Depuración

Hemos llegado al último capítulo del libro y en este capítulo aprenderemos las técnicas básicas de depuración.

La depuración es utilizada para corregir problemas en el programa. Algunas veces los programas tienen errores de sintaxis, pero principalmente de lógica.

Al depurarlos podemos observar su comportamiento de mejor forma y detectar el comportamiento que está dando problemas.

Cómo empezar a depurar un programa **374**

Para corregir los errores de compilación 374

Cómo corregir los errores en tiempo de ejecución 379

Cómo manejar los errores 383

Resumen **389**

Actividades **390**

CÓMO EMPEZAR A DEPURAR UN PROGRAMA

Para depurar un programa nosotros tenemos varias herramientas, pero es importante en primer lugar conocer los posibles errores que podemos tener. Básicamente podemos tener dos clases de errores bien diferenciados: por un lado los **errores de compilación** y por otro los **errores en tiempo de ejecución**.
Los errores de compilación son aquellos que impiden que el programa logre compilarse y generalmente son más fáciles de resolver. Muchos de estos errores se deben a problemas de sintaxis, es decir, son responsabilidad del usuario.
Los errores en tiempo de ejecución son aquellos que suceden cuando el programa se está ejecutando. Estos errores pueden deberse a problemas de lógica, es decir, que el algoritmo no fue diseñado correctamente. Pero también pueden ocurrir debido a un mal control de la información, por ejemplo, podemos tratar de leer más allá de los límites correspondientes a un arreglo determinado.

Para corregir los errores de compilación

La mejor forma de poder aprender a depurar programas es con la práctica. Para esto crearemos un pequeño programa de ejemplo. El siguiente programa tiene varios errores para poder identificar y corregir. En el siguiente bloque de código, los errores han sido colocados en forma deliberada.

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

namespace Cap12_1
{
    class Program
    {
        static void Main(string[] args)
```

III LOS PROBLEMAS DE SINTAXIS

Los problemas de sintaxis son sencillos de corregir, se deben a que se ha escrito algo de forma incorrecta. Algunas veces los nombres de variables se han escrito erróneamente, o hemos olvidado colocar **;** al final de la sentencia. Una revisión rápida de la línea donde está el error nos permite encontrarlo, con la práctica también se reduce la incidencia de estos errores.

```
        {
            // Variables necesarias
            int a = 5

            int b = 10;
            int c = 0;
            int r = 0;

            // Hacemos la division
            r = b / c;

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

            // Mostramos el resultado 5 veces
            for (int n = 0; n < 5; n++)
            {
                Console.WriteLine("El resultado es {0}", r);
            }

            // Invocamos la funcion
            MuestraValor();

        }

        static void MuestraValor(int n)
        {
            Console.WriteLine("El resultado es {0}", n);

