa que nos pide una clave de acceso y no nos deja entrar hasta que tecleemos la clave correcta:

```
// Ejemplo_02_02_02a.cs
// La orden "do..while" (repetir..mientras)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_02a
{
    static void Main()
    {
        int valida = 711;
        int clave;

        do
        {
            Console.Write("Introduzca su clave numérica: ");
            clave = Convert.ToInt32(Console.ReadLine());

            if (clave != valida)
                Console.WriteLine("No válida!");
        }
        while (clave != valida);

        Console.WriteLine("Aceptada.");
    }
}
```

En este caso, se comprueba la condición al final, de modo que se nos preguntará la clave al menos una vez. Mientras que la respuesta que demos no sea la correcta, se nos vuelve a preguntar. Finalmente, cuando tecleamos la clave correcta, el ordenador escribe "Aceptada" y termina el programa.

Como veremos con detalle un poco más adelante, si preferimos que la clave sea un texto en vez de un número, los cambios al programa son mínimos, basta con usar "string" e indicar su valor entre comillas dobles:

```
// Ejemplo_02_02_02b.cs
// La orden "do..while" (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_02b
{
    static void Main()
    {
        string valida = "secreto";
        string clave;

        do
        {
            Console.Write("Introduzca su clave: ");
            clave = Console.ReadLine();

            if (clave != valida)
                Console.WriteLine("No válida!");
        }
        while (clave != valida);

        Console.WriteLine("Aceptada.");
    }
}
```

Ejercicios propuestos:

(2.2.2.1) Crea un programa que pida números positivos al usuario, y vaya calculando y mostrando la suma de todos ellos (terminará cuando se teclea un número negativo o cero).

(2.2.2.2) Crea un programa que escriba en pantalla los números del 1 al 10, usando "do..while".

(2.2.2.3) Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "do..while".

(2.2.2.4) Crea un programa que pida al usuario su identificador y su contraseña (ambos numéricos), y no le permita seguir hasta que introduzca como identificador "1234" y como contraseña "1111".

(2.2.2.5) Crea un programa que pida al usuario su identificador y su contraseña, y no le permita seguir hasta que introduzca como nombre "Pedro" y como contraseña "Peter".

2.2.3. for

Ésta es la orden que usaremos habitualmente para crear partes del programa que **se repitan** un cierto número de veces. El formato de "for" es

```
for (valorInicial; CondiciónRepetición; Incremento)
    Sentencia;
```

Es muy habitual usar la letra "i" (abreviatura de "índice") como contador cuando se trata de tareas muy sencillas. Así, para **contar del 1 al 10**, tendríamos "i=1" como valor inicial, "i<=10" como condición de repetición, y el incremento sería de 1 en 1, con "i=i+1".

```
for (i=1; i<=10; i=i+1)
    ...
```

La orden para **incrementar** en una unidad el valor de una variable ("i = i+1") se puede escribir de la forma abreviada "i++", como veremos con más detalle en el próximo tema, de modo que la forma habitual de crear el contador anterior sería

```
for (i=1; i<=10; i++)
    ...
```

En general, en fragmentos de programa que no sean triviales, será preferible usar nombres de variable más descriptivos que "i". Así, un programa que escribiera los números del 1 al 10 podría ser:

```
// Ejemplo_02_02_03a.cs
// Uso básico de "for"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_03a
{
    static void Main()
    {
        int contador;

        for (contador = 1; contador <= 10; contador++)
            Console.Write("{0} ", contador);
    }
}
```

Ejercicios propuestos:

(2.2.3.1) Crea un programa que muestre los números del 10 al 20, ambos incluidos, usando "for".

(2.2.3.2) Crea un programa que escriba en pantalla los números del 1 al 50 que sean múltiplos de 3 (pista: habrá que recorrer todos esos números y ver si el resto de la división entre 3 resulta 0).
(2.2.3.3) Crea un programa que muestre los números del 100 al 200 (ambos incluidos) que sean divisibles entre 7 y a la vez entre 3.
(2.2.3.4) Crea un programa que muestre la tabla de multiplicar del 9.
(2.2.3.5) Crea un programa que muestre los primeros ocho números pares: 2 4 6 8 10 12 14 16 (pista: en cada pasada habrá que aumentar de 2 en 2, o bien mostrar el doble del valor que hace de contador).
(2.2.3.6) Crea un programa que muestre los números del 15 al 5, descendiendo (pista: en cada pasada habrá que descontar 1, por ejemplo haciendo `i=i-1`, que se puede abreviar `i--`).

Nota: como veremos también con más detalle en el próximo tema, existe una notación abreviada para incrementar en varias unidades el valor de una variable: "i = i+5" se puede escribir de forma alternativa como "i += 5" (no debe existir separación entre el símbolo "+" y el símbolo "="), y también se puede hacer lo mismo para decrementar en varias unidades: "i = i-4" se puede escribir como "i -= 4". Así, se puede mostrar los números pares del 2 al 10 con

```
for (i=2; i<=10; i+=2)
    ...
```

2.2.4. Bucles sin fin

Realmente, en un "for", la parte que hemos llamado "Incremento" no tiene por qué incrementar la variable, aunque ése es su uso más habitual. Es simplemente una orden que se ejecuta cuando se termine la "sentencia" y antes de volver a comprobar si todavía se cumple la condición de repetición.

Por eso, si escribimos la siguiente línea:

```
for (contador=1; contador<=10; )
```

la variable "contador" no se incrementa nunca, por lo que jamás se cumplirá la condición de salida: nos quedamos encerrados dando vueltas dentro de la orden que siga al "for". El programa no termina nunca. Se trata de un "**bucle sin fin**".

Un caso todavía más evidente de fragmento de programa a lo que se entra y de lo que no se sale nunca ("bucle sin fin") sería la siguiente orden:

```
for ( ;  ; )
```

También se puede crear un bucle sin fin usando "while" o usando "do..while", si se indica una condición que siempre vaya a ser cierta, como ésta:

```
    while (1 == 1)
```

Ejercicios propuestos:
(2.2.4.1) Crea un programa que contenga un bucle sin fin que escriba "Hola " en pantalla, sin avanzar de línea.
(2.2.4.2) Crea un programa que contenga un bucle sin fin que muestre los números enteros positivos a partir del uno, separados por un espacio en blanco.

2.2.5. Bucles anidados

Los bucles se pueden **anidar** (incluir uno dentro de otro), de modo que podríamos escribir las tablas de multiplicar del 1 al 5 con:

```
// Ejemplo_02_02_05a.cs
// "for" anidados
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_05a
{
    static void Main()
    {
        int tabla, numero;

        for (tabla = 1; tabla <= 5; tabla++)
            for (numero = 1; numero <= 10; numero++)
                Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                    tabla*numero);
    }
}
```

(Es decir: tenemos varias tablas, del 1 al 5, y para cada tabla queremos ver el resultado que se obtiene al multiplicar por los números del 1 al 10).

Ejercicios propuestos:
(2.2.5.1) Crea un programa escriba 4 veces los números del 1 al 5, en una misma línea, usando "for": 12345123451234512345.
(2.2.5.2) Crea un programa escriba 4 veces los números del 1 al 5, en una misma línea, usando "while": 12345123451234512345.
(2.2.5.3) Crea un programa que, para los números entre el 10 y el 20 (ambos incluidos) diga si son divisibles entre 5, si son divisibles entre 6 y si son divisibles entre 7, usando dos bucles anidados.

