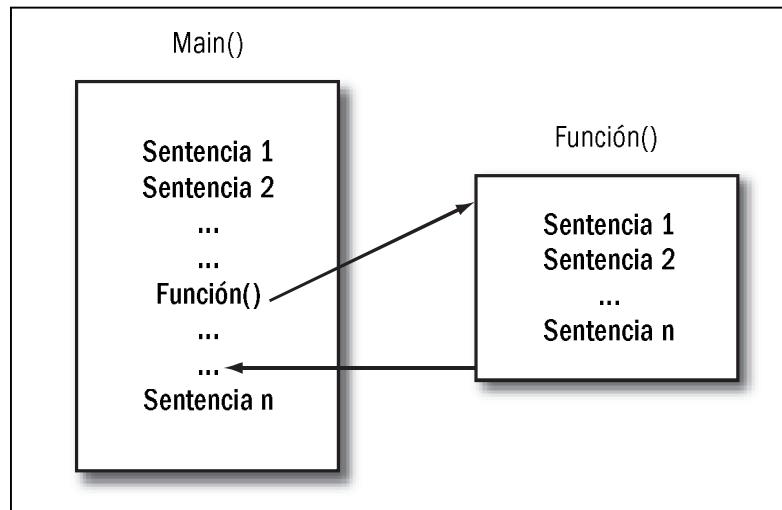


# LAS FUNCIONES

Empecemos a conocer las funciones. Veamos un ejemplo de un problema y luego la forma cómo la utilización de la función nos puede ayudar. La función es un elemento del programa que contiene código y se puede ejecutar, es decir, lleva a cabo una operación. La función puede **llamarse** o **invocarse** cuando sea necesario y entonces el código que se encuentra en su interior se va a ejecutar. Una vez que la función termina de ejecutarse el programa continúa en la sentencia siguiente de donde fue llamada.



**Figura 1.** Podemos observar cómo la ejecución del programa se dirige a la función y luego regresa a la sentencia siguiente de donde fue invocada.

Las funciones, para que sean útiles, deben estar especializadas. Es decir que cada función debe hacer solamente una cosa y hacerla bien. Nosotros ya hemos utilizado una función, es la función **Main()** y podemos crear más funciones conforme las necesitemos. Dentro de la programación orientada a objetos las clases tienen código y este código se encuentra adentro de funciones llamadas métodos. Por ejemplo, cuando hemos convertido de cadena a un entero, hacemos uso de una función llamada **ToIn32()** que se encuentra adentro de la clase **Convert**.

Las funciones constan de cinco partes:

```

modificador tipo Nombre(parámetros)
{
    código
}
    
```

Veremos ahora las cinco partes, pero conforme estudiemos a las funciones sabremos cómo son utilizadas. Las funciones pueden regresar información y esta información

puede ser cadena, entero, flotante o cualquier otro tipo. En la sección de **tipo** tenemos que indicar precisamente la clase de información que regresa. Si la función no regresa ningún valor entonces tenemos que indicar a su tipo como **void**. Todas las funciones deben identificarse y lo hacemos por medio de su **nombre**. Las funciones que coloquemos adentro de las clases deben de tener un nombre único. El nombre también es utilizado para invocar o ejecutar a la función.

Las funciones pueden necesitar de **datos** o **información** para poder trabajar. Nosotros le damos esta información por medio de sus **parámetros**. Los parámetros no son otra cosa que una lista de variables que reciben estos datos. Si la función no necesita usar a los parámetros, entonces simplemente podemos dejar los paréntesis vacíos. Nunca debemos olvidar colocar los paréntesis aunque no haya parámetros.

El código de la función se sitúa adentro de un **bloque de código**. En esta sección podemos colocar cualquier código válido de C#, es decir, declaración de variables, ciclos, estructuras selectivas e incluso invocaciones a funciones.

Las funciones al ser declaradas pueden llevar un **modificador** antes del tipo. Los modificadores cambian la forma como trabaja la función. Nosotros estaremos utilizando un modificador conocido como **static**. Este modificador nos permite usar a la función sin tener que declarar un objeto de la clase a la que pertenece.

Tenemos cuatro tipos básicos de funciones: las que solo ejecutan código, las que reciben parámetros, las que regresan valores y las que reciben parámetros y reciben valores. Durante este capítulo iremos conociéndolas.

Ahora que ya conocemos los elementos básicos de las funciones, podemos ver un ejemplo de donde su uso sería adecuado. Supongamos que tenemos un programa en el cual en diferentes secciones tenemos que escribir el mismo código, pero debido a su lógica no podemos usar un ciclo, ya que está en diferentes partes del programa. En este caso nuestra única solución hasta el momento sería simplemente escribir nuevamente el código. Otra forma de resolverlo y que nos evitaría tener código repetido es usar la función. El código que se va a utilizar repetidamente se coloca adentro de la función, y cada vez que necesitemos usarlo, simplemente se invoca a la función. En nuestro ejemplo del carpintero, el cual analizamos en el capítulo anterior, nos

### III

Es importante seleccionar correctamente el nombre de la función. El nombre debe hacer referencia al tipo de trabajo que lleva a cabo la función; en muchos casos podemos hacer uso de verbos. El nombre de la variable puede llevar números, pero debe empezar con una letra.

encargamos de realizar conversiones de pies y pulgadas a centímetros. La conversión se hacía con una fórmula sencilla pero un poco larga. Si ahora nuestro carpintero necesita un programa para transformar las medidas de una mesa completa a centímetros, entonces tendríamos que usar la fórmula adecuada en varios lugares. Para no copiar el mismo código varias veces, lo más conveniente sería colocar la fórmula en una función y simplemente invocarla donde fuera necesario. Empecemos a conocer cómo programar e invocar a las funciones.

## Funciones que ejecutan código

El primer tipo de funciones que vamos a conocer son las que ejecutan código. Estas funciones no reciben datos y no regresan ningún dato. Solamente llevan a cabo alguna operación. Aunque en este momento no parecen muy útiles, sí lo son.

Para poder utilizar la función, debemos **declararla**. La declaración se debe hacer adentro del bloque de código correspondiente a una clase. Para los ejemplos analizados en este libro, estamos usando la clase denominada **Program**. Nuestra función **Main()** se encuentra también adentro de esta clase.

En este momento tomaremos una estrategia de programación, que aún no pertenece a la programación estructurada, en el futuro veremos las técnicas orientadas a objetos. En esta técnica usamos a la función **Main()** como administradora de la lógica y la mayor parte del proceso se llevará a cabo en las funciones.

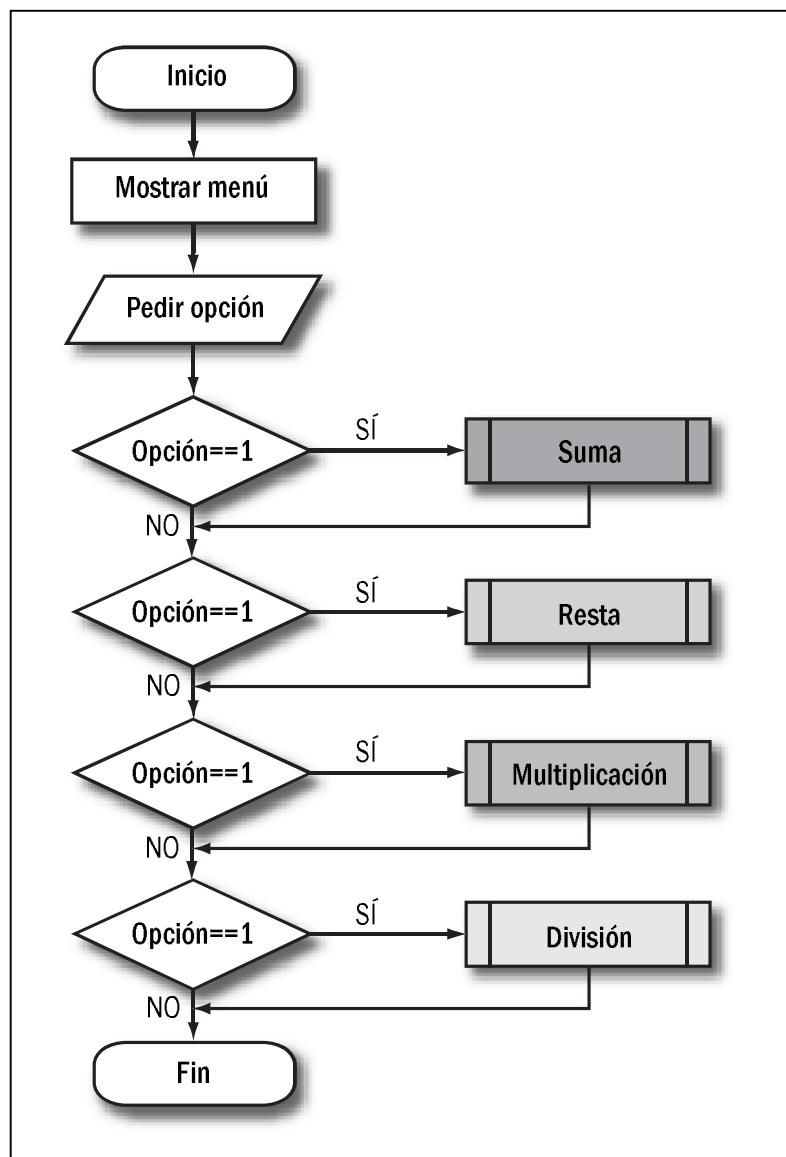
Crearemos una aplicación y le colocaremos cada uno de los tipos de funciones que vamos a aprender, dejando a **Main()** como nuestra administradora del programa. El programa va a ser conceptualmente sencillo, ya que lo que nos interesa es comprender el funcionamiento de las funciones.

El programa simplemente se encargará de preguntarle al usuario el tipo de operación aritmética que queremos llevar a cabo y luego la realizará con los datos dados que le proporcionemos. Este programa ya fue resuelto anteriormente, pero en este caso será implementado agregando el uso de las funciones.

En el diagrama de flujo indicamos a la función por medio de un rectángulo que tiene dos franjas a los lados. Cuando lo veamos, sabemos que tenemos que ir a la función y ejecutar su código. Veamos cómo quedaría el diagrama de flujo de la aplicación.



Las funciones no pueden ser declaradas adentro de otra función. Esto nos lleva a un error de sintaxis y de lógica que debe ser corregido lo antes posible. Es importante mantener las funciones ordenadas y tener cuidado de no desbalancear los **{ }** cuando se creen nuevas funciones. Hay que recordar que deben ir adentro de una clase.



**Figura 2.** Aquí podemos ver el diagrama de flujo y encontrar los lugares donde se invocan a las funciones.

Vemos que nuestro diagrama es muy sencillo y también podemos localizar inmediatamente el lugar donde invitamos a las funciones. Cada una de las operaciones tendrá su propia función. En este momento no programaremos todo, iremos haciendo crecer el programa poco a poco.

Empecemos por construir el programa, dejando la administración de la lógica a **Main()**, tal y como aparece en el diagrama de flujo.

```

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

namespace AplicacionBase
  
```

```
{  
    class Program  
    {  
        // Esta es la funcion principal del programa  
        // Aqui inicia la aplicacion  
        static void Main(string[] args)  
        {  
            // Variables necesarias  
            int opcion = 0;  
            string valor = “”;  
  
            // Mostramos el menu  
            Console.WriteLine(“1-Suma”);  
            Console.WriteLine(“2-Resta”);  
            Console.WriteLine(“3-Multiplicacion”);  
            Console.WriteLine(“4-Division”);  
  
            // Pedimos la opcion  
            Console.WriteLine(“Cual es tu opcion:”);  
            valor = Console.ReadLine();  
            opcion = Convert.ToInt32(valor);  
  
            // Checamos por la suma  
            if (opcion == 1)  
            {  
            }  
  
            // Checamos por la resta  
            if (opcion == 2)  
            {  
            }  
  
            // Checamos por la multiplicacion  
            if (opcion == 3)  
            {  
            }  
  
            // Checamos por la division
```

```

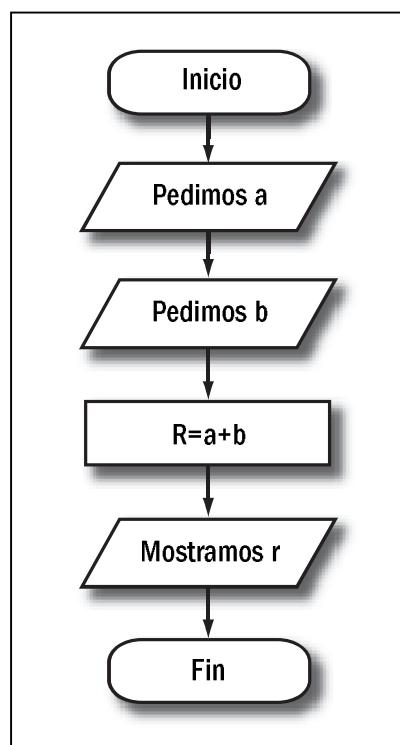
        if (opcion == 4)
        {
        }

    }
}

```

Como vemos en nuestra función **Main()** tenemos la lógica principal del programa. Hemos dejado en este momento los bloques de código de los **if** vacíos para ir colocando el código correspondiente según vallamos avanzando.

Empecemos a crear la función para la suma. Como esta función no va a recibir datos ni a regresar valores, todo se llevará a cabo en su interior. Será responsable de pedir los operandos, realizar la suma y mostrar el resultado. Entonces colocamos la declaración de la función. Esto lo hacemos después de la función **Main()**, hay que recordar tener cuidado con los **{ }**. También les debemos crear su diagrama de flujo y resolverlas de la forma usual. El diagrama de flujo es el siguiente.



**Figura 3.** Éste es el diagrama de flujo de la función, también debemos hacerlos.

Nuestra función queda de la siguiente forma:

```
static void Suma()
```

```

{
    // Variables necesarias
    float a=0;
    float b=0;
    float r=0;
    string numero="";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    a = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    b = Convert.ToSingle(numero);

    // Calculamos el resultado
    r = a + b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}",r);

}

```

Observamos que la función es **static**. Como la función no regresa ningún valor su tipo es **void**, el nombre de la función es **Suma**. Escogemos este nombre porque describe la actividad que lleva a cabo la función. Como no recibe ningún dato del programa principal no colocamos ningún parámetro y los paréntesis están vacíos. En el código tenemos todos los pasos para resolver la suma y ya los conocemos. Como podemos observar es posible declarar variables. Debemos saber que las variables que se declaran dentro de la función se conocen como **variables locales** y



Un error muy común cuando se empieza a programar las funciones es equivocarse al escribir el nombre de la función de forma diferente durante la invocación. Por ejemplo, si nuestra función se declara como **Suma()** cuando la invocamos no podemos usar **suma()**. La invocación debe usar el nombre de la función exactamente como fue declarado.

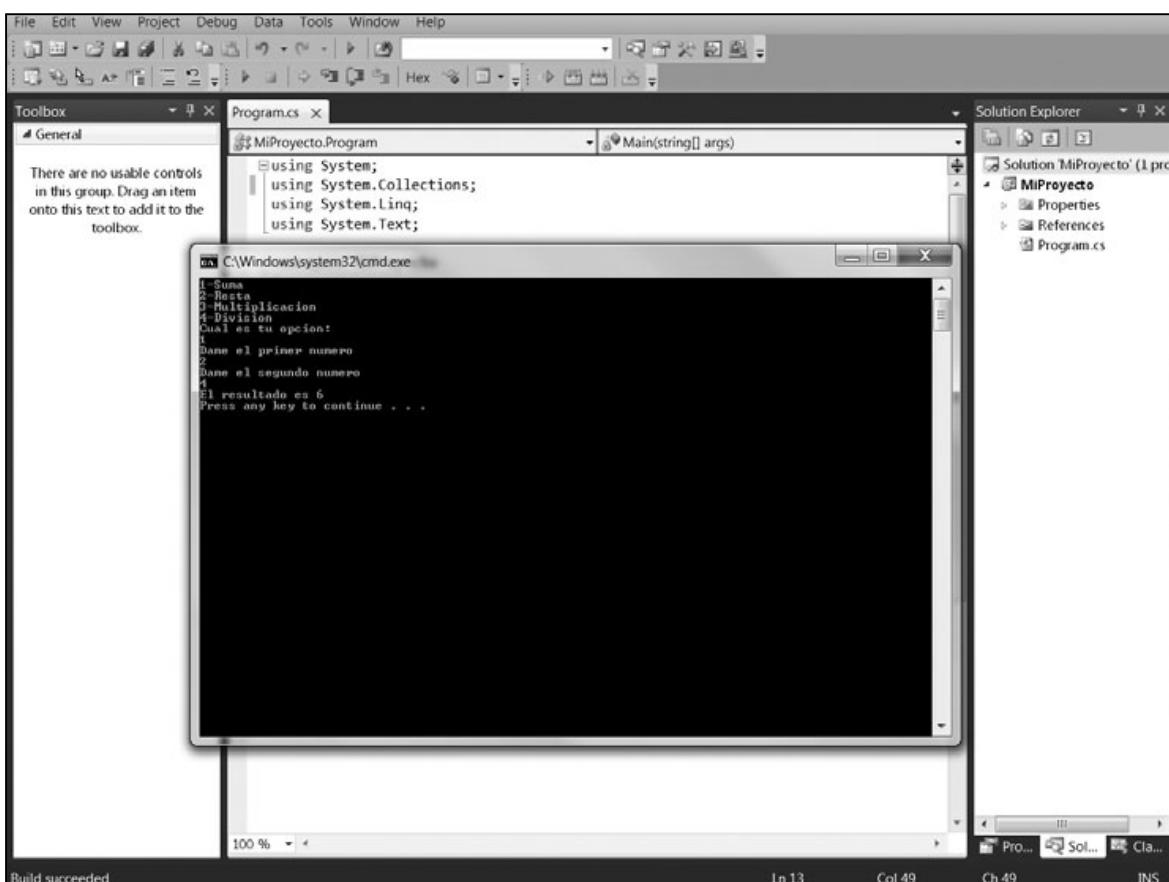
solamente serán conocidas adentro de esa función, de esta forma solo podrán ser invocadas dentro de la función correspondiente.

Con esto la función está lista, sin embargo, si ejecutamos el programa nada diferente sucede. Esto se debe a que no hemos invocado la función. Cuando invocamos a la función su código se ejecuta. Para invocar a una función que solamente ejecuta código simplemente colocamos su nombre seguido de () y punto y coma en el lugar donde queremos que se ejecute. En nuestro caso sería el bloque de código del if para la suma, de la siguiente manera:

```
// Checamos por la suma

if(opcion==1)
{
    Suma();
}
```

Ahora si ejecutamos el programa y seleccionamos la opción de suma, veremos que efectivamente se ejecuta el código propio de la función.