    }
}
```

III PARA EVITAR ERRORES CON LOS BLOQUES DE CÓDIGO

Cuando trabajamos con bloques de código un error común es olvidar cerrarlo. Para evitar esto, es buena costumbre cerrar el bloque de código inmediatamente después de abrirlo y luego colocar el código que va a llevar en su interior. Si abrimos y colocamos el código, puede suceder que olvidemos cerrarlo.

El programa tiene varios errores, algunos de ellos muy comunes, por lo que es importante aprender a reconocerlos y resolverlos.
Para comenzar, lo primero que hacemos es llevar a cabo una compilación. Si hay un error de compilación, inmediatamente aparece una ventana. Esta ventana se conoce como lista de errores. Podemos observar esta ventana en la siguiente figura.

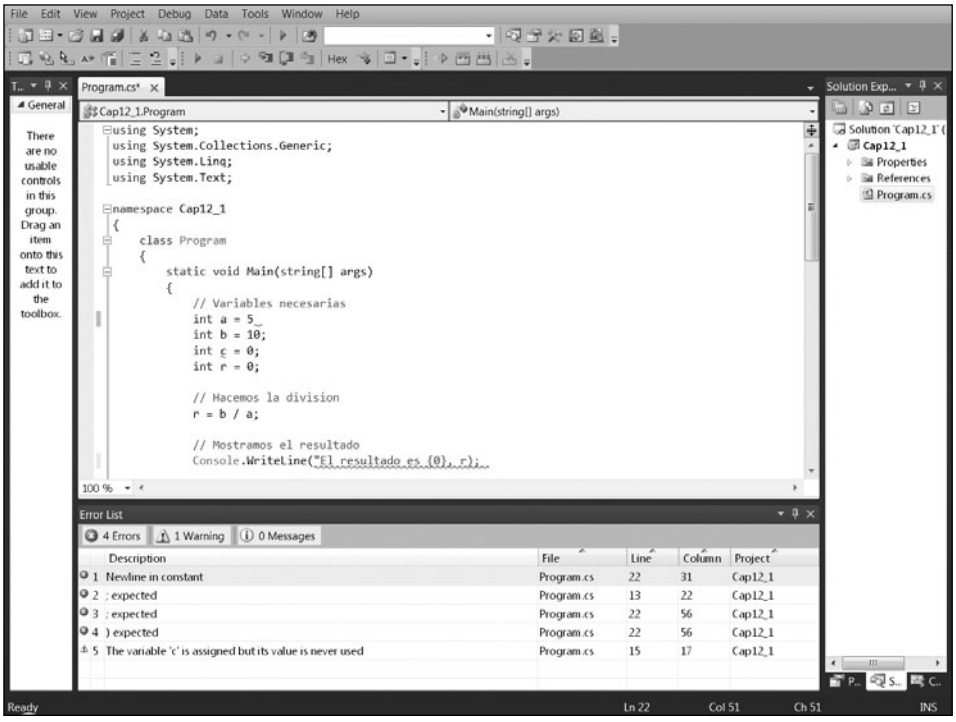

Figura 1. *Después de compilar el programa, podemos encontrar fácilmente la lista de errores.*

La ventana de la lista de errores contiene información importante que nos permite encontrar el error rápidamente. La lista tiene varias columnas. La primera columna nos indica si el elemento listado es un **error** o una **advertencia**. Esto depende del icono mostrado. Un icono con señal amarilla es una advertencia y el icono rojo un error. En la segunda columna tenemos el **número secuencial** de error.
La tercera línea es muy importante, ya que es una **descripción** del error. Esta descripción nos da pistas sobre la posible causa del error. La siguiente columna indica el **documento** en el cual ocurrió el error. Esto es útil cuando tenemos programas que se constituyen en varios documentos. Las dos siguientes líneas nos dicen el **lugar** donde está el posible error indicando la línea y la columna. Un punto importante a tomar en cuenta es que la posición es solamente un indicador. Nuestra última columna indica el **proyecto** en el cual ocurrió el error, esto llega a ser útil cuando tenemos aplicaciones con múltiples proyectos.

Si observamos el editor, veremos que hay texto que aparece con un subrayado rojo. Esto indica el texto que está relacionado con la lista de errores y facilita su localización.
Empecemos a corregir los errores. La depuración es un proceso iterativo, por lo que tenemos que ir corrigiendo errores hasta que no quede ninguno. En algunas ocasiones, cuando corrijamos un error aparecerán listados nuevos errores. No debemos preocuparnos por esto y tenemos que continuar corrigiendo los errores listados hasta que no quede ninguno.
Cuando corregimos los problemas de nuestro programa, debemos referirnos a la lista de errores. Siempre tenemos que empezar con el primer problema. Esto se debe a que a veces un error en la parte superior del programa puede ocasionar que líneas que sean correctas parezcan que tienen problemas.
En la lista debemos dar doble clic al primer error. Esto nos lleva automáticamente al código y señala la parte con el problema. Esto lo podemos observar en el ejemplo que se muestra en la siguiente figura.

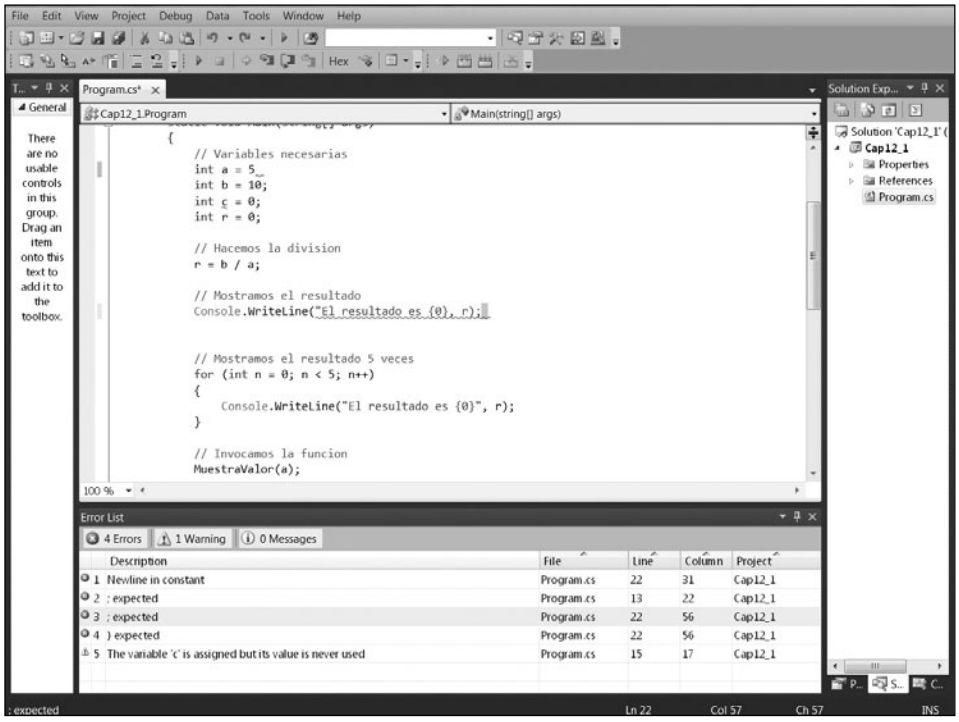

Figura 2. *Podemos observar que la línea donde se encuentra el problema ha sido señalada.*

El problema señalado indica que se tiene una nueva línea en una constante. Esto generalmente ocurre cuando usamos una cadena y olvidamos cerrarla. Si observamos la línea veremos que efectivamente la cadena de formato no se ha cerrado. Procedamos a cerrar la cadena. De tal forma que la línea quede de la siguiente forma:

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

Siempre que corregimos un problema, es bueno realizar una compilación. Esto actualizará la lista de errores y podemos proceder con el siguiente. Después de la compilación nuestra lista de errores se ha reducido a únicamente dos.

El siguiente error que corregiremos es muy común. Si vemos la descripción nos indica que se estaba esperando **}**. Este error ocurre cuando hemos olvidado cerrar un bloque de código. Puede ser a nivel de **namespace**, clase, método, ciclo, etc. La mejor forma de resolverlo es verificar nuestros bloques de código y adicionar el cierre donde corresponde. En muchas ocasiones el cierre no necesariamente es donde lo señala el editor, por lo que se necesita revisar correctamente dónde colocar el cierre. En nuestro caso es muy sencillo y lo adicionamos al final.

Ya cerrado el bloque de código, podemos compilar y ver la lista de errores. Ahora únicamente nos aparece un error en la lista. Inicialmente teníamos cinco, hemos corregido dos y solo queda uno. Estos cambios en la cantidad de errores son muy comunes. Esto se debe a que los errores pueden suceder en cascada, como comentábamos, un error en la parte superior puede crear errores fantasma en otras partes.

El error que tenemos que corregir en este momento es muy sencillo y ocurre cuando olvidamos terminar la sentencia con **;**. Simplemente lo adicionamos en la línea que corresponde. Al parecer ya hemos terminado de corregir los problemas. Compilemos y veamos qué sucede.

Después de corregir el último error, han aparecido dos errores más. Como mencionábamos la depuración es iterativa y podemos pensar que el compilador hace varias pasadas. En este caso se nos presentan dos problemas en los métodos.

El primer error que tenemos nos indica que no se tiene una definición para **WrteLine**. Este tipo de error ocurre generalmente en dos casos. El primero es cuando estamos intentando usar algo, ya sea método, variable, objeto, etc. que no hemos definido. Para corregirlo simplemente definimos lo que necesitamos usar. El segundo caso es más común y se relaciona a un error de escritura, es decir que hemos escrito mal el nombre de lo que queremos utilizar y al no reconocerlo el compilador nos marca este error.

III PARA EVITAR ERRORES DE ESCRITURA

Una forma de evitar errores de escritura para variables y métodos es apoyarnos en la función de auto completar del editor. Esta función nos muestra el elemento que podemos colocar cuando escribimos las primeras letras del nombre. Al dar la tecla de **ENTER**, lo escribe por nosotros. Hacer una buena selección de nombres también ayuda a reducir los errores de escritura.

En nuestro ejemplo, el error está referenciado a un problema de escritura. Simplemente debemos ir al código y escribir **WriteLine** de la forma correcta. Compilemos y continuamos con el siguiente error
Nuevamente nos queda un error. Este error nos dice que no hay un método sobrecargado. Este error suele pasar cuando tenemos una función o método y estamos pasando una cantidad errónea de parámetros o los parámetros son de un tipo no correcto. En nuestro programa vemos que el método **MuestraValor()** necesita de un parámetro, pero en la invocación no se está pasando ningún valor y esto genera el problema. Para resolverlo simplemente debemos colocar el valor a pasar. Por ejemplo podemos colocar la línea de la siguiente forma:

```
// Invocamos la funcion
MuestraValor(a);
```

Ahora podemos compilar nuevamente. La ventana de salida nos indica que se ha compilado exitosamente la aplicación. Esto significa que ya no tenemos errores de compilación, pero aún es posible tener errores en tiempo de ejecución.

Cómo corregir los errores en tiempo de ejecución

Los errores en tiempo de ejecución se hacen notar cuando el programa está corriendo. El error puede provocar que el programa deje de funcionar o se termine repentinamente. En otros casos el programa no da resultados correctos.
Cuando el programa no da resultados correctos, esto puede deberse a que nuestro algoritmo no está adecuadamente implementado. En este caso, lo mejor es primero revisar el análisis y tratar de encontrar el error a nivel de algoritmo. Si encontramos el error en el algoritmo, simplemente debemos corregirlo y luego modificar el programa para que se ejecute de acuerdo con el algoritmo correcto.
En el caso de que nuestro algoritmo esté correcto y el programa nos provee con resultados erróneos entonces es posible que la implementación, es decir la forma como pasamos el algoritmo a programa, este equivocada. La mejor forma de corregir esto es ir depurando paso a paso el programa, viendo cómo cambian los valores y de esta forma encontrar dónde falló la implementación. Las técnicas para llevar a cabo esto las veremos un poco más adelante.
Ahora podemos ver los errores en tiempo de ejecución que provocan la finalización inesperada del programa. Muchos de estos errores se deben a que hemos realizado una operación indebida con nuestro programa. A los errores de este tipo se les llama **excepciones** y cuando ocurren se dice que se han **levantado** o **aventado**.
Veamos cómo ocurren estos errores. Simplemente ejecutemos el programa anterior. El programa está listo para levantar una excepción.