2.2.6. Repetir sentencias compuestas

En los últimos ejemplos que hemos visto, después de "for" había una única sentencia. Al igual que ocurría con "if" y con "while", si queremos que se hagan varias cosas, basta definirlas como un **bloque** (una sentencia compuesta) encerrándolas entre llaves. Por ejemplo, si queremos mejorar el ejemplo anterior haciendo que deje una línea en blanco entre tabla y tabla, sería:

```
// Ejemplo_02_02_06a.cs
// "for" anidados (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_06a
{
    static void Main()
    {
        int tabla, numero;

        for (tabla=1; tabla<=5; tabla++)
        {
            for (numero=1; numero<=10; numero++)
                Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                    tabla*numero);

            Console.WriteLine();
        }
    }
}
```

Si repetimos la escritura de varias líneas, cada una formadas por varias columnas, podremos dibujar varias figuras geométricas sencillas (bastantes de las cuales quedarán propuestas para que tú las intentes). Por ejemplo, se puede dibujar un rectángulo con:

```
// Ejemplo_02_02_06b.cs
// Rectángulo de asteriscos
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_06b
{
    static void Main()
    {
        int fila, columna;
        int alto = 5;
        int ancho = 10;

        for (fila=1; fila <= alto; fila++)
        {
```

```
            for (columna=1; columna <= ancho; columna++)
                 Console.Write("*");
            Console.WriteLine();
        }
    }
}
```

que tendría como resultado:

```
**********
**********
**********
**********
**********
```

Ejercicios propuestos:

(2.2.6.1) Crea un programa que escriba 4 líneas de texto, cada una de las cuales estará formada por los números del 1 al 5.

(2.2.6.2) Crea un programa que pida al usuario el ancho (por ejemplo, 4) y el alto (por ejemplo, 3) y escriba un rectángulo formado por esa cantidad de asteriscos:

```
****
****
****
```

(2.2.6.3) Haz un programa que dibuje un cuadrado de asteriscos, cuyo ancho (y alto, que tendrá el mismo valor) será introducido por el usuario.

(2.2.6.4) Crea un triángulo de asteriscos, que mostrará uno en la primera fila, dos en la segunda, tres en la tercera y así sucesivamente, hasta llegar al tamaño indicado por el usuario.

(2.2.6.5) Dibuja un triángulo de asteriscos descendente. Por ejemplo, si el usuario escoge "4" como tamaño, la primera fila tendrá 4 asteriscos, la segunda tendrá 3, la siguiente tendrá 2 y la última tendrá 1.

2.2.7. Contar con letras

Para "contar" no necesariamente hay que usar números. Por ejemplo, podemos usar letras, si el contador lo declaramos como "char" (pronto hablaremos más de ese tipo de datos) y los valores inicial y final se detallan entre comillas simples, así:

```
// Ejemplo_02_02_07a.cs
// "for" que usa "char"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_07a
{
    static void Main()
```

```
    {
        char letra;

        for (letra='a'; letra<='z'; letra++)
            Console.Write("{0} ", letra);
    }
}
```

En este caso, empezamos en la "a" y terminamos en la "z", aumentando de uno en uno.

Como ya hemos comentado, si queremos contar de forma **decreciente**, o de dos en dos, o como nos interese, basta indicarlo en la condición de finalización del "for" y en la parte que cambia su valor. Así, podríamos escribir las letras de la "z" a la "a" de la siguiente manera:

```
// Ejemplo_02_02_07b.cs
// "for" que descuenta
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_07b
{
    static void Main()
    {
        char letra;

        for (letra='z'; letra>='a'; letra--)
            Console.Write("{0} ", letra);
    }
}
```

Ejercicios propuestos:
(2.2.7.1) Crea un programa que muestre las letras de la Z (mayúscula) a la A (mayúscula, descendiendo).
(2.2.7.2) Crea un programa que muestre 5 veces las letras de la L (mayúscula) a la N (mayúscula), en la misma línea, empleando dos "for" anidados.

2.2.8. Declarar variables en un "for"

Se puede incluso declarar una nueva variable en el interior de "for", y esa variable dejará de estar definida cuando el "for" acabe. Es una forma **recomendable** de trabajar, porque ayuda a evitar un fallo frecuente: reutilizar variables pero olvidar volver a darles un valor inicial:

```
for (int i=1; i<=10; i++) ...
```

Por ejemplo, el siguiente fuente compila correctamente y puede parecer mostrar dos veces la tabla de multiplicar del 3, pero el "while" no muestra nada, porque no hemos vuelto a inicializar la variable "n", así que ya está por encima del valor 10 y ya no es un valor aceptable para entrar al bloque "while":

```
// Ejemplo_02_02_08a.cs
// Reutilización incorrecta de la variable de un "for"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_08a
{
    static void Main()
    {
        int n = 1;

        // Vamos a mostrar la tabla de multiplicar del 3 con "for"
        for (n=1; n<=10; n++)
            Console.WriteLine("{0} x 3 = {1}", n, n*3);

        // Y ahora con "while"... pero no funcionará correctamente
        while (n<=10)
        {
            Console.WriteLine("{0} x 3 = {1}", n, n*3);
            n++;
        }
    }
}
```

Si declaramos la variable dentro del "for", la zona de "while" no compilaría, lo que hace que el error de diseño sea evidente:

```
// Ejemplo_02_02_08b.cs
// Intento de reutilización incorrecta de la variable
//   de un "for": no compila
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_08b
{
    static void Main()
    {
        // Vamos a mostrar la tabla de multiplicar del 3 con "for"
        for (int n=1; n<=10; n++)
            Console.WriteLine("{0} x 3 = {1}", n, n*3);

        // Y ahora con "while"... pero no compila
        while (n<=10)
        {
            Console.WriteLine("{0} x 3 = {1}", n, n*3);
            n++;
        }
    }
}
```

Esta idea sea puede aplicar a cualquier fuente que contenga un "for". Por ejemplo, el fuente 2.2.6a, que mostraba varias tablas de multiplicar, se podría reescribir de forma más segura así:

```
// Ejemplo_02_02_08c.cs
// "for" anidados, variables en "for"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_08c
{
    static void Main()
    {
        for (int tabla=1; tabla<=5; tabla++)
        {
            for (int numero=1; numero<=10; numero++)
                Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                    tabla*numero);

            Console.WriteLine();
        }
    }
}
```

Ejercicios propuestos:

(2.2.8.1) Crea un programa que escriba 6 líneas de texto, cada una de las cuales estará formada por los números del 1 al 7. Debes usar dos variables llamadas "linea" y "numero", y ambas deben estar declaradas en el "for".

(2.2.8.2) Crea un programa que pida al usuario el ancho (por ejemplo, 4) y el alto (por ejemplo, 3) y escriba un rectángulo formado por esa cantidad de asteriscos, como en el ejercicio 2.2.6.2. Deberás usar las variables "ancho" y "alto" para los datos que pidas al usuario, y las variables "filaActual" y "columnaActual" (declaradas en el "for") para el bloque repetitivo.

2.2.9. Las llaves son recomendables

Como ya hemos comentado, las "llaves" no son necesarias cuando una orden "for", "while", "do-while" o "if" va a repetir una única sentencia, sino sólo cuando se repite un bloque de dos o más sentencias. Pero un error frecuente consiste repetir inicialmente una única orden, añadir después una segunda orden repetitiva y olvidar las llaves. Otro error (menos frecuente) es querer incluir varias órdenes dentro de una de estas estructuras, tabular estas nuevas órdenes más a la derecha pero olvidar las llaves. Por eso, en programas no triviales es muy recomendable incluir siempre las llaves, aunque esperemos repetir sólo una orden.

