

Title: **Calling C Library DLLs from C#**
 Utilizing legacy software

Summary: For those who need to utilize legacy software, we provide techniques for calling unmanaged code written in C from C#.

The .NET framework was designed to be the “lingua franca” for Windows development, with the expectation that it will set a new standard for building integrated software for Windows. However, it is inevitable that there is a time lag before .NET is fully adopted and existing applications are recoded. In particular, there is a large body of legacy code that will likely never be rewritten in .NET. To address this situation, Microsoft provides attributes, assembly, and marshaling. At the Numerical Algorithms Group (where I work), our particular interest in using these techniques is to utilize numerical software developed in C from within the .NET environment. Because C# is the premier .NET language, the examples I present here are in C#. While I use an example of data types that are current in the NAG C Library, the techniques I present are general enough for calling unmanaged code written in C from C# directly.

The NAG C Library uses the following data types as parameters:

- Scalars of type *double*, *int*, and *Complex*. These are passed either by value or by reference (as pointers to the particular type).
- *enum* types.
- Arrays of type *double*, *int*, and *Complex*.
- A large number of structures, generally passed as pointers.
- A few instance of arrays which are allocated within NAG routines and have to be freed by users (these have type *double***).
- Function parameters (also know as “callbacks”). These are pointers to functions with particular signatures.

For convenience, I include a C DLL containing definitions of all the functions being called from C#. This DLL is available electronically from DDJ (see “Resource Center,” page 4) and NAG (<http://www.nag.com/public/ddj.asp>).

For instance, take the example of a C function that takes two parameters of the type *double* and *double**; that is, the first parameter is input (pass by value) and the second is output (pass by reference in non-C parlance). The corresponding C# signature for the C function is then *double, ref double*. Listing One presents the definition of the C function and its call from C#. In C#, you have to provide the DLL import attribute (line 5), specifying how the C signature maps to C#. Also the qualifier *ref* has to be used twice, in the declaration of the C function and in its call. Finally, note the use of the assembly directive, *System.Runtime.InteropServices* (line 3), which is important because it is the classes defined within the *InteropServices* that provide the mapping between managed code and unmanaged code.

Arrays are the bedrock of numerical programming. By definition arrays are passed by reference in both C and C#. A C function having a one-dimensional array as a parameter with the prototype:

```
void OneDArray(double AnArray[ ]);
```

has the C# signature given by:

```
public static extern void OneDArray(double [] AnArray);
```

With this proviso, the call is straightforward; see Listing Two.

Two-dimensional arrays are more interesting. In C, two-dimensional arrays are of the type pointer to pointer. For example, a two-dimensional array of doubles would have the prototype, *double ***. However, these are rarely used in practice as they imply noncontiguous storage, whereas numerics are best carried out using contiguous storage. Numerical C code using “notional” two-dimensional arrays frequently store data in row-major order. For example, to read a two-dimensional array of size $m*n$ (m rows and n columns) into contiguous storage, you might have the following:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    scanf("%lf", a[i*tda+j]);
```

where *tda* is the second dimension of *a*, in this case $tda=n$.

In Fortran the equivalent construction would be:

```
DO 10 I=1,M
  DO 10 J=1,N
  READ(5,*) A(I,J)
10 CONTINUE
```

Here the $A(I,J)$ construction is equivalent to (using C notation):

