Uso de Win32 y otras bibliotecas

Eric Gunnerson
Microsoft Corporation

16 de septiembre de 2002

Descargue csharp09192002_sample.exe de MSDN Code Center (en inglés).

En mi anterior artículo, ofrecía una introducción general a las diferentes formas de utilizar código existente desde C#. En esta ocasión, vamos a profundizar en el uso de Win32® y otras bibliotecas existentes desde nuestro código.

Dos preguntas que se plantean con frecuencia los usuarios de C# son: "¿Por qué tengo que escribir código especial para utilizar elementos que están integrados en Windows®? y ¿Por qué el marco no puede hacerlo directamente?". Cuando el equipo de marcos llevaba a cabo su trabajo en .NET, se plantearon el modo de hacer que Win32 estuviese disponible para los programadores de .NET. Descubrieron que el conjunto de API de Win32 era *inmenso*. No disponían de los recursos necesarios para codificar, probar y documentar interfaces administradas para la totalidad de Win32, por lo que se vieron obligados a establecer prioridades y centrarse en las más relevantes. Aunque numerosas operaciones habituales poseen interfaces administradas, existen secciones completas de Win32 que no quedan cubiertas.

El método más habitual de realizar esta acción es a través de Platform Invocation Services (servicios de invocación de plataforma; P/Invoke). Para utilizar P/Invoke, es necesario escribir un prototipo que describa el modo de llamar a la función. A continuación, el tiempo de ejecución utiliza esta información para realizar la llamada. Otra forma consiste en empaquetar las funciones mediante el uso de Extensiones administradas para C++, tema del que trataré en un próximo artículo.

La mejor forma de comprender esto es a través de ejemplos. En ciertos casos, únicamente presentaré parte del código; el código completo forma parte de la descarga.

Un ejemplo sencillo

En nuestro primer ejemplo, llamaremos a la API `Beep()` para que realice un sonido. Para comenzar, necesito escribir la definición apropiada de `Beep()`. Al examinar la definición en MSDN, averiguo que posee el siguiente prototipo:

```
BOOL Beep(
  DWORD dwFreq,      // frecuencia del sonido
  DWORD dwDuration   // duración del sonido
);
```

Para escribir este código en C#, necesito traducir los tipos de Win32 a los tipos adecuados de C#. Dado que `DWORD` es un entero de 4 bytes, se podría utilizar indistintamente `int` o `uint` como análogo en C#. Debido a que int es un tipo compatible con CLS (y, por tanto, se puede utilizar en todos los lenguajes .NET), se utiliza con más frecuencia que `uint`; no obstante, en la mayoría de los casos, la diferencia es irrelevante. El tipo `bool` es el análogo de `BOOL`. Por tanto, podemos escribir el siguiente prototipo en C#:

```
public static extern bool Beep(int frequency, int duration);
```

Se trata de una definición bastante estándar, si bien hemos utilizado "extern" para indicar que el código real para esta función se encuentra en otro lugar. Este prototipo indicará al tiempo de ejecución cómo llamar a la función; ahora debemos mostrarle dónde encontrarla.

Llegados a este punto, debemos volver a MSDN. En la información de referencia, encontramos que `Beep()` se define en `kernel32.lib`. Este hecho indica que el código del tiempo de ejecución está contenido en `kernel32.dll`. Por tanto, colocaremos un atributo DllImport en nuestro prototipo para indicar al tiempo de ejecución la ubicación del código:

```
[DllImport("kernel32.dll")]
```

Y eso es todo lo que debemos hacer. A continuación se muestra un ejemplo completo que genera el tipo de notas aleatorias tan habituales en ciertas películas de ciencia ficción de los años 60 de mala calidad.

```
using System;
using System.Runtime.InteropServices;

namespace Beep
{
   class Class1
   {
      [DllImport("kernel32.dll")]
      public static extern bool Beep(int frequency, int duration);

      static void Main(string[] args)
      {
         Random random = new Random();

         for (int i = 0; i < 10000; i++)
         {
            Beep(random.Next(10000), 100);
         }
      }
   }
}
```

Con toda seguridad este sonido molestará a cualquiera que lo oiga. Puesto que DllImport permite llamar a código arbitrario de Win32, existe la posibilidad de que se incluya código malintencionado. Por tanto, el tiempo de ejecución comprueba que el usuario es de confianza antes de que éste pueda realizar llamadas P/Invoke.