Por ejemplo, el siguiente fuente puede parecer correcto, pero si lo miramos con detenimiento, veremos que la orden "Console.WriteLine" del final, aunque esté tabulada más a la derecha, no forma parte de ningún "for", de modo que no se repite, y no se dejará ningún espacio en blanco entre una tabla de multiplicar y la siguiente, sino que sólo se escribirá una línea en blanco al final, justo antes de terminar el programa:

```
// Ejemplo_02_02_09a.cs
// "for" anidados de forma incorrecta, sin llaves
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_09a
{
    static void Main()
    {
        for (int tabla=1; tabla<=5; tabla++)
            for (int numero=1; numero<=10; numero++)
                Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                    tabla*numero);
            Console.WriteLine();
    }
}
```

Si incluimos llaves, incluso donde no son imprescindibles, el problema desaparece:

```
// Ejemplo_02_02_09b.cs
// "for" anidados, variables en "for", llaves "redundantes"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_09b
{
    static void Main()
    {
        for (int tabla=1; tabla<=5; tabla++)
        {
            for (int numero=1; numero<=10; numero++)
            {
                Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                    tabla*numero);
            }

            Console.WriteLine();
        }
    }
}
```

Ejercicios propuestos:

(2.2.9.1) Crea un programa que pida un número al usuario y escriba los múltiplos de 9 que haya entre 1 y ese número. Debes usar llaves en todas las estructuras de control, aunque sólo incluyan una sentencia.

(2.2.9.2) Crea un programa que pida al usuario dos números y escriba sus divisores comunes. Debes usar llaves en todas las estructuras de control, aunque sólo incluyan una sentencia.

2.2.10. Interrumpir un bucle: break

Podemos salir de un bucle antes de tiempo si lo interrumpimos con la orden "**break**":

```
// Ejemplo_02_02_10a.cs
// "for" interrumpido con "break"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_10a
{
    static void Main()
    {
        for (int contador=1; contador<=10; contador++)
        {
            if (contador==5)
                break;

            Console.Write("{0} ", contador);
        }
    }
}
```

El resultado de este programa es:

```
  1 2 3 4
```

(en cuanto se llega al valor 5, se interrumpe el "for", por lo que no se alcanza el valor 10).

Es una orden que se debe **tratar de evitar**, porque puede conducir a programas difíciles de leer, en los que no se compueba sólo si cumple la condición de repetición del bucle, sino que además éste se puede interrumpir por otros criterios. Como norma general, es preferible reescribir la condición del bucle de otra forma. En el ejemplo anterior, bastaría que la condición de "if" fuese "contador < 5". Un ejemplo ligeramente más complejo podría ser mostrar los números del 105 al 120 hasta encontrar uno que sea múltiplo de 13, que no se mostrará. Lo podríamos hacer de esta forma (poco correcta):

```
// Ejemplo_02_02_10b.cs
// "for" interrumpido con "break" (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_10b
{
    static void Main()
    {
        for (int contador=105; contador<=120; contador++)
        {
            if (contador % 13 == 0)
                break;

            Console.Write("{0} ", contador);
        }
    }
}
```

O reescribirlo de esta otra (preferible):

```
// Ejemplo_02_02_10c.cs
// Alternativa a un "for" interrumpido con "break"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_10c
{
    static void Main()
    {
        int contador=105;

        while ( (contador<=120) && (contador % 13 != 0) )
        {
            Console.Write("{0} ", contador);
            contador++;
        }
    }
}
```

(Sí, en la mayoría de los casos, un "for" se puede convertir en un "while"; veremos más detalles dentro de muy poco).

Ejercicios propuestos:

(2.2.10.1) Crea un programa que pida al usuario dos números y escriba su máximo común divisor (pista: una solución lenta pero sencilla es probar con un "for" todos los números descendiendo a partir del menor de ambos, hasta llegar a 1; cuando encuentres un número que sea divisor de ambos, interrumpe la búsqueda con "break").

(2.2.10.2) Crea un programa que pida al usuario dos números y escriba su mínimo común múltiplo (pista: una solución lenta pero sencilla es probar con un "for" todos los números a partir del mayor de ambos, de forma creciente; cuando encuentres un número que sea múltiplo de ambos, interrumpes la búsqueda con "break").
(2.2.10.3) Crea una versión alternativa del ejercicio 2.2.10.1 (máximo común divisor) usando "while", en vez de "for" y "break".
(2.2.10.4) Crea una versión alternativa del ejercicio 2.2.10.2 (mínimo común múltiplo) usando "while", en vez de "for" y "break".

2.2.11. Forzar la siguiente iteración: continue

Podemos saltar alguna repetición de un bucle con la orden "**continue**":

```
// Ejemplo_02_02_11a.cs
// "for" interrumpido con "continue"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_11a
{
    static void Main()
    {
        for (int contador=1; contador<=10; contador++)
        {
           if (contador==5)
                continue;

            Console.Write("{0} ", contador);
        }
    }
}
```

El resultado de este programa es:

```
  1 2 3 4 6 7 8 9 10
```

En él podemos observar que no aparece el valor 5. Al igual que ocurre con "break", su uso está desaconsejado. Como alternativa más legible, se podría haber utilizado un "if" opuesto al anterior, que escriba los valores que no sean 5, así:

```
// Ejemplo_02_02_11b.cs
// Alternativa a "for" interrumpido con "continue"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_11b
{
    static void Main()
    {
```

```
        for (int contador=1; contador<=10; contador++)
        {
            if (contador != 5)
                Console.Write("{0} ", contador);
        }
    }
}
```

Ejercicios propuestos:

(2.2.11.1) Crea un programa que escriba los números del 20 al 10, descendiendo, excepto el 13, usando "continue".

(2.2.11.2) Crea un programa que escriba los números pares del 2 al 106, excepto los que sean múltiplos de 10, usando "continue".

(2.2.11.3) Crea una versión alternativa del ejercicio 2.2.11.1, que no utilice "continue" sino el "if" contrario.

(2.2.11.4) Crea una versión alternativa del ejercicio 2.2.11.2, que no emplee "continue" sino el "if" contrario.

2.2.12. Equivalencia entre "for" y "while"

En la gran mayoría de condiciones, un bucle "for" equivale a un "while" compactado, de modo que casi cualquier "for" se puede escribir de forma alternativa como un "while", como en este ejemplo:

```
// Ejemplo_02_02_12a.cs
// "for" y "while" equivalente
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_12a
{
    static void Main()
    {
        for (int contador=1; contador<=10; contador++)
        {
            Console.Write("{0} ", contador);
        }

        Console.WriteLine();

        int n=1;
        while (n<=10)
        {
            Console.Write("{0} ", n);
            n++;
        }
    }
}
```

Incluso se comportarían igual si no se avanza de uno en uno, o se interrumpe con "break", pero no en caso de usar un "continue", como muestra este ejemplo:

```
// Ejemplo_02_02_12b.cs
// "for" y "while" equivalente... con "continue"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_12b
{
    static void Main()
    {
        for (int contador=1; contador<=10; contador++)
        {
            if (contador == 5)
                continue;
            Console.Write("{0} ", contador);
        }

        Console.WriteLine();

        int n=1;
        while (n<=10)
        {
            if (n == 5)
                continue;
            Console.Write("{0} ", n);
            n++;
        }
    }
}
```

En este caso, el "for" muestra todos los valores menos el 5, pero en el "while" se provoca un bucle sin fin y el programa se queda "colgado" tras escribir el número 4, porque cuando se llega al número 5, la orden "continue" hace que dicho valor no se escriba, pero que tampoco se incremente la variable, de modo que nunca se llega a pasar del valor 5.