```
a[(J-1)*lda + I-1]
```

where *lda* refers to the first dimension of *a*. This implies column major storage starting with indices starting at 1 rather than 0.

The point to note is that the $A(I,J)$ notation in Fortran squirrels away the complexity of array element access. If we had such a notation in C, it might have been $A[i,j]$. This is precisely what you have in C#. Hence, the notional two-dimensional array in C, *double fred[]* is represented in C# as *double [,] fred*. Listing Three shows an example of the use of a C function using a two-dimensional array from C#. It is worth noting how well the “notional” two-dimensional C array dovetails with the C# *double* type. The C# array is a proper class, with members available to provide us with information on the dimensions of the array; hence, the C# member function *callTwoDarray* needs to have just the one parameter.

Struct is a major type in C but in C# (being a value type), it is but a poor cousin to the class type. However, it can be mapped to a *struct* type in C#. The simplest and the most ubiquitous structure in numerics is the complex type, being defined as a pair of *reals*. In Listing Four, the *struct* type is defined in C and its equivalent in C#. In particular, you have to tell the C# compiler that the structure members are laid out sequentially in memory by the use of the attribute, $[StructLayout(LayoutKind.Sequential)]$. Given this information, you can treat the complex type as a regular object, passing it by value, reference, or as an array. Listing Four shows how you can access information from a C function of the type *Complex*,

which has three parameters, *inputVar* passed by value, *OutputVar* passed by reference, and an array of the *Complex* type. There is one further point to note here: As structures are of the type *value* in C#, we have to tell the compiler whether arrays are read or write. You do this by providing the attribute '[In, Out]' to the *array* parameter.

Structures can get very complicated indeed. Structure members can be scalars, arrays, pointers to other structures, and the like. Pointers being taboo (or at least very undesirable) in C# can be specified in C# as the *IntPtr* type. Listing Five presents a C and C# example showing the use of a *Marshal* class method to print the elements of an array that has been allocated internally within a C function. Memory has to be freed explicitly in this case within the unmanaged code.

Function parameters, also known as “callbacks”, play a central role in numerical software. These are required in cases where code to carry out some specific task has to be supplied to a function. This can occur, for example, in optimization software where the value of the objective or the first derivatives have to be computed on a problem-specific basis. The difficulty with callbacks is mainly that they imply a reversal of the situation I’ve been looking at so far; that is, managed code (in C#) calling unmanaged code. Instead, the callback function calls the managed code being called by C. C# provides the delegate type to cater for this situation. An appropriate signature of the callback function is provided and an instance of this type created using a construction such as:

```
NAG_E04CCC_FUN objfun = new NAG_E04CCC_FUN (funct);
```

Listing Six presents an example of a simple callback. This is a simple mechanism when the callback has simple types, but it gets more interesting when we have parameters of the callback that are arrays and structures that may have to carry information back to the unmanaged code. In this case, you have to use both marshaling techniques and attributes to the structure. This is illustrated in Listing Seven where I show how to handle arrays and structures within callbacks. The delegate in this case has an *array* parameter. If you use the following signature (which appears to be quite a reasonable signature in the first instance), we find that when the delegate is called from C, the array appears to be of length 1. This presumably has to do with the fact that C pointers do not carry any information on the length of the array. The trick is to specify the array pointer as of C# type *IntPtr* and subsequently copy data to and from the *IntPtr* parameter.

There are two more data types that occur in C and are worth mentioning. *Enum* types are an integral type that are mapped one to one between C and C#. Listing Eight illustrates how *enum* values may be passed from C# to C.

The final type to consider is the *string* type in C#, mapping to *char** type in C. When a *string* type is defined in C# and passed to C, the interop services provide the conversion to ASCII by default. I use the *StringBuilder* type because this is a reference type and can grow and shrink as required. Listing Eight illustrates a C function modifying a string. For users of the NAG C Library, we provide an *Assembly* of structures, functions, and delegate signatures within a *Nag* namespace. (See “Wrapping C with C# in .NET,” by George Levy, *C/C++ User Journal*, January 2004). We also provide examples of using this assembly from C# for some widely used NAG routines.

By Shah Datardina. Shah is a senior technical consultant for the Numerical Algorithms Group. He can be contacted at shah@nag.co.uk.

Courtesy of Dr. Dobb's Journal, www.ddj.com. Article originally published October 2005.

Numerical Algorithms Group

www.nag.com / info@nag.com (North America)

www.nag.co.uk / info@nag.co.uk (All Others)

Listing One

```

/*****
 *          C Function  Scalars          *
 *****/
#define NAG_CALL __stdcall
#define NAG_DLL_EXPIMP __declspec(dllexport)

NAG_DLL_EXPIMP void NAG_CALL Scalars(double, double*);
NAG_DLL_EXPIMP void NAG_CALL Scalars(double in, double *out)
{
    *out = in;
}
/*****
 *          C# Class          *
 *****/

using System;
using System.Runtime.InteropServices;
namespace DDJexamples
{
    public class ExerciseScalars
    {

        [DllImport("cmarshaldll")]
        public static extern void Scalars(double invar, ref double outvar);

        public static void CallScalars(double invar, ref double outvar)
        {
            Scalars(invar, ref outvar);
        }
        public static void Main()
        {
            double invar = 5.0;
            double outvar = 0.0;
            CallScalars(invar, ref outvar);
            Console.WriteLine("invar = {0}, outvar = {1}", invar, outvar);
        }
    }
}

```

Listing Two

