C# para programadores C++

Introducción

Este apéndice está destinado a aquellos programadores que están muy familiarizados con C++ y desean ver cuáles son las diferencias entre C++ y C#. Repasaremos el lenguaje C#, destacando específicamente aquellas áreas en las que C# se diferencia de C++. Dado que ambos lenguajes comparten una gran cantidad de sintaxis y metodología, los programadores de C++ pueden utilizar este apéndice como una vía rápida para el aprendizaje de C#.

Debe quedar claro desde el principio que C# es un lenguaje diferente a C++. Mientras que C++ fue diseñado para la programación orientada a objetos de propósito general en los días en que el ordenador típico era una máquina independiente que ejecutaba una interfaz de usuario basada en línea de comandos, C# ha sido diseñado específicamente para trabajar en la Plataforma .NET, y está orientado al entorno moderno de Windows y a las interfaces de usuario controladas mediante ratón, las redes e Internet. Sin embargo, es innegable que ambos lenguajes son muy similares tanto en su sintaxis como en que ambos han sido diseñados para facilitar el mismo paradigma de programación, en el que el código se estructura alrededor de jerarquías de clases heredadas. Esta similitud no es sorprendente dado que, como hemos destacado a lo largo de este libro, C# ha sido en gran medida diseñado como un lenguaje orientado a objetos que mantiene las bondades de los lenguajes orientados a objetos anteriores a él, de los cuales C++ es probablemente el ejemplo más exitoso hasta el presente, a la vez que aprende de sus errores de diseño.

Debido a las similitudes entre los dos lenguajes, los desarrolladores que dominan C++ pueden encontrar que la vía más rápida de aprender C# es tratarlo como si fuera C++ con unas pocas diferencias y aprender cuáles son esas diferencias. Este apéndice está diseñado para facilitar esa tarea. Comenzaremos por una presentación general que menciona cuáles son las diferencias principales entre los dos lenguajes a la vez que indica cuáles son los elementos que comparten en común. A continuación, mostraremos la apariencia del clásico programa 'Hello, World' en cada uno de los dos lenguajes. Por último, el grueso de este apéndice se dedica a un análisis punto por punto que ofrece una comparación detallada entre C# y C++ en cada una de las áreas principales de los lenguajes. Inevitablemente, un apéndice de este tamaño no puede ser exhaustivo, sino que ha sido diseñado para cubrir las diferencias fundamentales entre los lenguajes que Ud. encontrará en la programación diaria. Debemos notar, en todo caso, que C# depende fuertemente del soporte que le ofrece la librería de clases base

.NET en diversas áreas. En este apéndice limitaremos nuestra atención al lenguaje C# en sí, y no cubriremos extensamente las clases base.

Para la comparación, tomaremos ANSI C++ como punto de referencia. Microsoft ha añadido numerosas extensiones a C++, y el compilador de Visual C++ presenta unas pocas incompatibilidades con el estándar ANSI que señalaremos ocasionalmente; pero normalmente no haremos mención de ellas al comparar los dos lenguajes.

Convenios utilizados en este apéndice

En este apéndice adoptaremos un convenio adicional para mostrar el código. El código C# siempre se muestra del mismo modo que en el resto del libro, con sombreado en gris:

```
// este es código C#
class MyClass : MyBaseClass
{
```

Si deseamos resaltar cualquier código C# nuevo o importante, lo mostraremos en negrita:

```
// este es código C#
class MyClass : MyBaseClass        // ya hemos visto esto
{
   int X;                          // esto es interesante
```

El código C++ que se muestra para la comparación se presenta así:

```
// esto es código C++
class CMyClass : public CMyBaseClass
{
```

En el código de los ejemplos de este apéndice también hemos tenido en cuenta los convenios de nombres más comunes para los dos lenguajes bajo Windows. Por esa razón, los nombres de clases en los ejemplos C++ comienzan con 'C', mientras que los ejemplos correspondientes de C# no. Asimismo, la notación húngara se utiliza frecuentemente para los nombres de variables únicamente en los ejemplos C++.

Terminología

Ud. deberá tener en cuenta que en algunas construcciones del lenguaje se utiliza diferente terminología en C# con relación a C++. Las variables miembros de C++ se conocen en C# como **campos**, mientras que las funciones miembros de C++ se conocen como **métodos** en C#. En C#, el término **función** tiene un significado más general, y se refiere a cualquier miembro de una clase que contiene código. Ello significa que el término **función** cubre los métodos, propiedades, constructores, destructores, indizadores y sobrecargas de operadores. En C++, los términos 'función' y 'método' se utilizan en la conversación de forma intercambiable, aunque estrictamente hablando un método de C++ es una función miembro virtual.

Si le surge alguna confusión al respecto, la siguiente tabla deberá serle útil:

Significado	Término C++	Término C#
Variable que es miembro de una clase	Variable miembro	Campo
Cualquier miembro de una clase que contiene código	Función (o función miembro)	Función
Cualquier miembro de una clase que contiene código y que puede ser	Función (o función	Método

llamado por su nombre con la sintaxis `HacerAlgo(/*parámetros*/).`	miembro)	
Función virtual que se define como miembro de una clase	Método	Método virtual

Ud. también debe tener en cuenta las diferencias entre otro par de términos:

Término C++	**Término C#**
Instrucción compuesta	Instrucción de bloque
valor-l[1]	expresión-variable

En este apéndice utilizaremos siempre que sea posible la terminología asociada al lenguaje sobre el que estemos hablando.

Una comparación entre C# y C++

En esta sección resumiremos brevemente las diferencias y similitudes generales entre ambos lenguajes.

Diferencias

Las principales áreas en las que difieren C# y C++ son las siguientes:

- **Destino de la compilación** – El código C++ normalmente se compila a lenguaje ensamblador. Por el contrario, C# se compila a un **lenguaje intermedio (IL)**, que presenta alguna similitud con los códigos de bytes de Java. El lenguaje intermedio es convertido a continuación en código de máquina ejecutable mediante un proceso de compilación JIT. El código intermedio que se emite es almacenado en un fichero o conjunto de ficheros conocido como **unidad de ensamblaje** (**assembly**). Una unidad de ensamblaje conforma esencialmente la unidad en la que se empaqueta el IL, correspondiendo a una DLL o fichero ejecutable que sería creado por un compilador de C++.
- **Gestión de memoria** – C# ha sido diseñado para liberar al programador de las tareas relacionadas con la gestión de la memoria. Ello significa que en C# no es necesario liberar explícitamente la memoria que ha sido solicitada dinámicamente, como se haría en C++. En lugar de eso, un recolector de basura recupera periódicamente la memoria que deja de ser necesaria. Con vistas a facilitar este proceso, C# impone ciertas restricciones a cómo Ud. puede utilizar las variables que se almacenan en memoria dinámica, y es más estricto en la verificación de los tipos que C++.
- **Punteros** – Los punteros pueden utilizarse en C# del mismo modo que en C++, pero únicamente dentro de bloques de código que hayan sido específicamente marcados para ello. La mayor parte del tiempo C# se apoya en referencias al estilo de Java o VB para las instancias de clases, y el lenguaje ha sido diseñado de forma tal que los punteros no son necesarios con tanta frecuencia como lo son en C++.
- **Sobrecarga de operadores** – C# no permite la sobrecarga explícita de tantos operadores como C++. Esto se debe fundamentalmente a que el compilador de C# automatiza esta tarea hasta cierto punto utilizando cualquier sobrecarga definida para operadores elementales (como `=`) para resolver automáticamente la sobrecarga de operadores combinados (`+=`).

[1] La 'l' viene de 'left' (izquierda). Un valor-l (*l-value* en inglés) es una expresión que produce una referencia a memoria, y por tanto puede situarse en el lado izquierdo de una instrucción de asignación (N. del T.)

- **Librerías** – Tanto C++ como C# se apoyan en la existencia de una librería bastante amplia. En el caso de ANSI C++, estamos hablando de la librería estándar. C# se apoya en un conjunto de clases conocidas como las **clases base .NET**. Las clases base .NET están basadas en la herencia simple, mientras que la librería estándar de C++ se apoya en una mezcla de herencia y plantillas. Además, mientras que ANSI C++ mantiene la librería separada del lenguaje en sí, la interdependencia en C# es mucho mayor, y la implementación de muchas palabras reservadas de C# depende directamente de clases base particulares.
- **Plataformas de destino** – C# ha sido diseñado explícitamente para satisfacer las necesidades de programación en entornos dotados de interfaz gráfica de usuario (GUI) – no necesariamente Windows, aunque el lenguaje está disponible actualmente sólo para Windows, así como para el desarrollo de servicios de segundo plano, tales como servicios web. Esto realmente no afecta al lenguaje en sí, pero se refleja en el diseño de la librería de clases. Por el contrario, C++ fue diseñado para un uso más general en los días cuando las interfaces de usuario basadas en línea de comandos eran predominantes. Ni C++ ni la librería estándar incluyen ningún tipo de soporte para elementos de interfaz gráfica. En Windows, los programadores de C++ han tenido que apoyarse directa o indirectamente en el API de Windows para obtener ese soporte.
- **Directivas para el preprocesador** – C# incluye algunas directivas para el preprocesador, que siguen la misma sintaxis general de C++. Pero en general hay menos directivas para el preprocesador en C#, porque otras características del lenguaje hacen que aquellas sean menos importantes.
- **Enumeradores** – Están presentes en C# y son mucho más versátiles que sus equivalentes de C++, dado que son sintácticamente estructuras de pleno derecho, y soportan varias propiedades y métodos. Tenga en cuenta que este soporte existe a nivel de código fuente únicamente – cuando son compilados a código nativo, los enumeradores se implementan aún como tipos numéricos primitivos, para que no haya pérdida de rendimiento.
- **Destructores** – C# no garantiza cuándo serán llamados los destructores. En general, no se debe utilizar en C# el paradigma de programación basado en colocar código en los destructores de las clases C#, como se hace en C++, a menos que ello sea necesario para liberar recursos externos específicos, tales como ficheros o conexiones a bases de datos. Debido a que el recolector de basura recupera toda la memoria reservada dinámicamente, los destructores no son tan importantes en C# como lo son en C++. Para aquellos casos en que es importante liberar los recursos externos lo más rápidamente posible, C# implementa un mecanismo alternativo asociado a la interfaz `IDisposable`.
- **Clases vs. estructuras** – C# formaliza la diferencia entre las clases (generalmente utilizadas para objetos grandes con muchos métodos) y las estructuras (generalmente utilizadas para objetos pequeños que contienen poco más que algunas variables). Las clases y las estructuras se almacenan de forma diferente, no se permite la herencia de estructuras, y hay algunas otras diferencias más.

Similitudes

Entre las áreas en las que C# y C++ son muy similares podemos destacar:

- **Sintaxis** – En general, la sintaxis de C# es muy similar a la de C++, aunque existen numerosas diferencias menores.
- **Flujo de ejecución** – C++ y C# ofrecen prácticamente las mismas instrucciones de flujo de control, y generalmente su semántica es la misma en ambos lenguajes.
- **Excepciones** – El soporte para excepciones en C# es esencialmente idéntico al de C++, con la diferencia de que C# permite bloques `finally` e impone restricciones sobre el tipo de los objetos que pueden ser lanzados.
- **Modelo de herencia** – Las clases se heredan de la misma forma en C# que en C++. Los conceptos relacionados con la herencia, tales como clases abstractas y funciones virtuales se implementan de

la misma manera en ambos lenguajes, aunque existen algunas diferencias de sintaxis. Igualmente, C# soporta únicamente la herencia simple de clases. La similitud en las jerarquías de clases implica que los programas C# tendrán normalmente una estructura muy similar a la de los programas C++ correspondientes.

- **Constructores**. Funcionan del mismo modo en C# y en C++, aunque nuevamente hay ligeras diferencias de sintaxis.

Nuevas características

C# introduce varios conceptos nuevos que no forman parte de la especificación ANSI C++ (aunque la mayoría de ellos han sido introducidos por Microsoft como extensiones no estándar soportadas por el compilador de Microsoft C++). Estos son:

- **Delegados** – C# no soporta los punteros a funciones. Sin embargo, un efecto similar se obtiene encapsulando las referencias a métodos en un tipo especial de clase conocida como delegado. Los delegados pueden ser pasados de un método a otro, y utilizados para llamar a los métodos a los que ellos se refieren, de la misma forma en que se hacen llamadas a funciones a través de punteros en C++. Lo más significativo en relación con los delegados es que estos incorporan una referencia a un objeto a la vez que una referencia a un método. Eso significa que, a diferencia de un puntero a función, un delegado contiene información suficiente para llamar a un método de instancia para un objeto específico.
- **Eventos** – Los eventos son similares a los delegados, pero están diseñados específicamente para dar soporte al modelo de función de respuesta, en el que un cliente notifica a un servidor que desea ser informado cuando una acción específica tenga lugar. C# utiliza los eventos para encapsular los mensajes de Windows de forma muy similar a como lo hace VB.
- **Propiedades** – Esta idea, utilizada ampliamente en VB y en COM, ha sido importada a C#. Una propiedad es una pareja de métodos get/set en una clase que han sido disfrazados sintácticamente para que parezcan ante el mundo exterior como un campo. Las propiedades permiten escribir código como `MyForm.Height = 400` en lugar de `MyForm.SetHeight(400).`
- **Interfaces** – Una interfaz puede verse como una clase abstracta, cuyo propósito es definir un conjunto de métodos o propiedades que las clases pueden comprometerse a implementar. La idea tiene su origen en COM. Sin embargo, las interfaces de C# no son idénticas a las interfaces de COM – son simplemente listas de métodos, etc., mientras que las interfaces de COM pueden tener otras características asociadas, tales como GUIDs, pero el principio es muy similar. Esto significa que C# formalmente reconoce el principio de la herencia de interfaces, mediante la cual una clase hereda las definiciones de funciones, pero no sus implementaciones.
- **Atributos** – C# permite al programador decorar las clases, métodos, parámetros y otros elementos de código con meta-información conocida como atributos. Los valores de los atributos pueden ser obtenidos en tiempo de ejecución y utilizados para determinar las acciones a ejecutar.

Nuevas características en las clases base

Las siguientes características son nuevas a C# y no tienen un equivalente en el lenguaje C++. Sin embargo, el soporte para estas características proviene completamente de las clases base, sin ningún o casi ningún soporte en la sintaxis del lenguaje C# en sí, y no las trataremos en detalle en este apéndice. Todos los detalles se describen el Capítulo 7.

- **Múltiples hilos** – El lenguaje C# ofrece algún soporte para la sincronización de hilos mediante la instrucción `lock`. (C++ no ofrece soporte intrínseco para la gestión de hilos, para lo cual se debe utilizar librerías de código externas).
- **Reflexión** – C# permite al código obtener dinámicamente información relativa a las definiciones de clases almacenadas en unidades de ensamblaje compiladas (librerías y ejecutables). De hecho, ¡Ud.

puede escribir un programa en C# que visualice información sobre las clases y métodos que lo componen!

Características no soportadas

Las siguientes partes de C++ no tienen un equivalente en C#:

- **Herencia múltiple de implementación de clases** – Las clases C# soportan la herencia múltiple únicamente de interfaces.
- **Plantillas** – Actualmente no forman parte del lenguaje C#, pero Microsoft está investigando la posibilidad de añadir soporte para plantillas en versiones futuras de C#.

El ejemplo "Hello World"

Escribir un programa que imprima 'Hello World' es en el mundo de la programación casi lo menos original que puede haber. Pero una comparación directa de programas 'Hello World' en C++ y C# puede ser muy instructiva para ilustrar algunas de la diferencias entre los dos lenguajes. En esta comparación hemos intentado innovar un poco (y mostrar más características) presentando el mensaje `Hello World` tanto en la línea de comandos como en un cuadro de mensaje. Hemos hecho también un ligero cambio al texto del mensaje en la versión C++, algo que debe ser interpretado como una broma y no como una aseveración seria.

La versión C++ tiene la siguiente apariencia:

```
#include <iostream>
#include <Windows.h>
using namespace std;

int main(int argc, char *argv)
{
   cout << "Goodbye, World!";
   MessageBox(NULL, "Goodbye, World!", "", MB_OK);
   return 0;
}
```

Esta es la versión C#:

```
using System;
using System.Windows.Forms;

namespace Console1
{
   class Class1
   {
      static int Main(string[] args)
      {
         Console.WriteLine("Hello, World!");
         MessageBox.Show("Hello, World!");
         return 0;
      }
   }
}
```

La comparación de los dos programas nos indica que la sintaxis de ambos lenguajes es muy similar. En

particular, los bloques de código se delimitan mediante llaves `{ }`, y el carácter punto y coma se utiliza como terminador de instrucciones. Al igual que en C++, C# ignora los espacios en blanco entre las instrucciones.