Ejercicios propuestos:

(2.2.12.1) Crea un programa que escriba los números del 100 al 200, separados por un espacio, sin avanzar de línea, usando "for". En la siguiente línea, vuelve a escribirlos usando "while".

(2.2.12.2) Crea un programa que escriba los números pares del 20 al 10, descendiendo, excepto el 14, primero con "for" y luego con "while".

2.2.13. Ejercicios resueltos sobre bucles

Existen varios errores frecuentes en el manejo de los bucles. Por ejemplo, incluir un "punto" y coma tras una orden "for" o "while" puede hacer que nada se repita y que el programa se comporte de forma errónea. Por eso, aquí tienes varios ejercicios resueltos, que te ayudarán a "entrenar la vista" para localizar ese tipo de problemas:

- ¿Qué escribiría en pantalla este fragmento de código?

  ```
  for (i=1; i<4; i++) Console.Write("{0} ",i);
  ```

  Respuesta: los números del 1 al 3 (se empieza en 1 y se repite mientras sea menor que 4).

- ¿Qué escribiría en pantalla este fragmento de código?

  ```
  for (i=1; i>4; i++) Console.Write("{0} ",i);
  ```

  Respuesta: no escribiría nada, porque la condición es falsa desde el principio.

- ¿Qué escribiría en pantalla este fragmento de código?

  ```
  for (i=1; i<=4; i++); Console.Write("{0} ",i);
  ```

  Respuesta: escribe un 5, porque hay un punto y coma después del "for", de modo que repite cuatro veces una orden vacía, y cuando termina el "for", "i" ya tiene el valor 5.

- ¿Qué escribiría en pantalla este fragmento de código?

  ```
  for (i=1; i<4; ) Console.Write("{0} ",i);
  ```

  Respuesta: escribe "1" continuamente, porque no aumentamos el valor de "i", luego nunca se llegará a cumplir la condición de salida.

- ¿Qué escribiría en pantalla este fragmento de código?

  ```
  for (i=1; ; i++) Console.Write("{0} ",i);
  ```

Respuesta: escribe números crecientes continuamente, comenzando en uno y aumentando una unidad en cada pasada, pero sin terminar.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i = 0 ; i <= 4 ; i++) {
    if ( i == 2 ) continue ;
    Console.Write("{0} ",i);
}
```

Respuesta: escribe los números del 0 al 4, excepto el 2.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i = 0 ; i <= 4 ; i++) {
    if ( i == 2 ) break ;
    Console.Write("{0} ",i);
}
```

Respuesta: escribe los números 0 y 1 (interrumpe en el 2).

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i = 0 ; i <= 4 ; i++) {
    if ( i == 10 ) continue ;
    Console.Write("{0} ",i);
}
```

Respuesta: escribe los números del 0 al 4, porque la condición del "continue" nunca se llega a dar.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i = 0 ; i<= 4 ; i++)
    if ( i == 2 ) continue ;
    Console.Write("{0} ",i);
```

Respuesta: escribe 5, porque no hay llaves tras el "for", luego sólo se repite la orden "if".

2.2.14. Saltos incondicionales: goto

El lenguaje C# también permite la orden "**goto**", para hacer saltos incondicionales. **Su uso indisciplinado está muy mal visto**, porque puede ayudar a hacer programas llenos de saltos, muy difíciles de seguir. Pero en casos concretos puede ser útil, por ejemplo, para salir de un bucle muy anidado (un "for" dentro de otro "for", que a su vez está dentro de otro "for": en este caso, "break" sólo saldría del "for" más interno). Al igual que ocurría con la orden "break", será preferible replantear las condiciones de forma más natural, y no utilizar "goto".

El formato de "goto" es

```
goto etiqueta;
```

y la posición de salto se indica con una "etiqueta", un nombre seguido de un símbolo de dos puntos (:)

```
etiqueta:
```

como en el siguiente ejemplo, que escribe "Pasada 1", "Pasada 2" y así sucesivamente hasta que se detenga la ejecución cerrando la ventana del programa, porque el salto incondicional hacia un punto anterior crea un "bucle sin fin":

```
// Ejemplo_02_02_14a.cs
// Salto incondicional con "goto"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_14a
{
    static void Main()
    {
        int cantidad = 1;

        repetir:
        Console.Write("Pasada {0}; ", cantidad);
        cantidad++;
        goto repetir;
    }
}
```

Un ejemplo de uso más real, para salir de un bucle muy anidado, podría ser:

```
// Ejemplo_02_02_14b.cs
// "for" y "goto"
// Introducción a C#, por Nacho Cabanes
```

```
using System;

class Ejemplo_02_02_14b
{
    static void Main()
    {
        int i, j;

        for (i=0; i<=5; i++)
            for (j=0; j<=20; j=j+2)
            {
                if ((i==1) && (j>=7))
                    goto salida;
                Console.WriteLine("i vale {0} y j vale {1}.", i, j);
            }

        salida:
            Console.Write("Fin del programa");

    }
}
```

El resultado de este programa es:

```
i vale 0 y j vale 0.
i vale 0 y j vale 2.
i vale 0 y j vale 4.
i vale 0 y j vale 6.
i vale 0 y j vale 8.
i vale 0 y j vale 10.
i vale 0 y j vale 12.
i vale 0 y j vale 14.
i vale 0 y j vale 16.
i vale 0 y j vale 18.
i vale 0 y j vale 20.
i vale 1 y j vale 0.
i vale 1 y j vale 2.
i vale 1 y j vale 4.
i vale 1 y j vale 6.
Fin del programa
```

Vemos que cuando i=1 y j>=7, se sale de los dos "for", mientras que el programa "casi equivalente" que emplea "break" no se comporta igual, sino que continúa y da todas las pasadas para i=2, i=3, i=4 e i=5.

```
// Ejemplo_02_02_14c.cs
// "break" en vez de "goto"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_14c
{
    static void Main()
    {
        int i, j;

        for (i=0; i<=5; i++)
```

```
            for (j=0; j<=20; j=j+2)
            {
                if ((i==1) && (j>=7))
                    break;
                Console.WriteLine("i vale {0} y j vale {1}.", i, j);
            }

        Console.Write("Fin del programa");

    }
}
```

Ejercicios propuestos:

(2.2.14.1) Crea un programa que escriba los números del 1 al 10, separados por un espacio, sin avanzar de línea. No puedes usar "for", ni "while", ni "do..while", sólo "if" y "goto".

2.2.15. "for" y el operador coma

Cuando hemos empleado la orden "for", siempre hemos usado una única variable como contador. Esto es, con diferencia, lo más habitual, pero no tiene por qué ocurrir siempre. Vamos a verlo con un ejemplo:

```
// Ejemplo_02_02_15a.cs
// Operador coma
// Introducción a C#, por Nacho Cabanes

using System;

class OperadorComa
{
    static void Main()
    {
        int i, j;

        for (i=0, j=1; i<=5 && j<=30; i++, j+=2)
            Console.WriteLine("i vale {0} y j vale {1}", i, j);
    }
}
```

Vamos a ver qué hace este "for":

- Los valores iniciales son i=0, j=1.
- Se repetirá mientras que i <= 5 y j <= 30.
- Al final de cada paso, i aumenta en una unidad, y j en dos unidades.