**Figura 4.** En la ejecución observamos cómo se está ejecutando el código de la función.

## Funciones que regresan un valor

Nuestro siguiente tipo de función puede regresar un valor. Esto significa que cuando la función es invocada va a llevar a cabo la ejecución de su código. El código va a calcular un valor de alguna manera y este valor calculado por la función será regresado a quien haya invocado a la función. La invocación puede haber sido hecha por la función **Main()** o algún otra función.

Como la función va a regresar un valor, necesitamos indicar su **tipo**. El tipo va a depender del valor devuelto. Si el valor es un entero, entonces el tipo de la función es **int**. Si el valor es un flotante, la función tendrá tipo **float** y así sucesivamente. La función puede regresar cualquiera de los tipos definidos en el lenguaje y también tipos definidos por el programador y objetos de diferentes clases.

La función va a utilizar un comando especial para regresar el valor, este comando se conoce como **return** y se usa de la siguiente manera:

```
return variable;
```

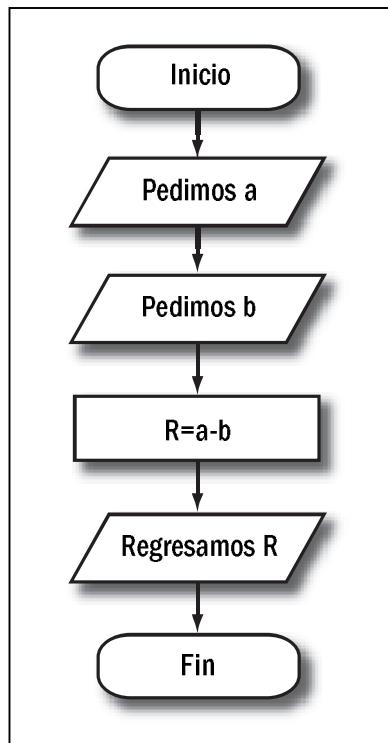
En cuanto la ejecución del programa encuentra un **return**, la función es finalizada aun si no ha llegado a su fin. Al mismo tiempo que finaliza la función el valor colocado después del **return** es regresado a quien hizo la invocación. El valor puede ser colocado con una variable o explícitamente con un valor en particular. No hay que olvidar colocar el punto y coma al finalizar la sentencia. La finalización de la función se lleva a cabo aunque tengamos código escrito después del **return**.

Como la función regresa un valor, del lado del invocador necesitamos tener alguien que pueda recibir el valor regresado. Generalmente usaremos una variable, pero en algunos casos puede ser una expresión que será evaluada con el valor regresado por la función. Supongamos que nuestra función regresa un valor entero. Entonces podemos tener un código como el siguiente.

```
int n;  
  
n=función();
```

De esta forma el valor regresado por la función queda guardado en la variable **n** y podemos hacer uso de él. El código lo entendemos de la siguiente forma. Tenemos una variable entera **n**. Luego tenemos una asignación para **n**. Si recordamos, la asignación siempre se lleva a cabo de derecha a izquierda. Se evalúa la expresión y la función se ejecuta y calcula un valor, el cual es regresado por medio de **return**. Este valor se considera como la evaluación de la expresión y es asignado a **n**. A partir de este momento podemos usar el valor según lo necesitemos.

Ahora ya podemos empezar a usar este tipo de funciones en nuestra aplicación. Para la operación de la resta haremos uso de ella. El código en la parte del **Main()** no será tan sencillo, ya que no solamente necesitamos una invocación, debemos de recibir un valor y hacer algo con él. De esta forma en el **if** para la resta, obtendremos el valor regresado por la función y luego lo presentaremos al usuario. La función **Resta()** tiene el siguiente diagrama de flujo.



**Figura 5.** Podemos observar la forma cómo se crea el diagrama de esta función, no hay que olvidar el retorno del valor.

El código de la función queda de la siguiente manera:

```

static float Resta()

{
  
```



Es importante que la variable que recibe el valor de regreso de la función tenga el mismo tipo que la función. Si no es así podemos tener desde problemas de compilación hasta perdida de precisión en valores numéricos. En algunos casos si los tipos son diferentes será necesario llevar a cabo conversiones entre los tipos.

```

    // Variables necesarias
    float a=0;
    float b=0;
    float r=0;
    string numero="";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    a = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    b = Convert.ToSingle(numero);

    // Calculamos el resultado
    r = a - b;

    // Retornamos el resultado
    return r;
}

```

Podemos observar que la función **Resta()** es de tipo **float** y al final hemos colocado el **return** indicando la variable cuyo valor deseamos regresar, en nuestro caso **r**. Pero no solamente debemos adicionar la función, también es necesario colocar nuevo código en el **if** que corresponde a la resta.

```

    // Checamos por la resta
    if(opcion==2)
    {

```



No es estrictamente necesario que el **return** se ubique al final de la función, aunque generalmente lo encontraremos ahí. También es posible tener varios **return** escritos, por ejemplo uno en cada caso de una escalera de **if**. Lo que no debemos olvidar es que cuando la ejecución encuentra el **return**, la función finaliza.

```

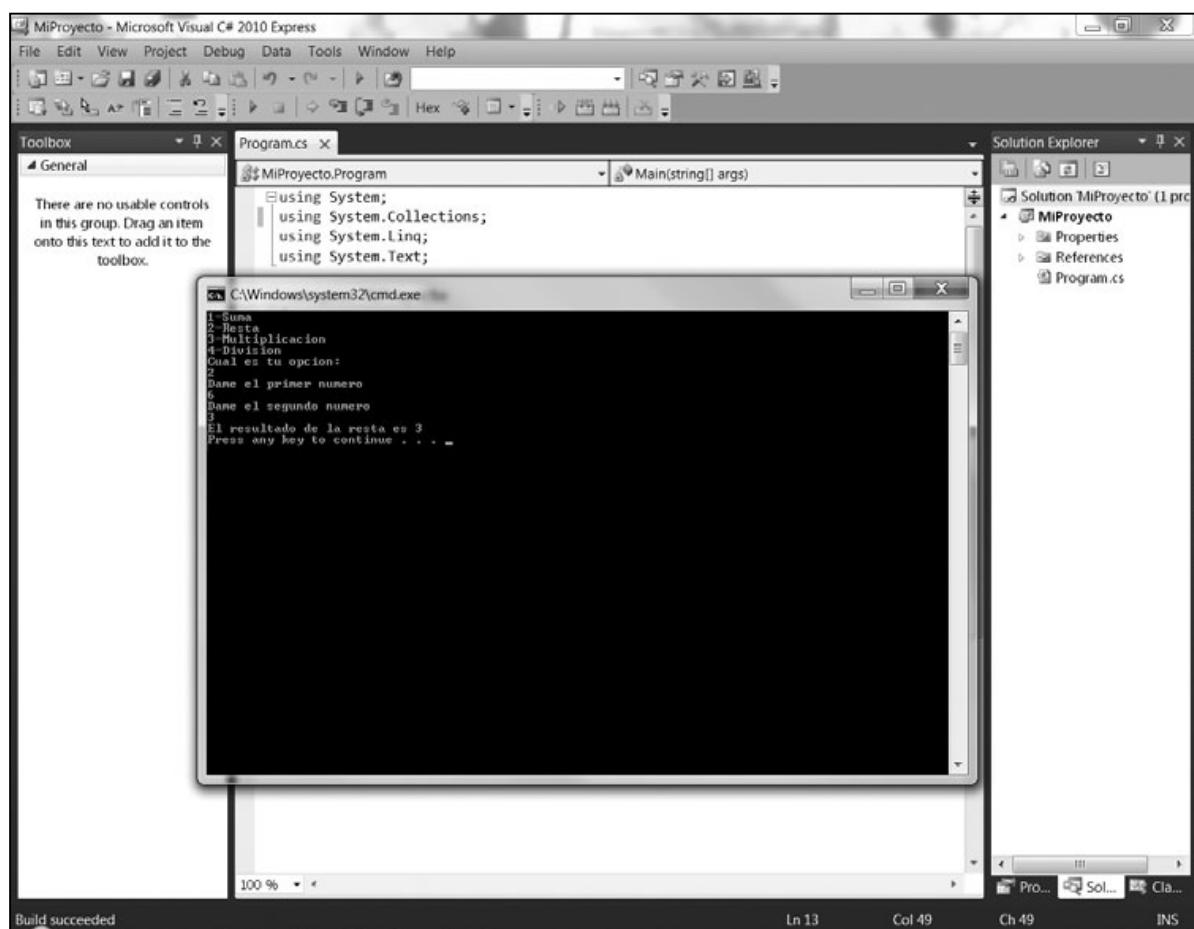
    // Variable para nuestro resultado
    float resultado=0;

    // Invocamos y obtenemos el resultado
    resultado=Resta();

    // Mostramos el resultado
    Console.WriteLine("El resultado de la resta es
                      {0}",resultado);
}

```

Ejecutemos el programa. Podemos observar que la función actúa como se esperaba.



**Figura 6.** Vemos cómo el valor regresado por la función es desplegado en quien lo invocó, en nuestro caso *Main()*.

## Funciones que reciben valores

Hasta el momento las funciones que hemos utilizado piden directamente al usuario los valores que necesitan para trabajar. Sin embargo, las funciones también

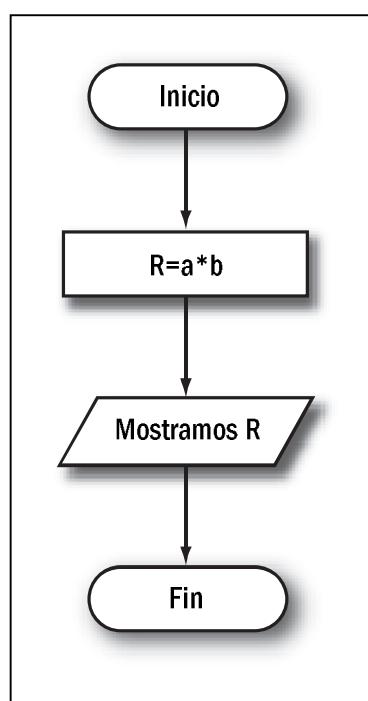
pueden recibir valores en el momento que son invocadas. De esta forma trabajarán con los valores pasados por el programa en lugar de pedirlos al usuario. Estos valores son conocidos como parámetros. Los parámetros pueden ser de cualquier tipo, ya sea de los tipos nativos de C# como entero, flotante, cadena o de tipos definidos por el programador como clases y estructuras.

Los parámetros deben llevar su tipo y su nombre. El nombre nos permite acceder a los datos que contiene y de hecho van a trabajar como si fueran variables locales a la función. Adentro de la función los usamos como variables normales. Los parámetros se definen en la declaración de la función. Adentro de los paréntesis de la función los listamos. La forma de listarlos es colocando primero en tipo seguido del nombre. Si tenemos más de un parámetro para la función, entonces debemos separarlos por medio de comas.

La invocación de la función es muy sencilla, simplemente debemos colocar el nombre de la función y, entre los paréntesis, los datos que vamos a enviar como parámetros. Los datos pueden ser situados por medio de variables o un valor colocado explícitamente.

Podemos hacer ahora la función que se encargará de la multiplicación. Esta función recibirá los operandos desde **Main()** por medio de los parámetros, realizará el cálculo y lo mostrará al usuario. Como la función **Main()** manda la información, entonces será responsabilidad de ella pedirlos a los usuarios.

El diagrama de flujo de la función **Multiplicacion()** se muestra a continuación.



**Figura 7.** La función **Multiplicacion()**  
es más sencilla, pues no pide los datos directamente al usuario.

El código de la función es el siguiente:

```
static void Multiplicacion(float a, float b)
{
    // Variables necesarias
    float r;

    // Calculamos el valor
    r=a*b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}",r);
}
```

Como podemos observar la función va a tener dos parámetros. Los parámetros son de tipo flotante y se llaman a y b. No olvidemos que están separados por una coma. Los parámetros a y b funcionarán como variables y su contenido serán los valores pasados en la invocación.

El otro lugar donde debemos de adicionar código es la función **Main()**, en este caso el **if** para la multiplicación.

```
// Checamos por la multiplicacion
if(opcion==3)
{
    // Variables necesarias
    float n1=0;
    float n2=0;
    string numero="";
}
```



Durante la invocación de la función los parámetros deben colocarse en el mismo orden en el cual fueron declarados. No hacerlo así puede llevarnos a que los parámetros reciban datos equivocados. Éste es uno de los puntos que debemos cuidar. También es bueno nombrar a los parámetros de tal manera que su nombre nos recuerde la información que va a colocársele.

```

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    n1 = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    n2 = Convert.ToSingle(numero);

    // Invocamos a la funcion
    Multiplicacion(n1, n2);

}

```

En el interior del **if** creamos dos variables flotantes y le pedimos al usuario los valores con los que vamos a trabajar. Estos valores quedan guardados adentro de las variables **n1** y **n2**. Veamos ahora la invocación de la función. Como la función fue definida para recibir parámetros entonces debemos de pasar datos en su invocación. Los parámetros son colocados entre los paréntesis. De la forma como lo tenemos el parámetro **a** recibirá una copia del valor contenido en **n1** y el parámetro **b** recibirá una copia del valor contenido en **n2**. Ya con esto la función puede llevar a cabo su trabajo. Las variables usadas en la invocación no necesitan llamarse igual que los parámetros declarados en la función. Si lo necesitáramos la invocación podría llevar un valor colocado explícitamente, por ejemplo:

```

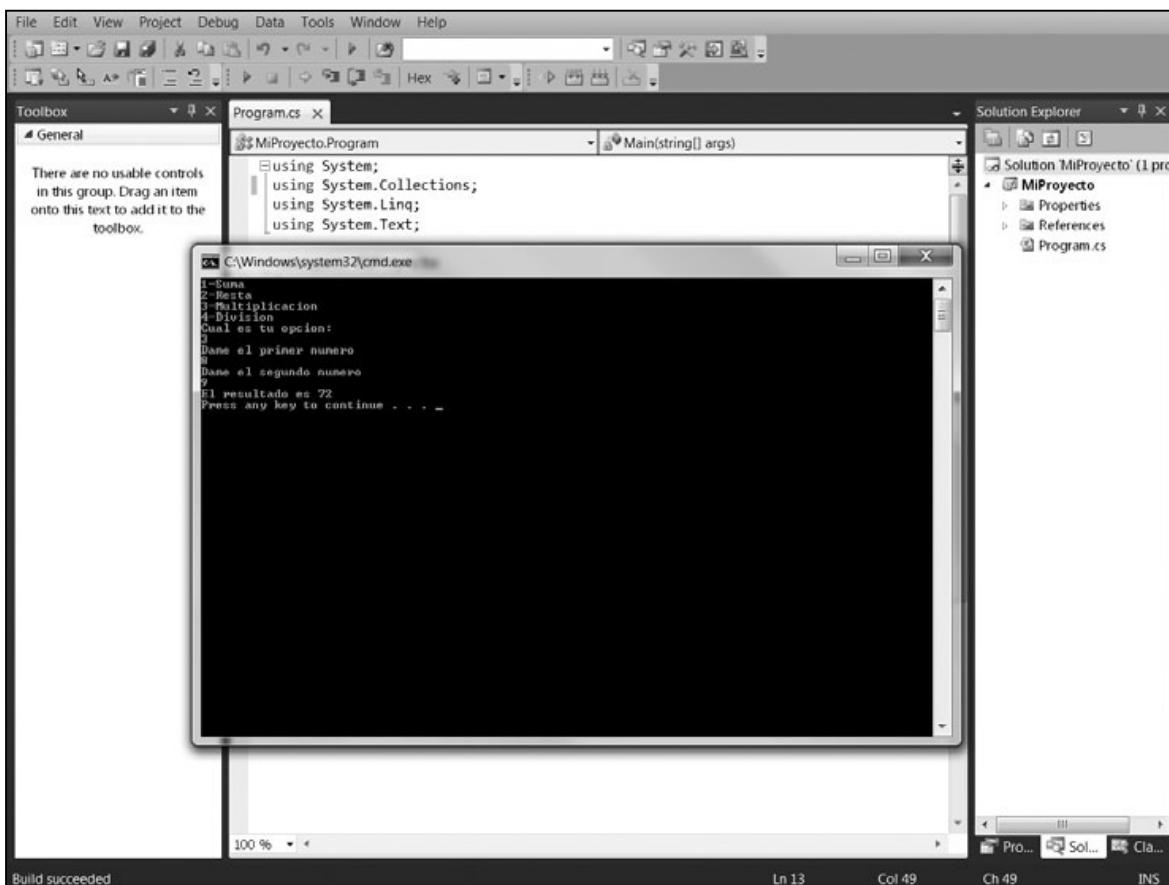
// Invocamos a la funcion
Multiplicacion(n1, 4.0f);

```

Cuando la función se invoca el parámetro **a** recibe una copia del valor contenido en **n1** y el parámetro **b** recibe el valor de **4**. Veamos cómo funciona.

### III

Los datos que enviamos deben de ser compatibles con los tipos que la función espera recibir como parámetros y de ser posible del mismo tipo. Podemos hacer uso de conversiones antes de enviar el dato hacia la función. Con datos no compatibles tendremos problemas de compilación, en datos compatibles, pero no del mismo tipo, tendremos problemas de perdida de precisión.



**Figura 8.** Vemos que la función imprime el resultado correcto, lo que nos muestra que está recibiendo la información pasada en los parámetros.

## Funciones que reciben parámetros y regresan un valor

Ya hemos visto tres tipos diferentes de funciones, y seguramente ya hemos comprendido cómo funcionan y podemos fácilmente imaginar lo que hará este tipo de función. Esta función va a recibir de quien la invoque información. La información es pasada por medio de parámetros. La función lleva a cabo alguna acción o cálculo y obtiene un valor que va a ser regresado. El valor regresado es recibido en el lugar donde se invocó a la función y se puede trabajar con él.

Los parámetros serán usados como variables y los declaramos adentro de los paréntesis de la función. Tenemos que indicar el tipo que va a recibir seguido de su nombre, si tenemos más de un parámetro, entonces debemos de separarlos por comas. Como la función regresa un valor, es necesario indicar su tipo en la declaración. La variable que va a recibir el valor retornado deberá tener de preferencia el mismo tipo que la función o cuando menos un tipo que sea compatible o posible de convertir. Como siempre haremos uso de **return** para poder regresar el valor calculado. De hecho este tipo de función utiliza todas las partes de las que consiste una función. En este momento debemos programar la función que lleva a cabo la división. La función recibirá dos valores flotantes, verificará que no llevemos a cabo la división entre cero, realiza el cálculo y regresa el valor.