Recorreremos los ejemplos línea a línea, examinando las características que ellas demuestran:

Directiva #include

La versión C++ de 'Hello World!' comienza con un par de directivas para el preprocesador que le ordenan incluir algunos ficheros de cabecera.

```
#include <iostream>
#include <Windows.h>
```

Estas líneas no aparecen en la versión C#, algo que ilustra un detalle importante en relación con el modo en que C# accede a las librerías. En C++ debemos incluir los ficheros de cabecera para que el compilador sea capaz de reconocer los símbolos relevantes del código. De forma independiente deberemos dar orden al enlazador de asociar las librerías – algo que se logra mediante parámetros de línea de comandos que se pasan al enlazador. C# no separa realmente las fases de compilación y enlace del modo en que lo hace C++. En C#, los parámetros de la línea de comandos son lo único necesario (y sólo en el caso de que esté Ud. accediendo a algo que no forma parte de la librería básica). Estos permiten al compilador encontrar todas las definiciones de clases; de aquí que las referencias explícitas en el código fuente sean innecesarias. Esta es una manera mucho más simple de trabajar – y de hecho, una vez que Ud. se ha acostumbrado al modelo de C#, la versión C++, donde todo debe ser referido dos veces, comienza a parecer extraño y engorroso.

Otro elemento a destacar aquí es que de las dos directivas `#include` en el código C++ anterior, la primera incluye una librería estándar ANSI (el módulo `iostream` de la librería estándar). La segunda se refiere a una librería específica de Windows, y es necesaria sólo para hacer posible la presentación del cuadro de mensaje. El código C++ para Windows frecuentemente necesita acceder a la API de Windows porque el estándar ANSI no incluye recursos para el manejo de ventanas. Por el contrario, las clases base .NET – el equivalente C# de la librería ANSI – sí ofrece tales facilidades, y sólo las clases base .NET son utilizadas aquí. Nuestro código C# no necesita ninguna característica no estándar (aunque podría objetarse que esto se debe al hecho de que el C# 'estándar' sólo está disponible en Windows en el presente).

Aunque el código C# anterior no incluye ninguna directiva `#include`, vale la pena señalar que en C# están disponibles algunas directivas para el preprocesador (aunque no `#include`), y mantienen la sintaxis de C++.

Espacios de nombres

El programa 'Hello, World!' de C# comienza con una declaración de espacio de nombres, cuyo alcance es el programa entero. Los espacios de nombres funcionan en C# exactamente de la misma forma que en C++, ofreciendo un mecanismo para eliminar posibles ambigüedades en los nombres de los símbolos de un programa. Colocar a los elementos en un espacio de nombres es opcional en ambos lenguajes, pero en C# el convenio es que todos los elementos deben formar parte de un espacio de nombres. Por lo tanto, aunque es muy común encontrar código C++ no contenido en ningún espacio de nombres, es extremadamente raro encontrar eso en C#.

En la siguiente parte del código, las versiones C# y C++ son muy similares – en ambas se utiliza la instrucción `using` para indicar el espacio de nombres en que cualquier símbolo debe ser buscado. La única diferencia es sintáctica: la instrucción en C# es solamente `namespace`, mientras que en C++ es `using namespace`.

> *Muchos programadores de C++ estarán acostumbrados a la vieja librería de C++, por lo que incluirán el fichero `iostream.h` en lugar de `iostream` – en cuyo caso la instrucción* `using namespace std;` *es innecesaria. La vieja librería de C++ está oficialmente considerada obsoleta y no será soportada a partir de Visual Studio 8 (la versión que seguirá a Visual Studio .NET). El código del ejemplo anterior muestra la forma correcta de incluir la*

librería `iostream` *en el código C++.*

El punto de entrada: Main() vs. main()

El siguiente elemento a analizar en nuestros ejemplos "Hello World" son los puntos de entrada de los programas. En el caso de C++, se trata de una función global llamada `main()`. C# hace más o menos lo mismo, aunque en este caso el nombre es `Main()`. Sin embargo, mientras que en C++ la función `main()` se define fuera de cualquier clase, en la versión C# se debe definir como un miembro estático de una clase. Esto se debe a que C# exige que todas las funciones y variables sean miembros de una clase o estructura. C# no admite otros elementos en el nivel más externo que no sean clases o estructuras. En ese sentido, puede decirse que C# refuerza el uso de prácticas de orientación a objetos más estrictas que C++. En cualquier caso, apoyarse excesivamente en el uso de variables o funciones globales y estáticas en el código C++ es considerado generalmente como un diseño de programas inadecuado.

Por supuesto, exigir que todo deba ser miembro de alguna clase trae a discusión el tema de dónde debe estar situado el punto de entrada de un programa. La respuesta es que el compilador de C# buscará un método estático que tenga el nombre `Main()`. Esta función puede ser miembro de cualquiera de las clases que conforma el programa, pero generalmente sólo una clase contendrá un método así (si más de una clase define ese método, se deberá utilizar una opción de compilación para indicar al compilador dónde está el punto de entrada del programa). Como en el caso de C++, `Main()` puede devolver un valor de tipo `void` o un `int`, aunque `int` es la opción más usual. También como su equivalente C++, `Main()` recibe argumentos similares – bien el conjunto de los parámetros de línea de comando pasados al programa, en forma de array de cadenas de caracteres, o ningún parámetro. Pero como puede verse en el código, las cadenas se definen de una manera más intuitiva en C# que en C++. De hecho, el identificador `string` es una palabra reservada de C#, y corresponde a una de las clases básicas de la librería de clases base de la Plataforma .NET, `System.String`. Además, los arrays son más sofisticados en C# que en C++. Cada array almacena la cantidad de elementos que contiene, además de los elementos en sí, por lo que no es necesario pasar de manera independiente la cantidad de elementos del array de cadenas, como debe hacerse en C++ mediante el parámetro `argc`.

Mostrando el mensaje

Finalmente, llegamos a las líneas de código que presentan el mensaje – primero en la consola, luego en un cuadro de mensaje. En ambos casos, estas líneas de código se apoyan en llamadas a las librerías de soporte de los lenguajes respectivos. El diseño de las clases de la librería estándar de C++ es obviamente muy diferente al de las clases de la librería de clases .NET, por lo que los detalles de las llamadas presentes en ambos ejemplos son muy diferentes. En C#, en ambos casos se realizan llamadas a métodos estáticos de clases base, mientras que para mostrar un cuadro de mensaje en C++ hay que apoyarse en una llamada no estándar a una función de la API de Windows, `MessageBox()`, que no es orientada a objetos.

Las clase base han sido diseñadas para ser muy intuitivas – probablemente más que en el caso de la librería estándar de C++. En la versión de C#, es inmediatamente obvio qué hace `Console.WriteLine()`. Mientras tanto, si no lo sabe con antelación, Ud. pasará un mal rato intentando averiguar qué significa `cout <<` (aunque sospecho que más de un programador de C++ extrañará el estilo artístico único de la construcción `cout <<`). Pero en el mundo comercial de la programación, que algo sea fácil de comprender es mucho más importante que el que sea artístico.

`MessageBox.Show()` recibe en este ejemplo menos parámetros que su equivalente C++, porque esa función está sobrecargada. Están igualmente disponibles otras variantes de la función que reciben parámetros adicionales.

Por último, otra diferencia que puede no ser notada a primera vista es que C# utiliza el punto ('`.`') en lugar de los dos puntos dobles ('`::`'), para la resolución de alcance. ¡ `Console` y `MessageBox` son nombres de clases, y no de instancias de clases! Para acceder a los miembros estáticos de las clases, C# siempre exige la sintaxis `<NombreClase>.<NombreMiembro>`, mientras que C++ permite elegir entre `<NombreClase>::<NombreMiembro>` y `<NombreInstancia>.<NombreMiembro>` (si una instancia de la clase existe dentro del alcance actual).

Comparación punto por punto

El ejemplo anterior debe haberle dado una visión general de algunas de las diferencias que encontrará entre los dos lenguajes. En el resto de este apéndice compararemos los dos lenguajes en detalle, recorriendo de modo sistemático las características de C++ y C#.

Arquitectura de programas

En esta sección presentaremos en términos generales cómo las características de los dos lenguajes afectan la arquitectura general de los programas.

Elementos de un programa

En C++ todo programa consta de un punto de entrada (en el ANSI C++ se trata de la función `main()`, aunque en las aplicaciones Windows generalmente se utiliza `WinMain()`), así como de diferentes clases, o estructuras, variables y funciones globales que se definen fuera de clase alguna. Aunque la mayoría de los desarrolladores considera un buen diseño orientado a objetos aquel en el que una gran mayoría de los elementos de nivel externo son objetos, C++ no obliga al programador a ello. Como ya hemos visto, C# sí obliga a utilizar este principio, y sienta las bases para un paradigma más exclusivamente orientado a objetos, exigiendo que todo sea miembro de alguna clase. En otras palabras, los únicos elementos de nivel superior que conforman los programas C# son las clases (u otros tipos que pueden ser considerados casos especiales de clases: enumeraciones, delegados e interfaces). En este sentido Ud. se dará cuenta de que se verá forzado a que su código C# sea más orientado a objetos de lo que sería necesario en C++.

Estructura de ficheros

En C++, la sintaxis mediante la cual se construyen los programas está en gran medida basada alrededor del fichero como unidad de código fuente. Por ejemplo, están los ficheros de código fuente (`.cpp`), cada uno de los cuales contendrá directivas `#include` al preprocesador para incluir los ficheros de cabecera relevantes. El proceso de compilación se basa en compilar de forma individual cada fichero fuente, después de lo cual los ficheros objetos correspondientes son enlazados para generar el ejecutable final. Aún cuando el ejecutable final no contendrá información alguna sobre los ficheros fuente o los ficheros objetos originales, C++ ha sido diseñado de un modo que exige al desarrollador codificar teniendo en cuenta explícitamente la estructura de ficheros elegida.

En C#, el compilador se encarga de los detalles relativos a la localización de las clases en los ficheros fuente. Ud. puede colocar su código en un solo fichero o, si lo prefiere, en varios ficheros, y eso será irrelevante para el compilador y no habrá necesidad de que ningún fichero haga referencia explícitamente a otros ficheros. En particular, no es necesario que un elemento cualquiera haya sido definido antes de que pueda ser referenciado en cualquier fichero individual, como ocurre en C++. El compilador localizará gustosamente la definición de cada elemento donde quiera que este pueda estar situado. Como efecto colateral de esto, el concepto de enlace en C# es muy diferente al de C++. El compilador de C# simplemente compilará todos sus ficheros fuente en una unidad de ensamblaje (aunque se puede hacer uso de otros conceptos, por ejemplo el módulo – una unidad que forma parte de una unidad de ensamblaje). El enlace tiene lugar en C#, pero se trata realmente de conectar su código con el código de librerías situadas en unidades de ensamblaje. No existe un concepto similar al de fichero de cabecera en C#.

Punto de entrada de un programa

En el ANSI C++ estándar, el punto de entrada de un programa es de forma predefinida una función llamada `main()`, que normalmente tiene la signatura:

```
int main(int argc, char *argv)
```

Aquí `argc` indica la cantidad de argumentos pasados al programa, y `argv` es un array de cadenas de

caracteres que contiene dichos argumentos. El primer argumento es siempre el comando utilizado para ejecutar el programa en sí. Windows modifica esto en cierto modo. Las aplicaciones Windows tradicionalmente arrancan por un punto de entrada llamado `WinMain()`, y las DLLs – por `DllMain()`. Estos métodos también reciben diferentes conjuntos de parámetros.

En C#, el punto de entrada sigue principios similares. Sin embargo, debido al requisito de que todos los elementos en C# deben ser parte de una clase, el punto de entrada no puede ser una función global. En lugar de eso, el requisito es que una de las clases del programa deberá tener un miembro estático llamado `Main()`, como hemos visto antes.

Sintaxis de los lenguajes

C# y C++ comparten una sintaxis virtualmente idéntica. Ambos lenguajes, por ejemplo, ignoran los espacios en blanco entre instrucciones, y utilizan el punto y coma como terminador de instrucciones y las llaves para unir instrucciones en bloques. Esto significa que, a primera vista, los programas escritos en ambos lenguajes tienen una apariencia muy similar. Debemos notar, sin embargo, las siguientes diferencias:

- C++ exige un punto y coma detrás de una definición de clase, mientras que C# no.
- C++ permite que las expresiones sean utilizadas como instrucciones incluso en el caso de que no tengan efecto colateral alguno, como por ejemplo, en la instrucción `i+1;` . En C#, esto será señalado como error.

Finalmente, debemos notar también que, al igual que C++, C# distingue entre mayúsculas y minúsculas. Sin embargo, debido a que C# ha sido diseñado para interoperar con VB.NET (que no hace distinción entre mayúsculas y minúsculas), se le recomienda encarecidamente no utilizar nombres públicos (o sea, que sean visibles a código situado fuera de su proyecto) que difieran únicamente en la utilización de letras mayúsculas y minúsculas. Si Ud. utiliza nombres públicos que se distingan únicamente por diferencias entre letras mayúsculas y sus correspondientes minúsculas, el código escrito en VB.NET no podrá acceder a sus clases (por cierto, si Ud. desarrolla código C++ controlado para la Plataforma .NET, el mismo consejo es aplicable).

Declaraciones adelantadas

Las declaraciones adelantadas no están soportadas ni permitidas en C#, dado que el orden en que los elementos de programa se definen en los ficheros fuente es irrelevante. Es perfectamente legal que un elemento de programa haga referencia a otro que está definido más adelante en el mismo fichero o en otro fichero – con tal de que esté definido en algún sitio. Esto contrasta con C++, en el que muchos símbolos sólo pueden ser utilizados en un fichero fuente si ya han sido declarados en el mismo fichero o en un fichero incluido.

No hay separación entre declaración y definición

Un elemento relacionado con la ausencia de declaraciones adelantadas en C# es que nunca hay separación entre declaración y definición. Por ejemplo, en C++ es común escribir una clase de la siguiente forma en el fichero de cabecera, en la que sólo se indican los prototipos de las funciones miembros, cuyas definiciones están especificadas en otro lugar:

```
class CMyClass
{
public:
   void MyMethod();   // definición de esta función en el fichero C++
                      // a menos que se trate de una función en línea
   // etc.
```

Esto no se hace así en C#. Los métodos siempre se definen completamente dentro de la definición de la clase:

```
class MyClass
{
   public void MyMethod()
   {
      // la implementación aquí
```

Ud. podrá pensar en primera instancia que esta característica hace que el código sea más difícil de leer. La belleza del modo de operar de C++ consiste en que sólo hace falta mirar el fichero de cabecera para ver qué funciones públicas la clase exporta, sin tener que ver la implementación de la clase. Sin embargo, esta facilidad se hace innecesaria en C#, en parte a causa de la potencia que ofrecen los editores modernos (el editor de Visual Studio .NET permite colapsar las implementaciones de los métodos), y en parte debido a que C# ofrece una utilidad para generar automáticamente la documentación del código en formato XML.

Control del flujo de programas

El control del flujo (el orden en que se ejecutan las instrucciones) de un programa es similar en C# y C++. En particular, las siguientes instrucciones funcionan exactamente igual en C# y en C++, y tienen exactamente la misma sintaxis:

- `for`
- `return`
- `goto`
- `break`
- `continue`

Hay un par de diferencias sintácticas en los casos de las instrucciones `if`, `while`, `do ... while` y `switch`, y C# ofrece una instrucción de control de flujo adicional, `foreach`.

if ... else

La instrucción `if` funciona exactamente de la misma forma y tiene exactamente la misma sintaxis en C# que en C++, con una pequeña pero importante distinción. La condición de toda cláusula `if` debe producir como resultado un valor `bool`. Por ejemplo, asumiendo que `x` es un entero, y no un `bool`, el siguiente código C++ producirá un error de compilación en C#:

```
if (x)
{
```

La sintaxis C# correcta es:

```
if (x != 0)
{
```

dado que el operador `!=` devuelve un valor `bool`.

Este requisito es un buen ejemplo de cómo la seguridad de tipos adicional que exige C# permite detectar los errores mucho antes. Son muy comunes en C++ los errores de ejecución provocados por haber escrito `if (a=b)` cuando se quería decir `if (a==b)`. En C# tales errores son detectados en tiempo de compilación.