```

/*****
 *                               C Function OneDArray                               *
 *****/

#define NAG_CALL __stdcall
#define NAG_DLL_EXPIMP __declspec(dllexport)

NAG_DLL_EXPIMP void NAG_CALL OneDArray(int n, double []);
NAG_DLL_EXPIMP void NAG_CALL OneDArray(int n, double anArray[])
{
    int i;
    for (i=0; i<n; i++)
        anArray[i] = 99.0;
}
/*****
 *                               C# Class                                       *
 *****/

using System;
using System.Runtime.InteropServices;
namespace DDJexamples
{

    public class ExerciseOneDArray
    {
        [DllImport("cmarshaldll")]
        public static extern void OneDArray(int n, double [] anArray);

        public static void CallOneDArray(double [] anArray)
        {
            OneDArray(anArray.GetLength(0), anArray);
        }
        public static void Main()
        {
            int n=2;
            double [] anArray = new double [n];
            CallOneDArray(anArray);
            for (int i=0; i<n; i++)
                Console.WriteLine("{0}", anArray[i]);
        }
    }
}

```

Listing Three

```

/*****
 *                               C Function TwoDArray                               *
 *****/

#define NAG_CALL __stdcall
#define NAG_DLL_EXPIMP __declspec(dllexport)

NAG_DLL_EXPIMP void NAG_CALL TwoDArray(int m, int n, double [], int tda);
#define A(I,J) a2dArray[I*tda+J]
NAG_DLL_EXPIMP void NAG_CALL TwoDArray(int m, int n, double a2dArray[], int tda)
{
    int i, j, k = 0;
    tda = n;
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            {
                A(i,j) = ++k;
            }
}
/*****
 *                               C# Class                                       *
 *****/

using System;
using System.Runtime.InteropServices;
namespace DDJexamples
{
    public class ExerciseTwoDArray
    {
        [DllImport("cmarshaldll")]
        public static extern void TwoDArray(int m, int n, double [,] a2dArray,
            int tda);

        public static void CallTwoDArray(double [,] a2dArray)
        {
            TwoDArray(a2dArray.GetLength(0), a2dArray.GetLength(1),
                a2dArray, a2dArray.GetLength(1));
        }
        public static void Main()
        {
            int m=3;
            int n=2;
            double [,] a2dArray = new double [m,n];
            CallTwoDArray(a2dArray);
            for (int i=0; i<m; i++)
            {
                for (int j=0; j<n; j++)
                    Console.Write("{0} ", a2dArray[i,j]);
                Console.WriteLine();
            }
        }
    }
}

```

Listing Four

```

/*****
 *          C Function TryComplex          *
 *****/

#define NAG_CALL __stdcall
#define NAG_DLL_EXPIMP __declspec(dllexport)

    typedef struct {                /* NAG Complex Data Type */
        double re,im;
    } Complex;

NAG_DLL_EXPIMP void NAG_CALL TryComplex(Complex inputVar, Complex *outputVar, int n, Complex array[]);
NAG_DLL_EXPIMP void NAG_CALL TryComplex(Complex inputVar, Complex *outputVar, int n, Complex array[])
{
    outputVar->re = ++inputVar.re;
    outputVar->im = ++inputVar.im;

    array[0].re = 99.0;
    array[0].im = 98.0;
    array[1].re = 97.0;
    array[1].im = 96.0;
}
/*****
 *          C# Class          *
 *****/
using System;
using System.Runtime.InteropServices;
namespace DDJexamples
{
    // Nag Complex type
    [StructLayout(LayoutKind.Sequential)]
    public struct Complex
    {
        public double re;
        public double im;
    };

    public class ExerciseTryComplex
    {
        [DllImport("cmarshaldll")]
        public static extern void TryComplex(Complex inputVar, ref Complex outputVar,
                                           int n, [In, Out] Complex [] array);

        public static void CallTryComplex(Complex inputVar, ref Complex outputVar, Complex [] array)
        {
            int n = 2;
            TryComplex(inputVar, ref outputVar, n, array);
        }
        public static void Main()
        {
            int n=2;
            Complex inputVar = new Complex();
            Complex outputVar = new Complex();
            Complex [] array = new Complex[n];
            inputVar.re = 1.0;
            inputVar.im = 2.0;