Si pasamos el diagrama de flujo a código, encontraremos algo interesante:

```

static float Division(float a, float b)
{
    // Variables necesarias
    float r=0;

    // Verificamos por division entre cero
    if(b==0)
    {
        Console.WriteLine("No es posible dividir
                           entre cero");
        return 0.0f;
    }
    else

    {
        r=a/b;
        return r;
    }
}

```

Adentro de la función tenemos dos **return**. ¿Esto significa que vamos a regresar dos valores? En realidad no. Únicamente podemos regresar un valor con este tipo de funciones. Lo que sucede es que cada uno de nuestros **return** se encuentra en una ruta de ejecución diferente. Observemos que una se lleva a cabo cuando el **if** se cumple y el otro se ejecuta cuando el **if** no se cumple. Para la función, solamente uno de ellos podrá ser ejecutado. También podemos notar que uno de los **return** está regresando un valor colocado explícitamente, en este caso **0.0**. Si colocamos valores de retorno explícitos, éstos deben de ser del mismo tipo que el tipo de la función. Éste es un concepto importante que nos da flexibilidad en nuestro código. Podemos regresar bajo diferentes condiciones, siempre y cuando solamente tengamos un **return** por ruta de ejecución. Si nuestra función debe regresar valores y tenemos varias rutas de ejecución, por ejemplo, varios **if** o **if** anidados, entonces debemos de colocar un **return** en cada ruta. No es posible dejar una ruta sin la posibilidad de regresar un valor. Debemos tomar en cuenta estas características para evitar problemas de lógica dentro de nuestro programa.

En el lado de la función **Main()** también debemos adicionar código. El código lo colocaremos en el bloque de código que corresponde al **if** de la división.

```
// Checamos por la division
if(opcion==4)
{
    // Variables necesarias
    float n1=0.0f;
    float n2=0.0f;
    float resultado=0.0f;
    string numero="";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    n1 = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    n2 = Convert.ToSingle(numero);

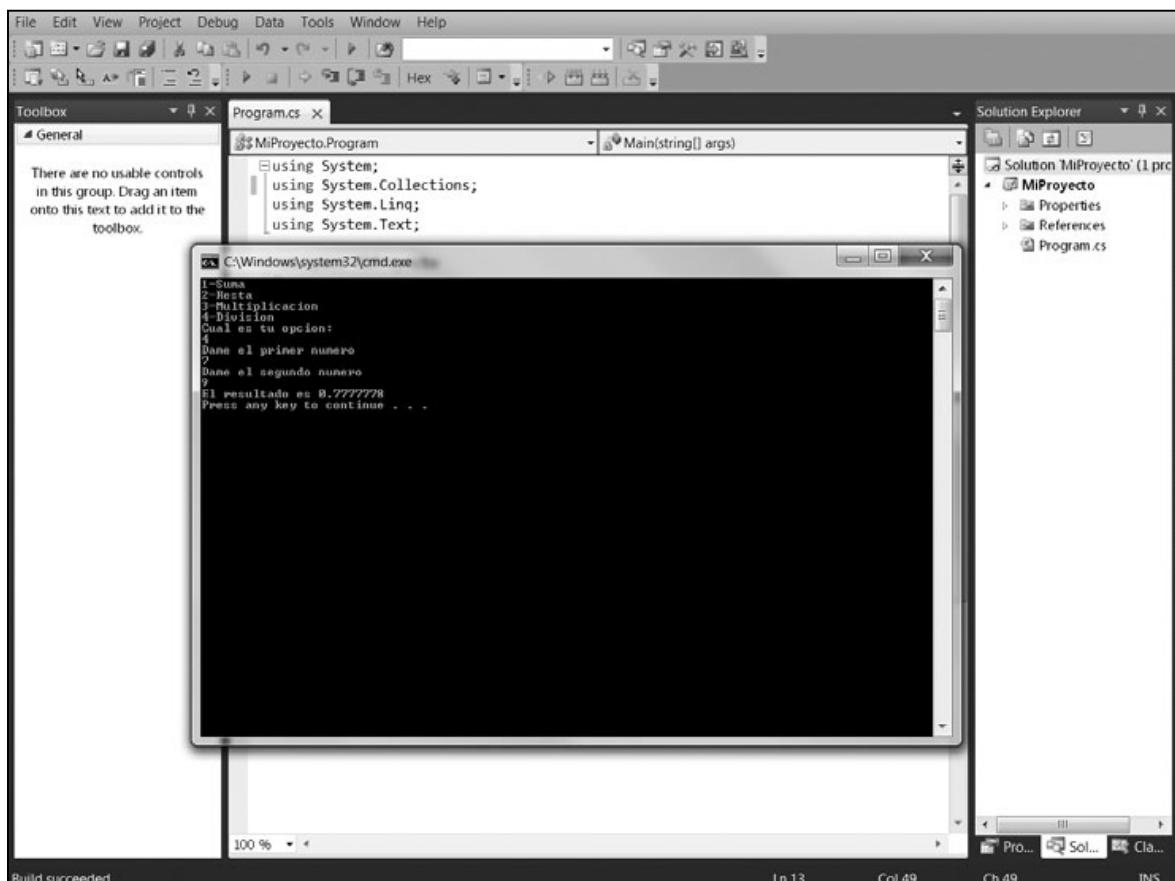
    // Invocamos a la funcion
    resultado=Division(n1, n2);

    // Mostramos el resultado
    Console.WriteLine("El resultado es
{0}",resultado);
}
```

Al igual que en casos anteriores, primero declaramos las variables necesarias y pedimos los valores. Luego invocamos a la función. En la invocación pasamos como parámetros a nuestras variables **n1** y **n2**. Se lleva a cabo la ejecución de la función y recibimos de regreso un valor. Este valor es asignado a la variable **resultado** y luego utilizado por **Main()** al mostrar el resultado.

### III

El ambiente de desarrollo .NET nos permite ver las funciones o métodos que tiene una clase en particular. Cuando se escribe el nombre del objeto de dicha clase seguido del operador punto nos aparece una lista de ellas. Esto lo podemos hacer con la clase **Console**. El investigar estas funciones nos permite aprender más sobre .NET y lo que podemos hacer con él.



**Figura 10.** La división se lleva a cabo y el valor de regreso es utilizado para mostrar el resultado.

Con esto ya tenemos el programa completo.

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

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {

            // Variables necesarias
            int opcion = 0;
            string valor = "";
        }
    }
}
```

```
// Mostramos el menu
Console.WriteLine("1-Suma");
Console.WriteLine("2-Resta");
Console.WriteLine("3-Multiplicacion");
Console.WriteLine("4-Division");

// Pedimos la opcion
Console.WriteLine("Cual es tu opcion:");
valor = Console.ReadLine();
opcion = Convert.ToInt32(valor);

// Checamos por la suma
if (opcion == 1)
{
    Suma();
}

// Checamos por la resta
if (opcion == 2)

{
    // Variable para nuestro resultado
    float resultado = 0;

    // Invocamos y obtenemos el resultado
    resultado = Resta();

    // Mostramos el resultado
    Console.WriteLine("El resultado de la resta es
{0}", resultado);
}

// Checamos por la multiplicacion
if (opcion == 3)
{

    // Variables necesarias
    float n1 = 0;
    float n2 = 0;
```

```
string numero = "";

// Pedimos los valores
Console.WriteLine("Dame el primer numero");
numero = Console.ReadLine();
n1 = Convert.ToSingle(numero);

Console.WriteLine("Dame el segundo numero");
numero = Console.ReadLine();
n2 = Convert.ToSingle(numero);

// Invocamos a la funcion
Multiplicacion(n1, n2);
}

// Checamos por la division
if (opcion == 4)
{
    // Variables necesarias
    float n1 = 0.0f;
    float n2 = 0.0f;

    float resultado = 0.0f;
    string numero = "";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    n1 = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    n2 = Convert.ToSingle(numero);

    // Invocamos a la funcion
    resultado = Division(n1, n2);

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}",
        resultado);
```

```
}

} // Cierre de Main

static void Suma()
{
    // Variables necesarias
    float a = 0;
    float b = 0;
    float r = 0;
    string numero = "";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
    numero = Console.ReadLine();
    a = Convert.ToSingle(numero);

    Console.WriteLine("Dame el segundo numero");
    numero = Console.ReadLine();
    b = Convert.ToSingle(numero);

    // Calculamos el resultado
    r = a + b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}", r);
}

static float Resta()
{
    // Variables necesarias
    float a = 0;
    float b = 0;
    float r = 0;
    string numero = "";

    // Pedimos los valores
    Console.WriteLine("Dame el primer numero");
```

```
numero = Console.ReadLine();
a = Convert.ToSingle(numero);

Console.WriteLine("Dame el segundo numero");
numero = Console.ReadLine();
b = Convert.ToSingle(numero);

// Calculamos el resultado
r = a - b;

// Retornamos el resultado
return r;
}

static void Multiplicacion(float a, float b)
{

    // Variables necesarias
    float r = 0;

    // Calculamos el valor

    r = a * b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}", r);
}

static float Division(float a, float b)
{

    // Variables necesarias
    float r = 0;

    // Verificamos por division entre cero
    if (b == 0)
    {
        Console.WriteLine("No es posible dividir entre cero");
        return 0.0f;
    }
}
```

```

    else
    {
        r = a / b;
        return r;
    }
}

```

## Optimizar con funciones

Ya conocemos cómo utilizar los diferentes tipos de funciones y ahora aprenderemos cómo podemos optimizar un programa haciendo uso de ellas. En el programa anterior podemos pensar que hemos utilizado todas las funciones que son necesarias. Pero nosotros sabemos que una buena oportunidad para hacer uso de las funciones es cuando tenemos código que se repite constantemente. Si observamos nuestro programa, podemos observar que esto ocurre.

En nuestro programa tenemos algo que sucede ocho veces. Se piden los valores a utilizar al usuario en varios lugares del programa. Cada vez que lo pedimos usamos tres líneas de código y la forma cómo se pide es muy similar entre ellas.

Cuando tenemos estos casos hay que llevar a cabo un poco de análisis. Lo primero que nos preguntamos es: ¿qué función lleva a cabo este código? La respuesta es pedir un valor flotante. Luego nos tenemos que preguntar si la función necesita de algún dato o datos para poder trabajar.

En una primera vista parece que no, ya que el usuario es el que va a introducir el dato. Sin embargo, si observamos mejor veremos que en realidad sí necesitamos un dato. La petición se lleva a cabo por medio de un mensaje y este mensaje puede ser diferente en cada caso. Podemos tener como dato el mensaje a mostrar por el usuario. Este va a entrar como parámetro en nuestra función. La última pregunta que nos debemos hacer es: ¿necesita la función regresar algo? En este caso es muy fácil deducir que sí. Lo que necesitamos regresar es el valor que el usuario nos ha dado.

Ya con esta información podemos construir nuestra función. Si la función fuera más complicada necesitaríamos hacer el diagrama de flujo correspondiente. Afortunadamente nuestra función es muy sencilla y queda de la siguiente manera:

```

static float PideFloatante(string mensaje)

{
    // Variables necesarias
    float numero=0.0f;
    string valor="";

    // Mostramos el mensaje
    Console.WriteLine(mensaje);

    // Obtenemos el valor
    valor = Console.ReadLine();
    numero = Convert.ToSingle(valor);

    // Regresamos el dato
    return numero;
}

```

Empecemos a ver cómo trabaja esta función que hemos creado.

El nombre es **PideFloatante** ya que como sabemos el nombre de la función tiene que indicar una referencia al tipo de trabajo que realiza. Como nuestra función regresa un valor de tipo flotante, entonces colocamos como su tipo a **float**.

La función tiene un único parámetro que es el mensaje que deseamos mostrar al usuario en la petición. Como es un mensaje de texto entonces el tipo que más conviene usar es **string**. A nuestro parámetro le nombramos **mensaje**.

En el interior de la función declaramos dos variables. Una es el número que vamos a recibir del usuario y la otra variable es la cadena que usamos con **ReadLine()**. Luego mostramos el mensaje de petición. De la forma habitual obtenemos el valor dado por el usuario. Ya para finalizar simplemente regresamos el número.

Para invocar a esta función, lo haremos de la siguiente manera:

```
n1 = PideFloatante("Dame el primer numero");
```

Ahora vemos que en lugar de utilizar las tres líneas de código por petición, simplemente hacemos una invocación a nuestra función. Pero este no es el único cambio que tenemos que hacer. En el programa original habíamos colocado variables de trabajo para apoyarnos en la petición del valor, pero ya no son necesarias.

Si cambiamos el programa para hacer uso de nuestra nueva función quedará como en el ejemplo que vemos a continuación:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {

            // Variables necesarias
            int opcion = 0;
            string valor = "";

            // Mostramos el menu
            Console.WriteLine("1-Suma");
            Console.WriteLine("2-Resta");
            Console.WriteLine("3-Multiplicacion");
            Console.WriteLine("4-Division");

            // Pedimos la opcion
            Console.WriteLine("Cual es tu opcion:");
            valor = Console.ReadLine();
            opcion = Convert.ToInt32(valor);

            // Checamos por la suma
            if (opcion == 1)
            {
                Suma();
            }

            // Checamos por la resta
            if (opcion == 2)
            {
                // Variable para nuestro resultado
                float resultado = 0;

                // Invocamos y obtenemos el resultado
            }
        }
    }
}
```

```
resultado = Resta();

// Mostramos el resultado
Console.WriteLine("El resultado de la resta es
{0}", resultado);

}

// Checamos por la multiplicacion
if (opcion == 3)
{
    // Variables necesarias
    float n1 = 0;
    float n2 = 0;

    // Pedimos los valores
    n1 = PideFloatante("Dame el primer numero");
    n2 = PideFloatante("Dame el segundo numero");

    // Invocamos a la funcion

    Multiplicacion(n1, n2);
}

// Checamos por la division
if (opcion == 4)
{
    // Variables necesarias
    float n1 = 0.0f;
    float n2 = 0.0f;
    float resultado = 0.0f;

    // Pedimos los valores
    n1 = PideFloatante("Dame el primer numero");
    n2 = PideFloatante("Dame el segundo numero");

    // Invocamos a la funcion
    resultado = Division(n1, n2);

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}",
```

```
        resultado);
    }

} // Cierre de Main

static void Suma()
{
    // Variables necesarias
    float a = 0;
    float b = 0;
    float r = 0;

    // Pedimos los valores
    a = PideFlotante("Dame el primer numero");
    b = PideFlotante("Dame el segundo numero");

    // Calculamos el resultado
    r = a + b;

    // Mostramos el resultado

    Console.WriteLine("El resultado es {0}", r);
}

static float Resta()
{
    // Variables necesarias
    float a = 0;
    float b = 0;
    float r = 0;

    // Pedimos los valores
    a = PideFlotante("Dame el primer numero");
    b = PideFlotante("Dame el segundo numero");

    // Calculamos el resultado
    r = a - b;

    // Retornamos el resultado
    return r;
```

```
}

static void Multiplicacion(float a, float b)
{

    // Variables necesarias
    float r = 0;

    // Calculamos el valor
    r = a * b;

    // Mostramos el resultado
    Console.WriteLine("El resultado es {0}", r);
}

static float Division(float a, float b)
{

    // Variables necesarias

    float r = 0;

    // Verificamos por division entre cero
    if (b == 0)
    {
        Console.WriteLine("No es posible dividir entre cero");
        return 0.0f;
    }
    else
    {
        r = a / b;
        return r;
    }
}

static float PideFlotante(string mensaje)
{
    // Variables necesarias
    float numero = 0.0f;
    string valor = "";
}
```

```

    // Mostramos el mensaje
    Console.WriteLine(mensaje);

    // Obtenemos el valor
    valor = Console.ReadLine();
    numero = Convert.ToSingle(valor);

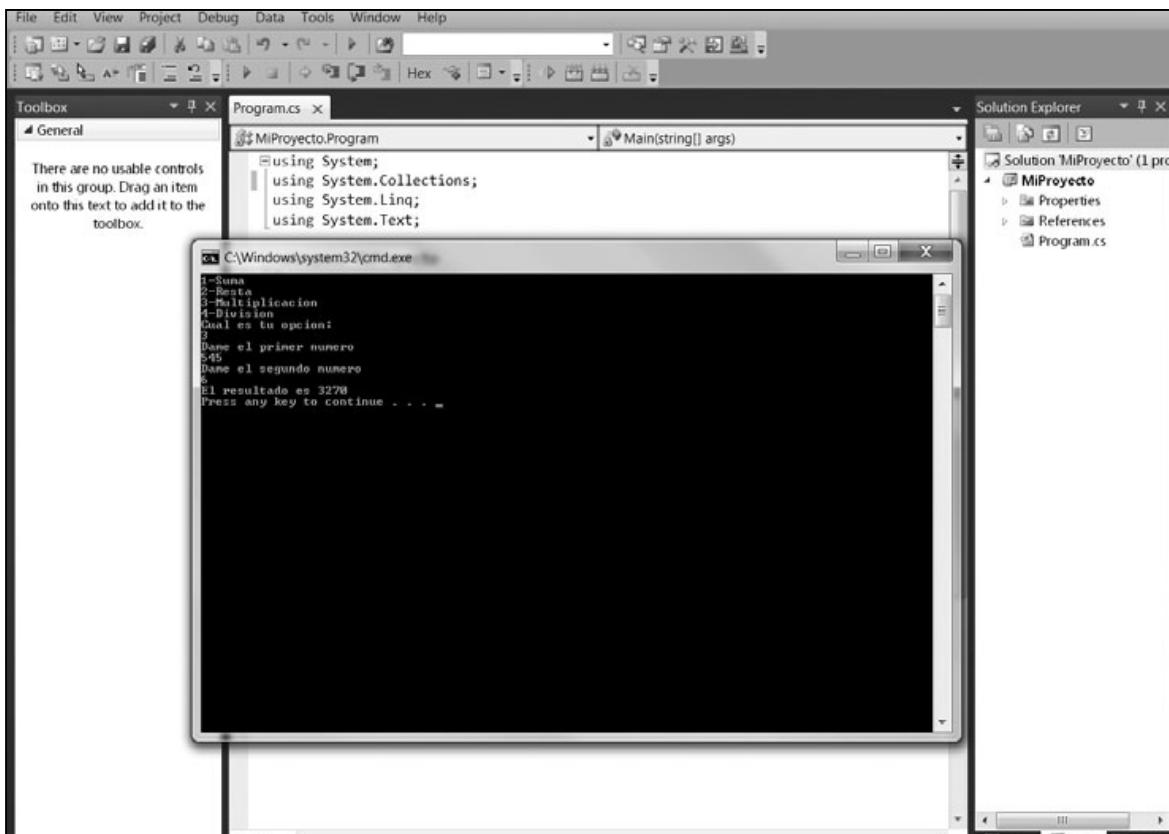
    // Regresamos el dato
    return numero;

}

}

```

Si observamos el código, podemos notar que es más fácil de leer y luce más ordenado. También hemos reducido los lugares donde podemos tener un error de sintaxis o lógica. Ejecutemos el programa para verificar si trabaja correctamente. Podemos seleccionar cualquier operación.