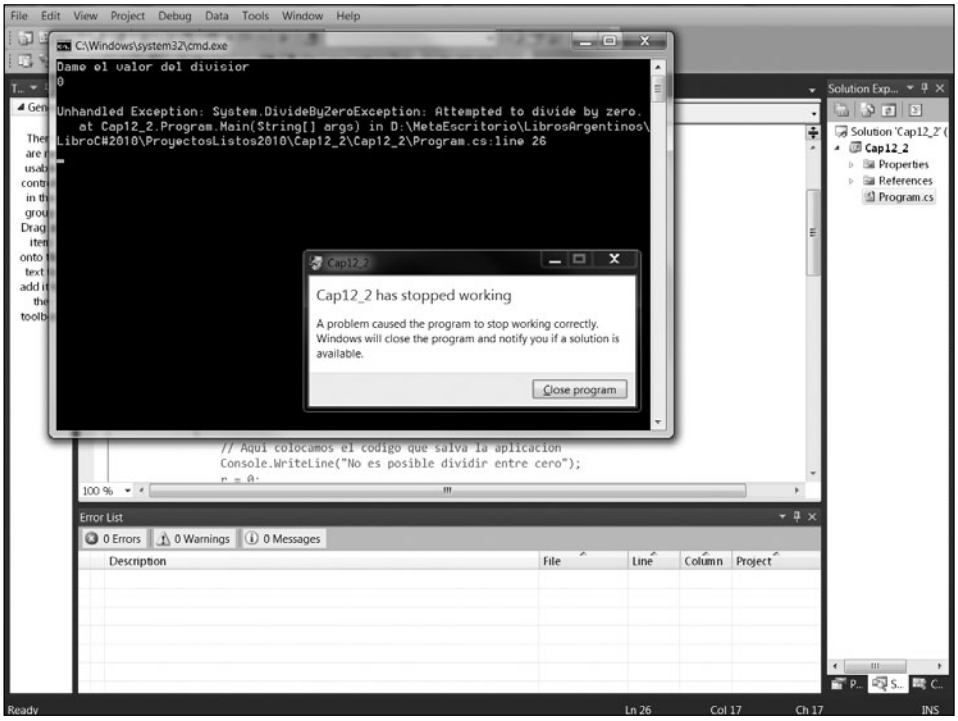

Figura 3. *En cuanto el programa levanta una excepción, finaliza su ejecución de forma inesperada y nos presenta un diálogo con información.*

Como vemos el programa ha levantado una excepción. Para poder depurar el programa debemos ejecutarlo en modo de depuración. Para esto seleccionamos el menú de **Debug** o **Depurar** y presionamos **Start Debugging**.

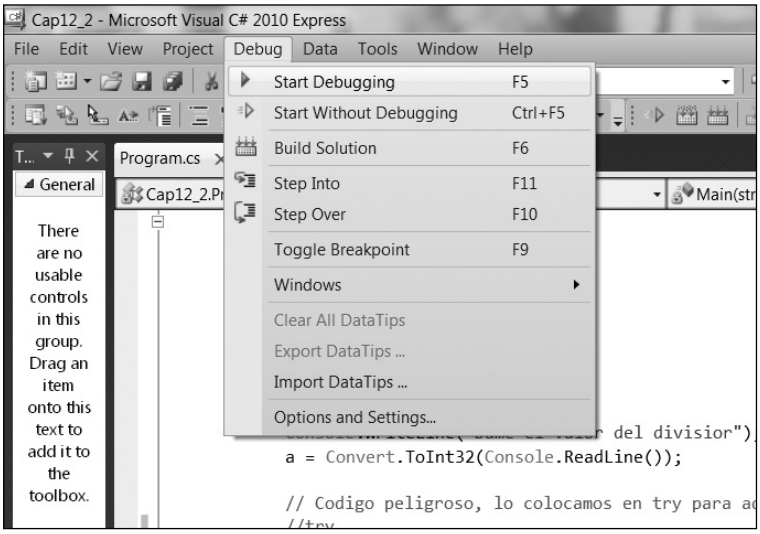

Figura 4. *Aquí podemos seleccionar dónde llevar a cabo la depuración.*

Ya que nos encontramos en el depurador, el lugar donde se generó la excepción aparece marcado en forma muy clara, y junto a él tendremos un cuadro de diálogo que contiene la información correspondiente.

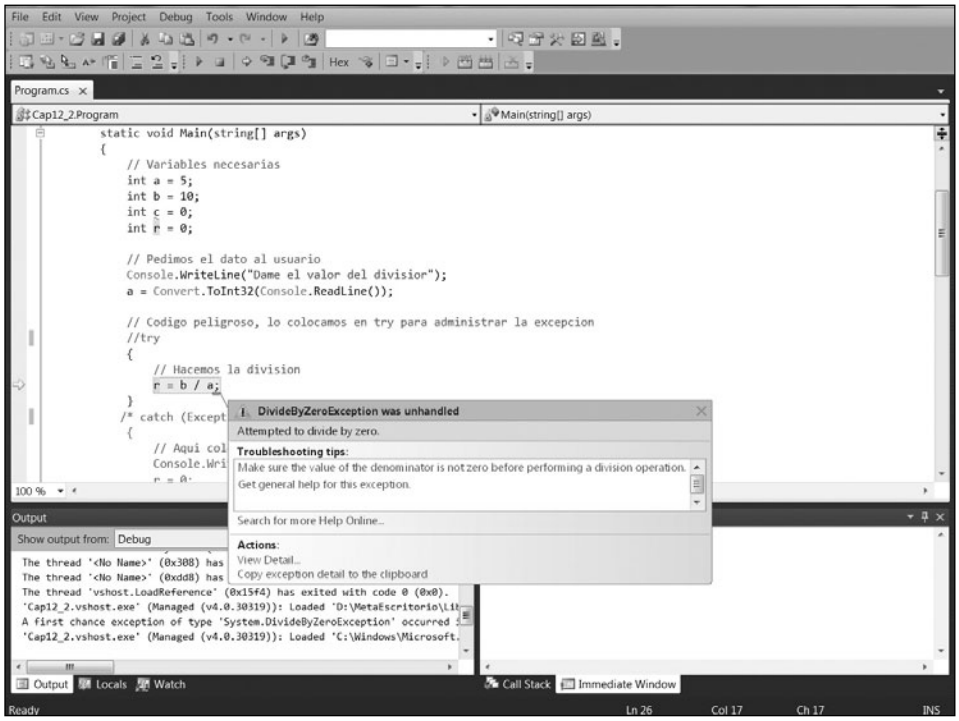

Figura 5. *El cuadro de dialogo nos indica el tipo de excepción que se levantó.*

Si observamos el cuadro de diálogo veremos que la excepción, es decir el error, fue provocado por un intento de dividir entre cero. La división entre cero no está definida y por eso es una operación inválida. Ahora que ya sabemos qué provoca el problema, simplemente podemos corregir el código.
Podemos pensar que la corrección es sencilla y simplemente dividir entre la variable que tiene un valor de cero, puede ser algo como lo que se muestra en el siguiente bloque de código, a modo de ejemplo:

III LA DESCRIPCIÓN DEL ERROR

Debemos acostumbrarnos a leer la descripción del error, es un punto muy importante. Con el tiempo nos acostumbraremos a la descripción y sabremos inmediatamente qué hacer para resolver el problema. Muchos programadores con malos hábitos solamente van a la línea del error, pero no leen la descripción del error. Esto los lleva a tiempos más largos para resolverlo.