Note que en C# no es posible convertir variables numéricas a `bool` o viceversa.

while y do … while

Al igual que en el caso de la instrucción `if`, estas instrucciones tienen en C# la misma sintaxis y propósito que en C++, con la excepción de que la expresión condicional debe producir un valor de tipo `bool`.

```
int X;
while (X) { /* instrucciones */}           // ERROR
while (X != 0) {/* instrucciones */}       // OK
```

switch

La instrucción `switch` sirve en C# para el mismo propósito que en C++. Es, sin embargo, más potente en C#, dado que es posible utilizar una cadena como variable de selección, algo que no es posible en C++:

```
string MyString;
// inicializar MyString
switch (MyString)
{
   case "Hello":
      // hacer algo
      break;
   case  "Goodbye":
      // etc.
```

La sintaxis en C# es ligeramente diferente por el hecho de que cada cláusula `case` debe garantizar una salida explícita. No se permite saltar de un `case` al siguiente `case`, a menos que el primero esté vacío. Para lograr este efecto, es necesario utilizar una instrucción `goto`.

```
   switch (MyString)
   {
      case "Hello":
         // hacer algo
         goto case "Goodbye";   // salta a las instrucciones del caso "Goodbye"
      case "Goodbye":
         // hacer otra cosa
         break;
      case "Black":      // OK para pasar al siguiente caso, éste está vacío
      case "White":
         // otra cosa más – se ejecutará si MyString contiene "Black" o "White"
         break;
      default:
         int j = 3;
         break;
   }
```

Microsoft ha decidido exigir la utilización del `goto` en este contexto para evitar los errores provocados por situaciones en las que se salta al siguiente caso cuando la intención real del programador es salir del `switch`.

foreach

C# ofrece una instrucción de control de flujo adicional, `foreach`. `foreach` permite recorrer todos los elementos de un array o colección sin necesidad de un índice explícito.

Un bucle `foreach` aplicado a un array puede tener la siguiente apariencia. En este ejemplo asumimos que `MyArray` es un array de elementos de tipo `double`, y deseamos mostrar cada uno de los valores en la ventana de la consola. Para ello podemos utilizar el siguiente código:

```
foreach(double SomeElement in MyArray)
{
   Console.WriteLine(SomeElement);
}
```

Note que en este bucle, `SomeElement` es el nombre que hemos elegido para representar a cada uno de los elementos que se visitan en el bucle – no es una palabra reservada, y podemos elegir cualquier nombre, siempre que no coincida con ningún otro nombre de variable.

Podríamos haber escrito también el bucle anterior de la siguiente forma:

```
foreach(double SomeElement in MyArray)
   Console.WriteLine(SomeElement);
```

dado que las instrucciones de bloque funcionan en C# de la misma forma que las instrucciones compuestas en C++.

Este bucle tiene exactamente el mismo efecto que:

```
for (int I=0 ; I < MyArray.Length ; I++)
{
   Console.WriteLine(MyArray[I]);
}
```

(Esta versión también ilustra cómo obtener la cantidad de elementos de un array en C#. Hablaremos de los arrays en C# más adelante en el apéndice).

Note sin embargo que, a diferencia del acceso a elementos de un array, el bucle `foreach` ofrece únicamente acceso de sólo lectura a sus elementos. Por lo tanto, el siguiente código no se compilará:

```
foreach(double SomeElement in MyArray)
   SomeElement*=2;            // ERROR – no se puede asignar a SomeElement
```

Hemos mencionado que el bucle `foreach` puede ser utilizado sobre arrays o colecciones. Las colecciones de C# no tienen contrapartida en C++, aunque el concepto se ha hecho popular en la programación para Windows gracias a su utilización en VB y COM. En esencia, una colección es una clase que implementa la interfaz `IEnumerable`. Dado que esto exige soporte de las clases base, explicamos las colecciones en el Capítulo 7.

Variables

Las definiciones de variables siguen básicamente el mismo patrón en C# que en C++:

```
int NCustomers, Result;
double DistanceTravelled;
double Height = 3.75;
const decimal Balance = 344.56M;
```

aunque, como era de esperar, algunos de los tipos son diferentes. Del mismo modo, como se ha señalado antes las variables pueden ser declaradas sólo localmente en un método o como miembros de una clase. C# no ofrece equivalentes a las variables globales o estáticas de C++. Como se ha mencionado antes, las variables que son miembros de una clase se denominan **campos** en C#.

Note que además C# distingue de una manera rígida entre los tipos de datos que se almacenan en la pila (los

tipos-valor) y los que se almacenan en memoria dinámica (los tipos-referencia). Examinaremos esta distinción con más detalle algo más adelante.

Tipos de datos básicos

Como en el caso de C++, C# ofrece varios tipos de datos predefinidos, y Ud. puede definir sus propios tipos en forma de clases o estructuras.

Los tipos de datos predefinidos de C# difieren de los que ofrece C++. Los tipos disponibles en C# son:

Nombre	Contenido	Símbolo
`sbyte`	Entero de 8 bits con signo	
`byte`	Entero de 8 bits sin signo	
`short`	Entero de 16 bits con signo	
`ushort`	Entero de 16 bits sin signo	
`int`	Entero de 32 bits con signo	
`uint`	Entero de 32 bits sin signo	`U`
`long`	Entero de 64 bits con signo	`L`
`ulong`	Entero de 64 bits sin signo	`UL`
`float`	Valor de punto flotante en 32 bits con signo	`F`
`double`	Valor de punto flotante en 64 bits con signo	`D`
`bool`	`true` o `false`	
`char`	Carácter Unicode en 16 bits	`''`
`decimal`	Valor de punto flotante con 28 dígitos significativos	`M`
`string`	Secuencia de caracteres Unicode de longitud variable	`""`
`object`	Utilizado cuando Ud. decide no especificar un tipo. El equivalente C++ más cercano es `void*`, con la diferencia de que `object` no es un puntero.	

En la tabla anterior, la tercera columna indica la letra que puede ser colocada detrás de una constante para indicar su tipo en situaciones en las que sea deseable indicar el tipo explícitamente. Por ejemplo, `28UL` representa el número 28, almacenado como un entero largo sin signo. Como en C++, se debe utilizar la comilla simple (apóstrofo) para representar caracteres, y las dobles comillas para las cadenas de caracteres. Sin embargo, en C# los caracteres siempre son caracteres Unicode, y `string` es un tipo-referencia predefinido, y no simplemente un array de caracteres.

Los tipos de datos en C# se definen con más exactitud que en C++. Por ejemplo, la expectativa natural en C++ es que un `int` ocupe 2 bytes (16 bits), pero la especificación ANSI C++ permite que esto sea un detalle dependiente de la plataforma. Por ello, en Windows un `int` de C++ ocupa 4 bytes, al igual que un `long`. Esto obviamente provoca algunos problemas de compatibilidad al transferir programas C++ entre plataformas. Por la otra parte, en C# para cada tipo de datos predefinido (¡excepto `string` y `object`, obviamente!) se define explícitamente su capacidad de almacenamiento.

Debido a que el tamaño de cada uno de los tipos primitivos de C# está perfectamente definido (se consideran tipos primitivos todos los anteriores, con excepción de `string` y `object`), hay menor necesidad de utilizar el

operador `sizeof`, que aunque existe en C#, sólo es permitido dentro del código no seguro (según se describe más adelante).

Aunque muchos nombres de C# son similares a los de C++, y por lo tanto se puede hacer un mapeado intuitivo entre muchos de los tipos correspondientes, algunos detalles sintácticos han cambiado. En particular, `signed` y `unsigned` no son palabras reservadas de C# (en C++ se pueden utilizar esas palabras, al igual que `long` y `short` para modificar otros tipos - por ejemplo, `unsigned long`, `short int`). Tales modificaciones no se permiten en C#, y por lo tanto la tabla de tipos anterior es la lista completa de tipos de datos predefinidos de C#.

Los tipos básicos como objetos

A diferencia de C++ (pero de modo similar a Java), los tipos de datos básicos de C# pueden además ser tratados como objetos, de manera que Ud. pueda aplicarles algunos métodos. Por ejemplo, en C# se puede convertir un entero a cadena de la siguiente forma:

```
int I = 10;
string Y = I.ToString();
```

Ud. puede incluso escribir:

```
string Y = 10.ToString();
```

El hecho de que es posible tratar los tipos básicos como objetos refleja la asociación directa entre C# y la librería de clases base .NET. C# realmente compila los tipos de datos básicos mediante un mapeado de cada uno de ellos a una de las clases base. Por ejemplo, `string` es mapeado a `System.String`, `int` a `System.Int32`, etc. Así que en un sentido real, en C# todo es un objeto. Sin embargo, note que esto sólo tiene lugar para propósitos sintácticos. En realidad, cuando su código es ejecutado estos tipos se implementan como los tipos correspondientes del lenguaje intermedio (IL), por lo que no hay ninguna pérdida de eficiencia derivada de tratar los tipos de datos básicos como objetos.

No presentaremos aquí todos los métodos disponibles a los tipos de datos básicos, pues todo los detalles están en MSDN. Sin embargo, sí haremos mención de los siguientes:

- Todos los tipos tienen un método `ToString()`. Para los tipos básicos, este método devuelve una representación en forma de cadena de caracteres de su valor.
- `char` ofrece una gran cantidad de propiedades que devuelven información sobre su contenido (`IsLetter`, `IsNumber`, etc.), y de métodos para efectuar conversiones (`ToUpper()`, `ToLower()`).
- `string` ofrece una gran cantidad de propiedades y métodos. Trataremos las cadenas independientemente.

También están disponibles varios métodos y propiedades estáticas. Entre ellos se incluyen:

- Los tipos enteros ofrecen `MinValue` y `MaxValue` para indicar los valores máximo y mínimo que permite almacenar el tipo.
- Los tipos `float` y `double` también ofrecen una propiedad, `Epsilon`, que indican el menor valor mayor que cero que estos tipos permiten representar.
- Para los tipos `float` y `double` están definidos los valores especiales `NaN` ('not a number', o sea indefinido), `PositiveInfinity` y `NegativeInfinity`. Los resultados de los cálculos producirán esos valores en los casos apropiados; por ejemplo dividir un número positivo entre cero producirá `PositiveInfinity`, mientras que dividir cero entre cero producirá `NaN`. Estos valores están disponibles como propiedades estáticas.
- Muchos tipos, incluyendo todos los numéricos, ofrecen un método estático `Parse()` que permite obtener un valor del tipo a partir de una cadena: `double D = double.Parse("20.5")`.

Note que para referirse a los métodos y propiedades estáticas en C# se debe incluir el nombre del tipo: `int.MaxValue` y `float.Epsilon`.

Conversión entre tipos básicos

La conversión de tipos es el proceso de transformar un valor almacenado en una variable de un tipo a un valor de otro tipo. En C++ esto puede hacerse de forma implícita o explícita:

```
float f1 = 40.0;
long l1 = f1;            // conversión implícita
short s1 = (short) l1;   // conversión explícita, estilo viejo (C)
short s2 = short (f1);   // conversión explícita, estilo nuevo (C++)
```

Si la conversión se especifica explícitamente, eso significa que Ud. ha indicado explícitamente el tipo de destino en su código. C++ le permite expresar conversiones explícitas en cualquiera de dos formas diferentes – el viejo estilo heredado de C, en el que el nombre del tipo se encierra entre paréntesis, o el nuevo estilo en el que lo que se encierra entre paréntesis es el nombre de la variable. Ambos estilos se muestra en el ejemplo anterior, y la elección está basada en preferencias sintácticas – dicha elección no tiene efecto sobre el código generado. En C++ es legal convertir entre cualesquiera de los tipos de datos básicos. Sin embargo, si hay riesgo de pérdida de datos porque el tipo de datos de destino tiene un rango de valores menor que el tipo de datos de origen, el compilador puede emitir una advertencia, en dependencia del nivel de advertencias para el que esté configurado. En el ejemplo anterior, la conversión implícita puede provocar una pérdida de información, por lo que el compilador emitirá típicamente una advertencia. La conversión explícita es en realidad una manera de indicarle al compilador que uno sabe lo que está haciendo, y por lo tanto no se emitirá advertencia alguna.

Dado que C# ha sido diseñado para ser más seguro en cuanto a tipos que C++, es menos flexible en lo que se refiere a las conversiones entre tipos de datos. Este lenguaje igualmente formaliza los conceptos de conversión explícita e implícita. Ciertas conversiones se definen como conversiones implícitas, lo que significa que Ud. puede realizarlas bien mediante la sintaxis explícita o la implícita. Otras conversiones sólo pueden efectuarse mediante sintaxis explícita, y el compilador emitirá un mensaje de error (¡no una advertencia, como en C++!) si Ud. intenta efectuar la conversión de forma implícita.

Las reglas de C# en relación a cuáles de los tipos numéricos básicos pueden ser convertidos a qué otros tipos son bastante lógicas. Las conversiones implícitas son aquellas en las que no hay riesgo de pérdida de información; por ejemplo, la conversión de `int` a `long` o de `float` a `double`. Las conversiones explícitas son aquellas en las que puede producirse una pérdida de información, debido a desbordamiento, error de signo o pérdida de la parte fraccionaria; por ejemplo, al convertir de `float` a `int`, de `int` a `uint`, o de `short` a `ulong`. Adicionalmente, como `char` se considera un tipo diferente de los tipos numéricos, sólo se puede convertir explícitamente de o hacia el tipo `char`.

Por lo tanto, los siguientes ejemplos son válidos en C#:

```
float f1 = 40.0F;
long l1 = (long)f1;            // explícito, debido a posible error de redondeo
short s1 = (short) l1;         // explícito, debido a posible desbordamiento
int i1 = s1;                   // implícito – ningún problema
uint i2 = (uint)i1;            // explícito, debido a posible error de signo
```

Note que en C# las conversiones explícitas siempre se realizan mediante el viejo estilo sintáctico de C. El nuevo estilo de C++ no puede utilizarse:

```
uint i2 = uint(i1);   // sintaxis incorrecta – esto NO compilará
```

Conversión verificada

C# ofrece la posibilidad de realizar las conversiones y otras operaciones aritméticas en un contexto en el que se verifica la corrección del resultado. Esto significa que el entorno de ejecución .NET detectará si se produce algún desbordamiento y lanzará una excepción (concretamente una `OverFlowException`) si ocurre algún desbordamiento. Esta característica no tiene contrapartida en C++.

```
checked
{
   int I1 = -3;
   uint I2 = (uint)I1;
}
```

Debido al contexto de verificación, la ejecución de la segunda instrucción del bloque lanzará una excepción. De no haber utilizado la instrucción `checked`, no se lanzaría ninguna excepción y la variable `I2` contendría "basura".

Cadenas de caracteres

El tratamiento de cadenas de caracteres es mucho más simple en C# de lo que nunca fue en C++. Esto se debe a la existencia de `string` como un tipo de datos básico que es reconocido por el compilador de C#. No hay necesidad alguna de tratar las cadenas como arrays de caracteres en C#.

El equivalente más cercano al tipo `string` de C# es la clase `string` de la librería estándar de C++. Sin embargo, una cadena de C# difiere de una cadena C++ en los siguientes aspectos fundamentales:

- Las cadenas C# contienen caracteres Unicode y no ANSI.
- La clase `string` de C# ofrece mucho más métodos y propiedades que la versión de C++.
- En C++, la clase `string` de la librería estándar no es más que una clase suministrada por la librería, mientras que en C# la sintaxis del lenguaje soporta específicamente la clase `string` como parte del lenguaje.

Secuencias de escape

C# utiliza el mismo método de C++ para denotar caracteres especiales mediante secuencias de escape, utilizando una barra invertida ('\'). La lista completa es la siguiente:

Secuencia de escape	Nombre del carácter	Codificación Unicode
`\'`	Comilla simple (apóstrofo)	`0x0027`
`\"`	Comilla doble	`0x0022`
`\\`	Barra invertida	`0x005C`
`\0`	Carácter nulo	`0x0000`
`\a`	Alerta	`0x0007`
`\b`	Retroceso	`0x0008`
`\f`	Cambio de página	`0x000C`
`\n`	Nueva línea	`0x000A`

\r	Retorno de carro	0x000D
\t	Tabulación horizontal	0x0009
\v	Tabulación vertical	0x000B

Esto significa básicamente que las secuencias de escape que se pueden utilizar en C# son las mismas que se utilizan en C++, con la excepción de que C# no reconoce \?.