**Figura 11.** El programa se ejecuta correctamente con nuestra función para pedir flotantes al usuario.

Otro punto que debemos de tener en cuenta es que las funciones están invocando a nuestra función sin problema. Las funciones pueden invocar a las funciones.

## Paso por copia y paso por referencia

Tenemos que aprender un concepto importante sobre los parámetros y las funciones. La mejor forma de hacerlo es por medio de un experimento sencillo. Ya anteriormente hemos comentado dos conceptos. El primero es que las variables tienen **ámbito**, es decir que las partes del programa donde se pueden utilizar depende de donde fueron declaradas. El otro concepto es que cuando invocamos a una función y pasamos parámetros, una copia del valor del parámetro es pasada a la función. Empecemos con nuestro experimento. Vamos a crear un programa con el siguiente código.

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

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {
            // Variables necesarias
            int numero = 5;

            // Valor antes de funcion
            Console.WriteLine("Valor antes de funcion {0}", numero);

            // Invocamos funcion
            Cambiar(numero);

            // Valor despues de funcion
            Console.WriteLine("Valor despues de funcion {0}", numero);
        }
        static void Cambiar(int numero)
        {
            // Cambiamos el valor
        }
    }
}
```

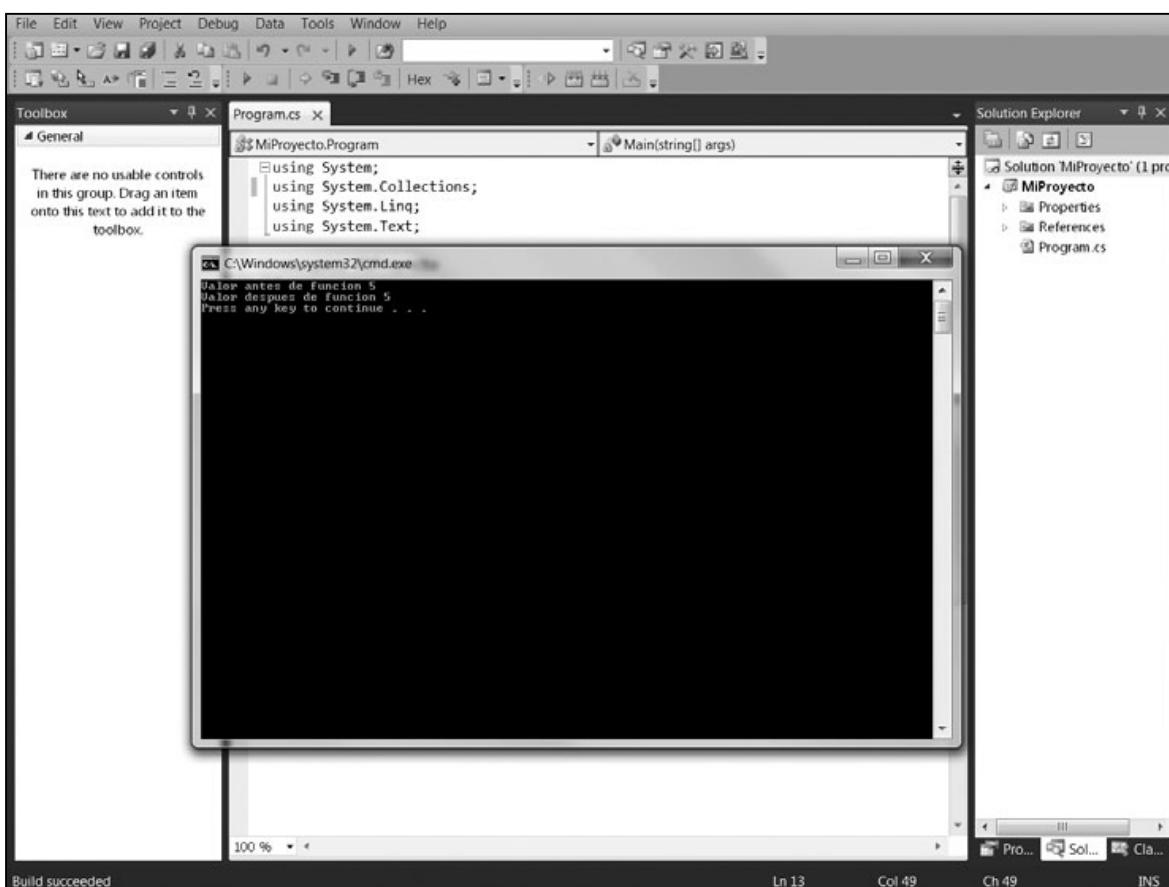
```

        numero = 17;
    }
}
}

```

En el programa tenemos dos funciones, la función **Main()** y la función **Cambiar()**. En **Main()** estamos declarando una variable y le asignamos el valor de **5**. Luego imprimimos un mensaje que muestra el valor de nuestra variable antes de invocar a la función. En seguida invocamos a la función y después de ésta imprimimos otro mensaje que nos muestra el valor de la variable.

Esto lo hacemos para poder verificar si la variable se ve afectada por la función. La función es muy sencilla, simplemente recibe un parámetro y le asigna un nuevo valor. Este programa podría parecer que presentaría un valor diferente antes y después de la función, pero ejecutémoslo y veamos qué ocurre.



**Figura 12.** El valor de la variable es el mismo antes y después de la ejecución de la función.

Al ejecutar notamos que el valor de la variable es igual antes y después de invocar a la función. Es decir que la función no alteró el valor de la variable, aunque parecía que lo iba a hacer. Veamos la razón de esto.

La primera razón de por qué esto sucede es que la variable fue declarada en la función **Main()**, es decir que solamente esta función la conoce y puede utilizarla directamente. **Main()** es el ámbito de la variable. Luego tenemos que cuando el parámetro se pasa de esta forma, en realidad no es la variable lo que se está pasando, sino una copia del valor de ella. Por eso cuando nuestra función trata de cambiar el valor, cambia a la copia y no a la variable original.

Cuando pasamos los parámetros de esta forma decimos que pasan por **copia**. Pero existe otra forma de pasar los parámetros y se conoce como **paso por referencia**. Cambiemos nuestro programa para que quede de la siguiente manera:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {
            // Variables necesarias
            int numero = 5;

            // Valor antes de funcion
            Console.WriteLine("Valor antes de funcion {0}",
                numero);

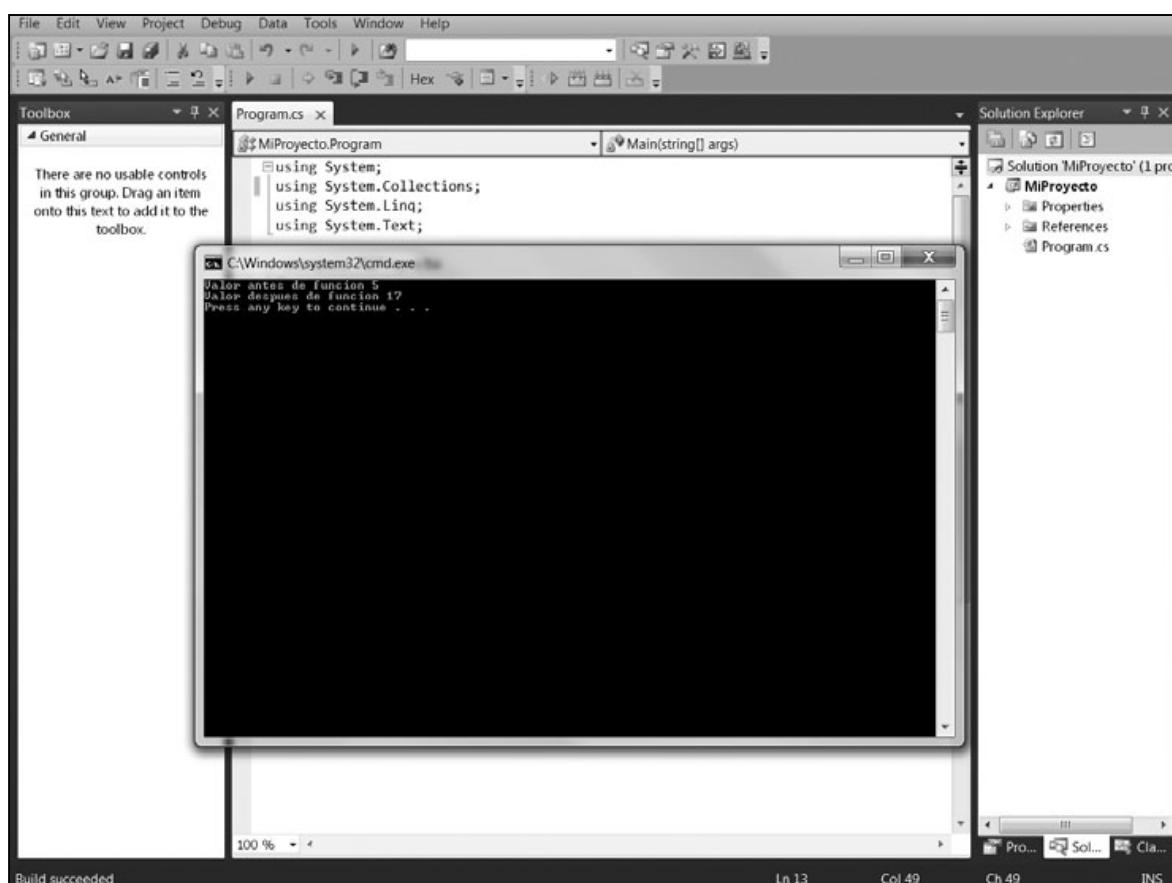
            // Invocamos funcion
            Cambiar(ref numero);

            // Valor despues de funcion
            Console.WriteLine("Valor despues de funcion {0}",
                numero);
        }

        static void Cambiar(ref int numero)
        {
            // Cambiamos el valor
            numero = 17;
        }
    }
}
```

}

Ejecutemos primero el programa y luego veamos qué ocurre.



**Figura 13.** En este caso el valor de la variable si es cambiado. Esto se debe a que se pasó la referencia a la variable.

En este caso estamos pasando el parámetro por referencia. Cuando esto sucede, en lugar de pasar una copia del valor, lo que pasamos es una referencia al lugar donde se encuentra la variable en memoria. De esta forma cualquier asignación a la variable se realiza sobre la variable original y su valor puede ser cambiado aunque haya sido definida en otro ámbito.

Para poder indicar que un parámetro necesita pasarse por referencia es necesario usar la instrucción **ref**, tanto en la invocación de la función como en su declaración. Podemos tener más de una variable pasada por referencia en una función.

El paso por referencia es muy útil, en especial para casos cuando es necesario que una función regrese más de un valor. Como sabemos, **return** solamente puede regresar un valor, pero al usar referencias podemos tener una función que regrese más de un valor por medio de la modificación de las variables pasadas.

Veamos un ejemplo de esto. Vamos a crear una función que permita intercambiar los valores de dos variables. Esto problema puede parecer sencillo, pero sin el uso de las referencias es imposible resolverlo con una sola función. A continuación veremos cómo queda nuestra programa.

```
using System;

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

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {

            // Variables necesarias
            int a = 5;
            int b = 3;

            // Valor antes de funcion
            Console.WriteLine("Valores antes de funcion a={0},
                b={1}", a,b);

            // Invocamos funcion
            Cambiar(ref a, ref b);

            // Valor despues de funcion

            Console.WriteLine("Valores despues de funcion a={0},
                b={1}", a, b);
        }

        static void Cambiar(ref int x, ref int y)
        {
            // Variable de trabajo
            int temp = 0;
```

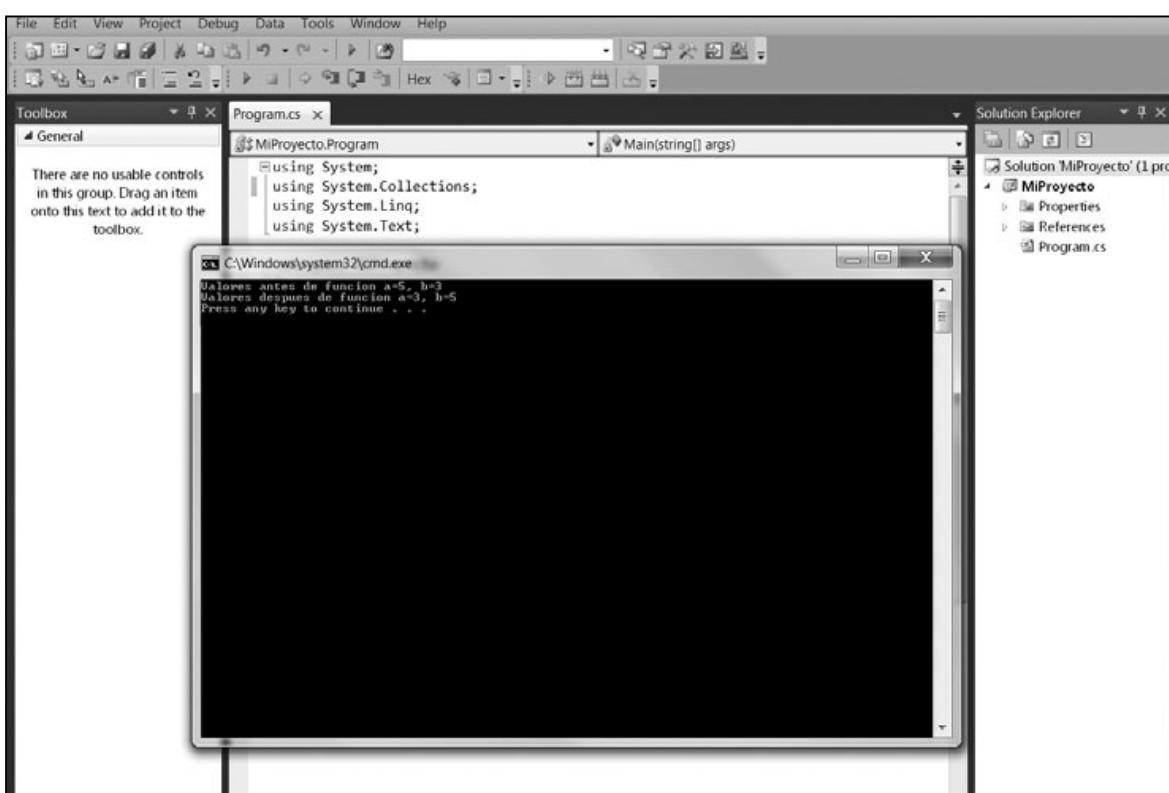
```

    // Cambiamos el valor
    temp = x;
    x = y;
    y = temp;
}

}
}

```

Si lo ejecutamos, obtenemos lo siguiente:



**Figura 14.** Los valores han sido intercambiados por la función.

Como vemos, hacemos uso de una variable de trabajo para ayudarnos en el intercambio de los valores. El paso por referencia nos permite modificar ambas variables.

## Uso de parámetros de default

Una de las características nuevas que encontramos en la última versión de C# es el uso de **parámetros de default** en las funciones. Con las funciones que hemos estado utilizando es necesario proveer a la función de todos los parámetros con los que fue declarada. Es decir, si la función tiene dos parámetros, cuando la invocamos debemos darle dos valores para que trabaje con ella.

Con los parámetros de default es posible colocar un valor predeterminado al parámetro, de tal forma que si en la invocación de la función no se da explícitamente el valor, entonces se usa el valor predeterminado. Esto puede resultar útil cuando tenemos parámetros que en la gran mayoría de los casos usan el mismo valor. De esta forma cuando invocamos a la función lo hacemos usando solamente los valores que cambien, lo cual nos permite escribir menos y solamente cuando es necesario colocamos el valor en el parámetro.