            CallTryComplex(inputVar, ref outputVar, array);
            Console.WriteLine("outputVar = ({0},{1})", outputVar.re, outputVar.im);
            Console.WriteLine("Array on output");
            for (int i = 0; i<array.GetLength(0); i++)
                Console.WriteLine("{0} {1}", array[i].re, array[i].im);
        }
    }
}

```

Listing Five

```

/*****
 *          C Function MarshalStructC          *
 *****/
#define NAG_CALL __stdcall
#define NAG_DLL_EXPIMP __declspec(dllexport)

#include <stdlib.h>
typedef struct
{
    int array_length;
    double *array;
} marshalStruct;
NAG_DLL_EXPIMP void NAG_CALL MarshalStructC(marshalStruct *pointerinStruct);
NAG_DLL_EXPIMP void NAG_CALL FreeMarshalStructptr(marshalStruct *pointerinStruct);
/**/
NAG_DLL_EXPIMP void NAG_CALL MarshalStructC(marshalStruct *pointerinStruct)
{
    int i;
    pointerinStruct->array = (double *)malloc((size_t)(sizeof(double)*pointerinStruct->array_length));
    for (i = 0; i < pointerinStruct->array_length; i++)
    {
        pointerinStruct->array[i] = (double)(i) + 1.0;
    }
}
NAG_DLL_EXPIMP void NAG_CALL FreeMarshalStructptr(marshalStruct *pointerinStruct)
{
    free(pointerinStruct->array);
    pointerinStruct->array = 0;
}
/*****
 *          C# Class          *
 *****/
using System;
using System.Runtime.InteropServices;
namespace DDJexamples
{
    [StructLayout(LayoutKind.Sequential)]
    public struct marshalStruct
    {
        public int array_length;
        public IntPtr array;
    };

    public class ExerciseMarshalStructC
    {
        [DllImport("cmarshaldll")]
        public static extern void MarshalStructC(ref marshalStruct pointerinStruct);

        [DllImport("cmarshaldll")]
        public static extern void FreeMarshalStructptr(ref marshalStruct pointerinStruct);

        public static void CallMarshalStructC(ref marshalStruct pointerinStruct)
        {
            MarshalStructC(ref pointerinStruct);
        }
        public static void Main()
        {
            marshalStruct pointerinStruct = new marshalStruct();
            pointerinStruct.array_length = 5;
            CallMarshalStructC(ref pointerinStruct);
            double [] x = new double[pointerinStruct.array_length];
            Marshal.Copy(pointerinStruct.array, x, 0, pointerinStruct.array_length);
            for (int i = 0; i < pointerinStruct.array_length; i++)
                Console.WriteLine("x[{0}] = {1}", i, x[i]);
            FreeMarshalStructptr(ref pointerinStruct);
        }
    }
}

```

Listing Six

```

/*****
 *                               C Function Callback                               *
 *****/

#define NAG_CALL __stdcall
#define NAG_DLL_EXPIMP __declspec(dllexport)

typedef void (NAG_CALL * NAG_D01_FUN)(double *);
NAG_DLL_EXPIMP void NAG_CALL Callback(NAG_D01_FUN f , double *output);
NAG_DLL_EXPIMP void NAG_CALL f(double *x);

/* */
NAG_DLL_EXPIMP void NAG_CALL Callback(NAG_D01_FUN f , double *output)
{
    (*f)(output);
}
NAG_DLL_EXPIMP void NAG_CALL f(double *x)
{
    *x = 100.0;
}
/*****
 *                               C# Class                               *
 *****/
using System;
using System.Runtime.InteropServices;
namespace DDJexamples
{
    // delegate
    public delegate void NAG_D01_FUN (ref double output);

    public class ExerciseSimpleCallback
    {
        [DllImport("cmarshaldll")]
        public static extern void Callback(NAG_D01_FUN f , ref double output);

        public static void CallCallback(NAG_D01_FUN f, ref double output)
        {
            Callback(f, ref output);
        }
        public static void Main()
        {
            double output = 0.0;
            NAG_D01_FUN F = new NAG_D01_FUN (f);
            CallCallback(F, ref output);
            Console.WriteLine("Ouput = {0}", output);
        }
        public static void f(ref double output)
        {
            {
                output = 100.0;
            }
        }
    }
}

```

Listing Seven