El resultado de este programa es:

```
i vale 0 y j vale 1
```

```
i vale 1 y j vale 3
i vale 2 y j vale 5
i vale 3 y j vale 7
i vale 4 y j vale 9
i vale 5 y j vale 11
```

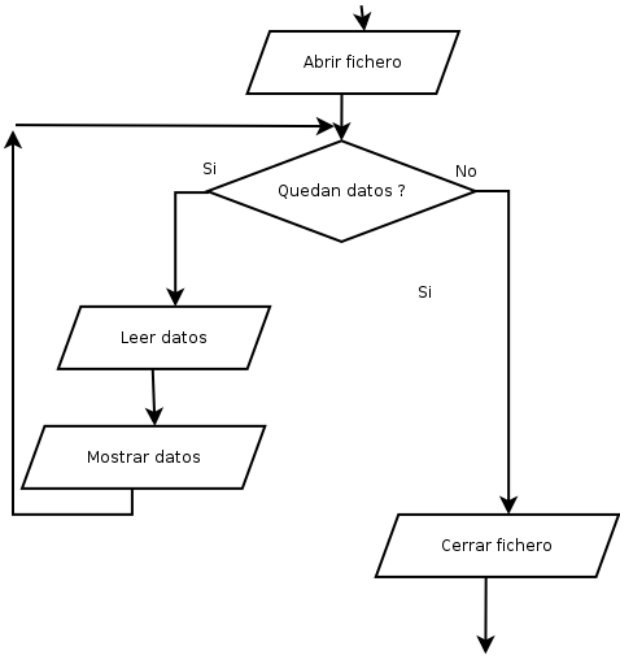
Nota: En el lenguaje C se puede "rizar el rizo" todavía un poco más: la condición de terminación también podría tener una coma, y entonces no se sale del bucle "for" hasta que se cumplen las dos condiciones (algo que no es válido en C#, ya que la condición debe ser un "booleano", algo que tenga como resultado "verdadero" o "falso", y la coma no es un operador válido para operaciones booleanas):

```
for (i=0, j=1; i<=5, j<=30; i++, j+=2)
```

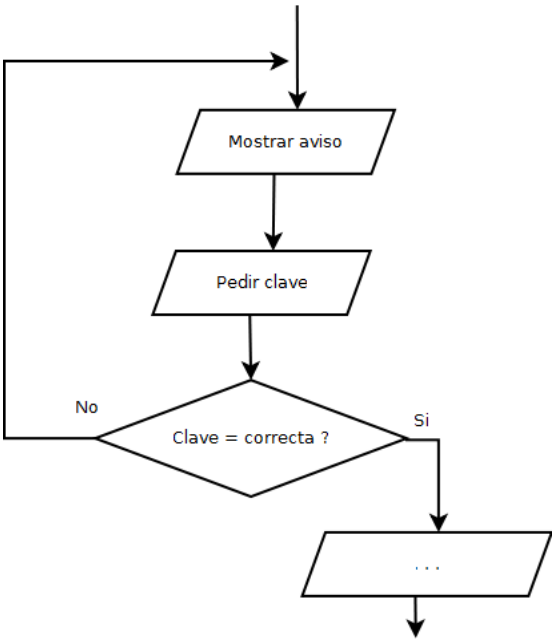
2.3. Más sobre diagramas de flujo. Diagramas de Nassi–Shneiderman

En el apartado 2.1.7 tuvimos un contacto con la forma de ayudarnos de los diagramas de flujo para plantear lo que un programa debe hacer. Si entendemos esta herramienta, el paso de esta notación a C# (o a casi cualquier otro lenguaje de programación) es sencillo. Pero este tipo de diagramas es antiguo, no tiene en cuenta todas las posibilidades del lenguaje C# (y de muchos otros lenguajes actuales). Por ejemplo, no existe una forma clara de representar una orden "switch", que equivaldría a varias condiciones encadenadas.

Por su parte, un bucle "**while**" se vería como una condición que hace que algo se repita (una flecha que vuelve hacia atrás, al punto en el que se comprobaba la condición):



Y un "**do...while**" se representaría como una condición al final de un bloque que se repite:



Aun así, existen otras notaciones más modernas y que pueden resultar más cómodas. Sólo comentaremos una: los **diagramas de Nassi–Shneiderman**, diagramas de cajas o diagramas de Chapin. En ellos se representa cada orden dentro de una caja, y el conjunto del programa es una serie de cajas apiladas de arriba (primera orden) a abajo (última orden):

Pedir primer número
Pedir segundo número
Mostrar primer num+segundo num

Las **condiciones** se denotan dividiendo las cajas en dos:

Pedir número n1	
Pedir número n2	
n1>n2?	
si	no
Decir "n1 es mayor"	Decir "n2 es mayor"

Y las condiciones repetitivas se indican dejando una barra vertical a la izquierda, que marca qué zona es la que se repite, tanto si la condición se comprueba al final (**do..while**):

	Escribir "Teclee su clave"
	Leer clave de acceso
Mientras clave ≠ correcta	

como si se comprueba al principio (**while**):

Abrir fichero	
Mientras haya datos en fichero	
	Leer dato
	Mostrar dato
Cerrar fichero	

Los "**for**" se suelen mostrar como un "while", pero detallando los valores: "para n = 1 hasta 5".

En la práctica, ambas notaciones **se usan poco** a nivel de programación formal, porque un programa simple puede necesitar más tiempo para representarse con una notación gráfica como éstas que para ser tecleado, y la diferencia es aún mayor cuando hay que hacer alguna modificación, que son más costosas en una notación gráfica que en un programa convencional. Aun así, en un momento inicial del aprendizaje (como éste), pueden ayudar a asentar conceptos básicos, como

decidir qué debe abarcar un "if" o si se debe optar por un "while" o un "do-while". En "el mundo real", se usan más en **diseño de alto nivel de programas complejos** que como paso previo en programas sencillos.

Ejercicios propuestos:

(2.3.1) Crea el diagrama de Nassi–Shneiderman para el programa que pide dos números al usuario y dice cuál es el mayor de los dos. Compáralo con el diagrama de flujo del ejercicio 2.1.7.1.

(2.3.2) Crea el diagrama de Nassi–Shneiderman para el programa que pide al usuario dos números y dice si uno de ellos es positivo, si lo son los dos o si no lo es ninguno. Compáralo con el diagrama de flujo del ejercicio 2.1.7.2.

(2.3.3) Crea el diagrama de Nassi–Shneiderman para el programa que pide tres números al usuario y dice cuál es el mayor de los tres. Compáralo con el diagrama de flujo del ejercicio 2.1.7.3.

(2.3.4) Crea el diagrama de flujo y el diagrama de Nassi–Shneiderman para un programa que pida pares de números al usuario y muestre el resultado dividir el primero entre el segundo, repitiendo hasta que el segundo número sea cero.

2.4. foreach

Nos queda por ver otra orden que permite hacer cosas repetitivas: "foreach" (se traduciría "para cada"). La veremos con detalle más adelante, cuando manejemos estructuras de datos más complejas, que es en las que la nos resultará útil para extraer los datos de uno en uno. De momento, el único dato compuesto que hemos visto (y todavía con muy poco detalle) es la cadena de texto, "string", de la que podríamos obtener las letras una a una con "foreach" así:

```
// Ejemplo_02_04a.cs
// Primer ejemplo de "foreach"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_04a
{
    static void Main()
    {
        Console.Write("Dime tu nombre: ");
        string nombre = Console.ReadLine();
        foreach(char letra in nombre)
        {
            Console.WriteLine(letra);
        }
    }
}
```

Ejercicios propuestos:

(2.4.1) Crea un programa que cuente cuantas veces aparece la letra 'a' en una frase que teclee el usuario, utilizando "foreach".