Esto lo podemos ver en el siguiente ejemplo. El cual tiene una función con parámetros de default y es invocada de forma tradicional y luego haciendo uso del parámetro de default. Cuando hace uso del parámetro de default impuesto usa el valor de **0.16**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cap5_5
{
    class Program
    {

        static void Main(string[] args)
        {
            double costo = 50.0;
            double imp = 0.0;
            double total = 0.0;

            // Hacemos uso de la función de manera tradicional

            imp = CalculaImpuesto(costo, 0.25);
            total = costo + imp;

            // Imprimimos resultado
            Console.WriteLine("El total es ${0}", total);

            // Hacemos uso de la función con parámetro de default
            // Hay que notar que solamente pasamos un parámetro,
            el otro usa

            // el valor predeterminado
    }
}
```

```
    imp = CalculaImpuesto(costo);
    total = costo + imp;

    // Imprimimos resultado
    Console.WriteLine("El total es ${0}", total);

}

public static double CalculaImpuesto(double cantidad,
    double impuesto = 0.16f)
{
    double impuestoCalculado;

    impuestoCalculado = cantidad * impuesto;
    return impuestoCalculado;
}
}
```

Con esto hemos visto los principios y los usos fundamentales de las funciones. Las funciones son muy útiles y las continuaremos utilizando frecuentemente.

## RESUMEN

Las funciones nos permiten tener secciones especializadas de código en nuestra aplicación, reducir la cantidad de código necesario y reutilizar el código de manera eficiente. Las funciones constan de cinco partes y hay cuatro tipos de ellas. La función puede recibir información por medio de los parámetros y regresar información por medio de return. Es posible pasar los parámetros por copia o por referencia.



## ACTIVIDADES

### TEST DE AUTOEVALUACIÓN

**1** ¿Qué son las funciones?

**2** ¿Qué es invocar una función?

**3** ¿Cuáles son las partes de las funciones?

**4** ¿Para qué nos sirve el modificador static?

**5** ¿Cuáles son los cuatro tipos de funciones?

**6** Cuando la función no regresa un valor,  
¿cuál es su tipo?

**7** ¿Qué tipos de valores pueden regresar las  
funciones?

**8** ¿Para qué sirve return?

**9** ¿Cómo se colocan los parámetros?

**10** ¿Cómo podemos usar las funciones para  
optimizar nuestro programa?

**11** ¿Qué es el paso por copia?

**12** ¿Qué es el paso por referencia?

### EJERCICIOS PRÁCTICOS

**1** Hacer una función para transformar de  
grados a radianes.

**2** Hacer una función para transformar de  
grados centígrados a grados Fahrenheit.

**3** Hacer una función que calcule el  
perímetro de cualquier polígono regular  
dando el número de lados y sus  
dimensiones.

**4** Hacer una función que calcule el factorial  
de un número.

**5** Hacer una función que dado un número  
nos regrese una cadena donde se  
encuentre escrito en palabras.

# Los arreglos

En los capítulos anteriores hemos utilizado las variables. Colocamos información en éstas para luego procesarlas. Nuestros programas pueden tener muchas variables, en especial cuando la cantidad de información a guardar es mucha. Los **arreglos** nos brindan una forma de poder administrar la información fácilmente, y son especialmente útiles cuando necesitamos muchas variables y el trabajo que se realiza sobre ellas es el mismo.

<b>Los arreglos</b>	<b>188</b>
Declaración de los arreglos de una dimensión	189
Asignación y uso de valores	191
Arreglos de dos dimensiones	196
Arreglos de tipo jagged	205
Los arreglos como parámetros a funciones	212
<b>Resumen</b>	<b>215</b>
<b>Actividades</b>	<b>216</b>

# LOS ARREGLOS

Pensemos en un problema donde los arreglos nos pueden ser útiles. En el **Capítulo 4** creamos un programa donde se calculaba el promedio de un grupo de alumnos. En él usamos un ciclo **for**, mediante el cual se pedía la calificación constantemente. Para la calificación siempre se usaba la misma variable. Pensemos ahora, que nos encontramos con la necesidad de que nuestro programa no solamente muestre el promedio, sino también la calificación más alta y más baja del salón. Estos cálculos requieren ser utilizados en diferentes partes del programa, por lo que no es posible colocar todo adentro del ciclo **for** con el que fue diseñado en un principio.

El problema al que nos estamos enfrentando es que, como sólo hemos hecho uso de una variable para las calificaciones, cuando necesitemos procesar nuevamente la información desde nuestro software, ya no tendremos almacenado el valor de la calificación anterior en la variable, sólo tendremos la última calificación capturada. ¿Cómo podemos resolver esto? Una forma de hacerlo sería crear y utilizar muchas variables como: **calif1**, **calif2**, **calif3**..., etcétera.

Al principio esta idea puede parecer correcta, pero manejar muchas variables de forma independiente, luego de un tiempo puede tornarse engorroso, y también podremos encontrar un caso en el que no sabemos cuántas variables necesitaremos. Sin embargo, esto nos puede dar una idea sobre lo que necesitamos.

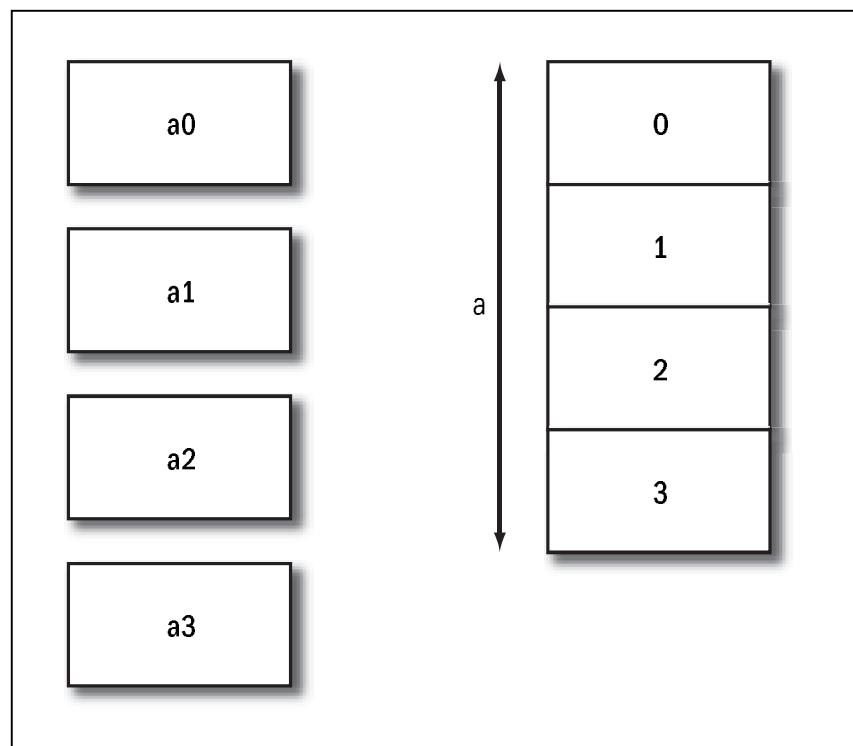
Si observamos el nombre de las variables, vemos que todas llevan en su nombre **calif**, es decir que todas se refieren a la calificación. Podemos pensar que están conceptualmente agrupadas para la calificación. Otro punto que notamos es que las variables están numeradas, como si tuvieran un **índice** que nos sirve para identificarlas.

Los arreglos son similares a estos conceptos ya que son grupos de variables y estas variables serán referenciadas por el mismo nombre. Para poder acceder a una variable del arreglo usaremos un número de índice, ya que todas las variables adentro de un arreglo serán de un mismo tipo. Un punto muy importante que no debemos olvidar cuando trabajemos con los arreglos es que éstos están basados en **índice cero**, esto quiere decir que el primer elemento del arreglo se encuentra en la posición **0**, no en la posición **1** como podríamos pensar. No olvidar este punto nos evitará muchos problemas de lógica en el futuro.



## EL TAMAÑO DEL ARREGLO

En un arreglo es conveniente colocar la cantidad correcta de elementos, ya que una vez creado, no puede crecer a menos que utilicemos una función especial. Si hacemos lo último en forma constante, esto indica que nuestro programa tiene problemas de diseño. En el próximo capítulo veremos estructuras que permiten variar la cantidad de elementos a guardar.



**Figura 1.** En esta figura vemos la comparación entre tener varias variables y un arreglo.

## Declaración de los arreglos de una dimensión

Los arreglos pueden tener diferentes **dimensiones**, y si el arreglo se parece a una simple lista, como la lista de calificaciones que tenemos, entonces decimos que es de una dimensión. Estos arreglos también se conocen como **monodimensionales** o **unidimensionales**. Si el arreglo es como una tabla con varios renglones y varias columnas, entonces es un arreglo de dos dimensiones o **bidimensional**. Primero trabajaremos con los arreglos de una dimensión y luego pasaremos a arreglos de dos dimensiones o más. Para poder trabajar un arreglo, primero es necesario declararlo. En la declaración nosotros indicamos su tipo, su nombre y su tamaño.

```
tipo[] nombre = new tipo[tamaño];
```



Los arreglos tienen una cantidad finita de elementos y ésta se indica en el momento en que se declaran. Cuando intentamos acceder a uno de sus elementos debemos colocar un valor de índice válido y que no exceda el tamaño del arreglo. Un error común con los arreglos es intentar acceder a elementos más allá de los que tiene.

La declaración puede parecer un poco extraña, por lo que a continuación haremos un ejemplo más claro y real, y lo explicaremos a fondo.

```
float[] calificaciones= new float[10];
```

En C# los arreglos son **objetos**, y deberemos usar **new** al declararlos. El arreglo será de tipo flotante, y usaremos **[ ]** para indicar su declaración. Luego debemos colocar el nombre con el que lo identificaremos. En nuestro ejemplo se llama: **calificaciones**. Del lado derecho de la sentencia tenemos la **instanciación** del arreglo. Indicaremos entre **[ ]** la cantidad de elementos que deseamos tener. La cantidad puede ser colocada de forma explícita, tal como está en el ejemplo, o por medio del contenido de una variable. Podemos ejemplificar esto de la siguiente forma:

```
int n = 10;  
  
float[] calificaciones = new float[n];
```

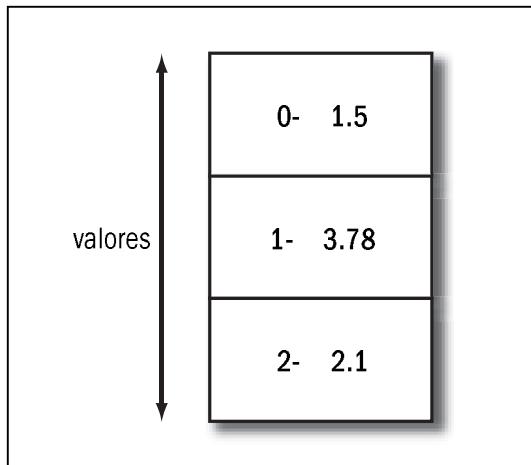
En este caso se tendrá la cantidad de elementos igual al valor guardado en la variable **n**. En algunas ocasiones podemos conocer los valores que colocaremos adentro del arreglo, por lo que podemos declararlo y asignarle sus valores en la misma sentencia. Esto lo hacemos indicando primero el tipo y los **[ ]** seguidos del nombre del arreglo y en el lado derecho de la sentencia colocamos entre **{ }** los elementos que se le desean asignar al arreglo. Estos elementos deberán estar separados por comas. Veámoslo ejemplificado de manera más clara:

```
float[] valores = { 1.5f, 3.78f, 2.1f };
```

En este caso hemos creado un arreglo llamado **valores** y tendrá tres elementos con los valores colocados. La siguiente figura nos muestra cómo quedaría.



La cantidad de datos debe ser un valor válido. No podemos colocar números negativos, ni un tamaño de cero ya que no tendría sentido. Si la cantidad de elementos a crear se pasará por medio de una variable, ésta debe de ser de tipo entera. No tener en cuenta esto traerá problemas al compilar la aplicación.



**Figura 2.** El arreglo tiene tres elementos con los índices 0, 1 y 2.

## Asignación y uso de valores

Ya hemos visto cómo declarar el arreglo. Ahora tenemos que aprender cómo poder colocar información en su interior y hacer uso de ésta.

Para poder asignarle un valor a alguno de los elementos del arreglo necesitamos hacer uso del índice del elemento que queremos utilizar, y como dijimos antes, no debemos olvidar que el primer elemento se encuentra en la posición **0**.

Supongamos que queremos asignarle la calificación **8.5** al tercer alumno.

```
calificaciones[2] = 8.5f;
```

Lo primero es usar el nombre del arreglo, tal y como con las variables, pero entre **[ ]** colocamos el índice del elemento al que se lo queremos asignar. El tercer alumno se encuentra en el índice **2**. Esto se debe a que empezamos a contar desde cero: **0, 1, 2**. Luego del lado derecho de la asignación colocamos el valor a asignar. El control del índice también se puede hacer por medio de una variable de tipo entero. El valor contenido en la variable será usado para acceder al elemento del arreglo.

```
calificaciones[indice] = 8.5f;
```

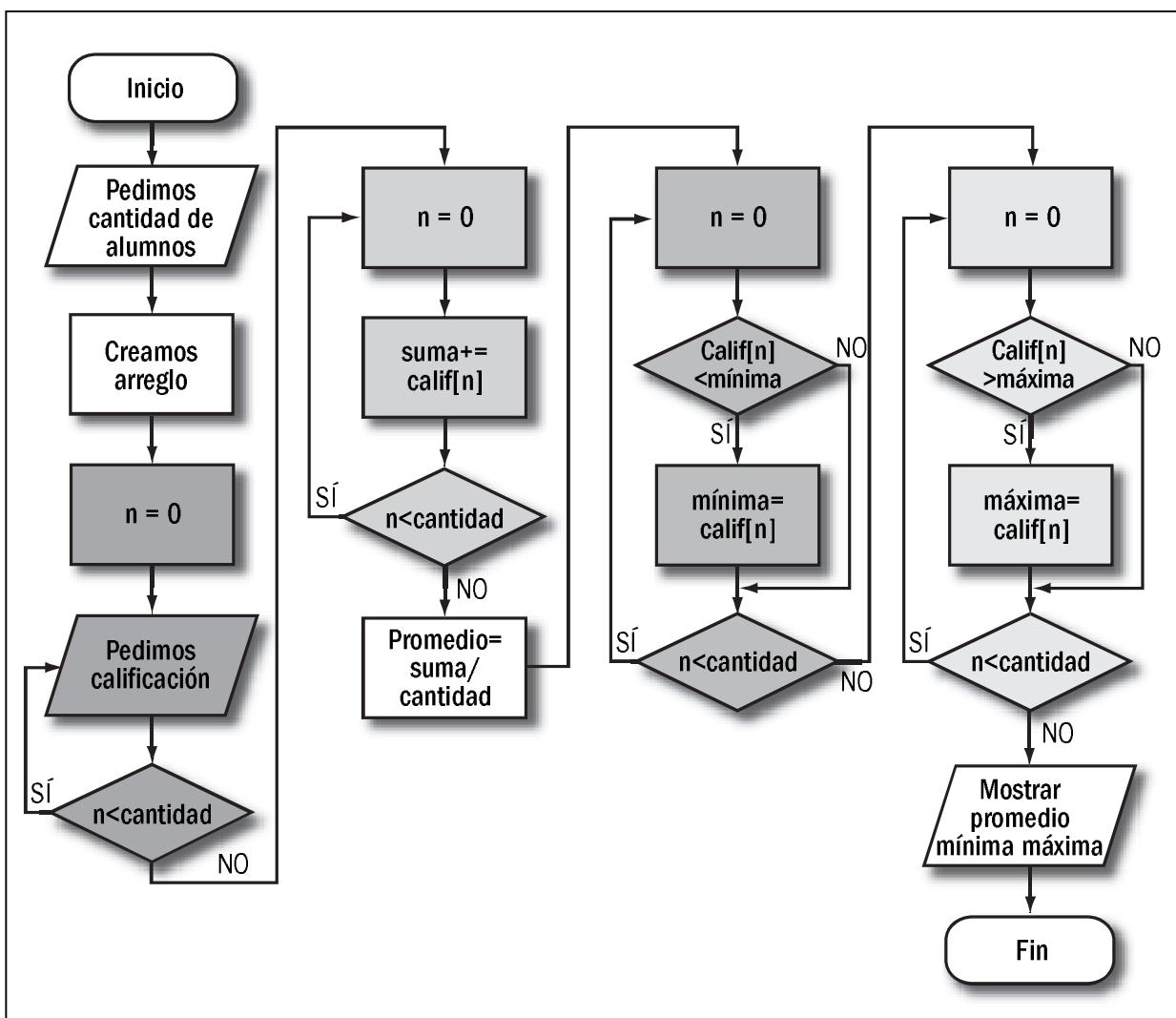
Para el índice, tener la capacidad de utilizar una variable es muy útil, ya que esto nos permitirá recorrer el arreglo con alguno de los ciclos estudiados en los primeros capítulos del libro. Los valores contenidos adentro del arreglo pueden usarse como variables normales, por ejemplo en un cálculo.

```
impuesto = costo[n] * 01.5f;
```

Y el desplegado es igualmente fácil.

```
Console.WriteLine("El valor es {0}", costo[n]);
```

Ahora que ya tenemos los conceptos básicos de los arreglos podemos hacer nuestro programa, donde se pedirá la cantidad de alumnos y las calificaciones. El programa encontrará el promedio, la calificación más alta y la calificación más baja. Para demostrar que los datos no se pierden al ser guardados en el arreglo haremos cada cálculo en un ciclo diferente, y para comprender mejor esto, veamos a continuación el diagrama de flujo correspondiente:



**Figura 3.** El diagrama nos muestra las diferentes partes donde usamos la información del arreglo.

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

```
using System.Text;
namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int cantidad = 0;      // Cantidad de alumnos
            int n = 0;              // Variable de control de ciclo
            string valor = "";

            // Variables para el promedio
            float suma = 0.0f;
            float promedio = 0.0f;

            float minima = 10.0f; // Variable para la
                                  // calificación mínima
            float máxima = 0.0f; // Variable para la calificación
                                  // máxima

            // Pedimos la cantidad de alumnos
            Console.WriteLine("Dame la cantidad de alumnos");
            valor = Console.ReadLine();
            cantidad = Convert.ToInt32(valor);

            // Creamos el arreglo
            float[] calif = new float[cantidad];

            // Capturamos la información
            for (n = 0; n < cantidad; n++)
            {
                Console.Write("Dame la calificación ");
                valor = Console.ReadLine();
                calif[n] = Convert.ToSingle(valor);
            }

            // Encontramos el promedio
```

```

        for (n = 0; n < cantidad; n++)
        {
            suma += calif[n];
        }

        promedio = suma / cantidad;

        // Encontramos la calificación mínima
        for (n = 0; n < cantidad; n++)
        {
            if (calif[n] < minima)
                minima = calif[n];
        }

        // Encontramos la calificación maxima
        for (n = 0; n < cantidad; n++)
        {
            if (calif[n] > maxima)
                máxima = calif[n];
        }

        // Desplegamos los resultados
        Console.WriteLine("El promedio es {0}", promedio);
        Console.WriteLine("La calificación mínima es {0}", minima);
        Console.WriteLine("La calificación máxima es {0}", máxima);