Enumeraciones y constantes

`Beep()` es perfecta para reproducir un sonido aleatorio. Sin embargo, en ocasiones deseamos reproducir el sonido asociado a un tipo de sonido determinado. En estos casos utilizaremos, en su lugar, `MessageBeep()`. MSDN ofrece el siguiente prototipo:

```
BOOL MessageBeep(
  UINT uType   // tipo de sonido
);
```

Parece bastante sencillo. Sin embargo, al leer las notas, observamos dos aspectos interesantes.

En primer lugar, el parámetro `uType` toma un conjunto de constantes predefinidas.

En segundo lugar, uno de los valores que puede tomar el parámetro es -1. Esto conlleva que, aunque está definido para tomar `uint`, resulta más apropiado tomar `int`.

Utilizar una enumeración es lo más lógico para un parámetro `uType`. MSDN enumera las constantes nombradas, pero no indica cuáles son los valores. Para ello, debemos examinar las API.

Si dispone de Visual Studio® y ha instalado C++, Platform SDK se encuentra en **\Archivos de programa\Microsoft Visual Studio .NET\Vc7\PlatformSDK\Include**.

Para hallar las constantes, simplemente tuve que ejecutar findstr en ese directorio:

```
findstr "MB_ICONHAND" *.h
```

De este modo, localicé las constantes en `winuser.h` y las utilicé para crear la enumeración y el prototipo:

```
public enum BeepType
{
   SimpleBeep = -1,
   IconAsterisk = 0x00000040,
   IconExclamation = 0x00000030,
   IconHand = 0x00000010,
   IconQuestion = 0x00000020,
   Ok = 0x00000000,
}

[DllImport("user32.dll")]
public static extern bool MessageBeep(BeepType beepType);
```

Ahora puedo llamarlo mediante:

```
MessageBeep(BeepType.IconQuestion);
```

Uso de estructuras

En algunas ocasiones me hubiese gustado poder saber cuál era el estado de la batería de mi portátil. Win32 ofrece funciones de administración de energía que permiten obtener esta información.

Si buscamos un poco en MSDN, encontraremos la función `GetSystemPowerStatus()`.

```
BOOL GetSystemPowerStatus(
  LPSYSTEM_POWER_STATUS lpSystemPowerStatus
);
```

Esta función toma un puntero a una estructura; algo que aún no hemos tratado. Para trabajar con estructuras, debemos, en primer lugar, definirlas en C#. Comenzaremos con la definición no administrada:

```
typedef struct _SYSTEM_POWER_STATUS {
  BYTE   ACLineStatus;
  BYTE   BatteryFlag;
  BYTE   BatteryLifePercent;
  BYTE   Reserved1;
  DWORD  BatteryLifeTime;
  DWORD  BatteryFullLifeTime;
} SYSTEM_POWER_STATUS, *LPSYSTEM_POWER_STATUS;
```

A continuación escribimos la definición en C#, para lo cual sustituimos los tipos de C por los de C#.

```
struct SystemPowerStatus
{
   byte ACLineStatus;
   byte batteryFlag;
   byte batteryLifePercent;
   byte reserved1;
   int  batteryLifeTime;
   int  batteryFullLifeTime;
}
```

Una vez realizado esto, resulta sencillo escribir el prototipo C#:

```
[DllImport("kernel32.dll")]
public static extern bool GetSystemPowerStatus(
   ref SystemPowerStatus systemPowerStatus);
```

En este prototipo, utilizamos "`ref`" para indicar que estamos pasando un puntero a la estructura, en lugar de la estructura por valor. Este es el método habitual de trabajar con las estructuras que se pasan mediante punteros.

Aunque esta función opera correctamente, los campos `ACLineStatus` y `batteryFlag` se definen mejor como enumeraciones:

```
   enum ACLineStatus: byte
   {
      Offline = 0,
      Online = 1,
      Unknown = 255,
   }

   enum BatteryFlag: byte
   {
      High = 1,
      Low = 2,
      Critical = 4,
      Charging = 8,
      NoSystemBattery = 128,
      Unknown = 255,
   }
```

Observe que, dado que los campos de la estructura son bytes, utilizamos `byte` como el tipo base para la enumeración.