2.5. Recomendación de uso para los distintos tipos de bucle

En general, nos interesará usar "**while**" cuando puede que la parte repetitiva no se llegue a repetir nunca (por ejemplo: cuando leemos un fichero, si el fichero está vacío o no existe, no habrá datos que leer).

De igual modo, "**do...while**" será lo adecuado cuando debamos repetir al menos una vez (por ejemplo, para pedir una clave de acceso, se le debe preguntar al menos una vez al usuario, o quizá más veces, si no la teclea correctamente).

En cuanto a "**for**", es equivalente a un "while", pero la sintaxis habitual de la orden "for" hace que sea especialmente útil cuando sabemos exactamente cuantas veces queremos que se repita (por ejemplo: 10 veces podría ser "`for (i=1; i<=10; i++)`"). Conceptualmente, si un "for" necesita un "break" para ser interrumpido en un caso especial, es porque realmente no se trata de un contador, y en ese caso debería ser reemplazado por un "while", para que el programa resulte más legible.

Ejercicios propuestos (utiliza en cada caso el tipo de bucle que consideres más adecuado):

(2.5.1) Crea un programa que muestre una cuenta atrás (3 2 1 0) desde el número que introduzca el usuario hasta cero. Ese número debe estar entre 1 y 10 (y el programa debe comprobar que realmente lo está, y volverlo a pedir tantas veces como sea necesario, en caso de que no sea así).

(2.5.2) Crea un programa en el que el usuario deba adivinar un número del 1 al 100 (prefijado en el programa). En cada pasada deberá avisar de si se ha pasado o se ha quedado corto.

(2.5.3) Haz un programa que dé al usuario la oportunidad de adivinar un número del 1 al 100 (prefijado en el programa) en un máximo de 6 intentos. En cada pasada deberá avisar de si se ha pasado o se ha quedado corto.

(2.5.4) Crea un programa que pida un número al usuario y diga si es primo (divisible sólo entre 1 y él mismo).

(2.5.5) Crea un programa que descomponga un número (que teclee el usuario) como producto de su factores primos. Por ejemplo, 60 = 2 · 2 · 3 · 5 (pista: como primera aproximación, puedes escribir siempre un "punto" después de cada

número y luego terminar con la cifra uno, así: 60 = 2 · 2 · 3 · 5 · 1; cuando lo consigas, piensa cómo harías para eliminar ese "• 1" del final).

(2.5.6) Crea un programa que calcule un número elevado a otro, usando multiplicaciones sucesivas.

(2.5.7) Crea un programa que "dibuje" un rectángulo hueco, cuyo borde sea una fila (o columna) de asteriscos y cuyo interior esté formado por espacios en blanco, con el ancho y el alto que indique el usuario. Por ejemplo, si desea anchura 4 y altura 3, el rectángulo sería así:

```
****
*  *
****
```

(2.5.8) Crea un programa que "dibuje" un triángulo creciente, alineado a la derecha, con la altura que indique el usuario. Por ejemplo, si el usuario dice que desea 4 caracteres de alto, el triángulo sería así:

```
   *
  **
 ***
****
```

(2.5.9) Crea un programa que devuelva el cambio de una compra, utilizando monedas (o billetes) del mayor valor posible. Supondremos que tenemos una cantidad ilimitada de monedas (o billetes) de 100, 50, 20, 10, 5, 2 y 1, y que no hay decimales. La ejecución podría ser algo como:

```
Precio? 44
Pagado? 100
Su cambio es de 56: 50 5 1

Precio? 1
Pagado? 100
Su cambio es de 99: 50 20 20 5 2 2
```

(2.5.10) Crea un programa que "dibuje" un cuadrado formado por cifras sucesivas, con el tamaño que indique el usuario, hasta un máximo de 9. Por ejemplo, si desea tamaño 5, el cuadrado sería así:

```
11111
22222
33333
44444
55555
```

2.6. Una alternativa para el control errores: las excepciones

La forma "clásica" del control de errores es utilizar instrucciones "if", que vayan comprobando cada una de lqs posibles situaciones que pueden dar lugar a un error, a medida que estas situaciones llegan. Esto tiende a hacer el programa más difícil de leer, porque la lógica de la resolución del problema se ve interrumpida por órdenes que no tienen que ver con el problema en sí, sino con las posibles situaciones de error. Por eso, los lenguajes modernos, como C#, permiten una alternativa: el manejo de "excepciones".

La idea es la siguiente: "intentaremos" dar una serie de pasos, y al final de todos ellos indicaremos qué hay que hacer en caso de que alguno no se consiga completar. Esto permite que el programa sea más legible que la alternativa "convencional".

Lo haremos dividiendo el fragmento de programa en **dos bloques**:

- En un primer bloque, indicaremos los pasos que queremos "**intentar**" (try).
- A continuación, detallaremos las posibles situaciones de error (excepciones) que queremos "**interceptar**" (catch), y lo que se debe hacer en ese caso.

Lo veremos más adelante con más detalle, cuando nuestros programas sean más complejos, especialmente en el **manejo de ficheros**, pero podemos acercarnos con un primer ejemplo, que intente dividir dos números, e intercepte los posibles errores:

```
// Ejemplo_02_06a.cs
// Excepciones (1)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_06a
{

    static void Main()
    {
        int numero1, numero2, resultado;

        try
        {
            Console.WriteLine("Introduzca el primer numero");
            numero1 = Convert.ToInt32( Console.ReadLine() );

            Console.WriteLine("Introduzca el segundo numero");
            numero2 = Convert.ToInt32( Console.ReadLine() );
```

```
            resultado = numero1 / numero2;
            Console.WriteLine("Su división es: {0}", resultado);
        }
        catch (Exception errorEncontrado)
        {
            Console.WriteLine("Ha habido un error: {0}",
                errorEncontrado.Message);
        }
    }
}
```

(La variable "errorEncontrado" es de tipo "Exception", y nos sirve para poder acceder a detalles como el mensaje correspondiente a ese tipo de excepción: errorEncontrado.Message)

En este ejemplo, si escribimos un texto en vez de un número, obtendríamos como respuesta

```
Introduzca el primer numero
hola
Ha habido un error: La cadena de entrada no tiene el formato correcto.
```

Y si el segundo número es 0, se nos diría

```
Introduzca el primer numero
3
Introduzca el segundo numero
0
Ha habido un error: Intento de dividir por cero.
```

Una alternativa más elegante es no "atrapar" todos los posibles errores a la vez, sino uno por uno (con varias sentencias "catch"), para poder tomar distintas acciones, o al menos dar mensajes de error más detallados, así:

```
// Ejemplo_02_06b.cs
// Excepciones (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_06b
{
    static void Main()
    {
        int numero1, numero2, resultado;

        try
        {
            Console.WriteLine("Introduzca el primer numero");
            numero1 = Convert.ToInt32( Console.ReadLine() );
```

```
            Console.WriteLine("Introduzca el segundo numero");
            numero2 = Convert.ToInt32( Console.ReadLine() );

            resultado = numero1 / numero2;
            Console.WriteLine("Su división es: {0}", resultado);
        }

        catch (FormatException)
        {
            Console.WriteLine("No es un número válido");
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("No se puede dividir entre cero");
        }
    }
}
```

Como se ve en este ejemplo, si no vamos a usar detalles adicionales del error que ha afectado al programa, no necesitamos declarar ninguna variable de tipo Exception: nos basta con construcciones como "catch (FormatException)" en vez de "catch (FormatException e)".