```

/*****
 *          C Function CallbackWithStruct          *
 *****/
#define NAG_CALL __stdcall
#define NAG_DLL_EXPIMP __declspec(dllexport)

typedef struct {
    int flag;
} Nag_Comm;

typedef void (NAG_CALL * NAG_E04UCC_FUN)(int, double *, Nag_Comm *);

extern NAG_DLL_EXPIMP void NAG_CALL CallbackWithStruct(NAG_E04UCC_FUN funct, int array_length,
double *a, Nag_Comm *user_comm);
void NAG_CALL funct(int n, double *x, Nag_Comm *user_comm);

/* */
NAG_DLL_EXPIMP void NAG_CALL CallbackWithStruct(NAG_E04UCC_FUN funct , int n, double *a, Nag_Comm
*user_comm)
{
    (*funct)(n, a, user_comm);
    if (user_comm->flag == 99)
    {
        a[0] = 99.0;
    }
}

void NAG_CALL funct(int n, double *x, Nag_Comm *user_comm)
{
    int i;
    for (i=0; i<n; i++)
    {
        x[i]++;
    }
    if (x[0] < 3.0)
    {
        user_comm->flag = 99;
    }
}

/*****
 *          C# Class          *
 *****/
using System;
using System.Runtime.InteropServices;
namespace DDJexamples
{

    [StructLayout(LayoutKind.Sequential)]
    public struct Nag_Comm
    {
        public int flag;
    };

    // delegate
    public delegate void NAG_E04UCC_FUN (int array_length, IntPtr a, ref Nag_Comm comm);

    public class ExerciseCallBackWithStruct
    {
        [DllImport("cmarshaldll")]
        public static extern void CallbackWithStruct(NAG_E04UCC_FUN f , int array_length, double
[] a, ref Nag_Comm user_commt);
    }
}

```

```
public static void CallCallbackWithStruct(NAG_E04UCC_FUN f, int array_length, double [] a,
ref Nag_Comm user_comm)
{
    CallbackWithStruct(f, array_length, a, ref user_comm);
}
public static void Main()
{
    double [] a = {1.0, 2.0, 3.0, 4.0, 5.0};
    int array_length = a.GetLength(0);
    Nag_Comm user_comm = new Nag_Comm();
    NAG_E04UCC_FUN F = new NAG_E04UCC_FUN (funct);
    CallCallbackWithStruct(F, array_length, a, ref user_comm);
    Console.WriteLine("user_comm.flag = {0}", user_comm.flag);
    Console.WriteLine("a[0] altered further as a result of user_comm.flag, a[0] = {0}",
a[0]);
}
public static void funct(int n, IntPtr xptr, ref Nag_Comm user_comm)
{
    double [] x = new double[n];
    Marshal.Copy( xptr, x, 0, n );

    int i;
    for (i=0; i<n; i++)
    {
        x[i]++;
    }
    if (x[0] < 3.0)
    {
        user_comm.flag = 99;
    }
    Marshal.Copy( x, 0, xptr, n);
}
}
```

Listing Eight

```

/*****
 *          C Function EnumString          *
 *****/
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define NAG_CALL __stdcall
#define NAG_DLL_EXPIMP __declspec(dllexport)

typedef enum { red=101, green, blue,black } colour;
NAG_DLL_EXPIMP void NAG_CALL EnumString(colour rainbow, char *rainbowcolour);
NAG_DLL_EXPIMP void NAG_CALL EnumString(colour rainbow, char *rainbowcolour)
{
    if (rainbow == black )
    {
        strcpy(rainbowcolour, "Black is not a rainbow colour");
    }
    else
    {
        strcpy(rainbowcolour, "This is a rainbow colour");
    }
}
/*****
 *          C# Class          *
 *****/
using System;
using System.Runtime.InteropServices;
using System.Text;
namespace DDJexamples
{
    public enum colour { red=101, green, blue,black };

    public class ExerciseEnumString
    {
        [DllImport("cmarshaldll")]
        public static extern void EnumString(colour rainbow, StringBuilder rainbowcolour);

        public static void CallEnumString(colour rainbow, StringBuilder rainbowcolour)
        {
            EnumString(rainbow, rainbowcolour);
        }
        public static void Main()
        {
            StringBuilder colourstring=new StringBuilder("once upon a time ... ");
            colour somecolour = colour.black;
            CallEnumString(somecolour, colourstring);
            Console.WriteLine("{0}", colourstring);
        }
    }
}

```