Existe un par de diferencias entre los caracteres de escape en C++ y C#:

- La secuencia de escape \0 no es reconocida por C#. Sin embargo, el carácter nulo no se utiliza como terminador de cadenas en C#. Las cadenas C# almacenan de forma independiente su longitud, por lo que no se utiliza ningún carácter como terminador. Por eso las cadenas de C# pueden contener cualquier carácter Unicode.
- C# ofrece una secuencia de escape adicional: la secuencia \uxxxx (o su equivalente, \Uxxxx), donde xxxx representa un número hexadecimal de 4 dígitos. \uxxxx representa al carácter con código Unicode xxxx; por ejemplo, \u0065 es lo mismo que 'e'. Sin embargo, a diferencia del resto de las secuencias de escape, \uxxxx puede ser utilizado en nombres de variables, así como en constantes de caracteres y cadenas. Por ejemplo, el siguiente código C# es válido:

```
int R\u0065sult;       // el mismo efecto que int Result;
Result = 10;
```

C# ofrece además una sintaxis alternativa para representar las cadenas que es más conveniente en el caso de cadenas que contienen caracteres especiales: colocando un carácter @ delante de la cadena se impide que busquen en ella secuencias de escape. A estas cadenas se les denomina **cadenas verbatim** (tal cual). Por ejemplo, para representar la cadena C:\Book\Chapter2, podríamos escribir bien "C:\\Book\\Chapter2" o bien @"C:\Book\Chapter2". Por cierto, esto permite incluir retornos de carro en cadenas sin utilizar secuencias de escape para ellos:

```
string Message = @"Esta cadena empieza en una línea
y continúa en la siguiente ";
```

Tipos-valor y tipos-referencia

C# divide todos los tipos de datos en dos categorías: los tipos-valor y los tipos-referencia. Esta distinción no tiene equivalente en C++, donde las variables siempre contienen valores, a menos que una variable se declare explícitamente como una referencia a otra variable.

En C#, un tipo-valor contiene realmente su valor. Todos los tipos predefinidos de C# son tipos-valor, a excepción de object y string. Si Ud. define sus propias estructuras o enumeraciones, éstas también serán tipos-valor. Esto significa que los tipos de datos simples de C# generalmente funcionan exactamente igual que en C++ cuando Ud. les asigna un valor.

```
int I = 10;
long J = I;   // crea otra copia del valor 10
I = 15;       // no tiene efecto sobre J
```

Un tipo-referencia, como su nombre implica, contiene solamente una referencia al lugar donde el dato es almacenado en memoria. Sintácticamente, esto funciona de modo similar a como funcionan las referencias en C++, pero en términos de lo que ocurre realmente, las referencias de C# están más cercanas a los punteros de C++. En C#, object y string son tipos-referencia, como también lo son las clases que Ud. defina. A las referencias de C# se les puede asignar nuevos valores para que apunten a elementos de datos diferentes, de la misma manera que con los punteros de C++. Asimismo, a las referencias de C# se les puede asignar el valor null para indicar que no se refieren a nada. Por ejemplo, suponga que tenemos una clase llamada MyClass, que tiene una propiedad pública Width.

```
MyClass My1 = new MyClass();  // en C#, new simplemente llama a un constructor
My1.Width = 20;
MyClass My2 = My1;            // My2 apunta al mismo objeto que My1

My2.Width = 30;               // Ahora My1.Width = 30 también, porque My1 y My2
                              // apuntan al mismo objeto
My2 = null;                   // Ahora My2 no apunta a nada
                              // My1 aún se refiere al mismo objeto
```

No es posible en C# declarar una variable particular como un tipo-valor o un tipo-referencia – eso queda determinado única y exclusivamente por el tipo de la variable.

Los tipos-valor y los tipos-referencia tienen implicaciones sobre la gestión de memoria, debido a que los tipos-referencia siempre se almacenan en el *heap* (memoria dinámica), mientras que los tipos-valor se almacenan generalmente en el *stack* (pila). Esto se describe con más detalle en la próxima sección sobre la gestión de memoria.

Inicialización de variables

En C++ las variables nunca se inicializan a menos que Ud. lo haga explícitamente (o en el caso de las clases, suministre constructores). Si Ud. no lo hace, las variables contendrán cualquiera que sea el dato aleatorio que estuviera almacenado anteriormente en la zona de memoria asignada a esa variable – esto refleja el énfasis en el rendimiento que hace C++. C# hace más hincapié en evitar los errores de ejecución, y por lo tanto es más estricto en cuanto a la inicialización de las variables. Las reglas de C# son las siguientes:

- Las variables que son campos miembros se inicializan por defecto a cero en caso de que Ud. no las inicialice explícitamente. Esto significa que las variables de tipos numéricos contendrán el valor cero, las variables de tipo `bool` contendrán `false`, y todos los tipos-referencia (incluyendo `string` y `object`) contendrán la referencia nula. Las estructuras tendrán cada uno de sus miembros puestos a cero.
- Las variables locales a métodos no se inicializan por defecto. Sin embargo, el compilador producirá un error si una variable es utilizada antes de haber sido inicializada. Ud. puede, si lo desea, inicializar una variable llamando a su constructor por defecto (que inicializa la memoria a ceros).

```
// variables locales a un método
int X1;           // en este punto X1 contiene un valor aleatorio
//int Y = X1;     // esta línea provocaría un ERROR, por cuanto
                  // X1 se utilizaría antes de ser inicializada
X1 = new int();   // ahora X1 contiene cero y está inicializada
```

Enmarque (boxing)

En algunos caso, Ud. puede desear tratar un tipo-valor como si fuera un tipo-referencia. Esto se logra mediante un proceso conocido como **enmarque** (**boxing**). Sintácticamente, enmarcar una variable sólo significa convertir la variable en un objeto:

```
int J = 10;
object BoxedJ = (object) J;
```

El enmarque opera como cualquier otra conversión, pero Ud. debe saber que implica que el contenido de la variable será copiado al *heap* y se creará una referencia (dado que el objeto `BoxedJ` es un tipo-referencia).

La razón usual para enmarcar un valor es para pasarlo como parámetro a un método que espera un tipo-referencia como parámetro. Ud. puede además desenmarcar un valor enmarcado, indicando sencillamente una conversión al tipo original.

```
int J = 10;
object BoxedJ = (object) J;
int K = (int) BoxedJ;
```

Note que el proceso de desenmarque elevará una excepción si Ud. intenta convertir a un tipo inadecuado.

Gestión de memoria

En C++, las variables (incluyendo las instancias de clases y estructuras) pueden almacenarse en la pila o la memoria dinámica. En general, una variable es alojada en memoria dinámica si ella, o la clase en la que ella está contenida, ha sido reservada mediante `new()`, y en la pila en caso contrario. Esto significa que Ud., a través de su selección a la hora de reservar memoria para la variable, tiene total libertad para elegir si una variable será almacenada en la memoria dinámica o en la pila (aunque obviamente, debido a la forma en que trabaja la pila, los datos almacenados en ella sólo existirán mientras la variable correspondiente esté dentro de su alcance).

C# opera de modo muy diferente a este respecto. Una manera de comprender esta situación es pensando en dos escenarios comunes en C ++. Considere estas dos declaraciones de variable en C++:

```
int j = 30;
CMyClass *pMine = new CMyClass;
```

Aquí el contenido de `j` se almacena en la pila. Esta es exactamente la situación en el caso de los tipos-valor de C#. Nuestra instancia de `MyClass` es, sin embargo, almacenada en memoria dinámica, y un puntero a ella está situado en la pila. Esta es básicamente la situación en el caso de los tipos-referencia de C#, con la excepción de que en C# la sintaxis disfraza el puntero como una referencia. El equivalente en C# es:

```
int J = 30;
MyClass Mine = new MyClass();
```

Este código tiene prácticamente el mismo efecto en términos de dónde son almacenados los objetos que el código anterior en C++ – la diferencia está en que `MyClass` es tratada sintácticamente como una referencia y no como un puntero.

La gran diferencia entre C++ y C# es que C# no permite elegir cómo reservar memoria para una instancia particular. Por ejemplo, en C++ Ud. podría decir lo siguiente:

```
int* pj = new int(30);
CMyClass Mine;
```

Esto haría que el entero se almacenara en el heap, y la instancia de `CMyClass` – en la pila. Ud. no puede hacer eso en C#, porque C# obliga a que un `int` sea un tipo-valor, mientras que cualquier clase es siempre un tipo-referencia.

La otra diferencia es que no existe un equivalente al operador `delete` de C++ en C#. En lugar de ello, el recolector de basura del runtime de la Plataforma .NET periódicamente revisa todas las referencias contenidas en el código para identificar qué áreas del heap están actualmente siendo utilizadas por su programa, y es capaz de eliminar automáticamente todos los objetos que dejan de estar en uso. Esta técnica le libera a Ud. de tener que liberar personalmente la memoria utilizada.

En C#, los siguientes tipos son siempre tipos-valor:

- Todos los tipos predefinidos (exceptuando `object` y `string`)
- Todas las estructuras
- Todas las enumeraciones

Los siguientes son siempre tipos-referencia:

- `object`
- `string`
- Todas las clases

El operador new

El operador `new` tiene un sentido muy diferente en C# comparado con C++. En C++, `new` indica una solicitud de memoria dinámica. En C#, `new` simplemente indica que se está llamando al constructor de una variable. Sin embargo, la acción es similar, hasta el punto de que si la variable es de un tipo-referencia, la llamada a su constructor implícitamente significa que se reserve memoria dinámica para el objeto. Por ejemplo, suponga que tenemos una clase, `MyClass`, y una estructura, `MyStruct`. De acuerdo con las reglas de C#, las instancias de `MyClass` siempre serán alojadas en el heap, y las instancias de `MyStruct`, en el stack.

```
MyClass Mine;              // Sólo declara una referencia. Similar a declarar
                           // un puntero no inicializado en C++

Mine = new MyClass();      // Crea una instancia de MyClass. Llama al constructor
                           // sin parámetros. En el proceso, se reserva memoria
                           // en el heap

MyStruct Struct;           // Crea una instancia de MyStruct, pero no llama a
                           // ningún constructor. Los campos de MyStruct quedan
                           // sin inicializar

Struct = new MyStruct();   // Llama al constructor, inicializando los campos,
                           // pero no reserva ninguna memoria, por cuanto
                           // Struct ya existe en el stack
```

Es posible utilizar `new` para llamar al constructor para los tipos predefinidos:

```
int X = new int();
```

Esto tiene el mismo efecto que:

```
int X = 0;
```

Note que esto no es lo mismo que:

```
int X;
```

Esta última instrucción deja sin inicializar a `X` (si `X` es una variable local).

Métodos

Los métodos en C# se definen de la misma forma que las funciones en C++, salvo por la diferencia de que los métodos de C# deben siempre ser métodos de una clase, y que la definición y la declaración siempre van juntas en C#:

```
class MyClass
{
   public int MyMethod()
   {
      // implementación
```

Una restricción, sin embargo, es que los métodos miembros no pueden ser declarados `const` en C#. La posibilidad de C++ de definir explícitamente los métodos como `const` (en otras palabras, que no modifican la instancia de clase que los contienen) parecía originalmente una buena verificación de tiempo de compilación contra posibles errores, pero tiende a causar problemas en la práctica. Esto se debe a que es común para los métodos que no alteran el estado público de una clase alterar los valores de las variables miembro privadas, por ejemplo, en el caso de variables a las que se asigna valor en su primer acceso. No es poco común en código C++ encontrar el operador `const_cast`, utilizado salvar la dificultad de que un método ha sido declarado como `const`. Debido a esos problemas, Microsoft decidió no permitir los métodos `const` en C#.

Parámetros de métodos

Como en C++, los parámetros se pasan a los métodos por valor de forma predefinida. Si Ud. desea modificar esto, puede utilizar la palabra reservada `ref` para indicar que un parámetro es pasado por referencia, o la palabra reservada `out` para indicar que se trata de un parámetro de salida (siempre pasado por referencia). En tales casos, es necesario indicar el mecanismo de paso de parámetros tanto en la definición del método como en las llamadas al mismo:

```
public void MultiplyByTwo(ref double d, out double square)
{
   d *= 2;
   square = d*d;
}

// más adelante, al llamar al método:
double Value = 4.0, Square;
MultiplyByTwo(ref Value, out Square);
```

El paso por referencia implica que el método puede modificar el valor del parámetro. Ud. puede también utilizar el paso por referencia para mejorar el rendimiento cuando esté pasando como parámetro estructuras de gran tamaño, dado que, como en C++, el paso por referencia significa que sólo se pasa la dirección del parámetro. Note, sin embargo, que si Ud. utiliza el paso por referencia por razones de rendimiento, el método llamado podrá modificar el valor del parámetro: C# no permite asociar el modificador `const` a los parámetros del modo que lo hace C++.

Los parámetros de salida funcionan de modo muy similar a los parámetros por referencia, con la diferencia de que deben ser utilizados en los casos en que el método llamado suministra el valor del parámetro en lugar de modificarlo. De ahí que los requisitos a la hora de inicializarlos sean diferentes. C# exige que un parámetro `ref` sea inicializado antes de ser pasado a un método, y que un parámetro `out` sea inicializado dentro del método llamado antes de ser utilizado.

Sobrecarga de métodos

Los métodos pueden ser sobrecargados en C# del mismo modo que en C++. Sin embargo, C# no permite los

parámetros con valor por defecto. Esto debe simularse mediante la sobrecarga.

En C++, Ud. puede hacer esto:

```
double DoSomething(int someData, bool Condition = true)
{
   // etc.
```

Mientras que en C#, Ud. tendrá que hacer esto:

```
double DoSomething(int someData)
{
   DoSomething(someData, true);
}

double DoSomething(int someData, bool condition)
{
   // etc.
```

Propiedades

Las propiedades no tienen equivalente en ANSI C++, aunque han sido introducidas como extensiones al lenguaje en Microsoft Visual C++. Una propiedad es un método o par de métodos que se disfrazan sintácticamente para hacer parecer al código externo que está tratando con un simple campo. Ellas existen para aquellos casos en que es más intuitivo que un método sea llamado con la sintaxis de un campo – un ejemplo obvio puede ser el caso de un campo privado que se desea encapsular mediante métodos de acceso públicos. Suponga que una clase tiene un campo `length`, de tipo `int`. Entonces, en C++ podríamos encapsularlo mediante los métodos `GetLength()` y `SetLength()`, y deberíamos acceder a él desde fuera de la clase de la siguiente forma:

```
// MyObject es una instancia de la clase en cuestión
MyObject.SetLength(10);
int Length = MyObject.GetLength();
```

En C# podemos implementar estos métodos como métodos de acceso de lectura (`get`) y escritura (`set`) de una propiedad, `Length`. Entonces podríamos escribir:

```
// MyObject es una instancia de la clase en cuestión
MyObject.Length = 10;
int Length = MyObject.Length;
```

Para definir esos métodos de acceso, definiríamos la propiedad del siguiente modo:

```
class MyClass
{
   private int length;

   public int Length
   {
      get
      {
         return length;
      }
      set
      {
         Length = value;
      }
```

Aunque hemos implementado los métodos de acceso para que simplemente devuelvan o asignen un valor al campo que contiene la longitud, podríamos colocar cualquier código C# que deseáramos dentro de esos métodos de acceso. Por ejemplo, podríamos añadir validación de datos al método de acceso de escritura. Note

que el método de acceso de escritura devuelve `void` y recibe un parámetro adicional implícito, que tiene el nombre genérico `value`.

Es posible omitir el método de acceso de lectura o el de escritura de la definición de una propiedad, en cuyo caso la propiedad se convertiría en una propiedad de sólo escritura o de sólo lectura, respectivamente.

Operadores

Los significados y sintaxis de los operadores es muy similar en C# y C++. Los siguientes operadores tienen por defecto en C# la misma sintaxis y semántica que en C++:

- Los operadores aritméticos binarios `+`, `-`, `*`, `/`, `%`
- Los operadores aritméticos de asignación correspondientes `+=`, `-=`, `*=`, `/=`, `%=`
- Los operadores unarios `++` y `--` (tanto en versión prefija como postfija)
- Los operadores de comparación `!=`, `==`, `<`, `<=`, `>`, `>=`
- Los operadores de desplazamiento de bits `>>` y `<<`
- Los operadores lógicos `&`, `|`, `&&`, `||`, `~`, `^`, `!`
- Los operadores de asignación correspondientes a los operadores lógicos: `>>=`, `<<=`, `&=`, `|=`, `^=`
- El operador ternario (condicional) `? :`

Los símbolos `()`, `[]`, y `,` (coma) tienen también, en general, el mismo efecto en C# que en C++.