Cadenas

Si bien únicamente existe un tipo de cadena .NET, en el ámbito de lo no administrado encontramos múltiples posibilidades. Existen punteros de caracteres y estructuras con matrices de caracteres incrustadas, las referencias de cada una de las cuales se deben calcular correctamente.

Asimismo, en Win32 se utilizan dos representaciones de cadena diferentes:

- ANSI
- Unicode

La línea original de Windows utilizaba caracteres de un solo byte que ahorraban espacio de almacenamiento. No obstante, presentaba una codificación compleja de múltiples bytes para numerosos idiomas. Cuando surgió Windows NT®, utilizaba una codificación Unicode de dos bytes. Para solucionar esta diferencia, la API de Win32 usa una inteligente estratagema. Define un tipo `TCHAR`, que es un carácter de un solo byte en las plataformas Win9x y un carácter Unicode de dos bytes en las plataformas WinNT. Para cada función que

toma una cadena o una estructura que contiene datos de caracteres, define dos versiones de dicha estructura, con un sufijo `A` que indica que es Ansi y `W` que indica que es amplia (es decir, Unicode). Si compila un programa C++ para bytes únicos, obtendrá la variante `A`. Si, por el contrario, compila para Unicode, obtendrá la variante `W`. Las plataformas Win9x contienen la versión Ansi, mientras que las plataformas WinNT contienen la versión `W`.

Dado que los diseñadores de P/Invoke quisieron evitarle la molestia de tener que averiguar la plataforma en la que se encuentra, proporcionaron compatibilidad integrada para utilizar bien la versión `A` o la `W` de forma automática. Si la función a la que llama no existe, la capa de interoperabilidad buscará la versión `A` o `W` automáticamente y la utilizará en su lugar.

Existen ciertos detalles de la compatibilidad para cadenas que se pueden explicar con más claridad mediante ejemplos.

Cadenas sencillas

A continuación se muestra un ejemplo sencillo de una función que toma un parámetro de cadena:

```
BOOL GetDiskFreeSpace(
  LPCTSTR lpRootPathName,          // ruta de acceso raíz
  LPDWORD lpSectorsPerCluster,     // sectores por clúster
  LPDWORD lpBytesPerSector,        // bytes por sector
  LPDWORD lpNumberOfFreeClusters,  // clústeres libres
  LPDWORD lpTotalNumberOfClusters  // clústeres totales
);
```

La ruta de acceso raíz se define como `LPCTSTR`. Esta es la versión independiente de la plataforma de un puntero de cadena.

Puesto que no hay ninguna función llamada `GetDiskFreeSpace()`, el contador de referencias buscará automáticamente la variante "`A`" o "`W`" y llamará a la función apropiada. Utilizaremos un atributo para indicar al contador de referencias el tipo de cadena que requiere la API.

A continuación se muestra la definición completa para la función, tal como la definí inicialmente:

```
[DllImport("kernel32.dll")]
static extern bool GetDiskFreeSpace(
   [MarshalAs(UnmanagedType.LPTStr)]
   string rootPathName,
   ref int sectorsPerCluster,
   ref int bytesPerSector,
   ref int numberOfFreeClusters,
   ref int totalNumberOfClusters);
```

Desafortunadamente, cuando probé este código no funcionó. El problema radica en que, de forma predeterminada, el contador de referencias trata de hallar la versión Ansi de una API sin tener en cuenta el sistema en el que nos encontramos, y como el código `LPTStr` significa que las cadenas de Unicode se deben utilizar en las plataformas Windows NT, acabamos intentando llamar a la función Ansi con una cadena de Unicode; y eso no funciona.

Hay dos formas de conseguir que funcione. La más sencilla consiste simplemente en quitar el atributo `MarshalAs`. De esta forma, siempre se llamará a la versión `A` de la función; esto ya es suficiente si ésta existe en todas las variantes de plataforma que le puedan interesar. Sin embargo, con esta opción el código se ralentiza, puesto que el contador de referencias transforma la cadena .NET de Unicode a múltiples bytes, llama a la versión `A` de la función, que convertirá la cadena de nuevo en Unicode, y, a continuación, llama a la versión `W` de la función.