    } // Cierre de main

} // Cierre de la clase
}

```

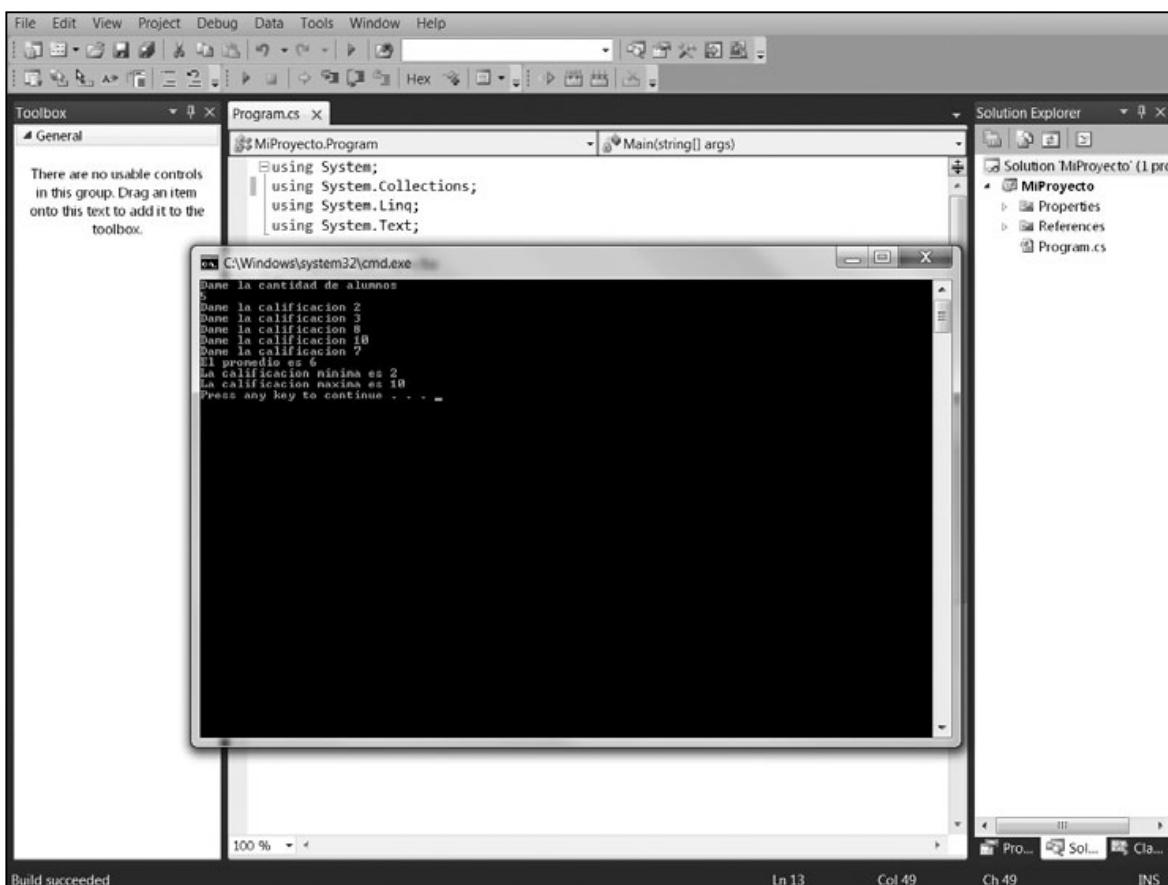
Inicialmente declaramos las variables que necesitará la aplicación. Sin embargo, aún no hemos declarado el arreglo. Esto se debe a que el tamaño del arreglo dependerá de la cantidad de alumnos. Una vez que tenemos esa cantidad creamos el arreglo. Luego, por medio de un ciclo **for** capturamos la información. Debemos notar que el ciclo inicia en **0**. Esto se debe a que el primer elemento del arreglo se encuentra en el índice **0**. El ciclo se recorre tantas veces como elementos tenga el arreglo. Para calcular el promedio, recorremos el arreglo de nuevo por medio de un ciclo **for** con la variable **suma** para obtener la sumatoria de todas las calificaciones. Luego simplemente calculamos el promedio, dividiendo la suma entre la cantidad de alumnos.

Para poder calcular la calificación menor declaramos una variable llamada **minima**. A esta variable la inicializamos con el valor más alto posible para las calificaciones (la razón de esto tendrá sentido enseguida). Creamos nuevamente un ciclo **for** y recorremos el arreglo. Para cada elemento del arreglo comparamos su valor con el valor de nuestra variable. Si el valor del elemento del arreglo es menor que **minima**, eso significa que hemos encontrado un valor menor, por lo que **minima** se actualiza con este nuevo valor menor. Esto se repite hasta que hemos terminado con todos los elementos del arreglo y **minima** en cada caso tendrá el valor más pequeño encontrado hasta esa vuelta del ciclo.

De igual forma encontramos la calificación más alta del arreglo pero en este caso creamos una variable llamada máxima. Esta variable se inicializa con el valor más pequeño que puede tener una calificación. Hacemos un ciclo y recorremos todo el arreglo, y a cada elemento contenido en éste lo comparamos con el valor de **máxima**. Si es mayor eso significa que hemos encontrado un nuevo máximo, por lo que actualizamos la variable. Si lo anterior se cumple, al finalizar el ciclo tendremos la calificación más grande para ese grupo de alumnos.

Para finalizar el programa, simplemente mostramos en la consola o línea de comandos, los resultados que hemos obtenido.

Ejecutemos y probemos cómo funciona la teoría hasta aquí explicada.

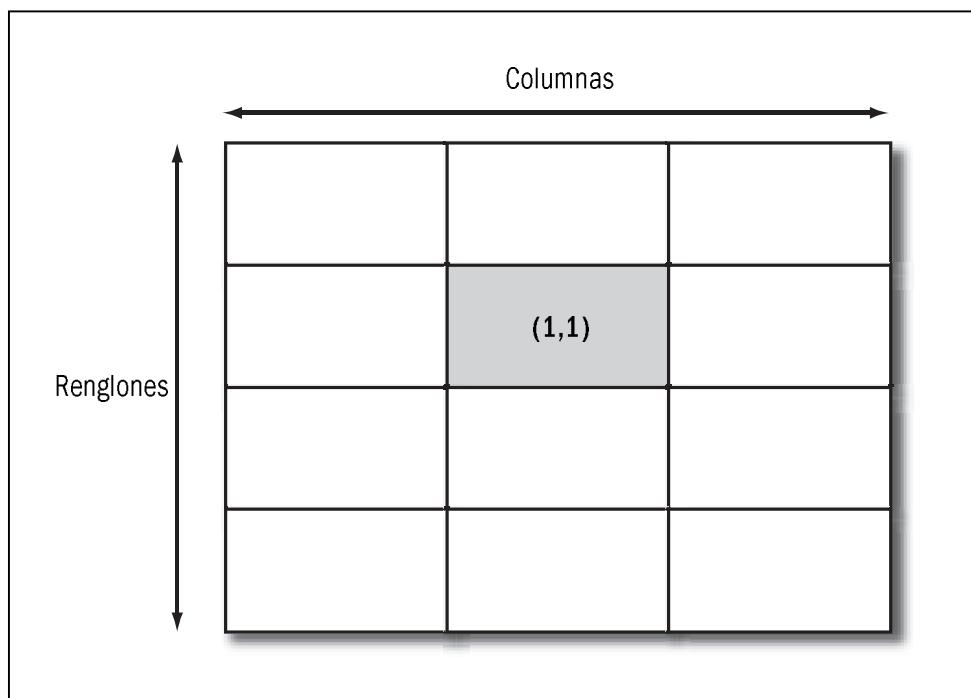


**Figura 4.** El programa guarda la información en un arreglo y podemos utilizarla cuando sea necesario.

## Arreglos de dos dimensiones

Hasta aquí hemos visto que los arreglos nos ayudan a guardar información y a trabajar de una manera más cómoda con ella. Cuando tenemos mucha información que almacenar es más fácil manipular un arreglo que muchas variables. Los arreglos que vimos son similares a una lista, pero no todos los problemas se pueden resolver con este esquema, y veamos por qué. Supongamos que ahora debemos hacer un nuevo programa para una fábrica que produce automóviles y desea tener la información de las unidades producidas a diario. La información será procesada por semana, con el promedio semanal de vehículos producidos. Hasta el momento el problema sería muy similar al anterior. Podríamos crear un arreglo de 7 elementos llamado semana y estaría resuelto, pero ahora también desean la información por mes. El mes tiene cuatro semanas, por lo que podemos pensar en tener cuatro arreglos, uno que se corresponda con cada semana, pero en realidad existe una forma mejor de solucionar este problema.

Si pensamos un poco más en el problema veremos que podemos guardar la información en una tabla, cada **columna** sería una semana y cada **renglón** representaría un día. Esto se conoce como **matriz**, y es un arreglo de dos dimensiones. En los arreglos de dos dimensiones tenemos que utilizar dos índices. Uno controlará el renglón y el otro la columna. Con la creación de estos dos índices es posible que accedamos a cualquier **celda** ubicada dentro de la matriz.



**Figura 5.** El diagrama nos muestra un arreglo de dos dimensiones y cómo podemos localizar una celda por medio de su renglón y de su columna.

Al igual que con los arreglos de una sola dimensión, los índices están basados en **0**. Esto es tanto para los renglones como para las columnas.

## Declaración de los arreglos de dos dimensiones

La declaración es similar al arreglo de una dimensión, pero indicamos la cantidad de elementos en cada dimensión.

```
float[,] tabla = new float[5,3];
```

En este caso, hemos creado una matriz de valores flotantes llamada **tabla**. Hay que notar que hemos colocado una coma entre los **[ ]** para indicar que serán dos dimensiones. En el lado derecho indicamos el tamaño de cada dimensión. Para este ejemplo tenemos cinco columnas y tres renglones.

También es posible declarar la matriz por medio de variables.

```
float[,] tabla = new float[n, m];
```

## Acceso a un arreglo de dos dimensiones

La utilización del arreglo de dos dimensiones es muy sencilla, podemos acceder a la información si indicamos el renglón y la columna de la celda que nos interesa. Por ejemplo, si deseamos hacer uso del valor que se encuentra almacenado en la celda **(3, 2)** hacemos lo siguiente:

```
impuesto = producto[3,2] * 0.15;
```

En este caso se toma el valor de la celda **(3,2)** del arreglo **producto**, se multiplica por **0.15** y el valor resultante se asigna a **impuesto**.

De igual forma podemos hacer la asignación al arreglo.

```
producto[3,2] = 17.50;
```

Para mostrar el valor en la consola hacemos lo siguiente:

```
Console.WriteLine("El costo es {0}", producto[3,2]);
```

No debemos olvidar que los índices siempre se inician en cero, y que no es posible colocar índices más grandes que la cantidad de elementos contenidos en el arreglo. Con este conocimiento podemos comenzar a trabajar en un ejemplo. Tomaremos

el código del ejemplo de la escuela y ahora haremos posible que trabaje para varios salones. Para este caso pediremos la cantidad de salones y la cantidad de alumnos. Cada columna representará el salón y los renglones guardarán la información de los alumnos. De esta forma podemos capturar toda la información de la escuela y calcular el promedio, y la calificación mínima y máxima de toda la escuela.

Para que este programa funcione de forma eficiente utilizaremos la sentencia **for** y también un tipo de ciclo mencionado en un capítulo anterior. Éstos son los **ciclos enlazados**, o simplemente, un ciclo dentro de otro ciclo. La forma de utilizarlo es la siguiente: tendremos un ciclo para recorrer cada uno de los salones y dentro de éste tendremos otro ciclo que se dedicará a recorrer los estudiantes. De esta forma podemos cubrir todos los salones y todos los estudiantes de cada salón.

A continuación un ejemplo práctico de ciclos enlazados.

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

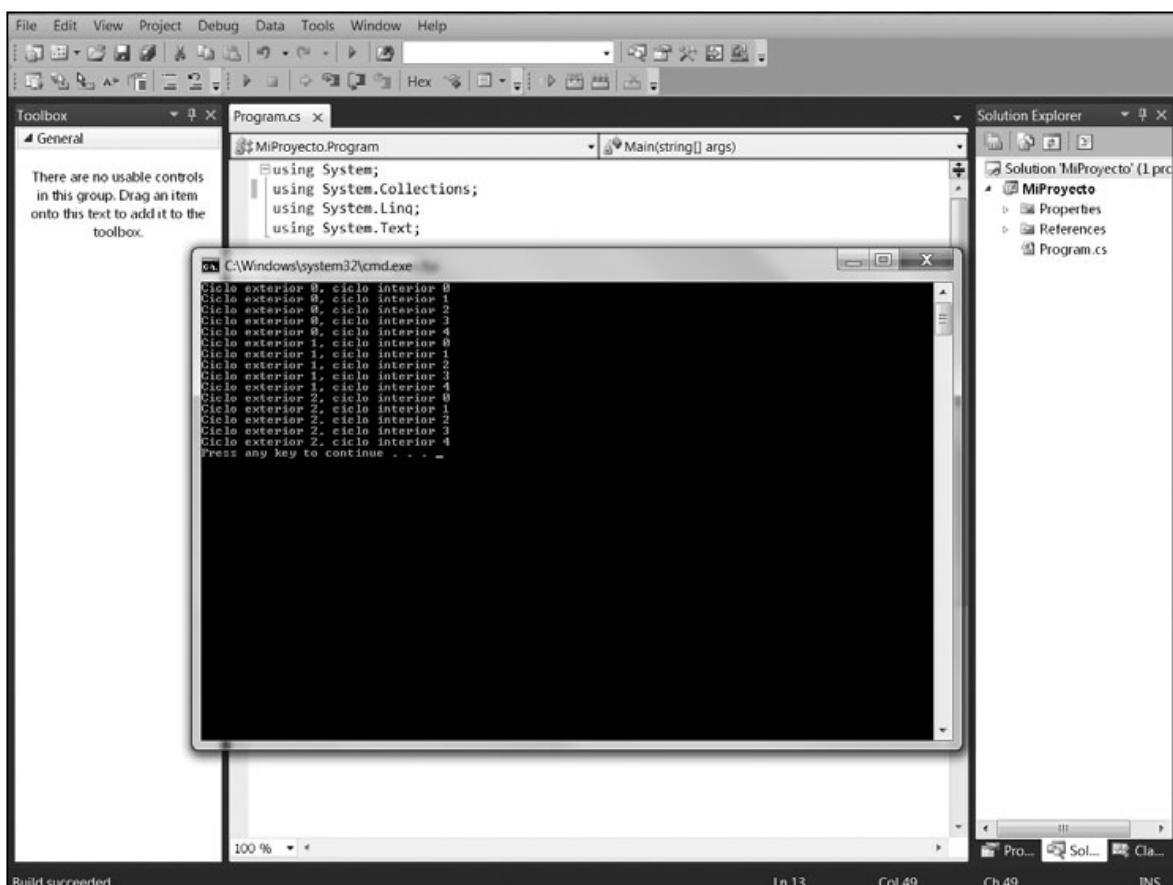
namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int n = 0; // Ciclo exterior
            int m = 0; // Ciclo interior

            // Ciclos enlazados
            for (n = 0; n < 3; n++)
            {
                for (m = 0; m < 5; m++)
                {
                    Console.WriteLine("Ciclo exterior {0}, ciclo interior
{1}", n, m);

                } // fin del ciclo m
            } // fin del ciclo n
        }
    }
}
```

}

Compilemos y ejecutemos el programa.



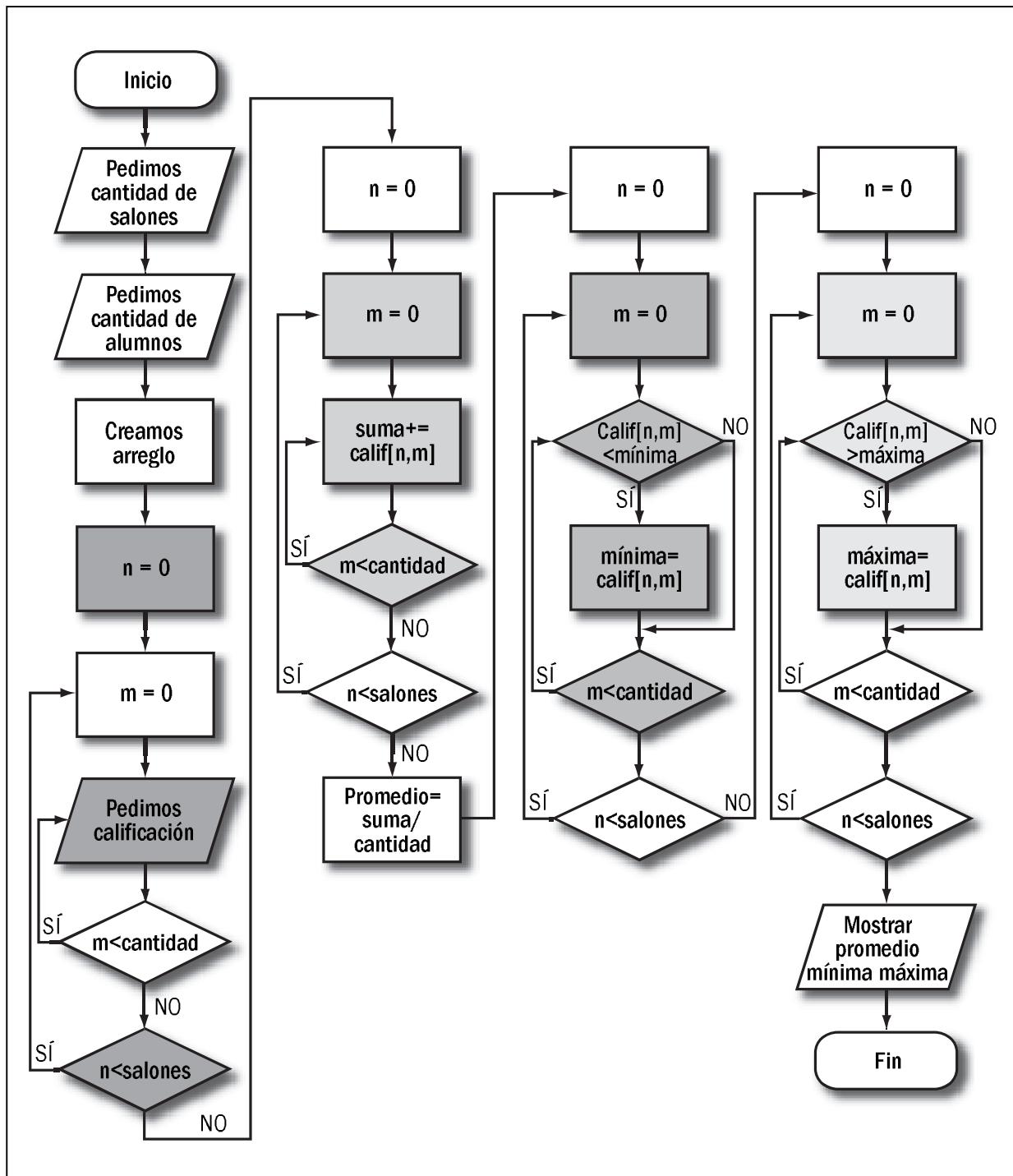
**Figura 6.** Vemos cómo se recorren los ciclos. Para cada paso del ciclo exterior tenemos una vuelta completa del ciclo interior.