¿Y cómo sabemos qué excepciones debemos interceptar? La mejor forma es mirar en la "referencia oficial" para programadores de C#, la MSDN (Microsoft Developer Network): si tecleamos en un buscador de Internet algo como "msdn convert toint32" nos llevará a una página en la que podemos ver que hay dos excepciones que podemos obtener en ese intento de conversión de texto a entero: FormatException (no se ha podido convertir) y OverflowException (número demasiado grande). Otra alternativa más arriesgada es "probar el programa" y ver qué errores obtenemos en pantalla al introducir un valor no válido. Esta alternativa es la menos deseable, porque quizá pasemos por alto algún tipo de error que pueda surgir y que nosotros no hayamos previsto. En cualquier caso, volveremos a las excepciones más adelante.

Ejercicios propuestos:

(2.6.1) Crea un programa que pregunte al usuario su edad y su año de nacimiento. Si la edad que introduce no es un número válido, mostrará un mensaje de aviso y se detendrá. Lo mismo ocurrirá si el año de nacimiento no es un número válido.

(2.6.2) Crea un programa que pregunte al usuario su edad y su año de nacimiento. Si la edad que introduce no es un número válido, mostrará un mensaje de aviso, pero aun así le preguntará su año de nacimiento.

2.7. Conceptos básicos sobre depuración

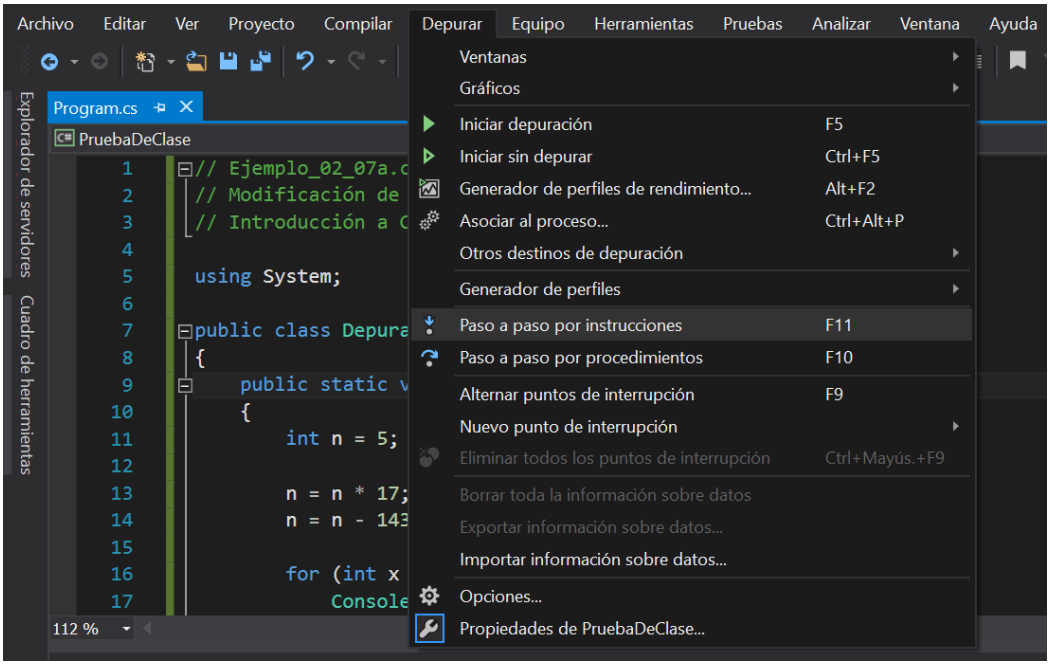
La depuración es el análisis de un programa para descubrir fallos. El nombre en inglés es "debug", porque esos fallos de programación reciben el nombre de "bugs" (bichos).

Para eliminar esos fallos que hacen que un programa no se comporte como debería, se usan unas herramientas llamadas "depuradores". Estos nos permiten avanzar paso a paso para ver cómo transcurre realmente nuestro programa, y también nos dejan analizar los valores de las variables en cada momento.

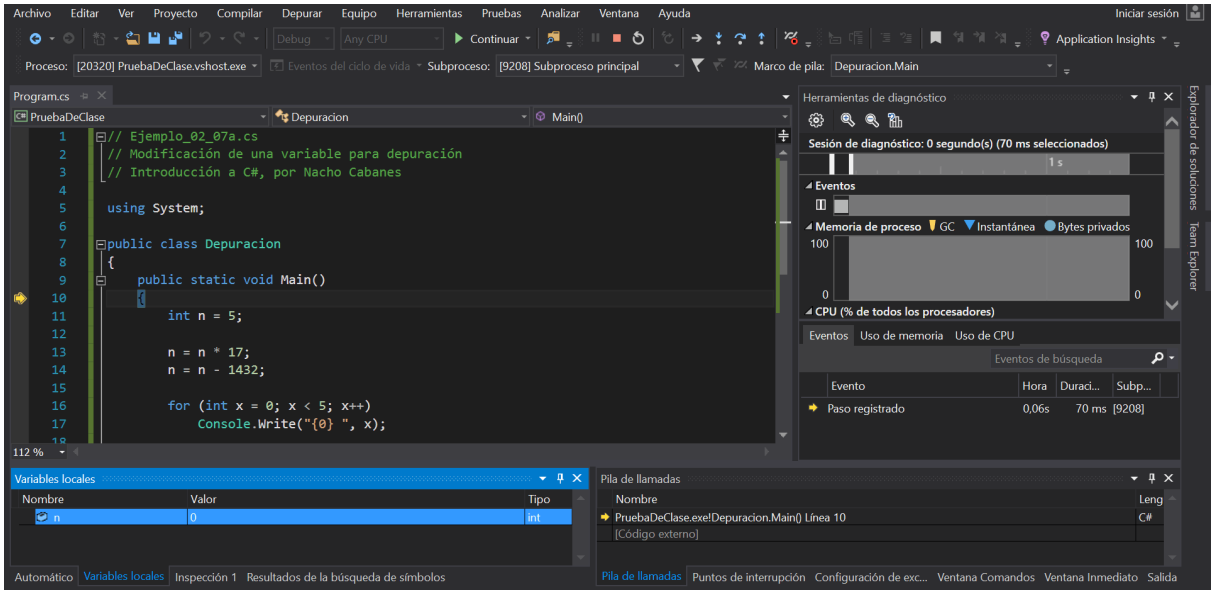
Veremos como ejemplo el caso de Visual Studio 2015 Community, pero las diferencias con otras versiones de este entorno deberían ser mínimas. Vamos a partir de un programa sencillo que manipule un par de variables, como:

```
// Ejemplo_02_07a.cs
// Modificación de una variable para depuración
// Introducción a C#, por Nacho Cabanes

using System;

class Depuracion
{
    static void Main()
    {
        int n = 5;

        n = n * 17;
        n = n - 1432;

        for (int x = 0; x < 5; x++)
            Console.Write("{0} ", x);

        for (int i = 1; i < n; i++)
            Console.Write("{0} ", i);
    }
}
```