```
// Hacemos la division
r = b / a;
```

Si procedemos con la compilación, no aparece ningún problema, pero veamos qué sucede cuando ejecutemos nuevamente el programa.

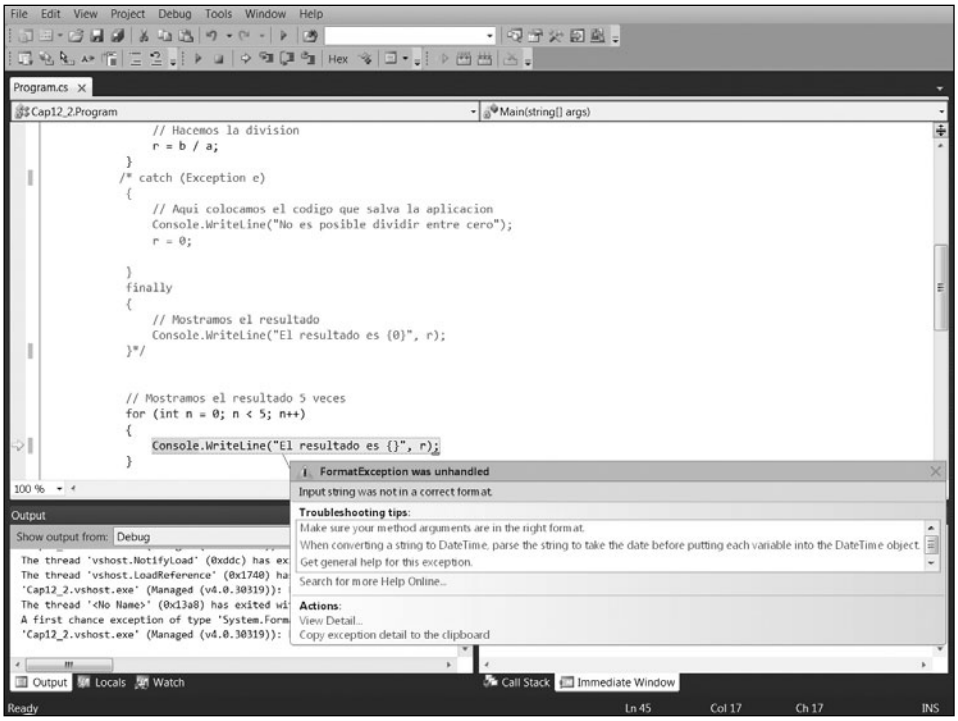

Figura 6. *Una nueva excepción ha ocurrido en el programa.*

El programa nuevamente tiene problemas de ejecución. Si observamos la información que nos provee el depurador inmediatamente nos damos cuenta de que la excepción ha ocurrido porque no estamos colocando el formato correcto en la cadena. Al ver la línea de código podemos identificar que efectivamente el formato es incorrecto, la línea con el formato adecuado debe ser como la siguiente:

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

Después de corregir, nuevamente debemos compilar y ejecutar. Todo debe estar bien y el programa puede ejecutarse sin problema. El resultado lo vemos en el ejemplo que se muestra en la siguiente figura:

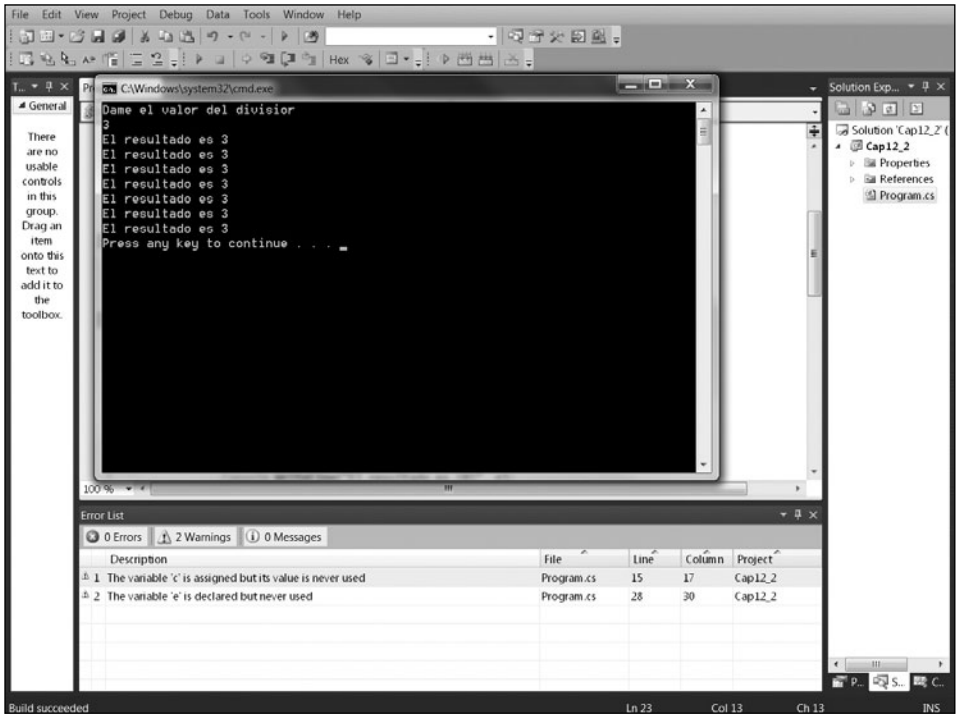

Figura 7. *El programa por fin se ejecuta y podemos ver en la consola los resultados.*

Ya vimos cómo localizar el código que nos genera excepciones. Pero lo mejor sería poder evitarlas o controlarlas de alguna manera.

Cómo manejar los errores

No es posible evitar las excepciones completamente. Esto se debe a que no siempre tenemos control sobre la información o el manejo que va a tener el usuario en nuestro programa. Por ejemplo, en el programa anterior, la excepción se eliminó al dividir entre la variable que no tiene el valor de cero. Pero imaginemos que el usuario es el que debe colocar el valor del divisor. En este caso no podemos forzar a una variable con un valor diferente de cero. En algunas ocasiones el usuario colocará un valor adecuado, pero también es posible que dé el valor de cero. En esos casos, el programa terminará por el error.

Como no podemos preveer todas las excepciones que pueden ocurrir, lo mejor que podemos hacer es tener una forma de manejar o administrar los errores. Esto permitirá que el programa detecte la excepción y entonces puede hacer algo para salvar la ejecución de programa.

C# nos permite hacer esto ya que provee un sistema de manejo estructurado de excepciones. Para ver cómo poder lograr esto, modificaremos el programa y le daremos el manejo de las excepciones. El programa modificado queda así:

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

namespace Cap12_2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Variables necesarias
            int a = 5;
            int b = 10;
            int c = 0;
            int r = 0;

            // Pedimos el dato al usuario
            Console.WriteLine("Dame el valor del divisior");
            a = Convert.ToInt32(Console.ReadLine());

            // Hacemos la division
            r = b / a;

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


            // Mostramos el resultado 5 veces
            for (int n = 0; n < 5; n++)
            {
                Console.WriteLine("El resultado es {0}", r);
            }

            // Invocamos la funcion
            MuestraValor(a);

        }

        static void MuestraValor(int n)
        {
```