Al primer ciclo lo llamaremos **ciclo exterior** y al ciclo que se coloca adentro lo llamaremos **ciclo interior**. Cada vez que el ciclo exterior avanza un paso, el ciclo interior da una vuelta completa. Este tipo de comportamiento es el que nos



Los arreglos tienen un método conocido como **Array.Clear()**. Este método limpia el arreglo y al hacerlo coloca todos sus elementos en **0**, **false** o **null** dependiendo del tipo del arreglo. El método tiene tres parámetros. El primero es el arreglo a borrar. Luego debemos colocar el índice a partir de donde deseamos borrar y por último la cantidad de elementos a limpiar.

permite recorrer toda la tabla. Para cada columna recorremos todos los renglones. Veamos en el diagrama de flujo del programa cómo sería:



**Figura 7.** El diagrama muestra cómo podemos localizar los ciclos enlazados fácilmente.

Nuestro programa queda constituido de la siguiente manera:

```

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

```
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {

            // Variables necesarias
            int cantidad = 0;      // Cantidad de alumnos
            int salones = 0;       // Cantidad de salones
            int n = 0;             // Variable de control de ciclo salones
            int m = 0;             // Variable de control del ciclo alumnos
            string valor = "";

            // Variables para el promedio
            float suma = 0.0f;
            float promedio = 0.0f;

            float minima = 10.0f; // Variable para la calificación mínima
            float máxima = 0.0f; // Variable para la calificación máxima

            // Pedimos la cantidad de salones
            Console.WriteLine("Dame la cantidad de salones");
            valor = Console.ReadLine();
            salones = Convert.ToInt32(valor);

            // Pedimos la cantidad de alumnos
```



Podemos tener arreglos de más de dos dimensiones. El mecanismo para declararlos y utilizarlos es el mismo. Simplemente debemos tener cuidado con el manejo de los índices para cada dimensión y si el arreglo es de tres dimensiones necesitaremos tres índices, si es de cuatro, cuatro índices y así sucesivamente.

```

Console.WriteLine("Dame la cantidad de alumnos por salón");
valor = Console.ReadLine();
cantidad = Convert.ToInt32(valor);

// Creamos el arreglo
float[,] calif = new float[salones, cantidad];

// Capturamos la información
for (n = 0; n < salones; n++) // Ciclo salones

{
    Console.WriteLine("Salón {0}", n);
    for (m = 0; m < cantidad; m++) // Ciclo alumnos
    {
        Console.Write("Dame la calificación ");
        valor = Console.ReadLine();
        calif[n, m] = Convert.ToSingle(valor);
    }
}

// Encontramos el promedio
for (n = 0; n < salones)           // Ciclo salones
{
    for (m = 0; m < cantidad; m++) // Ciclo alumnos
    {
        suma += calif[n, m];
    }
}
promedio = suma / (cantidad * salones);

// Encontramos la calificación mínima

```

### III

Aunque no parezca, cuando usamos arreglos de muchas dimensiones la cantidad de memoria necesaria para guardarlos puede crecer rápidamente sin notarlo. Un arreglo de tres dimensiones de enteros con un tamaño de 256x256x256 no puede parecer mucho, pero necesita 67,108,864 bytes de memoria.

```

        for (n = 0; n < salones; n++) // Ciclo salones
        {
            for (m = 0; m < cantidad; m++) // Ciclo alumnos
            {
                if (calif[n, m] < minima)
                    minima = calif[n, m];
            }
        }

        // Encontramos la calificación maxima
        for (n = 0; n < salones; n++) // Ciclo salones
        {
            for (m = 0; m < cantidad; m++) // Ciclo alumnos
            {
                if (calif[n, m] > maxima)
                    máxima = calif[n, m];
            }
        }

        // Desplegamos los resultados
        Console.WriteLine("El promedio es {0}", promedio);
        Console.WriteLine("La calificación mínima es {0}", minima);
        Console.WriteLine("La calificación máxima es {0}",
                          máxima);

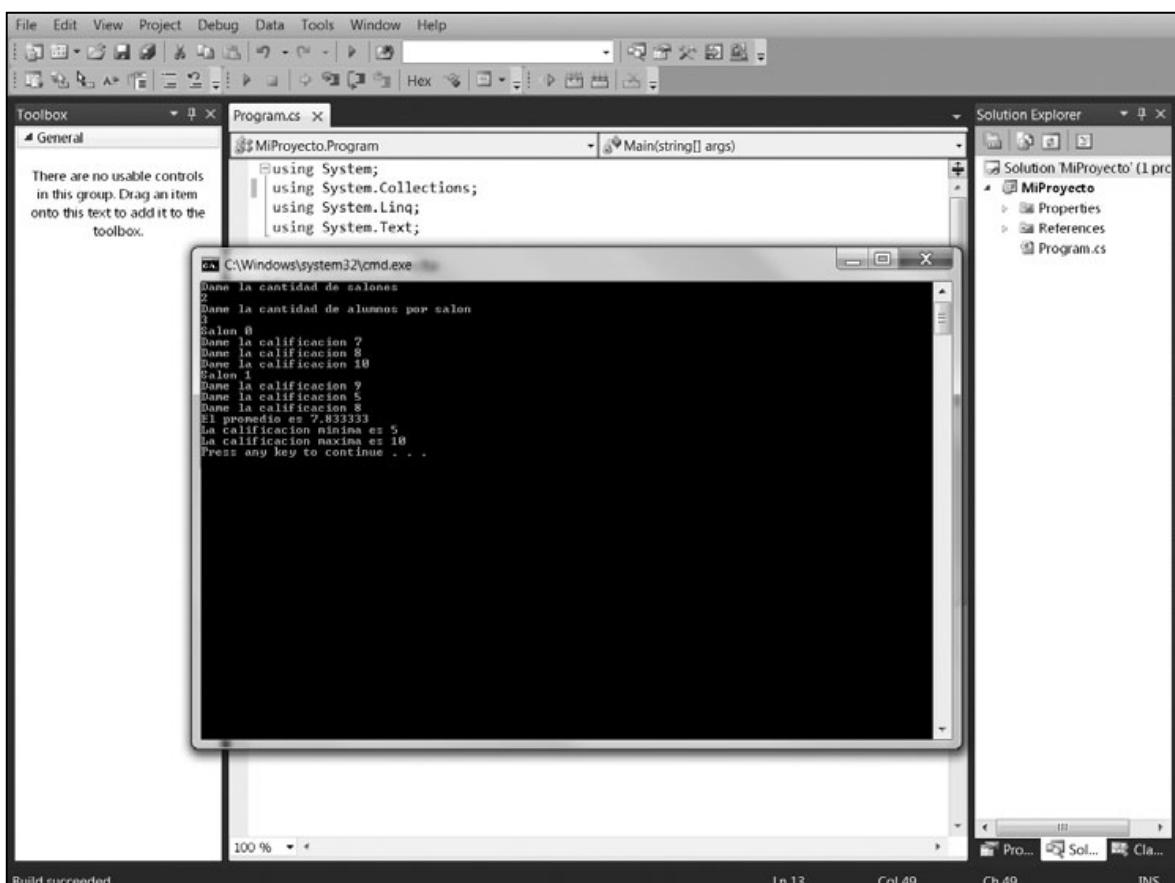
    } // Cierre de main

}
}

```

Lo primero que necesitamos es obtener las dimensiones de nuestra matriz. Para esto debemos saber cuántos salones o cuántas columnas y qué cantidad de alumnos o renglones necesitaremos. Como podemos observar, en el código tenemos los ciclos enlazados. Un ciclo recorre los salones y el otro recorre los alumnos. Cuando hacemos la petición de la información, es posible asignarla a la celda adecuada por medio de las variables **m** y **n**. De esta forma capturamos toda la información y queda guardada en las celdas del arreglo.

Teniendo esta información, podemos procesarla. Como trabajamos sobre toda la tabla, nuevamente usamos los ciclos anidados, desplegando al final sólo los resultados. Ejecutemos el programa y veamos cómo funciona:



**Figura 8.** Observemos que capturamos la información y luego la procesamos en diferentes partes del programa para obtener los resultados finales.

Es posible que nosotros hayamos procesado una columna en particular. Para esto simplemente mantenemos fijo el valor de la columna y recorremos los renglones con un ciclo. Un ejemplo de esto sería el siguiente:

```

for (n = 0; n < 10; n++)
{
    Console.WriteLine("El valor es {0}", producto[5,n]);
}

```

En este caso recorremos la columna **5** de la matriz `producto`. El índice del renglón es colocado por la variable de control del ciclo.  
También es posible mantener fijo el renglón y recorrer todas las columnas. Para esto nuevamente es necesario un solo ciclo.

```

for (n = 0; n < 10; n++)
{
}

```

```

Console.WriteLine("El valor es {0}", producto[n,3]);
}

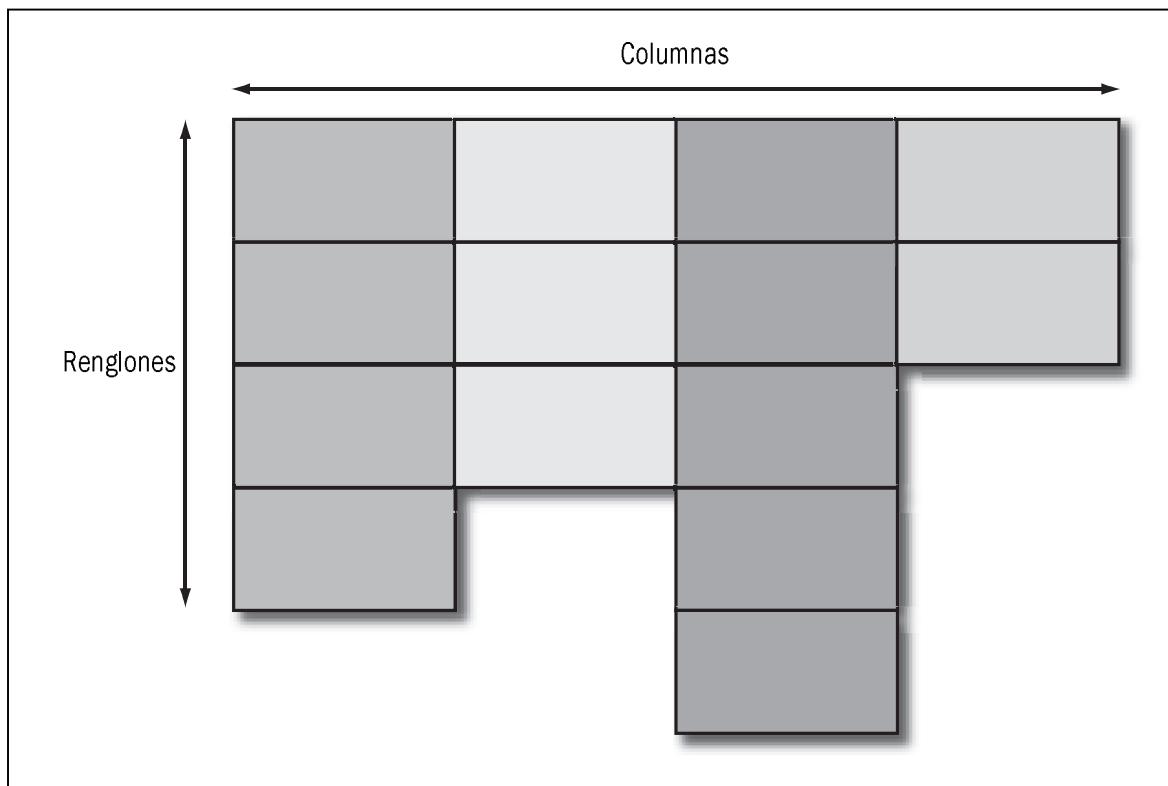
```

En el caso anterior recorrimos todas las columnas del renglón 3.

## Arreglos de tipo jagged

En el ejemplo anterior todas las columnas tienen la misma cantidad de renglones, es decir, que todos los salones tendrían la misma cantidad de alumnos, sin embargo, en la práctica esto no sucederá. Tendremos salones con diferentes cantidades de alumnos, y los arreglos que hemos visto hasta el momento no permiten tener columnas con diferente cantidad de renglones. Esto puede ser una limitación, y puede significar que tengamos renglones sin utilizar en varias columnas. Esto nos lleva a desperdiciar memoria. Una solución podría ser tener un arreglo que nos permita lograr esto. Para hacerlo tenemos que crear un arreglo de arreglos. Esto se conoce como **arreglo jagged**. En lugar de usar una matriz, lo que haremos es crear un arreglo, pero cada elemento de este arreglo será a su vez otro arreglo. Así podemos controlar de forma independiente la cantidad de renglones en cada arreglo. Estos arreglos son más flexibles que los tradicionales, pero requieren que seamos más cuidadosos con ellos.

La siguiente figura nos muestra cómo está constituido un arreglo jagged.



**Figura 9.** Tenemos un arreglo y cada elemento del arreglo es a su vez otro arreglo.

Al arreglo que contiene los demás arreglos lo llamaremos **arreglo contenedor** para que podamos reverenciarnos a él fácilmente.

## Declaración de un arreglo jagged

La declaración de los arreglos jagged es ligeramente más complicada que la de los tradicionales. Cuando los declaramos debemos declarar en primer lugar el arreglo contendor y luego cada uno de los arreglos independientes que tiene.

Veamos un primer ejemplo de declaración:

```
// Declaramos arreglo contenedor
int[][] Costos = new int[3][];

// Declaramos los arreglos
Costos[0] = new int[15];
Costos[1] = new int[20];
Costos[2] = new int[10];
```

Lo que declaramos es un arreglo jagged de tipo entero que se llamará **Costos**. Como observamos, junto al **int** tenemos **[ ][ ]**. Esto indica que es un arreglo de arreglos y no una matriz. En el lado derecho de la asignación tenemos algo similar. Aquí indicamos que nuestro arreglo tiene 3 columnas, pero inmediatamente dejamos **[ ]** vacíos. Esto es lo que nos permite tener la flexibilidad de darle un tamaño propio a cada renglón. Ya tenemos inicializado el arreglo contenedor. Ahora necesitamos inicializar cada uno de los arreglos internos. Primero indicamos cuál es la columna a inicializar. En el ejemplo tenemos **Costos[0]**, es decir, que inicializaremos el arreglo que se encuentra en la primera columna. Del lado derecho está **int[15]**, con lo que indicamos que tendrá quince elementos, es decir, 15 renglones para la columna **0**.

De forma similar la columna **1** tendrá **20** renglones y la columna **2** tendrá **10** renglones. Podemos ver que ahora tenemos diferente cantidad de renglones para cada columna. Al igual que con los arreglos tradicionales, la cantidad de elementos que tendrá el arreglo puede ser colocada de forma explícita o por medio de



Como en un arreglo de tipo jagged nosotros podemos tener diferente cantidad de renglones, es necesario tener un mecanismo que nos permita saber cuántos renglones tenemos, ya que de lo contrario corremos el riesgo de colocar índices indebidos que provocarán errores de lógica. Estos errores pueden ser difíciles de corregir, por lo que es mejor estar preparados antes de utilizarlos.

una variable. Si conocemos con anterioridad la información que tendrá el arreglo jagged podemos hacer lo siguiente:

```
int[][] valores = new int[3][];
valores[0] = new int[] { 9, 3, 1, 7, 2, 4 };
valores[1] = new int[] { 2, 9 };
valores[2] = new int[] { 3, 5, 2, 9 };
```

En este caso creamos un arreglo jagged de tres columnas y luego, al momento de crear cada uno de los arreglos internos, los instanciamos colocando los valores directamente. En este caso cada columna tendrá la cantidad de renglones según la cantidad de valores utilizados en su instanciación. La columna **0** tendrá seis renglones, la columna **1** tendrá dos renglones y la columna **2** tendrá cuatro renglones.

A continuación se muestra otra forma para inicializar este tipo de arreglos cuando conocemos previamente los elementos que utilizaremos adentro de él:

```
int[][] valores = new int[][] {
{
    new int[] { 9, 3, 1, 7, 2, 4 },
    new int[] { 2, 9 },
    new int[] { 3, 5, 2, 9 }
};
```

En el ejemplo listado no indicamos directamente la cantidad de columnas. C# encontrará este valor dependiendo de la cantidad de arreglos que se declaren adentro del bloque de código. En este caso declaramos tres arreglos internos, por lo que la cantidad de columnas es de tres. Al igual que en el caso anterior, la cantidad de renglones por columnas dependerá de la cantidad de elementos que declaramos en cada inicialización, entonces tendremos las mismas cantidades que en el ejemplo anterior.

## Acceder a un arreglo jagged

Para acceder a los elementos guardados adentro de un arreglo jagged también necesitamos utilizar índices. Un índice será aplicado para indicar cuál elemento del arreglo contenedor utilizaremos, es decir el número de columna. El otro índice entonces nos indicará el elemento del arreglo interno que queremos acceder.

Por ejemplo, para asignar un valor realizamos lo siguiente:

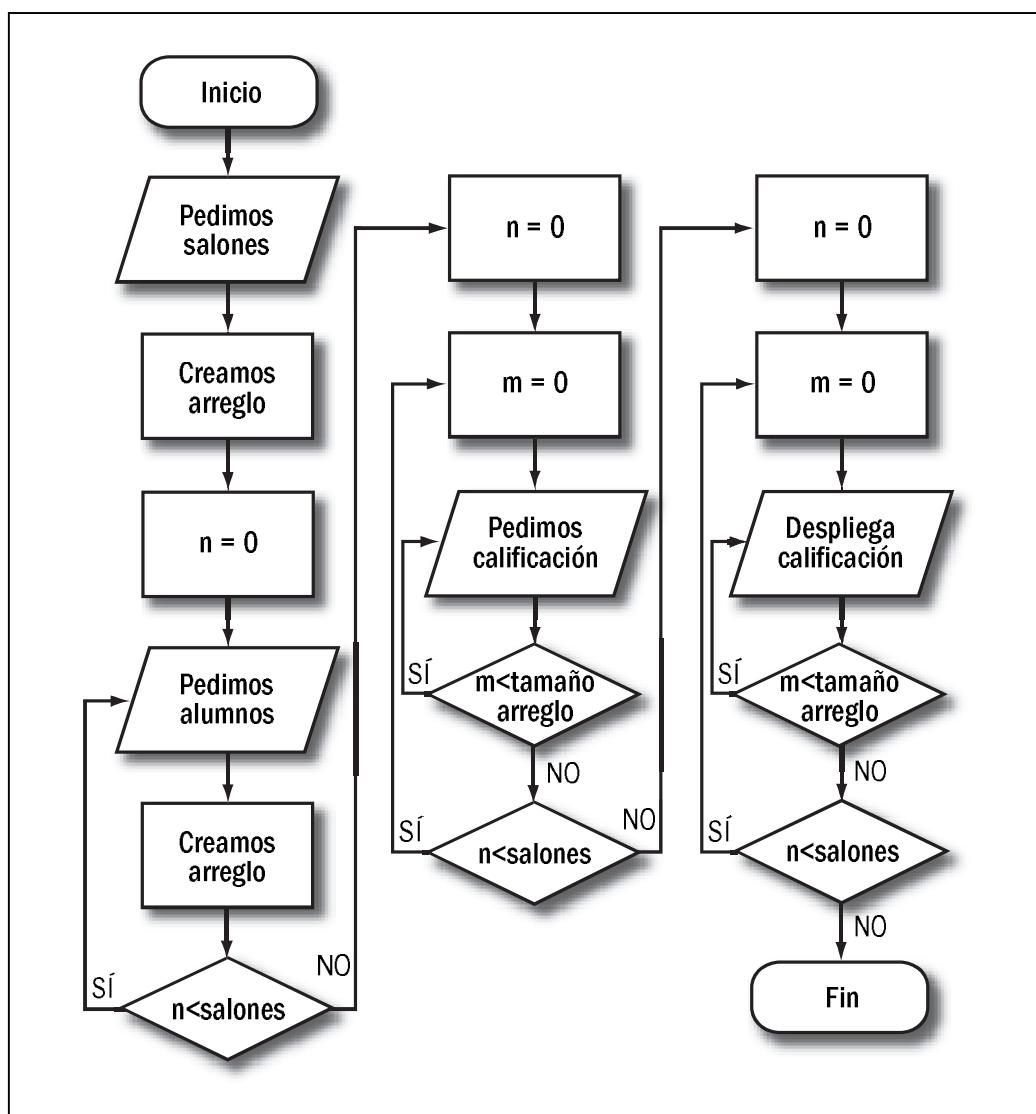
```
productos[6][7] = 5.7f;
```

En este ejemplo vemos que se selecciona el elemento **6** del arreglo contenedor, o si lo preferimos la columna **6**. Adentro de esa columna seleccionamos el elemento **7** y ahí es dónde se coloca el valor **5.7**. Si lo que necesitamos es mostrar el contenido de un elemento, el esquema es similar a lo que ya conocemos.