Deberá tener cuidado con los siguientes operadores, que funcionan de modo diferente en C# y C++:

- Asignación (`=`), `new`, `this`.

El operador de resolución de alcance en C# se representa mediante el punto (`.`), y no mediante `::` (que no tiene significado alguno en C#). Además, los operadores `delete` y `delete[]` no existen en C#. No son necesarios, debido a que el recolector de basura gestiona automáticamente la recuperación de la memoria dinámica. Sin embargo, C# ofrece tres operadores nuevos que no existen en C++: `is`, `as` y `typeof`. Estos operadores tienen que ver con la obtención de información sobre el tipo de un objeto o clase.

El operador de asignación (=)

Para los tipos de datos simples, `=` simplemente copia los datos. Sin embargo, cuando Ud. define sus propias clases, C++ considera que es responsabilidad del programador indicar el significado del operador `=` para sus clases. Por defecto, en C++ el operador = implica la realización de una copia "superficial" de cualquier variable, clase o estructura. Sin embargo, los programadores sobrecargan este operador para implementar operaciones de asignación más complejas.

En C#, las reglas que gobiernan el significado del operador de asignación son mucho más sencillas; el lenguaje no permite sobrecargar el operador `=`, estando su significado definido implícitamente en todas las situaciones.

La situación en C# es la siguiente:

- Para los tipos de datos simples, `=` simplemente copia los valores como en C++.
- Para las estructuras, `=` realiza una copia superficial de la estructura – o sea, una copia directa de los datos de la instancia en memoria. Este comportamiento es similar al de C++.
- Para las clases, `=` copia la referencia, o sea, la dirección y no el objeto. Este **NO** es el comportamiento de C++.

Si Ud. desea ser capaz de copiar instancias de clases, la manera usual de C# consiste en redefinir un método, `MemberwiseCopy()`, que todas las clases en C# heredan de `System.Object`, el ancestro absoluto del que todas las clases de C# heredan implícitamente.

this

El operador `this` tiene el mismo significado que en C++, pero produce una referencia en lugar de un puntero. Por ejemplo, en C++ Ud. puede hacer esto:

```
this->m_MyField = 10;
```

Sin embargo, en C#, Ud. deberá hacer esto:

```
this.MyField = 10;
```

`this` se utiliza de la misma manera en C# y C++. Por ejemplo, Ud. puede pasarlo como parámetro en llamadas a métodos, o usarlo para hacer explícito que está accediendo a un campo miembro de una clase. En C#, hay un par de situaciones adicionales que exigen la utilización de `this`, que mencionaremos en la sección relativa a las clases.

new

Como se ha mencionado anteriormente, el operador `new` tiene un significado muy diferente en C#, siendo interpretado como un constructor, hasta el extremo de obligar a un objeto a inicializarse, y no como una solicitud de memoria dinámica.

Clases y estructuras

En C++, las clases y las estructuras son extremadamente similares. Formalmente, las únicas diferencias son que los miembros de una estructura son públicos por defecto, mientras que los miembros de una clase son privados por defecto. En la práctica, sin embargo, muchos programadores prefieren utilizar las clases y estructuras de forma diferente, reservando el uso de las estructuras para objetos de datos que contengan únicamente variables miembros (en otras palabras, sin funciones miembros o constructores explícitos).

C# refleja esta diferencia tradicional de uso: en C# una clase es un tipo de objeto muy diferente de una estructura, por lo que Ud. deberá considerar cuidadosamente si un objeto dado se define mejor como una clase o como una estructura. Las diferencias más importantes entre las clases y estructuras de C# son las siguientes:

- Las estructuras no permiten la herencia, salvo el hecho de que ellas heredan de `System.ValueType`. No es posible heredar de una estructura, ni una estructura puede heredar de otra estructura o clase.
- Las estructuras son tipos-valor. Las clases son siempre tipos-referencia.
- Las estructuras le permiten organizar la manera en que los campos se disponen en la memoria, y definir el equivalente de las uniones de C++.
- El constructor por defecto (sin parámetros) de una estructura es siempre suministrado por el compilador y no puede ser reemplazado.

Dado que las clases y las estructuras son tan diferentes en C#, las trataremos en este apéndice por separado.

Clases

Las clases en C# siguen en general los mismos principios que en C++, aunque hay algunas diferencias tanto en características como en sintaxis. En esta sección repasaremos las diferencias entre las clases de C++ y las de C#.

Definición de una clase

Las clases se definen en C# utilizando una sintaxis que en principio puede parecer muy similar a la de C++:

```
class MyClass : MyBaseClass
{
   private string SomeField;
   public int SomeMethod()
   {
      return 2;
   }
}
```

Detrás de esa similitud superficial, existen numerosas diferencias en los detalles:

- No se aplican modificadores de acceso al nombre de la clase base. La herencia es siempre pública.
- Una clase puede heredar únicamente de una clase base (aunque puede derivarse también de cualquier cantidad de interfaces). Si no se especifica explícitamente ninguna clase base, entonces la clase se deriva implícitamente de `System.Object`, lo cual garantiza que la clase responda a todos los métodos definidos en `System.Object`, el más utilizado de los cuales es `ToString()`.
- Cada miembro se declara explícitamente con un modificador de acceso. Esto no tiene equivalente en la sintaxis de C++, donde un modificador de acceso puede aplicarse simultáneamente a varios miembros.

  ```
  public:   // no se puede utilizar esta sintaxis en C#
     int MyMethod();
     int MyOtherMethod();
  ```

- Los métodos no pueden ser declarados como `inline`. Esto es debido a que C# es compilado a lenguaje intermedio (IL). Cualquier transformación en línea se producirá en la segunda fase de la compilación – cuando el compilador Just-In-Time convierta el código IL a código de máquina. El compilador JIT tiene acceso a toda la información del lenguaje intermedio para determinar qué métodos pueden ser sustituidos en línea, sin necesidad de orientación alguna por parte del desarrollador en el código fuente.
- La implementación de los métodos siempre se coloca junto con la declaración. No es posible codificar la implementación fuera de la clase, como permite C++.
- Mientras que en ANSI C++ los únicos tipos de miembros posibles son las variables, funciones, constructores, destructores y sobrecargas de operadores, C# también permite delegados, eventos y propiedades.
- Los modificadores `public`, `private` y `protected` tienen el mismo significado que en C++, pero están disponibles dos modificadores adicionales:
  - `internal` restringe el acceso al código situado dentro de la misma unidad de ensamblaje.
  - `protected internal` restringe el acceso a las clases derivadas que estén situadas dentro de la misma unidad de ensamblaje.
- Se permite la inicialización de variables en las definiciones de clases en C#.
- C++ exige un punto y coma detrás de la llave que cierra la definición de una clase. Esto no es necesario en C#.

Inicialización de campos miembros

La sintaxis que se utiliza para inicializar los campos miembros en C# es muy diferente de la utilizada en C++, aunque el efecto final es idéntico.

Miembros de instancia

En C++, los campos miembros de instancia se inicializan generalmente a través de la lista de inicializaciones del constructor:

```
MyClass::MyClass()
   : m_MyField(6)
{
   // etc.
```

En C# esta sintaxis es incorrecta. Lo único que puede colocarse en el inicializador del constructor (el equivalente C# de la lista de inicializaciones del constructor de C++) es una llamada a otro constructor. Las inicializaciones de campos se llevan a cabo dentro de la definición del miembro en la definición de la clase:

```
class MyClass
{
   private int MyField = 6;
```

Note que en C++ esto sería un error, dado que C++ utiliza una sintaxis parecida para definir funciones virtuales puras. En C# esto es correcto, dado que C# no utiliza la sintaxis `=0` para este propósito, sino que utiliza en lugar de ello la palabra reservada `abstract`.

Campos estáticos

En C++, los campos estáticos se inicializan mediante una definición separada fuera de la clase:

```
int MyClass::MyStaticField = 6;
```

De hecho, en C++ aunque Ud. no desee inicializar el campo estático, debe incluir esta instrucción para evitar un error de enlace. En contraste, en C# no es necesaria esa instrucción, dado que en C# las variables se declaran en un solo lugar.

```
class MyClass
{
   private static int MyStaticField = 6;
```

Constructores

La sintaxis para la declaración de constructores en C# es la misma que se utiliza para los constructores en línea en las definiciones de clases de C++:

```
class MyClass
{
   public MyClass()
   {
      // código del constructor
   }
```

Al igual que en C++, Ud. puede definir tantos constructores como desee, siempre que éstos acepten conjuntos de parámetros diferentes. (Note que, al igual que en el caso de los métodos, no se admiten parámetros con valor por defecto – esto debe simularse mediante la sobrecarga).

Para las clases derivadas de una jerarquía, los constructores funcionan en C# básicamente de la misma manera que en C++: por defecto, el constructor de la clase más alta de la jerarquía (que es siempre `System.Object`) se ejecuta primero, y a partir de ahí se continúa bajando por el árbol de clases, ejecutando los constructores en orden.

Constructores estáticos

C# da soporte al concepto de constructor estático, que sólo es ejecutado una vez, y puede ser utilizado para inicializar los campos estáticos. Este concepto no tiene equivalente en C++.

```
class MyClass
{
   static MyClass()
   {
      // código del constructor estático
   }
```

Los constructores estáticos son muy útiles, porque permiten inicializar los campos estáticos con valores que se determinan en tiempo de ejecución (por ejemplo, se les puede asignar valores leídos de una base de datos). Esta clase de efecto puede lograrse en C++ pero requiere cierto trabajo, y la solución casi siempre luce engorrosa. La manera más común de hacerlo sería tener una función que accede a la variable miembro estática, e implementar la función de forma que asigne el valor a la variable la primera vez que sea llamada.

Note que un constructor estático no debe tener asociado un modificador de acceso – no debe ser declarado privado, público u otra cosa. Un modificador de acceso no tendría sentido aquí, dado que el constructor es llamado únicamente por el runtime de la Plataforma .NET, cuando la definición de clase sea cargada. Los constructores estáticos no pueden ser llamados desde ningún código C#.

C# no especifica exactamente cuándo un constructor estático será ejecutado; únicamente garantiza que éste será llamado después que los campos estáticos hayan sido inicializados, pero antes de que ningún objeto de la clase haya sido instanciado o que cualquier método estático de la clase sea utilizado.

Constructores por defecto

Como en C++, las clases de C# generalmente ofrecen un constructor por defecto sin parámetros, que simplemente llama al constructor sin parámetros de la clase base inmediata y luego inicializa todos los campos a sus valores por defecto. También a semejanza de C++, el compilador genera este constructor en el caso de que Ud. no suministre explícitamente ningún constructor en la definición de la clase. Si uno o más constructores estuvieran presentes en la definición de la clase, haya o no entre ellos un constructor sin parámetros, entonces esos constructores serían los únicos disponibles.

Del mismo modo que en C++, es posible evitar la instanciación de una clase declarando un constructor privado como único constructor:

```
class MyClass
{
   private MyClass()
   {
   }
```

Esto impedirá también la instanciación de las clases que hereden de ésta. Sin embargo, si una clase o cualquiera de sus métodos son declarados abstractos, esto impedirá la creación de instancias de la clase, pero no necesariamente de clases derivadas de ella.

Listas de inicialización de constructores

Los constructores de C# ofrecen un recurso a todas luces similar a las listas de inicialización de constructores de C++. Sin embargo, en C# esta lista puede contener a lo sumo un elemento, y se conoce como **inicializador de constructor**. El elemento presente en este inicializador debe ser bien una llamada a un constructor de la clase ancestro inmediata, o una llamada a otro constructor de la misma clase. Las sintaxis para ambas opciones se apoya en el uso de las palabras reservadas `base` y `this`, respectivamente:

```
class MyClass : MyBaseClass
{
   MyClass(int X)
   : base(X)      // se ejecuta el constructor de un parámetro de MyBaseClass
   {
         // otras inicializaciones aquí
   }

   MyClass()
   : this (10)   // se ejecuta el constructor de un parámetro de MyClass,
                 // pasándole el valor 10
   {
         // más inicializaciones aquí
   }
```

Si Ud. no suministra explícitamente una lista de inicialización de constructores, el compilador insertará implícitamente una, consistente de la llamada `base()`. En otras palabras, el inicializador por defecto llama al constructor por defecto de la clase base. Este comportamiento es similar al de C++.

A diferencia de C++, Ud. no puede colocar variables miembros en una lista de inicialización de constructores. Sin embargo, se trata de un asunto puramente sintáctico – el equivalente C# consiste en asignar los valores iniciales a los campos en la definición de la clase.

Una diferencia más seria es el hecho de que sólo se puede colocar a un constructor en la lista. Esto afectará la manera en que Ud. diseñará sus constructores, pero podría decirse que es beneficioso porque le obliga a utilizar un paradigma bien definido y efectivo de organizar sus constructores. Este paradigma es visible en el código de ejemplo anterior: el orden en que los diferentes constructores son ejecutados sigue siempre una ruta lineal.

Destructores

C# implementa un modelo de programación muy diferente del de C++ en relación con los constructores. Esto se debe a que el mecanismo de recolección automática de basura presente en C# implica que:

- Los destructores son menos necesarios, dado que la memoria reservada dinámicamente es liberada de forma automática.
- Dado que no es posible, en general, predecir cuándo el recolector de basura liberará un objeto dado, no es posible predecir exactamente el momento en el que el destructor se ejecutará, en caso de que un destructor haya sido definido.

Gracias a que la memoria es liberada "por detrás del telón" en C#, Ud. notará que sólo una pequeña parte de sus clases necesita un destructor. Para aquellas que lo necesiten (fundamentalmente clases que mantengan recursos externos no controlados, tales como ficheros o conexiones a bases de datos), C# ofrece un mecanismo de destrucción en dos fases:

1. La clase debe heredar la interfaz `IDisposable`, e implementar el método `Dispose()`. Este método debe ser llamado explícitamente por el código cliente para indicar que se ha terminado de trabajar con el objeto, y que se deben liberar los recursos a él asociados (hablaremos de las interfaces más adelante en este apéndice).
2. La clase debe implementar por separado un destructor, que tiene la consideración de mecanismo 'de reserva', para el caso de que el código cliente no haga una llamada a `Dispose()`.

La implementación típica de `Dispose()` tiene la siguiente apariencia:

```
public void Dispose()
{
   // devolución de recursos
   System.GC.SuppressFinalize(this);
}
```

`System.GC` es una clase base que representa al recolector de basura. `SuppressFinalize()` es un método que informa al recolector de basura que no es necesario llamar al destructor del objeto que está liberando. Es muy importante llamar a `SuppressFinalize()`, porque hay una pérdida de rendimiento asociada a la llamada al destructor de un objeto cuando el recolector de basura está recuperándolo; la consecuencia de esto será que la verdadera liberación de la memoria asociada al objeto se retrasará considerablemente.

La sintaxis para los destructores es básicamente la misma en C# que en C++. Note que en C# no hay necesidad de declarar el destructor como virtual – el compilador asume que lo es. Tampoco debe asociársele un modificador de acceso:

```
class MyClass
{
   ~MyClass()
   {
      // liberar los recursos
   }
```

Aunque el método `Dispose()` normalmente será llamado explícitamente por los clientes, C# permite una sintaxis alternativa que asegura que el compilador garantizará que sea llamado. Si la variable es declarada dentro de un bloque `using()`, entonces su alcance estará limitado al bloque `using`, y el método `Dispose()` será llamado a la salida del bloque:

```
using (MyClass MyObject = new MyClass())
{
   // código
}  // MyObject.Dispose() será llamado implícitamente al salir de este bloque
```

Note que el código anterior sólo se compilará en caso de que `MyClass` herede de `IDisposable` e implemente el método `Dispose()`. Si Ud. no desea utilizar esta sintaxis basada en `using`, entonces puede tomarse la libertad de no implementar alguno de los dos pasos asociados a la destrucción del objeto (implementar `Dispose()` e implementar el destructor), pero normalmente se deben implementar ambos. Ud. puede también implementar `Dispose()` sin heredar de `IDisposable`, pero si hace esto no le será posible utilizar la sintaxis `using` para que `Dispose()` sea llamada automáticamente para las instancias de la clase.