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

        }
    }
}
```

Veamos cómo este programa puede ejecutarse correctamente a veces y con problemas en otros casos. Si ejecutamos el programa y damos como valor **5**, el programa se ejecuta correctamente. Cuando ejecutemos el programa nuevamente, si el valor que damos es cero, entonces la excepción ocurrirá. Esto lo vemos claramente en las dos figuras que se mostrarán a continuación.

Es sencillo saber qué métodos pueden generar excepciones. Simplemente debemos ir a MSDN y buscar la documentación del método a utilizar. Dentro de esta documentación hay una sección que describe las posibles excepciones que pueden ocurrir. De esta forma hacemos que nuestro programa las evite.

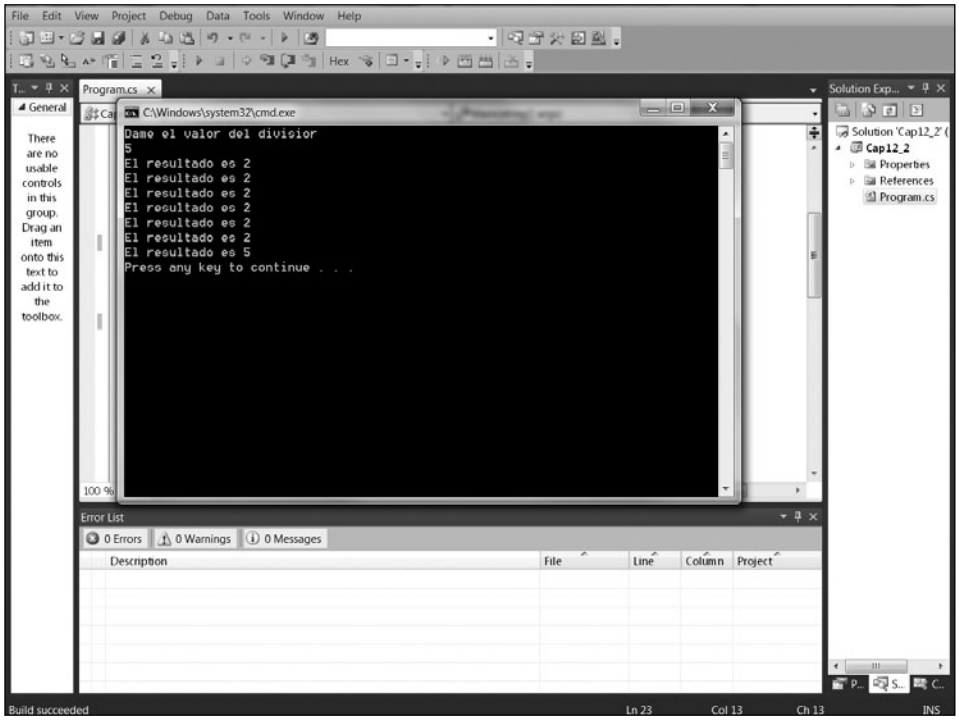

Figura 8. *En esta figura observamos el caso en el cual la aplicación funciona correctamente.*

Para llevar a cabo la administración de excepciones vamos a tener tres bloques de código conocidos como: **try**, **catch** y **finally**.

Lo primero que tenemos que hacer es detectar dónde se encuentra el código peligroso, es decir, dónde puede ocurrir la excepción. En nuestro caso es fácil de identificar ya que sabemos que es la división.

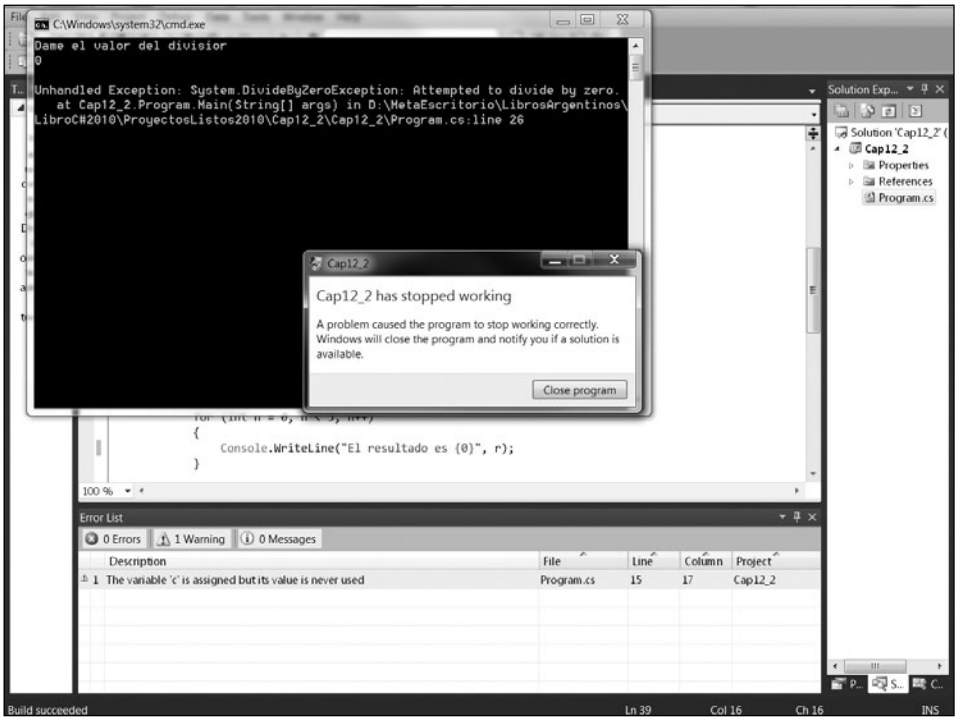

Figura 9. *Ahora podemos observar cómo la excepción se ha levantado y el programa finaliza su ejecución.*

El código que tiene el riesgo de levantar la excepción debe ser colocado en el bloque de **try**. Por ejemplo, en nuestro caso colocaremos lo siguiente:

```
// Codigo peligroso, lo colocamos en try para
      administrar la excepcion
```

III NO EXAGERAR CON LAS EXCEPCIONES

Existen muchos métodos y sentencias que pueden generar excepciones. Pero no es bueno exagerar. Una gran cantidad de excepciones se evitan simplemente al usar correctamente los métodos y hacer buen uso de nuestras variables y sus valores. Solamente debemos implementar la administración de excepciones en las partes del programa que realmente lo necesiten.