```
Console.WriteLine("El valor es {0}", productos[5][n]);
```

Ahora modificaremos algunas partes del programa de la escuela para usar el arreglo de tipo jagged. Lo que haremos es tener salones con diferente cantidad de alumnos y luego simplemente mostraremos las calificaciones de cada salón.

El diagrama es el siguiente:



**Figura 10.** Hemos agregado un ciclo para la creación de cada uno de los arreglos internos.

El código del programa quedará de la siguiente forma:

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

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Variables necesarias
            int cantidad=0;          // Cantidad de alumnos
            int salones=0; // Cantidad de salones
            int n=0;      // Variable de control de ciclo
                           salones
            int m=0;      // Variable de control del ciclo
                           alumnos
            string valor="";

            // Variables para el promedio
            float suma=0.0f;
            float promedio=0.0f;

            float minima=10.0f; //Variable para la
                               calificación
                               mínima
            float maxima=0.0f; //Variable para la calificación
                               maxima

            // Pedimos la cantidad de salones
            Console.WriteLine("Dame la cantidad de salones");
            valor=Console.ReadLine();
            salones=Convert.ToInt32(valor);

            // Creamos el arreglo
            float[][] calif= new float [salones][];
            // Pedimos los alumnos por salón
```

```
for(n=0;n<salones) // Ciclo salones
{
    Console.WriteLine("Dame la cantidad de
                      alumnos
                      para el salon {0}",n);
    valor=Console.ReadLine();
    cantidad=Convert.ToInt32(valor);

    // Instanciamos el arreglo
    calif[n]=new float[cantidad];

}

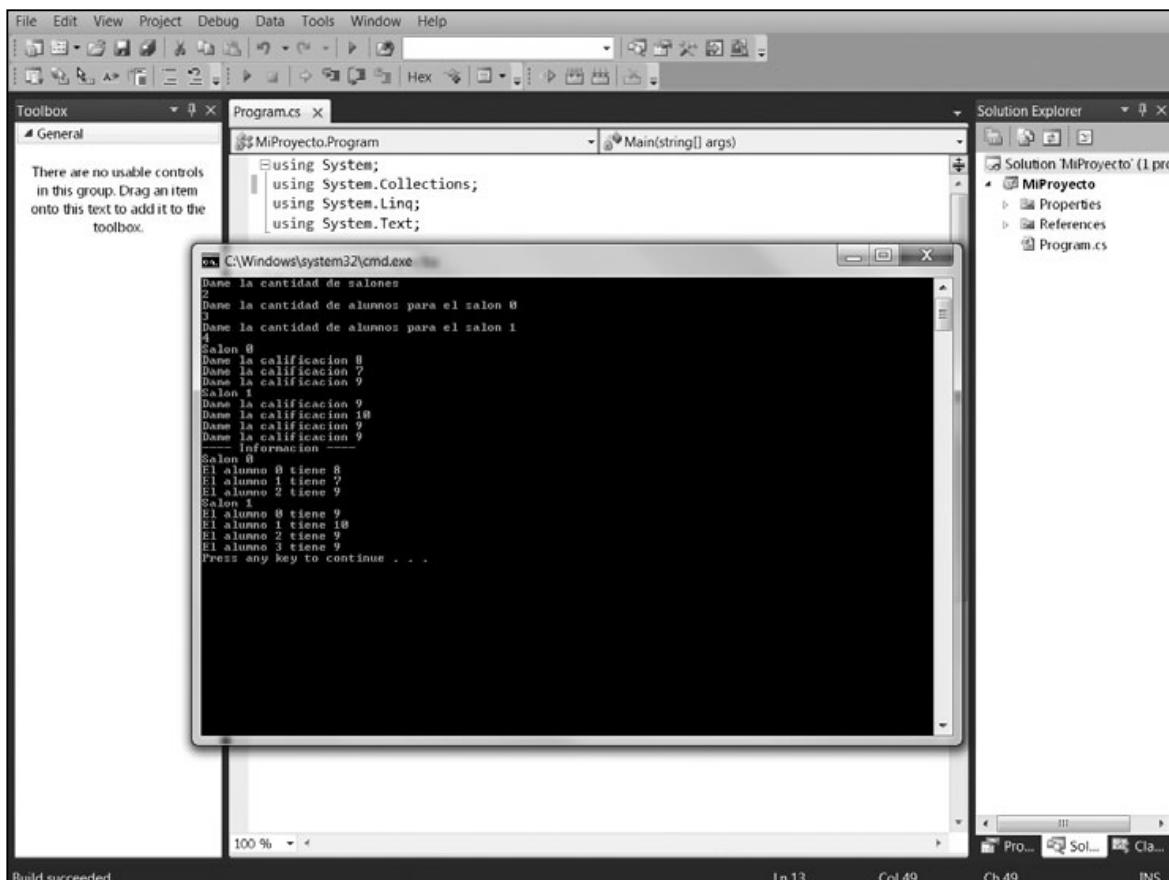
// Capturamos la información
for(n=0;n<salones) // Ciclo salones
{
    Console.WriteLine("Salon {0}",n);
    for(m=0;m<calif[n].GetLength(0);m++) //
                      Ciclo
                      alumnos
    {
        Console.Write("Dame la calificación ");
        valor=Console.ReadLine();
        calif[n][m]=Convert.ToSingle(valor);
    }
}

// Desplegamos la información
Console.WriteLine("— Información —");
for (n = 0; n < salones; n++) // Ciclo salones
{
    Console.WriteLine("Salon {0}", n);
    for (m = 0; m < calif[n].GetLength(0); m++) // Ciclo
                      alumnos
    {
        Console.WriteLine("El alumno {0} tiene {1} ", m,
                          calif[n][m]);
    }
}
```

Podemos observar que como primera medida hemos creado el arreglo contenedor de acuerdo con la cantidad de salones que tendremos, luego, por medio de un ciclo, recorremos cada uno de los salones preguntando en cada uno la cantidad de alumnos, entonces ahí creamos el arreglo interno con el tamaño obtenido. Debemos tener cuidado y prestar mucha atención en el uso de la sintaxis para procesar cada arreglo que manejaremos.

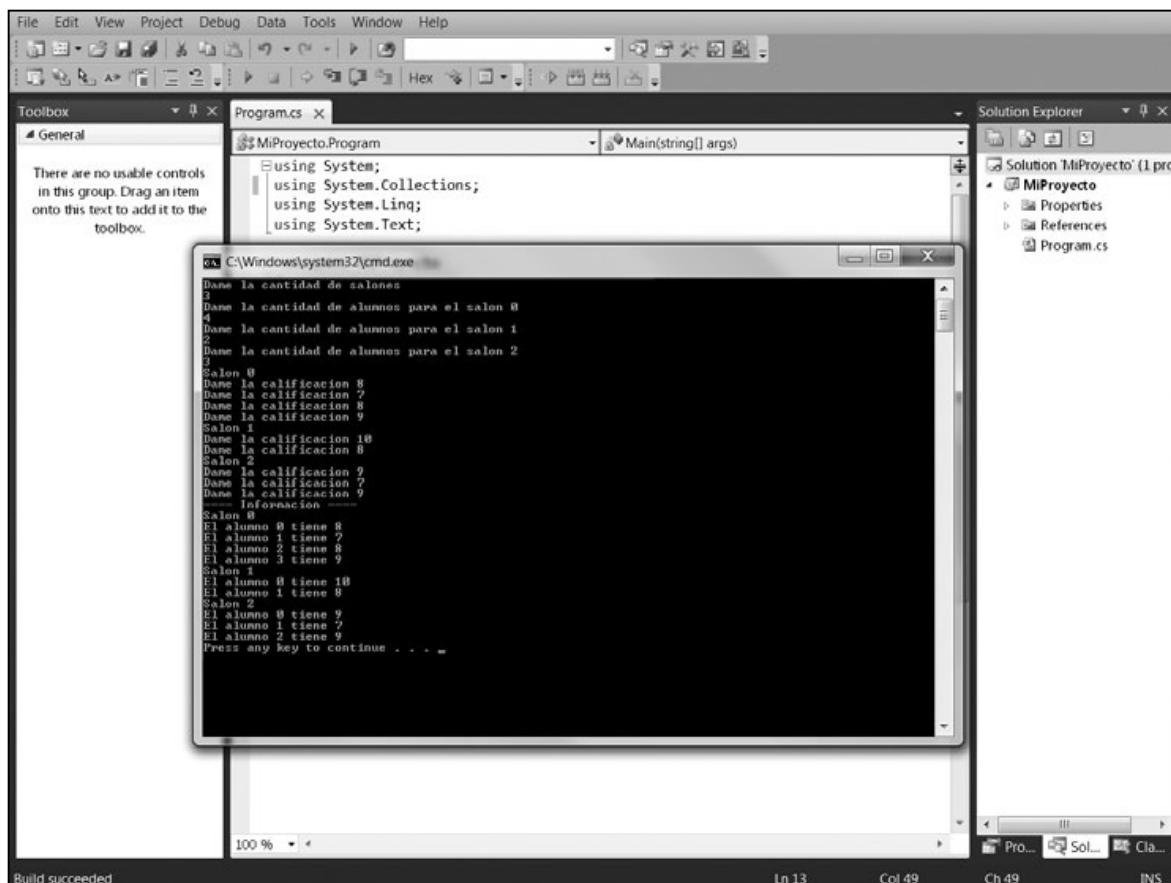
Veamos que del lado derecho de la sentencia tenemos el índice del arreglo que crearemos y en el lado izquierdo indicamos el tamaño. De esta forma, quedará creado un arreglo de ese tamaño en tal posición del arreglo contenedor.

Ya creado el arreglo **jagged** procedemos a colocar la información en su interior. Al igual que antes usamos ciclos enlazados, y solamente deberemos cambiar la asignación de acuerdo con la sintaxis del arreglo jagged. Lo mismo ocurre cuando queremos imprimir los contenidos de arreglo jagged en la consola.



**Figura 11.** Podemos observar que efectivamente cada salón tiene un número diferente de alumnos.

Para verificar que efectivamente podemos crear columnas con diferentes tamaños cada vez, ejecutemos el programa de nuevo con otros valores.



**Figura 12.** En esta ocasión hemos creado un arreglo jagged de diferente tamaño.

## Los arreglos como parámetros a funciones

Aprendimos a usar los arreglos. Hasta el momento han sido utilizados adentro de la función **Main()**. A medida que nuestros programas sean más grandes y especializados, tendremos que usar funciones y éstas necesitarán procesar la información contenida en los arreglos. Al igual que con las variables, es posible pasar un arreglo como parámetro. De esta forma podremos encargarnos de enviar la información desde el arreglo hacia la función correspondiente.



Cuando enviamos un arreglo como parámetro a una función, contemplemos los siguientes factores. El primero es que el arreglo enviado y el arreglo del parámetro deben ser el mismo tipo de dato, el segundo es que deben tener las mismas dimensiones, es decir ambos son como lista o como matriz, y el último es que ambos deben ser del mismo estilo: normal o jagged.

Nuestra función debe estar definida de forma similar, como se muestra a continuación:

```
static void Imprime(int[] arreglo)
{
    ...
}
```

Como podemos ver, en la lista de parámetros indicamos que el parámetro es un arreglo al colocar `[ ]` y es de tipo **int**. El nombre del parámetro en este caso es arreglo y lo podremos utilizar internamente dentro de la función como variable local.

La invocación se llevaría a cabo de la siguiente forma:

```
int[] numeros = new int[5];
...
...
Imprime(numeros);
```

Vemos que tenemos un arreglo de enteros que se llama **números**. En el momento de invocar la función lo hacemos por medio del nombre de la función y colocamos como parámetro solamente el nombre del arreglo, es decir, pasamos el arreglo pero sin colocarle `[ ]`. Esto se debe a que pasamos un arreglo completo y no solamente un elemento que se encuentra adentro del arreglo. Ahora veremos un ejemplo donde podamos hacer uso de esto. En la función **Main()** crearemos un arreglo y capturaremos los datos necesarios. Luego tendremos una función especializada en imprimir los contenidos del arreglo.

Nuestro código queda de la siguiente forma:

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

namespace AplicacionBase
{
    class Program
    {
```

```
// Esta es la función principal del programa
// Aquí inicia la aplicación
static void Main(string[] args)
{
    // Variables necesarias
    int[] numeros = new int[5];
    int n = 0;
    string valor = "";

    // Pedimos los números
    for (n = 0; n < 5; n++)
    {
        Console.WriteLine("Dame un número ");
        valor = Console.ReadLine();
        numeros[n] = Convert.ToInt32(valor);
    }

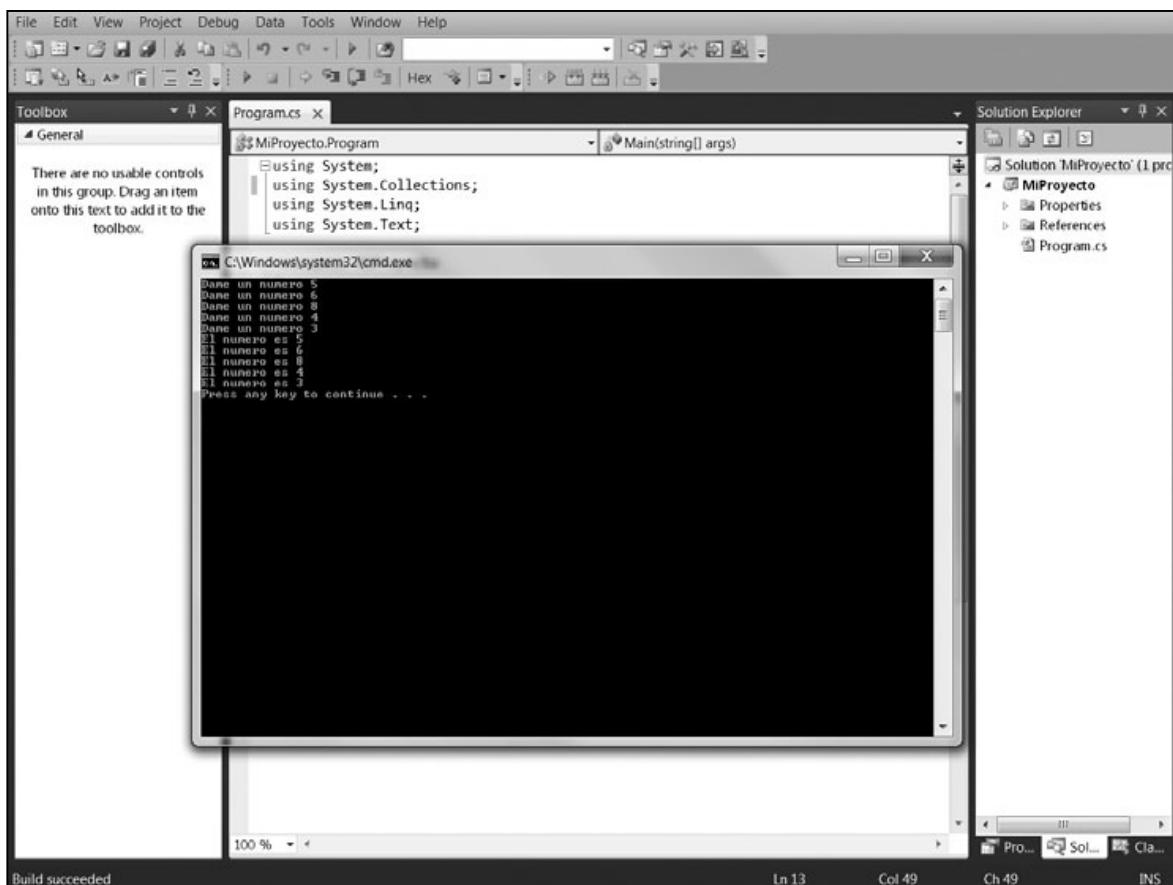
    // Invocamos a la función
    Imprime(numeros);
}

// Esta es la función de impresión
static void Imprime(int[] arreglo)
{
    int n = 0;

    for (n = 0; n < 5; n++)
    {
        Console.WriteLine("El número es {0}", arreglo[n]);
    }
}
```

Vemos que es muy sencillo llevar a cabo el paso del arreglo como parámetro y la función **Imprime()** puede usar la información que contiene sin problemas.

Si compilamos y ejecutamos la aplicación obtendremos el siguiente resultado.



**Figura 13.** Podemos observar que la información capturada efectivamente es pasada a la función y ésta la utiliza.

Con esto hemos visto los puntos más importantes de los arreglos y cómo utilizarlos.

## RESUMEN

Los arreglos nos permiten guardar información adentro de un grupo de variables que son todas del mismo tipo y a las que se puede acceder por un nombre en común. Para acceder a un elemento en particular necesitamos usar su número de índice. En los arreglos el índice está basado en cero, es decir que el primer elemento se encuentra en la posición cero, a diferencia de los arrays, que pueden tener varias dimensiones dependiendo de nuestras necesidades. Los arrays de tipo jagged en realidad son arrays de arrays. Éstos nos permiten tener columnas con diferente cantidad de renglones. Al ser clases, los arrays tienen funciones de apoyo que nos facilitan su uso y pueden ser pasados como parámetros a funciones.