Herencia

La herencia funciona básicamente de la misma manera en C# que en C++, con la excepción de que la herencia múltiple de implementación no está soportada. Microsoft ha estimado que la herencia múltiple de implementación produce código menos estructurado y más difícil de mantener, y por ello tomó la decisión de dejar esta característica fuera de C#.

```
class MyClass : MyBaseClass
{
   // etc.
```

En C++, un puntero a una clase puede apuntar a una instancia de la clase o de una clase derivada (¡las funciones virtuales dependen de hecho de este mecanismo!). En C#, las clases son accedidas a través de

referencias, pero la regla equivalente también tiene lugar. Una referencia a una clase puede referirse a una instancia de esa clase o a una instancia de cualquier clase derivada.

```
MyBaseClass Mine;
Mine = new MyClass();   //OK si MyClass hereda de MyBaseClass
```

Si Ud. desea que una referencia pueda "apuntar" a cualquier objeto (el equivalente de `void*` en C++), puede definir la referencia como `object`, dado que C# mapea `object` a la clase `System.Object`, de la que todas las demás clases heredan:

```
object Mine2 = new MyClass();
```

Funciones virtuales y no virtuales

C# soporta las funciones virtuales del mismo modo que C++. Sin embargo, hay algunas diferencias sintácticas en C# que han sido diseñadas para eliminar ciertas ambigüedades potenciales de C++. Esto garantiza que ciertos tipos de errores que en C++ sólo se identifican en tiempo de ejecución, se detectarán en tiempo de compilación en C#.

Note además que en C# las clases siempre se utilizan a través de referencias (mecanismo equivalente al acceso a través de un puntero en C++).

En C++, si Ud. necesita que una función sea virtual, deberá utilizar la palabra reservada `virtual` tanto en la clase base como en la clase derivada. Por el contrario, en C# Ud. deberá declarar la función como `virtual` en la clase base, y utilizar la palabra reservada `override` en las versiones de las clases derivadas.

```
class MyBaseClass
{
   public virtual void DoSomething(int X)
   {
      // etc.
   }
   // etc.
}

class MyClass : MyBaseClass
{
   public override void DoSomething(int X)
   {
      // etc.
   }
   // etc.
}
```

La ventaja de esta sintaxis es asegura que el deseo del programador sea interpretado exactamente por el compilador y que se elimina la posibilidad de situaciones en las que, por ejemplo, Ud. introduce una redefinición de un método en una clase derivada con una signatura ligeramente diferente a la de la clase base, y por lo tanto termina definiendo una nueva versión cuando lo que intentaba era redefinir una existente. El compilador indicará un error si encuentra una función que incluye el modificador `override` y no puede identificar la versión correspondiente en la clase base.

Si la función no es virtual, Ud. puede aún definir otra versión de ese método en la clase derivada, en cuyo caso se dice que la versión de la clase derivada **esconde** a la versión de la clase base. En tales casos, qué método será llamado en un momento dado depende únicamente del tipo de la referencia utilizada para acceder a la

clase, del mismo modo en que depende del tipo de puntero utilizado para acceder a una clase en C++.

En C#, si la versión de una función en una clase derivada debe esconder a la función correspondiente de la clase base, Ud. puede indicarlo explícitamente mediante la palabra reservada `new`:

```
class MyBaseClass
{
   public void DoSomething(int X)
   {
      // etc.
   }
   // etc.
}

class MyClass : MyBaseClass
{
   public new void DoSomething(int X)
   {
      // etc.
   }
   // etc.
}
```

Si Ud. no marca explícitamente la nueva versión de la función como `new`, el código se compilará, pero el compilador emitirá una advertencia. Esta advertencia está destinada a proteger contra errores sutiles en los que, por ejemplo, se crea una nueva versión de una clase base, y se añade a ésta un método que resulta tener el mismo nombre que un método existente en una clase derivada.

Ud. puede declarar funciones abstractas en C# del mismo modo que en C++ (en C++ se les conoce también como funciones virtuales puras). La sintaxis, sin embargo, es diferente en C#: en lugar de utilizar `=0` al final de la definición se debe utilizar la palabra reservada `abstract`.

C++:

```
public:
   virtual void DoSomething(int X) = 0;
```

C#:

```
      public abstract void DoSomething(int X);
```

Como en C++, Ud. sólo podrá instanciar una clase en caso de que no contenga ella misma métodos abstractos, y ofrezca implementaciones para los métodos abstractos que hayan sido definidos en sus clases base.

Las estructuras

La sintaxis para la definición de estructuras en C# se asemeja a la que se utiliza para definir clases:

```
struct MyStruct
{
   private SomeField;
   public int SomeMethod()
   {
      return 2;
   }
}
```

La herencia y sus conceptos asociados, como las funciones virtuales y abstractas, no están permitidas para las

estructuras. Por lo demás, la sintaxis básica es idéntica a la utilizada para las clases, con la excepción de que la palabra reservada `struct` sustituye a `class` en la definición.

Hay, sin embargo, un par de diferencias entre las estructuras y las clases en lo que se refiere a su construcción. En particular, las estructuras siempre tienen un constructor por defecto que inicializa a ceros todos los campos, y este constructor sigue estando presente aún cuando Ud. defina otros constructores. Tampoco es posible definir explícitamente un constructor sin parámetros para reemplazar al predefinido. Ud. sólo puede definir constructores con parámetros. En este aspecto, las estructuras en C# difieren de sus homólogas en C++.

A diferencia de las clases en C#, las estructuras son tipos-valor. Eso significa que instrucciones como:

```
MyStruct Mine;
```

crean una instancia de `MyStruct` en la pila, exactamente igual que lo que ocurriría en C#. Sin embargo, en C#, esta instancia no será inicializada, a menos que explícitamente se llame al constructor:

```
MyStruct Mine = new MyStruct();
```

Si los campos miembros de `MyStruct` son todos públicos, Ud. podrá de forma alternativa inicializar la estructura inicializando por separado cada campo miembro.

Constantes

La palabra reservada `const` de C++ tiene toda una diversidad de usos. Por ejemplo, Ud. puede declarar variables como `const`, lo que indica usualmente que sus valores se asignan en tiempo de compilación y no podrán ser modificados en tiempo de ejecución (aunque hay un poco de flexibilidad en esto, dado que por ejemplo el valor de una variable miembro declarada como `const` puede ser asignado en una lista de inicialización de un constructor, lo que implica que ese valor puede ser calculado en tiempo de ejecución). Ud. puede también aplicar `const` a punteros y referencias para evitar que esos punteros o referencias sean utilizados para modificar los datos a los que ellos apuntan, y utilizar `const` para modificar las definiciones de parámetros que se pasan a las funciones, en cuyo caso `const` indica que una variable que ha sido pasada por referencia a través de un puntero no debe ser modificada por la función. Además, como se ha mencionado anteriormente, las funciones miembros pueden ser declaradas como `const` para indicar que no modifican a la instancia de la clase en la que están definidas.

C# permite el uso de la palabra reservada `const` para indicar que una variable no puede ser modificada. En muchos aspectos, sin embargo, el uso de `const` es más restrictivo en C# que en C++. En C#, el único uso válido de `const` es para fijar el valor de una variable (o de una referencia) en tiempo de compilación. No puede ser aplicada a métodos o parámetros. Por otra parte, la sintaxis de C# permite una mayor flexibilidad que la de C++ a la hora de inicializar campos `const` en tiempo de ejecución.

La sintaxis para declarar constantes es muy diferente en C# con relación a C++, por lo que la cubriremos con cierto nivel de detalle. La sintaxis de C# hace uso de dos palabras reservadas, `const` y `readonly`. La palabra reservada `const` indica que un valor es asignado en tiempo de compilación, mientras que `readonly` implica que el valor es asignado una vez en tiempo de ejecución, en un constructor.

Dado que en C# debe estar definido dentro de una clase o estructura, no existe por supuesto un equivalente directo en C# de las constantes globales en C++. Esta funcionalidad debe lograrse bien mediante enumeraciones o mediante campos miembros estáticos de una clase.

Constantes asociadas a una clase (constantes estáticas)

La manera usual de definir una constante estática en C++ es añadir un miembro `static const` a la clase. C# enfoca este asunto del mismo modo, pero con una sintaxis más simple:

Sintaxis C++:

```
int CMyClass :: MyConstant = 2;

class CMyClass
{
public:
   static const int MyConstant;
```

Sintaxis C#:

```
class MyClass
{
   public const int MyConstant = 2;
```

Note que en C# no declaramos explícitamente la constante como estática – eso provocaría un error de compilación. Por supuesto, la constante es implícitamente estática, dado que no tiene sentido almacenar un valor constante más de una vez, y por lo tanto el acceso siempre debe efectuarse utilizando la sintaxis para campos estáticos:

```
   int SomeVariable = MyClass.MyConstant;
```

Las cosas se ponen algo más interesantes cuando Ud. desea que sus constantes estáticas sean inicializadas con valores calculados en tiempo de ejecución. C++ simplemente no ofrece ninguna característica del lenguaje para lograr esto. Si Ud. desea lograr el mismo efecto, entonces deberá encontrar una manera de inicializar la variable la primera vez que es utilizada, lo que significa en primer lugar que no podrá declararla como `const`. En este caso C# ofrece ventaja sobre C++, dado que las constantes estáticas inicializadas en tiempo de ejecución se definen muy fácilmente. Ud. debe declarar el campo como `readonly`, e inicializarlo en el constructor estático:

```
class MyClass
{
   public static readonly int MyConstant;

   static MyClass()
   {
      // asignar el valor inicial de MyConstant aquí
   }
```

Constantes de instancia

Las constantes asociadas a instancias de clases siempre son inicializadas con valores calculados en tiempo de ejecución (si sus valores fueran calculados en tiempo de compilación, eso las haría por definición estáticas).

En C++, tales constantes deben ser inicializadas en la lista de inicializaciones de un constructor. Esto limita hasta cierto punto la flexibilidad para calcular los valores de esas constantes, dado que el valor inicial debe ser una expresión capaz de ser situada en una lista de inicialización de constructor.

```
class CMyClass
{
public:
   const int MyConstInst;

   CMyClass()
```

```
      : MyConstInst(45)
   {
```

En C# el principio es similar, pero la constante se declara como `readonly` en lugar de `const`. Eso significa que su valor es asignado en el cuerpo del constructor, lo cual ofrece mayor flexibilidad, ya que se puede utilizar cualquier conjunto de instrucciones C# para calcular el valor inicial (recuerde que en C# no se puede asignar valores a campos miembros en inicializadores de constructor –sólo se puede llamar a otro constructor).

```
class MyClass
{
   public readonly int MyConstInst;

   MyClass()
   {
      // inicializar MyConstInst aquí
```

En C#, si un campo es declarado como `readonly`, entonces puede recibir un valor sólo dentro de un constructor.

Sobrecarga de operadores

La sobrecarga de operadores sigue principios similares en C# y C++, pero hay algunas diferencias. La más notable es que C++ permite que la inmensa mayoría de los operadores sean sobrecargados. C# tiene más restricciones. Para muchos operadores compuestos, C# automáticamente obtiene el significado del operador a partir del significado de los operadores constituyentes, mientras que C++ permite su sobrecarga directa. Por ejemplo, en C++, Ud. puede sobrecargar independientemente los operadores + y +=. En C# sólo se puede sobrecargar +. El compilador siempre utilizará la sobrecarga del operador + para obtener automáticamente el significado de += para esa clase o estructura.

Los siguientes operadores pueden ser sobrecargados en C#, al igual que en C++:

- Los operadores aritméticos binarios + - * /%
- Los operadores unarios ++ y -- (versión prefija)
- Los operadores de comparación !=, == <, <=, >, >=
- Los operadores de manipulación de bits &, |, ~, ^, !
- Los valores lógicos `true` y `false`

Los siguientes operadores, que Ud. puede haber sobrecargado alguna vez en C++, no pueden ser sobrecargados en C#.

- Los operadores de asignación aritméticos *=, /=, +=, -=, %=. El compilador deduce el significado de estos operadores a partir de los significados del operador aritmético correspondiente y del operador de asignación, que no puede ser sobrecargado.
- Los operadores de incremento y decremento postfijos. El compilador deduce el significado de estos operadores a partir del significado del operador prefijo correspondiente (son implementados mediante una llamada al operador prefijo correspondiente, pero devuelven el valor original en lugar del nuevo valor).
- Los operadores de asignación con manipulación de bits &=, |=, ^=, >>= y <<=.
- Los operadores lógicos &&, ||. Son implementados por el compilador a partir de los operadores de manipulación de bits correspondientes.

- El operador de asignación, =. El significado de este operador está predefinido en C#.

También existe la restricción de que los operadores de comparación deben ser sobrecargados por parejas; en otras palabras, si Ud. sobrecarga el operador == deberá también sobrecargar != y viceversa. De forma similar, si sobrecarga uno de los operadores < ó <=, deberá sobrecargar también el otro, y lo mismo es aplicable a > y >=. La razón para esto es asegurar un soporte consistente para los tipos asociados a bases de datos que incluyan el valor `null`, y para los cuales, por ejemplo, == no significa necesariamente lo contrario que !=.

Una vez que Ud. haya comprobado que el operador que desea sobrecargar puede ser sobrecargado en C#, la sintaxis para hacerlo es mucho más simple que la correspondiente sintaxis de C++. El único punto con el que debe ser cuidadoso es que los operadores en C# siempre deben declararse como miembros estáticos de una clase. Esto contrasta con la situación en C++, donde Ud. puede elegir entre definir el operador mediante una función miembro estática de la clase, una función miembro de instancia con un parámetro menos, o una función que no sea miembro de ninguna clase.

La razón por la que definir sobrecargas de operadores en C# es mucho más simple no tiene nada que ver con las sobrecargas en sí, sino con la manera en que el mecanismo de gestión automática de memoria de C# ayuda al programador de un modo natural. La definición de sobrecargas de operadores en C++ es un área repleta de trampas en las que puede caer el principiante. Considere, por ejemplo, un intento de sobrecargar el operador de adición para una clase C++. Asumiremos que la clase `CMyClass` tiene una variable miembro `x`, y que añadir algo a una instancia significa añadir al miembro `x`. El código podría ser como sigue:

```
static CMyClass operator + (const CMyClass &lhs, const CMyClass &rhs)
{
   CMyClass Result;
   Result.x = lhs.x + rhs.x;
   return Result;
}
```

Note que los parámetros se han declarado `const` y se pasan por referencia, para asegurar un rendimiento óptimo. Esto de por sí no está tan mal. Sin embargo, con vistas a devolver un resultado, necesitamos crear una instancia temporal de `CMyClass` dentro de la implementación de la sobrecarga. La instrucción final `return Result;` parece inocua, pero compilará únicamente en el caso de que exista un operador de asignación adecuado para copiar `Result` fuera de la función. Si Ud. ha definido su propio constructor de copia para `CMyClass`, podrá necesitar definir también el operador de asignación para asegurarse de que la asignación se comporta de la forma apropiada. Esta no es una tarea trivial, y si no utiliza las referencias correctamente cuando defina el operador, será muy fácil obtener accidentalmente una versión que se llame a sí misma recursivamente hasta provocar un desbordamiento de pila. ¡Decididamente, la sobrecarga de operadores en C++ no es una tarea para un programador novel! No es difícil comprender por qué Microsoft decidió hacer no sobrecargables algunos operadores en C#.

En C# el cuadro es muy diferente. No es necesario pasar por referencia, dado que las clases de C# son referencias en sí mismas (y para las estructuras, el paso por referencia tiende a degradar el rendimiento en lugar de mejorarlo). Y devolver un valor es una tarea trivial. Ya se trate de una clase o de una estructura, simplemente bastará con devolver el valor del resultado temporal, y el compilador de C# asegurará que todos los campos miembros del resultado son copiados (para los tipos-valor) o la dirección copiada (para los tipos-referencia). La única desventaja es que no se puede utilizar la palabra reservada `const` para obtener del compilador la garantía de que la sobrecarga del operador no modifica las parámetros. Además, C# no ofrece la ventaja del rendimiento mejorado que garantiza la expansión en línea de C++.

```
static MyClass operator + (MyClass lhs, MyClass rhs)
{
   MyClass Result = new MyClass();
   Result.x = lhs.x + rhs.x;
   return Result;
```

```
   }
```

Indizadores

C# no permite en el sentido estricto la sobrecarga del operador `[]`. Sin embargo, le permite utilizar un mecanismo adicional, los indizadores de clases, para obtener el mismo efecto. La sintaxis para definir un indizador es muy similar a la sintaxis para definir una propiedad. Suponga que deseamos tratar las instancias de `MyClass` como un array de elementos de tipo `long`, en el que cada elemento es accesible mediante un índice de tipo `int`. Debemos escribir:

```
   class MyClass
   {
      public long this[int x]
      {
         get
         {
            // código para obtener el elemento x
         }
         set
         {
            // código para asignar al elemento x
         }
      }
      // etc.
```

El código situado dentro del bloque `get` se ejecutará siempre que la expresión `Mine[x]` aparezca en la parte derecha de una asignación (suponiendo que `Mine` es una instancia de `MyClass` y que `x` es un `int`), mientras que el bloque `set` será ejecutado cuando `Mine[x]` aparezca en la parte izquierda de una asignación. El bloque `set` no debe devolver nada, y dentro de él se utiliza la palabra reservada `value` para indicar la cantidad que aparece en la parte derecha de la asignación. El bloque `get` debe devolver un valor del mismo tipo de datos que el indizador.

Es posible sobrecargar los indizadores para que reciban cualquier tipo de datos dentro de los corchetes, o cualquier cantidad de argumentos – permitiendo simular el efecto de un array multidimensional.

Conversiones definidas por el usuario

Al igual que en el caso de los indizadores y `[]`, C# no permite formalmente la sobrecarga de `()`. Sin embargo, sí permite la definición de conversiones definidas por el programador, que producen el mismo efecto. Por ejemplo, suponga que tiene dos clases (o estructuras) llamadas `MySource` y `MyDest`, y que Ud. desea definir una conversión de tipos de `MySource` a `MyDest`. La sintaxis sería la siguiente:

```
      public static implicit operator MyDest (MySource Source)
      {
         // código para la conversión. Debe devolver una instancia de MyDest
      }
```

La conversión debe definirse como un miembro estático de cualquiera de las clases `MyDest` o `MySource`. También puede declararse como `implicit` o `explicit`. Si se declara la conversión como implícita, entonces podrá ser utilizada de forma implícita, como sigue:

```
      MySource Source = new MySource();
      MyDest Dest = MySource;
```

Si se declara el operador como explícito, entonces la conversión sólo podrá ser utilizada de manera explícita:

```
      MySource Source = new MySource();
```

```
      MyDest Dest = (MyDest) MySource;
```

Ud. debe definir conversiones implícitas en el caso de las conversiones que siempre funcionarán, y conversiones explícitas en los casos en que exista el peligro de pérdida de información o pueda ser lanzada una excepción.

Al igual que en C++, si el compilador de C# se enfrenta a una situación en la que se le exige convertir entre tipos de datos y no existe una conversión directa, buscará el 'mejor' camino utilizando las conversiones que estén disponibles. Aquí sin aplicables las mismas reglas generales de C++ en relación a que Ud. planifique las conversiones de forma tal que éstas sean intuitivas, y que diferentes rutas para aplicar conversiones no produzcan resultados incompatibles.

C# no permite definir conversiones entre clases que son derivadas una de otra. Tales conversiones ya están disponibles – implícitamente de una clase derivada a su clase base, y explícitamente de una clase base a su derivada.

Note que si Ud. intenta realizar una conversión de una clase base a una derivada, y el objeto en cuestión no es una instancia de la clase derivada (o de una clase derivada de ella) una excepción será lanzada.

En C++, no es difícil convertir un puntero a objeto al tipo de objeto equivocado. Esto es simplemente imposible en C# utilizando referencias. Por esa razón, la conversión de tipos en C# es mucho más segura que en C++.

```
      // se asume que MyDerivedClass es derivada de MyBaseClass
      MyBaseClass MyBase = new MyBaseClass();
      MyDerivedClass MyDerived = (MyDerivedClass) MyBase;   // esto provocará
                                                 // que se lance una excepción
```

Si Ud. no desea que como resultado de intentar una conversión sea lanzada una excepción, puede utilizar la palabra reservada `as`. Utilizando `as`, si la conversión falla simplemente se devolverá el valor `null`.

```
      // se asume que MyDerivedClass es derivada de MyBaseClass
      MyBaseClass MyBase = new MyBaseClass();
      MyDerivedClass MyDerived = MyBase as MyDerivedClass;  // esto devolverá null
```

Arrays

Los arrays son un área en la que la similitud superficial en sintaxis entre C++ y C# esconde el hecho de que lo que ocurre por detrás del telón en ambos lenguajes es muy diferente. En C++, un array es esencialmente un conjunto de variables dispuestas secuencialmente en memoria y accesibles mediante un puntero. Por otra parte, en C# una array es una instancia de la clase base `System.Array`, y es por lo tanto un objeto con todas las de la ley, residente en memoria dinámica bajo el control del recolector de basura. C# utiliza una sintaxis similar a la de C++ para acceder a los métodos de esta clase, creando la ilusión de que se está trabajando con arrays al estilo de C++. La parte negativa de este enfoque es que el rendimiento es inferior al de los arrays de C++, pero la ventaja es que con los arrays de C# es mucho más cómodo trabajar. Por ejemplo, todos los arrays de C# disponen de una propiedad, `Length`, que indica la cantidad de elementos en el array, ahorrando al programador tener que almacenar este dato independientemente. Los arrays de C# son también mucho más seguros – por ejemplo, la verificación de la validez de un acceso mediante índice se realiza automáticamente.

Si Ud. desea disponer en C# de un tipo de array simple, que no sufra de la sobrecarga que representa la clase `System.Array`, puede hacerlo, aunque para ello necesitará utilizar punteros y bloques de código no seguro.

Arrays de una dimensión

En el caso de los arrays de una sola dimensión (en la terminología de C#, arrays de **rango** 1), la sintaxis para el acceso a un elemento en los dos lenguajes es idéntica, encerrando entre corchetes el subíndice que indica el elemento deseado. Los índices comienzan a partir de cero en ambos lenguajes.

Por ejemplo, para multiplicar por 2 todos los elementos de un array de valores de tipo `float`:

```
// El array está declarado como array de floats
// Este código funciona tanto en C++ como en C#
for (int i=0 ; i<10 ; i++)
   Array[i] *= 2.0f;
```

Como se ha mencionado antes, sin embargo, los arrays de C# ofrecen la propiedad `Length`, que puede utilizarse para conocer cuántos elementos el array contiene.

```
// El array está declarado como un array de floats
// Este código sólo compila en C#
for (int i=0 ; i<Array.Length ; i++)
   Array[i] *= 2.0f;
```

En C# se puede utilizar adicionalmente la instrucción `foreach`, presentada anteriormente, para acceder a los elementos de un array.

La sintaxis para declarar un array es ligeramente diferente en C#, dado que los arrays de C# son siempre tipos-referencia.

```
double [] Array;             // simplemente declara una referencia,
                             // sin instanciar un array
Array = new double[10];      // crea (instancia) un objeto System.Array,
                             // y le asigna un tamaño 10.
```

Combinando estos elementos, se puede escribir:

```
double [] Array = new double[10];
```

Note que el tamaño del array sólo se especifica cuando es instanciado. La declaración de la referencia utiliza únicamente los corchetes para indicar que el rango del array es 1. En C#, el rango es considerado parte del tipo del array, mientras que la cantidad de elementos no.

El equivalente más cercano en C++ a la definición anterior sería:

```
double *pArray = new double[10];
```

Esta instrucción C++ de hecho ofrece una analogía bastante cercana, dado que en este caso tanto la versión de C++ como la de C# están situadas en el heap. Note que la versión C++ representa simplemente a un área de memoria que contiene 10 variables de tipo `double`, mientras que la versión de C# instancia un objeto verdadero. La versión más simple, basada en la pila, de C++:

```
double pArray[10];
```

no tiene una contrapartida directa en C#, aunque la directiva de C# `stackalloc` permite alcanzar el equivalente a esta instrucción utilizando punteros. Esto se discute más adelante, en la sección dedicada al

código no seguro.

Los arrays en C# pueden ser inicializados explícitamente al ser instanciados:

```
double [] Array = new double[10]
          {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0);
```

También está disponible una sintaxis más corta:

```
double [] Array = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0);
```

Si un array no es inicializado explícitamente, el constructor por defecto será llamado automáticamente para cada uno de sus elementos (los elementos de arrays son formalmente tratados como campos miembros de una clase). Este comportamiento es muy diferente al de C++, que no permite ninguna clase de inicialización automática de arrays creados en memoria dinámica mediante `new` (aunque C++ soporta arrays situados en la pila).

Arrays multidimensionales

C# se aleja significativamente de C++ en lo que respecta a los arrays multidimensionales, dado que C# soporta tanto arrays rectangulares como rasgados.

Un array rectangular es una verdadera rejilla de elementos. En C#, esto se indica mediante una sintaxis en la que las comas separan la cantidad de elementos de cada dimensión o rango. Por ejemplo, un array rectangular de dos dimensiones podría definirse de la siguiente forma:

```
int [,] MyArray2d;
MyArray2d = new int[2,3]  { {1, 0}, {3, 6}, {9, 12} };
```

Esta sintaxis no es más que una extensión bastante intuitiva de la sintaxis para los arrays de una dimensión. La lista de inicialización en el código anterior podría estar ausente. Por ejemplo:

```
int [,,] MyArray3d = new int[2,3,2];
```

Esto hará que el constructor por defecto sea llamado para cada elemento, inicializando cada `int` a cero. En este ejemplo particular, estamos ilustrando la creación de un array de tres dimensiones. La cantidad total de elementos en el array es de 2 x 3 x 2 = 12. La característica básica de los arrays rectangulares es que cada fila tiene la misma cantidad de elementos.

A los elementos de arrays rectangulares se accede utilizando una sintaxis similar:

```
int X = MyArray3d[1,2,0] + MyArray2d[0,1];
```

Los arrays rectangulares de C# no tienen contrapartida directa en C++. Los arrays rasgados de C#, sin embargo, se corresponden bastante directamente con los arrays multidimensionales de C++. Por ejemplo, si Ud. declara el siguiente array en C++:

```
int MyCppArray[3][5];
```

lo que está declarando no es un array de 3x5, sino un array de arrays – un array de tamaño 3, cada uno de cuyos elementos es a su vez un array de tamaño 5. Esto se ve más claramente en el caso de un array alojado en memoria dinámica. En ese caso deberíamos escribir:

```
int pMyCppArray = new int[3];
for (int i=0 ; i<3 ; i++)
   pMyCppArray[i] = new int[5];
```

Debe quedar claro de este código que no hay razón alguna para que cada fila contenga la misma cantidad de elementos (aunque eso es lo que ocurre en el ejemplo). Como ejemplo de array rasgado en C++, que realmente contiene una cantidad de elementos diferente en cada fila, podríamos escribir:

```
int pMyCppArray = new int[3];
for (int i=0 ; i<3 ; i++)
   pMyCppArray[i] = new int[2*i + 2];
```

Las filas respectivas de este array tienen dimensiones 2, 4 y 6.

C# obtiene el mismo resultado de un modo muy similar, aunque en el caso de C#, la sintaxis indica la cantidad de dimensiones de forma más explícita:

```
int [][] MyJaggedArray = new int[3][];
for (int i=0 ; <3 ; i++)
   MyJaggedArray[i] = new int[2*i + 2];
```

Para el acceso a los elementos de un array rasgado se utiliza la misma sintaxis que en C++:

```
int X = MyJaggedArray[1][3];
```

Aquí hemos mostrado un array rasgado de rango 2. Sin embargo, del mismo modo que en C++, Ud. puede definir un array rasgado con tantas dimensiones como desee – sólo tiene que añadir más corchetes a la definición.

Validación de fronteras

Un aspecto en el que se hace evidente la naturaleza orientada a objetos de los arrays de C# es la validación de las fronteras del array cuando se realiza un acceso a un elemento. Si Ud. intenta acceder a un elemento de un array en C# especificando un índice que está fuera del rango de las fronteras del array, esto será detectado en tiempo de ejecución, y será lanzada una excepción `IndexOutOfBoundsException`. En C++ esto no ocurre así, y por ello pueden producirse errores muy sutiles en tiempo de ejecución. Nuevamente, C# realiza esta comprobación adicional a expensas de la velocidad de ejecución. Aunque Ud. puede esperar que esta validación provoque una pérdida de rendimiento, ofrece la ventaja de que el motor de ejecución .NET puede verificar el código para asegurarse de que es seguro, en el sentido de que no intentará acceder a zonas de memoria que no han sido reservadas para sus variables. Esto también ofrece ventajas de rendimiento, dado que por ejemplo diferentes aplicaciones podrán ejecutarse en el mismo proceso con la total seguridad de que esas aplicaciones estarán aisladas unas de otras. También esto ofrece ventajas para la seguridad, dado que es posible predecir más exactamente qué intentará o no intentará hacer un programa.

Por otra parte, no es poco común en estos días que los programadores de C++ utilicen alguna de las varias clases de la librería estándar o de las MFC en lugar de los arrays, para obtener los beneficios de la validación de índices y otras varias características – aunque en este caso sin los beneficios de rendimiento y seguridad asociados a poder analizar el programa antes de ejecutarlo.

Redimensionamiento de arrays

Los arrays de C# son dinámicos en el sentido de que Ud. puede especificar la cantidad de elementos de cada dimensión en tiempo de ejecución (como en el caso de los arrays reservados dinámicamente en C++). Sin embargo, no es posible redimensionarlos una vez que han sido instanciados. Si Ud. necesita esta

funcionalidad, deberá estudiar otras clases similares, situadas en el espacio de nombres `System.Collections` de la librería de clases base, como `System.Collections.ArrayList`. En ese aspecto, C# tampoco es diferente de C++. Los arrays 'puros' de C++ no pueden ser redimensionados, pero el programador tiene a su disposición varias funciones de la librería estándar para obtener ese efecto.

Enumeraciones

En C#, es posible definir una enumeración utilizando la misma sintaxis que en C++:

```
// válido en C++ y C#
enum TypeOfBuilding {Shop, House, OfficeBlock, School};
```

Note, sin embargo, que el punto y coma final en C# es opcional, dado que en C# una definición de enumeración es efectivamente una definición de estructura, y las definiciones de estructuras no necesitan el punto y coma final.

```
// válido sólo en C#
enum TypeOfBuilding {Shop, House, OfficeBlock, School}
```

Otra diferencia estriba en que en C# a la enumeración debe asociársele un nombre, mientras que en C++ suministrar un nombre es opcional. Al igual que C++, C# numera los elementos de la enumeración comenzando desde cero, a menos que Ud. especifique que un elemento debe tener un valor específico.

```
enum TypeOfBuilding {Shop, House=5, OfficeBlock, School=10}
// Shop tendrá el valor 0, OfficeBlock tendrá el valor 6
```

La manera en que se accede a los valores de los elementos es diferente en C#, dado que en C# el nombre de la enumeración es obligatorio:

Sintaxis C++:

```
TypeOfBuilding MyHouse = House;
```

Sintaxis C#:

```
TypeOfBuilding MyHouse = TypeOfBuilding.House;
```

Ud. puede pensar que esto es una desventaja porque obliga a escribir más, pero esto de hecho refleja que las enumeraciones son mucho más potentes en C#. En C#, cada enumeración es una estructura por derecho propio (derivada de `System.Enum`), y por lo tanto ofrece diversos métodos. Por ejemplo, para cualquier enumeración es posible hacer lo siguiente:

```
TypeOfBuilding MyHouse = TypeOfBuilding.House;
string Result = MyHouse.ToString();   // Result contendrá "House"
```

Esto es algo imposible de lograr en C++.

Ud. puede también realizar la conversión inversa en C#, utilizando el método estático `Parse()` de la clase `System.Enum`, aunque la sintaxis es algo más engorrosa:

```
TypeOfBuilding MyHouse = (TypeOfBuilding)Enum.Parse(typeof(TypeOfBuilding),
                                                    "House", true);
```

`Enum.Parse()` devuelve una referencia a objeto, y por lo tanto debe ser explícitamente convertida (desenmarcada) al tipo `enum` apropiado. El primer parámetro de `Parse()` es un objeto de tipo `System.Type`, que describe a qué enumeración la cadena representa. El segundo parámetro es la cadena en sí, y el tercer parámetro indica si se debe ignorar la distinción entre mayúsculas y minúsculas. Existe una versión sobrecargada de este método que omite el tercer parámetro, y tiene en cuenta las diferencias entre mayúsculas y minúsculas.

C# también permite indicar el tipo base utilizado para almacenar los valores de la enumeración:

```
enum TypeOfBuilding : short {Shop, House, OfficeBlock, School};
```

Si no se especifica un tipo, se asume `int`.

Excepciones

Las excepciones se utilizan de la misma manera en C# que en C++, con las dos siguientes diferencias:

- C# define el bloque `finally`, que contiene código que siempre será ejecutado al finalizar el bloque `try`, independientemente de si se ha producido una excepción o no. La ausencia de esta característica es una causa común de queja de los programadores C++. El bloque `finally` es ejecutado tan pronto como el control abandona un bloque `catch` o `try`, y generalmente contiene código que se encarga de la liberación de los recursos reservados en el bloque `try`.
- En C++, el valor lanzado como consecuencia de una excepción puede ser de cualquier tipo. C#, sin embargo, exige que el valor lanzado sea un objeto de una clase derivada de `System.Exception`.

Las reglas para el flujo del control a través de los bloques `try` y `catch` son idénticos en C++ y C#. La sintaxis utilizada también es idéntica, con una diferencia – en C# un bloque `catch` que no especifique una variable para recibir al objeto de excepción se denota mediante una instrucción `catch` por sí sola:

Sintaxis C++:

```
catch (...)
{
```

Sintaxis C#:

```
catch
{
```

En C#, este tipo de instrucción `catch` puede ser útil para capturar excepciones lanzadas por código escrito en otros lenguajes (y que por tanto puede que no se derive de `System.Exception` – el compilador de C# producirá un error si Ud. intenta definir un objeto de excepción que no lo sea, ¡pero eso no ocurre en otros lenguajes!).

La sintaxis completa de `try` ... `catch` ... `finally` en C# tiene la siguiente apariencia:

```
try
{
   // código normal
}
catch (MyException e)      // MyException derivada de System.Exception
{
```

```
   // código de gestión de error
}
// opcionalmente, más bloques catch
finally
{
   // código de liberación de recursos
}
```

Note que el bloque `finally` es opcional. También es posible que no haya bloques `catch` – en cuyo caso la construcción `try` … `finally` simplemente sirve como una manera de asegurar que el código situado en el bloque `finally` siempre será ejecutado cuando el bloque `try` finalice. Esto podría ser útil, por ejemplo, si el bloque try contiene varias instrucciones `return`, y Ud. desea que se realice cierta liberación de recursos antes de que el método finalice.

Punteros y código no seguro

En C# es posible definir punteros y utilizarlos de un modo muy similar a como se hace en C++. Sin embargo, esto puede hacerse únicamente dentro de bloques de código no seguro.

Ud. puede declarar cualquier método como no seguro:

```
public unsafe void MyMethod()
{
```

De forma alternativa, Ud. puede declarar cualquier clase o estructura como no segura:

```
unsafe class MyClass
{
```

Declarar una clase como no segura implica que todos sus miembros son tratados como no seguros. Ud. puede además declara cualquier campo miembro (aunque no las variables locales) como no seguros, si tiene un campo miembro de tipo puntero:

```
private unsafe int* pX;
```

También es posible marcar un bloque de código como no seguro de la siguiente forma:

```
unsafe
{
   // instrucciones que utilizan punteros
}
```

Las sintaxis para declarar, acceder, derreferenciar y realizar operaciones aritméticas sobre punteros son las mismas de C++:

```
// Este código compila tanto en C++ como en C#,
// y produce el mismo efecto en ambos lenguajes
int X = 10, Y = 20;
int *pX = &X;
*pX = 30;
pX = &Y;
++pX;    // añade sizeof(int) a pX
```

Tenga en cuenta los siguientes detalles, sin embargo:

- En C# no se permite derreferenciar punteros `void*`, ni tampoco se pueden efectuar operaciones aritméticas sobre punteros `void*`. La sintaxis `void*` se ha mantenido simplemente por compatibilidad descendente, para llamar a funciones externas de la API de Windows que exigen punteros `void*` como parámetros.
- Los punteros no pueden apuntar a tipos-referencia (clases o arrays). Tampoco pueden apuntar a estructuras que contengan tipos-referencia como miembros. Esto es un intento de proteger los datos utilizados por el motor de ejecución .NET y el recolector de basura (aunque en C#, como en C++, una vez que Ud. comienza a utilizar punteros casi siempre encontrará un modo de salvar cualquier restricción realizando operaciones aritméticas sobre los punteros y luego derreferenciando).
- Además de declarar las partes relevantes de su código como no seguras, Ud. deberá especificar la opción `/unsafe` al compilador al compilar código que trabaje con punteros.
- Los punteros no pueden apuntar a variable que estén embebidas en tipos-referencia (por ejemplo, miembros de clases), a menos que el puntero esté declarado en un bloque `fixed`.

Fijación de datos en memoria dinámica

Se permite asignar la dirección de un tipo-valor a un puntero incluso si ese tipo-valor está embebido como campo miembro dentro de un tipo-referencia. Sin embargo, ese puntero debe ser declarado dentro de una instrucción `fixed`. La razón para esto es que los tipos-referencia pueden ser movidos a conveniencia en memoria dinámica por el recolector de basura. El recolector de basura sabe qué referencias existen a cada objeto, y puede actualizarlas si es necesario, pero no controla los punteros. Por lo tanto, si un puntero está apuntando a un campo de un objeto en el heap y el recolector de basura mueve la instancia entera, el puntero quedaría apuntando a un sitio incorrecto. La instrucción `fixed` impide que el recolector de basura mueva en memoria la instancia de clase especificada durante la ejecución del bloque `fixed`, asegurando así la integridad de los valores de los punteros.

```
class MyClass
{
   public int X;
   // etc.
}

// en otra parte del código...
MyClass Mine = new MyClass();
// procesamiento
fixed(int *pX = Mine.X)
{
   // se puede utilizar pX en este bloque
}
```

Es posible anidar bloques `fixed` para declarar más de un puntero. También es posible declarar más de un puntero en un mismo bloque `fixed`, siempre que ambos punteros apunten al mismo tipo:

```
fixed(int *pX= Mine.X, *pX2 = Mine2.X)
{
```

Declaración de arrays en la pila

C# ofrece un operador, `stackalloc`, que puede utilizarse en combinación con los punteros para definir un array "de bajo consumo" en la pila. El array reservado de esta forma no será un objeto C# de tipo `System.Array`, sino un simple array de elementos exactamente análogo a un array unidimensional de C++. Los elementos de este array no son inicializados, y se accede a ellos utilizando la misma sintaxis de C++,

aplicando corchetes al puntero.

El operador `stackalloc` exige la especificación de l tipo de datos y la cantidad de elementos para los que se desea reservar espacio:

Sintaxis C++:

```
unsigned long pMyArray[20];
```

Sintaxis C#:

```
ulong *pMyArray = stackalloc ulong [20];
```

Tenga en cuenta que, aunque esos arrays son exactamente análogos, la versión C# permite que el tamaño sea especificado en tiempo de ejecución:

```
int X;
// inicializar X
ulong *pMyArray = stackalloc ulong [X];
```

Interfaces

Las interfaces son un aspecto de C# que no tienen análogo directo en ANSI C++, aunque Microsoft ha introducido las interfaces en C++ a través de una palabra reservada específica a Microsoft C++. La idea de una interfaz es una evolución de las interfaces de COM, que fueron pensadas como contratos que indican qué métodos o propiedades un objeto implementa.

Una interfaz en C# no es exactamente lo mismo que una interfaz COM, por cuanto no tiene un GUID asociado, no se deriva de `IUnknown`, y no tiene entradas del Registro asociadas (aunque es posible mapear una interfaz de C# a una interfaz COM). Una interfaz de C# es simplemente un conjunto de definiciones de métodos y propiedades. Puede considerarse un análogo de una clase abstracta, y se define utilizando una sintaxis similar a la de una clase.

```
interface IMyInterface
{
   void MyMethod(int X);
}
```

Notará, sin embargo, las siguientes diferencias sintácticas con una definición de clase:

- Los métodos no incluyen modificadores de acceso.
- Los métodos nunca se implementan en una interfaz.
- Los métodos no pueden ser declarados como virtuales, ni explícitamente como abstractos. La elección de cómo implementar los métodos es responsabilidad de la clase que implemente la interfaz.

Una clase implementa una interfaz heredando de ella. Aunque una clase puede heredar de a lo sumo una clase, también puede heredar de tantas interfaces como sea necesario. Si una clase implementa una interfaz, está obligada a ofrecer una implementación a cada uno de los métodos definidos por la interfaz.

```
class MyClass : MyBaseClass, IMyInterface, IAnotherInterface // etc.
{
```

```
      public virtual void MyMethod(int X)
      {
         // implementación
      }
      // etc.
```

En este ejemplo, hemos decidido implementar `MyMethod` como un método virtual con acceso público.

Las interfaces pueden a su vez heredar de otras interfaces, en cuyo caso la interfaz derivada contiene, además de sus métodos, los de la interfaz base:

```
   interface IMyInterface : IBaseInterface
```

Ud. puede verificar si un objeto implementa una interfaz bien utilizando el operador `is`, o bien utilizando el operador `as` para convertir a dicha interfaz. De forma alternativa, puede intentar la conversión directamente, pero en ese caso obtendrá una excepción si el objeto no implementa la interfaz, por lo que este enfoque sólo se recomienda si se sabe a ciencia cierta que la conversión será exitosa. Ud. puede utilizar la referencia a interfaz obtenida de la conversión para llamar a los métodos de la interfaz (cuyas implementaciones son suministradas por la instancia de la clase).

```
   IMyInterface MyInterface;
   MyClass Mine = new MyClass();
   MyInterface = Mine as IMyInterface;
   if (MyInterface != null)
      MyInterface.MyMethod(10);
```

Los usos principales de las interfaces son:

- Para la interoperabilidad y la compatibilidad descendente con componentes COM.
- Para servir como contratos para otras clases .NET. Una interfaz puede ser utilizada para indicar que una clase implementa ciertas características. Por ejemplo, el bucle `foreach` funciona internamente verificando que la clase a la que es aplicado implementa la interfaz `IEnumerable`, y luego llamando a los métodos definidos por esa interfaz.

Delegados

Un delegado en C# no tiene equivalente directo en C++, y realiza las mismas funciones que un puntero a función en C++. La idea es que un delegado es un puntero a método, encapsulado en una clase especializada conjuntamente con una referencia al objeto al que el método será aplicado (en el caso de un método de instancia, o la referencia nula en el caso de un método estático). Esto implica que, a diferencia de un puntero a función de C++, un delegado de C# contiene suficiente información para llamar a un método de instancia.

Formalmente, un delegado es una clase que se deriva de la clase `System.Delegate`. Por lo tanto, instanciar un delegado es un proceso de dos fases: la definición de esta clase derivada, y la declaración de una variable del tipo apropiado. La definición de una clase-delegado incluye los detalles de la signatura (incluyendo el tipo del valor a devolver) del método que el delegado encapsula.

La utilización principal de los delegados consiste en pasar de un método a otro las referencias a métodos y permitir las llamadas indirectas: las referencias a métodos no pueden ser pasadas como parámetros directamente, pero sí como parte de un delegado. Los delegados aseguran la seguridad de tipos, y evitan las llamadas a métodos con signaturas incorrectas. El método contenido en el delegado puede ser llamado a través del delegado. El siguiente código muestra los principios generales.

En primer lugar, debemos definir la clase-delegado:

```
   // Define una clase-delegado que representa un método que recibe un entero
```

```
// y no devuelve nada
delegate void MyOp(int X);
```

A continuación, para los propósitos del ejemplo declararemos una clase que contiene los métodos a llamar:

```
// una definición de clase
class MyClass
{
   void MyMethod(int X)
   {
      // etc.
   }
}
```

Más adelante, probablemente en la implementación de alguna otra clase, tenemos el método al que se le ha de pasar una referencia a método a través de un delegado:

```
void MethodThatTakesDelegate(MyOp Op)
{
   // llamar al método, pasándole el valor 4
   Op(4);
}
```

Por último, este es el código que utiliza el delegado:

```
MyClass Mine = new MyClass();

// instanciar un delegado MyOp. Hacer que apunte al método MyMethod de Mine
MyOp DoIt = new MyOp(Mine.MyMethod);
```

Una vez que la variable-delegado ha sido declarada, podremos llamar al método a través del delegado:

```
DoIt();
```

O pasarla como parámetro a otro método:

```
MethodThatTakesDelegate(DoIt);
```

En el caso particular en que un delegado represente a un método que devuelve `void`, se dice que el delegado es un delegado múltiple (**multicast**), y puede representar simultáneamente a más de un método. La llamada a ese delegado implica que todos los métodos a los que él representa sean ejecutados secuencialmente. Los operadores `+` y `+=` pueden utilizarse para añadir un método a un delegado, mientras que `–` y `-=` pueden utilizarse para eliminar un método del delegado. Los delegados se explican con mayor nivel de detalle en el Capítulo 6.

Eventos

Los eventos son formas especializadas de delegados que se utilizan para soportar el modelo de función de respuesta necesario para implementar las notificaciones, típicas en los sistemas con interfaz de usuario gráfica. Un evento es un delegado que tiene la signatura:

```
delegate void EventClass(obj Sender, EventArgs e);
```

Esta es la signatura que debe tener obligatoriamente el gestor de eventos que será llamado como respuesta a un suceso determinado. `Sender` debe ser una referencia al objeto que ha provocado (lanzado) el evento, mientras que `System.EventArgs` (o cualquier clase derivada de `EventArgs`, que también podrá ser pasada como parámetro) es la clase utilizada por el runtime .NET para pasar información genérica concerniente a los detalles de un evento.

La sintaxis especial para declarar un evento es la siguiente:

```
public event EventClass OnEvent;
```

Los clientes utilizarán la sintaxis de los delegados múltiples basada en el uso del operador `+=` para indicar al evento que desean que se les envíe una notificación.

```
// EventSource es la instancia de clase que contiene el evento
EventSource.OnEvent += MyHandler;
```

El objeto que origina el evento sencillamente llama al evento cuando es necesario, utilizando la misma sintaxis vista anteriormente para los delegados. Dado que el evento es un delegado múltiple, todos los gestores de evento serán llamados en el proceso. Los eventos se explican con mayor detalle en el Capítulo 6.

```
OnEvent(this, new EventArgs());
```

Atributos

Los atributos son un concepto que no tiene equivalente en ANSI C++, aunque están soportados por el compilador de Microsoft C++ como una extensión específica para Windows. En la versión C#, se trata de clases .NET que se derivan de `System.Attribute`. Los atributos pueden ser aplicados a diferentes elementos presentes en el código C# (clases, enumeraciones, métodos, parámetros, etc.) para generar información adicional acerca del elemento en la unidad de ensamblaje compilada, con vistas a la documentación. Adicionalmente, ciertos atributos son reconocidos por el compilador C# y producen un efecto sobre la compilación. Estos incluyen:

Atributo	Descripción
`DllImport`	Indica que un método está definido en una DLL externa.
`StructLayout`	Permite especificar la colocación en memoria de los campos de una estructura. Permite implementar el equivalente de una unión en C#.
`Obsolete`	El compilador genera un error o advertencia en caso de que este método sea utilizado.
`Conditional`	Fuerza la compilación condicional de código. Este método y todas las referencias a él serán ignoradas, a menos que un símbolo del preprocesador específico esté presente.

Existe una gran cantidad de atributos predefinidos, y también es posible definir nuestros propios atributos. El uso de los atributos se discute en los Capítulos 6 y 7.

Los atributos aparecen en el código inmediatamente antes del objeto al que se aplican, encerrados entre corchetes. Esta es la misma sintaxis que utiliza para los atributos Microsoft C++.

```
[Conditional("Debug")]
void DisplayValuesOfImportantVariables()
{
   // etc.
```

Directivas al preprocesador

C# soporta las directivas al preprocesador de un modo similar al de C++, con la excepción de que en C# hay menos directivas. En particular, C# no soporta la ampliamente utilizada directiva `#include` de C++ (que no es necesaria porque C# no necesita declaraciones adelantadas).

La sintaxis de las directivas al preprocesador es la misma en C# que en C++. Las directivas soportadas por C# son:

Directiva	Significado
`#define/#undef`	El mismo que en C++, con la excepción de que deben aparecer al principio del fichero, antes de cualquier código C#.
`#if/#elif/#else/#endif`	El mismo de C++.
`#line`	El mismo de C++.
`#warning/#error`	El mismo de C++.
`#region/#endregion`	Marca zonas de código como una **región**. Las regiones son reconocidas por ciertos editores (como el editor de Visual Studio .NET) y pueden ser utilizadas para mejorar la disposición del código que se presenta al usuario durante la edición.