```
try
{
    // Hacemos la division
    r = b / a;
}
```

El código adentro de **try** tiene dos opciones: levantar o no la excepción. Si la excepción se levanta entonces en el bloque de **catch** podemos colocar el código que se encargue de evitar que el programa termine inesperadamente. Aquí podemos poner código que corrija lo ocurrido.
En nuestro caso, no hay mucho que hacer, simplemente podemos enviar un mensaje al usuario y dar un valor adecuado a la variable de resultado.

```
catch (Exception e)
{
    // Aqui colocamos el codigo que salva la a
              plicacion
   Console.WriteLine("No es posible dividir entre cero");
    r = 0;

}
```

Vemos que este bloque tiene un parámetro llamado e. Adentro de este parámetro podemos encontrar información relacionada con la excepción. Lo que coloquemos en el interior del bloque depende de la lógica de nuestra aplicación. Si no hubiera excepción el código adentro del bloque **catch** no se ejecuta.
Tenemos otro bloque llamado **finally**. Este bloque es opcional. El código que se coloque en el bloque **finally** se ejecuta siempre sin importar si la excepción se llevó a cabo o no. Siempre se ejecuta una vez que **try** o **catch** han finalizado de ejecutarse. **Finally** puede ser utilizado para limpiar o liberar cualquier recurso que se haya utilizado en **try** o **catch**. En nuestro caso puede utilizarse simplemente para darle al usuario el valor del resultado.

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

Nuestro programa completo con la administración de la excepción es el siguiente:

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

namespace Cap12_2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Variables necesarias
            int a = 5;
            int b = 10;
            int c = 0;
            int r = 0;

            // Pedimos el dato al usuario
            Console.WriteLine("Dame el valor del divisior");
            a = Convert.ToInt32(Console.ReadLine());

            // Codigo peligroso, lo colocamos en try para
                  administrar la excepcion
            try
            {
                // Hacemos la division
                r = b / a;
            }
            catch (Exception e)
            {
               // Aqui colocamos el codigo que salva la aplicacion
               Console.WriteLine("No es posible dividir entre cero");
                r = 0;

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

```
            }


            // Mostramos el resultado 5 veces
            for (int n = 0; n < 5; n++)
            {
                Console.WriteLine("El resultado es {0}", r);
            }

            // Invocamos la funcion
            MuestraValor(a);

        }

        static void MuestraValor(int n)
        {
            Console.WriteLine("El resultado es {0}", n);

        }
    }
}
```

Veamos si esto funciona adecuadamente. Compilemos el programa y al ejecutarlo vamos a dar el valor de cero. El programa ahora no deberá generar problemas, pues hemos capturado la excepción y ejecutado código que evita que el programa finalice de forma no adecuada.

... RESUMEN

La depuración nos permite corregir los problemas que pueda tener nuestro programa. Los errores de compilación impiden que el programa pueda ser compilado y muchas veces son simples errores de sintaxis. Tenemos una ventana que nos muestra la lista de errores, siempre debemos corregir el primer error de la lista, ya que muchos errores se generan en cascada. La depuración es un proceso iterativo. Si tenemos código peligroso que puede generar excepciones, es recomendable administrar ese código para evitar que la excepción termine con nuestro programa, cuando administramos la excepción, podemos colocar código que salve al programa.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es la depuración?

2 ¿Cuántos tipos de errores tenemos?

3 ¿Por qué algunos programas no pueden compilar?

4 ¿Qué es un error de sintaxis?

5 ¿En dónde podemos ver los errores que tenemos?

6 ¿Qué son los errores en tiempo de ejecución?

7 ¿Qué es un error de lógica?

8 ¿Qué es una excepción?

9 ¿Qué es la administración de excepciones?

10 ¿Cuáles son los bloques de código para administrar la excepción?

11 ¿Qué es un punto de interrupción?

12 ¿Cómo podemos depurar paso a paso?

EJERCICIOS PRÁCTICOS

1 Usar la depuración paso a paso para observar cómo cambia el valor de la variable en el programa del factorial.

2 Utilizar el método **WriteLine()** de la clase **Debug** para que las funciones nos indiquen cuando entramos y salimos de ellas.

3 Buscar en MSDN cuáles son las excepciones que pueden ocurrir con los arreglos.

4 Buscar en MSDN cuáles son las excepciones que pueden ocurrir con los streams.

5 Buscar en MSDN cuáles son las excepciones que pueden ocurrir con el método **WriteLine()**.