Para evitar esto, deberá indicar al contador de referencias que busque la versión `A` cuando se encuentre en plataformas Win9x y a la versión W cuando se encuentre en plataformas NT. Para ello, defina `CharSet` como parte del atributo `DllImport`:

```
[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
```

Comprobé que esta opción resultaba un 5 por ciento más rápida que la opción anterior.

Definir el atributo `CharSet` y utilizar `LPTStr` para los tipos de cadena funciona para la mayor parte de las API de Win32. Sin embargo, existen otras funciones que no utilizan el mecanismo `A/W`, y para las que deberá proceder de un modo diferente.

Búferes de cadena

El tipo de cadena en .NET es inmutable, lo que significa que su valor nunca varía. Para las funciones que copian un valor de cadena en un búfer de cadena, no se podrá utilizar una cadena. Si se hiciese, en el mejor de los casos, se dañaría el búfer temporal creado por el contador de referencias al traducir una cadena. Pero aún podría ocurrir algo peor: que se dañase el montón administrado, lo que generalmente se traduce en graves consecuencias. En cualquier caso, resulta poco probable que se devuelva el valor correcto.

Para conseguir que esto funcione, deberemos utilizar un tipo diferente. El tipo `StringBuilder` se ha diseñado para actuar como búfer, y lo utilizaremos en lugar de la cadena. Veamos un ejemplo:

```
[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
```

```
public static extern int GetShortPathName(
   [MarshalAs(UnmanagedType.LPTStr)]
   string path,
   [MarshalAs(UnmanagedType.LPTStr)]
   StringBuilder shortPath,
   int shortPathLength);
```

Utilizar esta función resulta sencillo:

```
StringBuilder shortPath = new StringBuilder(80);
int result = GetShortPathName(
@"d:\test.jpg", shortPath, shortPath.Capacity);
string s = shortPath.ToString();
```

Observe que `Capacity` de `StringBuilder` se pasa como el tamaño del búfer.

Estructuras con matrices de caracteres incrustadas

Ciertas funciones toman estructuras que presentan matrices de caracteres incrustadas. Por ejemplo, la función `GetTimeZoneInformation()` toma un puntero a la siguiente estructura:

```
typedef struct _TIME_ZONE_INFORMATION {
    LONG       Bias;
    WCHAR      StandardName[ 32 ];
    SYSTEMTIME StandardDate;
    LONG       StandardBias;
    WCHAR      DaylightName[ 32 ];
    SYSTEMTIME DaylightDate;
    LONG       DaylightBias;
} TIME_ZONE_INFORMATION, *PTIME_ZONE_INFORMATION;
```

Para utilizar este código desde C# se requieren dos estructuras. `SYSTEMTIME` es fácil de configurar:

```
   struct SystemTime
   {
      public short wYear;
      public short wMonth;
      public short wDayOfWeek;
      public short wDay;
      public short wHour;
      public short wMinute;
      public short wSecond;
      public short wMilliseconds;
   }
```

Hasta aquí ningún problema. Sin embargo, la definición de `TimeZoneInformation` es algo más complicada:

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
struct TimeZoneInformation
{
   public int bias;
   [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
   public string standardName;
   SystemTime standardDate;
   public int standardBias;
   [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
   public string daylightName;
   SystemTime daylightDate;
   public int daylightBias;
}
```

En esta definición debemos destacar dos detalles de gran importancia. El primero de ellos es el atributo `MarshalAs`:

```
   [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
```

Si echamos un vistazo a los documentos sobre `ByValTStr`, comprobaremos que éste se utiliza para matrices de caracteres incrustadas. `SizeConst` se utiliza para definir el tamaño de las matrices.

La primera vez que codifiqué esta cadena, surgieron errores en el motor de ejecución. Este tipo de errores suele indicar que se está sobrescribiendo cierta memoria como parte de la interoperabilidad, lo que sugería que el tamaño de la estructura era incorrecto. Por tanto, utilicé Marshal.SizeOf() para obtener el tamaño que el contador de referencias utilizaba: 108 bytes. Tras una breve investigación, pronto recordé que el tipo de carácter predeterminado para la interoperabilidad es Ansi, es decir, un solo byte. Los caracteres de la definición de la función son de tipo `WCHAR`, es decir, de dos bytes, y esa era la causa del problema.

Solucioné el error agregando el atributo `StructLayout`. El diseño predeterminado para las estructuras es secuencial, lo que conlleva que todos los campos se distribuyen en el orden en el que aparecen enumerados. El valor `CharSet` se define como Unicode para utilizar

siempre el tipo de carácter adecuado.

Una vez hecho esto, la función funcionaba correctamente. Quizás se esté preguntando porqué no utilicé `CharSet.Auto` en esta función. Esta es una de esas funciones que no tienen variantes `A` y `W`, sino que utilizan siempre cadenas Unicode. Por ese motivo la incluí en un código no modificable.

Funciones con devoluciones de llamada

Cuando las funciones de Win32 deben devolver más de un elemento de datos, generalmente lo hacen a través de un mecanismo de devolución de llamada. El desarrollador pasa un puntero de función a la función. A continuación, se hace una llamada a la función del desarrollador para cada elemento que aparezca enumerado.

En lugar de tener punteros de función, C# posee delegados, que se utilizan como sustitutos de los punteros de función al llamar a las funciones de Win32.

Un ejemplo de este tipo de funciones es `EnumDesktops()`:

```
BOOL EnumDesktops(
  HWINSTA hwinsta,            // identificador de una estación de ventanas
  DESKTOPENUMPROC lpEnumFunc, // función de devolución de llamada
  LPARAM lParam               // valor de la función de devolución de llamada
);
```

El tipo `HWINSTA` se sustituye por `IntPtr`, y `LPARAM` por int. `DESKTOPENUMPROC` resulta algo más complicado. Esta es la definición de MSDN:

```
BOOL CALLBACK EnumDesktopProc(
  LPTSTR lpszDesktop,  // nombre del escritorio
  LPARAM lParam        // valor definido por el usuario
);
```

Podemos convertir esta definición en el siguiente delegado:

```
delegate bool EnumDesktopProc(
   [MarshalAs(UnmanagedType.LPTStr)]
   string desktopName,
   int lParam);
```

Una vez definido, podemos escribir la definición para `EnumDesktops()`:

```
[DllImport("user32.dll", CharSet = CharSet.Auto)]
static extern bool EnumDesktops(
   IntPtr windowStation,
   EnumDesktopProc callback,
   int lParam);
```

Con esto es suficiente para que la función opere correctamente.

Cuando utilice delegados en la capa de interoperabilidad, tenga en cuenta la siguiente sugerencia importante: El contador de referencias crea un puntero de función que hace referencia al delegado, y ese es el puntero de función que se pasa a la función no administrada. El contador de referencias no puede conocer, sin embargo, qué es lo que hace la función no administrada con el puntero de función, por lo que asume que sólo debe ser válido durante la llamada a la función.

Esto quiere decir que si llama a una función, por ejemplo `SetConsoleCtrlHandler()`, en la que el puntero de función se guarda para su uso posterior, debe asegurarse de que en el código se hace referencia al delegado. De lo contrario, la función puede parecer que funciona correctamente, pero la próxima recolección de elementos no utilizados eliminará el delegado, lo que causará problemas serios.

Funciones más avanzadas

Todos los ejemplos que hemos visto hasta ahora han sido relativamente sencillos, pero existen numerosas funciones de Win32 con un mayor nivel de complejidad. Veamos un ejemplo:

```
DWORD SetEntriesInAcl(
  ULONG cCountOfExplicitEntries,           // número de entradas
  PEXPLICIT_ACCESS pListOfExplicitEntries, // búfer
  PACL OldAcl,                             // ACL original
  PACL *NewAcl                             // ACL nueva
);
```

El control de los dos primeros parámetros resulta bastante sencillo. `ulong` carece de cualquier complejidad, y se pueden calcular las referencias del búfer utilizando `UnmanagedType.LPArray`.

En cambio, el tercer y cuarto parámetro plantean ciertos conflictos. El problema principal radica en la forma en que se define una `ACL`. La estructura `ACL` únicamente define el encabezado de la ACL, mientras que el resto del búfer se compone de varios elementos `ACE`. Este

elemento puede ser uno de los diferentes tipos de `ACE`, los cuales tienen diferentes longitudes.

Puede resolver este problema en C#, siempre que esté dispuesto a realizar la asignación del búfer y utilizar una buena cantidad de código no seguro. Sin embargo, esto conlleva una inmensa labor y su depuración resultará extremadamente difícil. Es mucho más sencillo realizar esta API en C++.

Otras opciones para los atributos

Los atributos `DLLImport` y `StructLayout` poseen una serie de opciones de gran utilidad cuando se utiliza P/Invoke. He realizado un listado completo de todas ellas.

DLLImport

CallingConvention

Se utiliza para indicar al contador de referencias la convención de llamada que utiliza la función. Debe definir esta opción en la convención de llamada de su función. En general, si utiliza esta opción incorrectamente, el código no funcionará. Sin embargo, si se trata de una función `Cdecl` y hace la llamada mediante `StdCall` (predeterminado), la función operará correctamente, aunque los parámetros de ésta nunca se quitarán de la pila, por lo que esta última se podría saturar.

CharSet

Controla si se llama o no a las variantes `A` o `W`.

EntryPoint

Esta propiedad se utiliza para definir el nombre que buscará el contador de referencias en la DLL. Al definir esta propiedad, puede cambiar el nombre de la función C#.

ExactSpelling

Si define esta propiedad como True, el contador de referencias desactiva la búsqueda de `A` y `W`.

PreserveSig

La interoperabilidad COM hace que una función con un parámetro de salida final parezca que devuelve ese valor. Mediante esta propiedad se evita que esto ocurra.

SetLastError

Comprueba que se llama a la API de Win32 `SetLastError()`, de modo que se pueda averiguar la causa del error.

StructLayout

LayoutKind

La distribución predeterminada de las estructuras es secuencial, lo que funciona en la mayoría de las ocasiones. Si necesita controlar con exactitud la ubicación de los miembros de la estructura, puede utilizar `LayoutKind.Explicit` y, a continuación, colocar un atributo `FieldOffset` en cada miembro de la estructura. Esto es lo que se suele hacer para crear una unión.

CharSet

Controla el tipo de caracteres predeterminado de los miembros `ByValTStr`.

Pack

Define el tamaño del empaquetado de la estructura, lo que permite controlar la alineación de la misma. Si la estructura de C utiliza un empaquetado diferente, puede que tenga que definir esta propiedad.

Size

Define el tamaño de la estructura. No se usa habitualmente, pero puede resultar útil si necesita espacio asignado adicional al final de la estructura.

Carga desde ubicaciones diferentes

No hay forma de especificar la ubicación en la que desea que DLLImport busque un archivo en tiempo de ejecución. Sin embargo, hay un truco al que puede recurrir.

DllImport llama a `LoadLibrary()` para cumplir su función. Si ya se ha cargado una DLL específica en un proceso determinado, `LoadLibrary()` tendrá éxito, aún cuando la ruta especificada para la carga sea diferente.

Esto significa que si llama a `LoadLibrary()` directamente, podrá cargar la DLL desde cualquier ubicación, y a continuación, `LoadLibrary()` de DllImport utilizará esa versión.

Debido a este comportamiento, se puede llamar a `LoadLibrary()` de manera anticipada para reenviar las llamadas a una DLL distinta. Si va a escribir una biblioteca, puede evitar que esto ocurra llamando a `GetModuleHandle()`. De este modo, podrá asegurarse antes de realizar la primera llamada P/Invoke de que la biblioteca no se ha cargado previamente.

Solución de problemas de P/Invoke

Cuando las llamadas P/Invoke dan error, generalmente se debe a que ciertos tipos no se han definido correctamente. Aquí se muestran algunos de los problemas más frecuentes:

- `Long` != `long`. En C++, `long` es un entero de 4 bytes, mientras que en C# es un entero de 8 bytes.
- El tipo de cadena no se define correctamente.

Para el próximo mes

El próximo mes, trataremos las funciones de empaquetado en C++.

Eric Gunnerson es administrador de programas del equipo de Visual C#, miembro del equipo de diseño de C# y autor de la introducción a C# para programadores, A Programmer's Introduction to C# (en inglés). Posee la suficiente experiencia como para reconocer qué es un disquete de 8 pulgadas y hasta es capaz de montar cintas con una sola mano. Durante su tiempo libre, realiza una tesis de doctorado sobre malabarismos con gatos, centrándose particularmente en los persas.

Ultima actualización: 12 de Diciembre de 2002