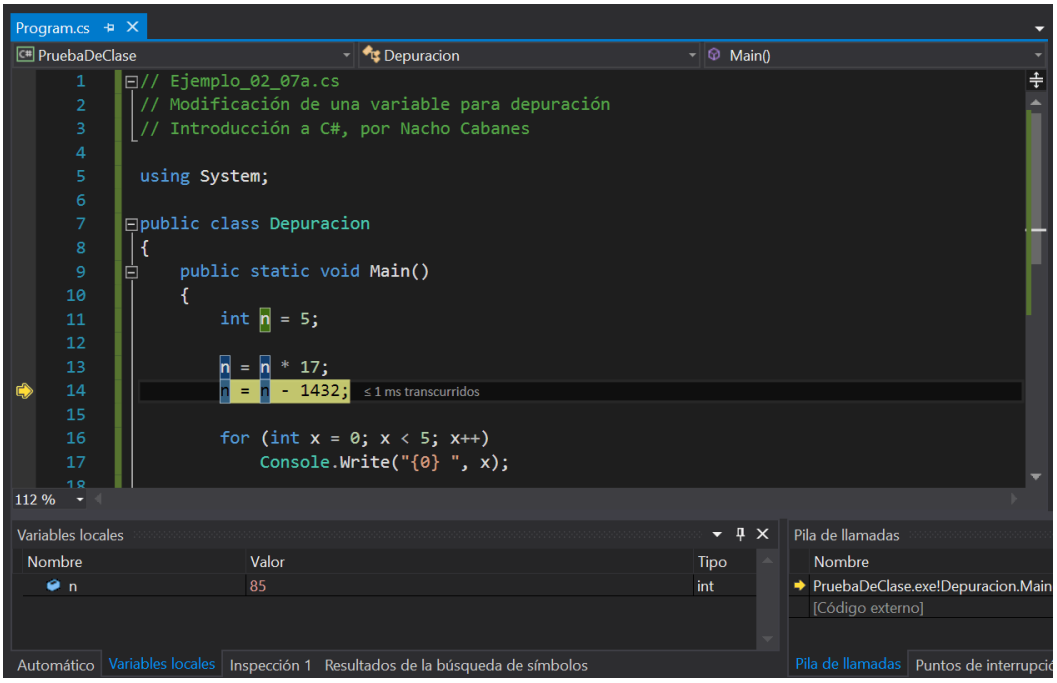
Para avanzar paso a paso y ver los valores de las variables, entramos al menú "Depurar". En él aparece la opción "Paso a paso por instrucciones" (a la que corresponde la tecla F11):



Si escogemos esa opción del menú o pulsamos F11, aparece una ventana inferior con la lista de variables (inicialmente, sólo "n"), y una flecha amarilla que señala el punto del programa en el que nos encontramos (actualmente al principio del programa):

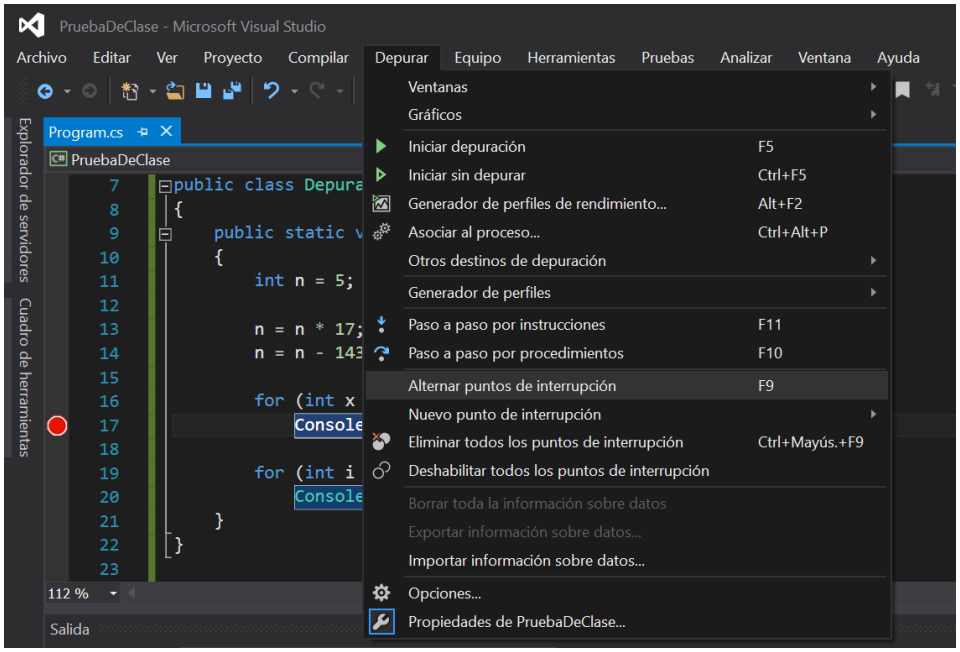


Cada vez que pulsemos nuevamente F11 (o vayamos al menú, o al botón correspondiente de la barra de herramientas), el depurador analiza una nueva línea de programa, muestra los valores de las variables correspondientes (el cambio más reciente se verá en color rojo), y se vuelve a quedar parado, realzando con fondo amarillo la siguiente línea que se analizará:



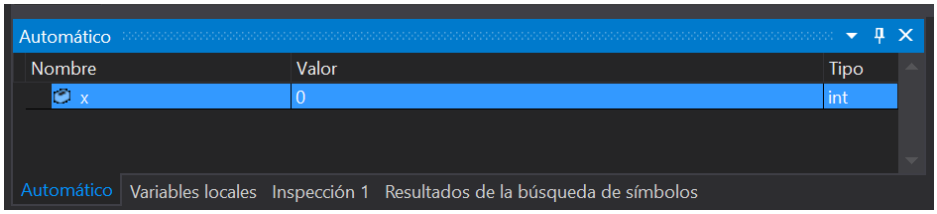
En este caso, hemos dado dos pasos: n = 5 y n = n * 17, de modo que actualmente n vale 85. El cursor está en la siguiente línea que queremos se va a procesar, que aparece destacada con fondo amarillo.

En este primer contacto, hemos avanzado paso desde el principio del programa, pero eso no es algo totalmente habitual. Es más frecuente que supongamos en qué zona del programa se encuentra el error, y sólo queramos depurar una parte de programa. La forma de conseguirlo es desplazarnos hasta la primera línea que queramos analizar y escoger otra de las opciones del menú de depuración: "Alternar puntos de interrupción" (tecla F9). Aparecerá una marca de color rojo en la línea actual. Como alternativa, podemos hacer clic con el ratón en el margen izquierdo del programa, junto a esa línea:

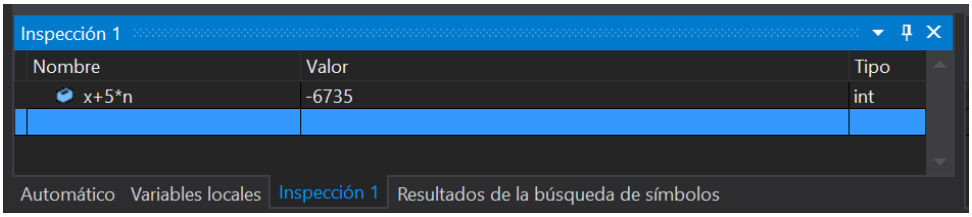


Si ahora iniciamos la depuración del programa, avanzará hasta ese punto, y será en esa línea en la que se detenga. A partir de ahí, podemos seguir depurando paso a paso como antes, pulsando F11.

Si tenemos muchas variables, nos puede interesar más la pestaña "Automático" que la la de "Variables locales", porque ésta mostrará sólo aquellas que han cambiado recientemente y no veremos las que Visual Studio considere que son menos relevantes en este momento:



Además, existe también una pestaña "Inspección", en la que podemos escribir nosotros cualquier expresión cuyo valor queramos analizar:



Y en ciertas versiones de Visual Studio es posible que se nos muestre también en la parte derecha de la pantalla una ventana de "Herramientas de diagnóstico",

que, entre otros detalles, nos puede informar del consumo de memoria y de CPU por parte de nuestro programa. En programas más complejos, un uso de memoria que aumente continuamente sería un claro indicador de un error en nuestro programa: