Passing Array

Möchte man .Net-Arrays an C++ übergeben, kommt man nicht darüber hinweg sich ein bisschen mit Marshalling zu befassen.

Nun was ist Marshalling?

Gemäss Wikipedia ist Marshalling das Umwandeln von strukturierten oder elementaren Daten in ein Format, das die Übermittlung an andere Prozesse oder Programme ermöglicht. Auf Empfängerseite werden aus diesem Format die Daten in ihrer ursprünglichen Struktur wiederhergestellt, was als Unmarshalling oder Demarshalling bezeichnet wird.

Und was bedeutet Marshalling im .Net-Framework?

In .Net heisst es Interop-Marshalling, und ist eine run-time Aktivität, die vom Marshalling-Service der Common Language Runtime zur Verfügung gestellt wird. Die meisten Datentypen in .Net besitzen im verwalteten (managed) wie auch im nichtverwaltetem (unmanaged) Speicher ähnliche oder sogar die gleichen Darstellungsformen. Interop-Marshalling “managed“ und “regelt“ quasi den korrekten Umgang primitiver Datentypen zwischen beider Speicher. Jedoch gilt das nicht für alle Datentypen, denn es gibt auch mehrdeutige Datentypen oder Datentypen die gar nicht dargestellt werden können.

Wie man erkennen kann ist Interop-Marshalling ein sehr umfangreiches Kapitel, die auch eine entsprechende vielfältige Klasse zur Verfügung stellt. Für das Übergeben von Arrays beschränken wir uns auf einen kleinen Rahmen von Pointers. Es ist nicht Ziel hier, plattformübergreifende oder sogar COM (Component Object Model) Überlegungen einfliessen zu lassen. Das würde den Rahmen dieses Tutorials sprengen. Es soll lediglich Möglichkeiten gezeigt werden, wie so etwas auf einfache Art bewerkstelligt werden kann.

Was wird hier gezeigt?

Dieses kleine Tutorial zeigt wie man Arrays der Datentypen Byte, Int32, Int64, Double und String mit wenig Aufwand korrekt an C++ (Nativ) übergeben kann. Hierbei handelt es sich um 1-, 2- oder 3-dimensionale Arrays in Form von Multidimensional oder Jagged Arrays.

Um den schweregrad ein bisschen zu erhöhen, werden die Array einmal auf der .Net-Seite erzeugt, und einmal auf der C++ Seite.

.Net-seitig erzeugte Arrays werden an C++ zum Füllen mit Werten übergeben. C++-seitig erzeugte Arrays werden gefüllt und an .Net übergeben. (sinnbildlich erklärt)

Liste primitiver Datentypen

Primitive Datentypen müssen nicht speziell behandelt werden, und können direkt so an C++ übergeben werden. Aus Performancegründen sollten primitive Datentypen immer bevorzugt werden, sofern es möglich ist.

.Net Type	C++ Type	stdint.h Type
byte	unsigned char	uint8_t
sbyte	char	int8_t
short	short	int16_t
ushort	unsigned short	uint16_t
char	unsigned short	uint16_t
double	double	
float	float	

int	int long	(Nur auf 32-bit Platformen)	int32_t
uint	unsigned int unsigned long	(Nur auf 32-bit Platformen)	uint32_t
long	long __int64 long long	(Nur auf 64-bit Platformen) MSVC GCC	int64_t
ulong	unsigned long unsigned __int64 unsigned long long	Nur auf 64-bit Platformen MSVC GCC	uint64_t

1D-Arrays

1-dimensionale Arrays mit primitiven Datentypen können gleich direkt an C++ übergeben werden, und müssen nicht vorgängig behandelt werden.

Ausnahme die String-Array. Streng genommen handelt es ich bei der String-Array nicht um einen primitiven Datentyp einerseits, und anderseits darf die String-Array nicht als 1-dimensionales Array betrachtet werden. String-Arrays werden ganz am Schluss erklärt.

Vb.Net

```
Dim b = New Byte(size - 1) {}

Dim len = FillRngByt(b, size)

<DllImport(DllName, EntryPoint:="fill_rng_byt",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function FillRngByt(buffer() As Byte, size As Int32) As Int32
End Function
```

C++

```
typedef int i32;
typedef unsigned char b8;

EXP32 i32 APIENTRY fill_rng_byt(b8 * buffer, i32 szbuffer);
```

Wie man erkennen kann wird die Methode `FillRngByt` mit dem Attribut `DllImport` für die unmanaged DLL (Dynamic Link Library) als statischer Einstiegspunkt verfügbar gemacht.

Mit `New Byte(size - 1) {}` wird eine Array erzeugt mit dem primitiven Datentyp Byte. Um mehr muss man in .Net sich nicht kümmern, den .Net übernimmt die Speicherfreigabe wie auch die Speicherbereichskontrolle. Darum wird es auch als “managed“ bezeichnet.

Der Methode `FillRngByt` wird eine Array mit dem primitiven Datentyp Byte übergeben, die von C++ als `unsigned char` (`b8`) in der Methode `fill_rng_byt` entgegen genommen wird.

C++ füllt nun den Speicherbereich hinter der Variable `buffer` (Pointer) mit Werten. D.h. ganz genau genommen wird der reservierte Speicher in .Net mit Werten gefüllt, und die Array b die eine Referenz auf den entsprechenden Speicher hält zeigt die gefüllten Werte.

Vb.Net

```
Dim b = New Byte(size - 1) {}
Dim ptr = CreateRngByt(size)
Marshal.Copy(ptr, b, 0, b.Length)

<DllImport(DllName, EntryPoint:="create_rng_byt",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function CreateRngByt(size As Int32) As IntPtr
End Function
```

C++

```
typedef int i32;
typedef unsigned char b8;

EXP32 b8 * APIENTRY create_rng_byt(i32 szbuffer);
```

Hier wird der Methode `CreateRngByt` nur die Arraylänge übergeben. Das lässt vermuten, dass die Array in C++ also unmanaged seitig erzeugt wird. Genauso ist es.

Damit das auch funktioniert, kann in C++ ein Puffer bereitgestellt werden, der den allozierten Speicherbereich festhält, bis er mit `delete` wieder freigegeben wird.

C++

```
b8** res = 0;
res = &BUFFER_BYT;
if (!*res)delete[](*res), (*res) = 0;
*res = new b8[szbuffer + 1];
```

Nun kann der Speicherbereich hinter `BUFFER_BYT` mit Werten gefüllt werden. `BUFFER_BYT` ist übrigens auch ein Pointer mit Adresse den man problemlos an .Net übergeben kann. .Net übernimmt den Pointer mit den Datentyp IntPtr.

Wichtig: Speicher der auf der unmanaged Seite reserviert wird, muss genau dort wieder freigegeben werden, um Memory-Leaks zu vermeiden. Dazu kann man sich ruhig mit einer separaten Methode bedienen, in dem der IntPtr an C++ wieder übergeben wird, und dort auch zurückgesetzt wird.

Oder man macht es so wie in diesem Falle hier, wo der Speicher hinter `BUFFER_BYT` durch die Variable bekannt ist und so bei Bedarf wiederum mittels einer Methode zurückgesetzt werden kann.

```
EXP32 void APIENTRY dispose_buffer();
```

Mehrdimensionale-Arrays

.Net verfügt über zwei Arten von mehrdimensionalen Arrays.

- Jagged 2D — `New Byte(size1 – 1)() {}`
- Jagged 3D — `New Byte(size1 – 1)()() {}`
- Multidimensional-Array 2D — `New Byte(size1 – 1, size2 - 1) {}`
- Multidimensional-Array 3D — `New Byte(size1 – 1, size2 - 1, size3 - 1) {}`

2D-3D-Arrays

Vb.Net

```
Dim emptyjagged2d = Get2dj(Of Byte)(size1, size2)
Dim ptr = agch.GetPtr(emptyjagged2d)
Dim len = FillRngByt2dj(ptr, size1, size2)
Dim jagged2d = GetJagged(Of Byte)(ptr, size1, size2)

<DllImport(DllName, EntryPoint:="fill_rng_byt_2d",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function FillRngByt2d(ptr As IntPtr, size1 As Int32, size2 As Int32) As Int32
End Function
```

Vb.Net

```
Dim emptyarray2d = Get2D(Of Byte)(size1, size2)
Dim ptr = Agch.GetPtr(emptyarray2d)
Dim len = FillRngByt2d(ptr, size1, size2)
Dim array2d = GetArray(Of Byte)(ptr, size1, size2)
```

```
<DllImport(DllName, EntryPoint:="fill_rng_byt_2dj",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function FillRngByt2dj(ptr As IntPtr, size1 As Int32, size2 As Int32) As Int32
End Function
```

Wie man erkennen kann, sind die Übergabeparameter für mehrdimensionale Arrays gleich. Es spielt keine Rolle, ob es sich um eine 2D oder 3D Array handelt. Der einzige Unterschied zeigt sich in der Anzahl der Size-Parameter. Für 2D Arrays müssen zwei Size-Parameter übergeben werden, bei 3D Arrays drei.

`emptyarray` wird zuerst in ein IntPtr gemarshallt. Dies ist wichtig, da die Speicherverwaltung von .Net und C++ sich grundlegend unterscheiden. Während in C++ durch den Aufruf von `new` und `delete` der Speicherbereich mit seiner zugewiesenen Adresse stets am selben Ort bleibt, verhält sich die Speicherverwaltung in .Net komplett anders. .Net besitzt einen `Garbage Collector` der sich um die angeforderten Speicherbereiche kümmert. Darum spricht man in Bezug auf die Speicherverwaltung jeweils von `managed` (.Net) und `unmanaged` (C++). Möchte man die Speicherverwaltung von .Net durch den `Garbage Collector` unterbinden, braucht es wie oben erwähnt das Interop-Marshalling.

C# besitzt noch die Möglichkeit den Speicherbereich mittels **fixed** während der gesamten Lebensdauer auf der gleichen Adresse festzuhalten. So etwas kennt man in Vb.net leider nicht, und daher kann der Umgang zwischen managed und unmanaged nur mit Interop-Marshalling erfolgen.

Wieder zurück zu unseren Methoden. Bei der Rückgabe handelt es sich um einen Int32-Datentyp, also eine Primitive, die direkt von C++ an .Net übergeben werden kann.

C++

```
typedef int i32;
typedef unsigned char b8;

EXP32 i32 APIENTRY fill_rng_byt_2d(b8 * buffer, i32 szbuffer1, i32 szbuffer2);
EXP32 i32 APIENTRY fill_rng_byt_2dj(b8 ** buffer, i32 szbuffer1, i32 szbuffer2);
```

C++ übernimmt `IntPtr` je nach Array als normaler `unsigned char (b8)` Pointer, oder wenn es sich um eine Jagged-Array handelt als Double- oder Triple-Pointer des Datentyp `unsigned char (b8)`.

C++ füllt den Speicherbereich hinter `IntPtr`. Nun muss der entsprechende Speicherbereich wieder .Net zugefügt werden. Ich habe das hier bewusst so gemacht, um zu zeigen, dass die Speicherbereiche hinter `emptyarray` und `ptr (IntPtr)` nicht die gleichen sind.

Will man die `emptyarray` bzw. `emptyjagged` füllen, so bedient man sich einfach dieser Methoden.

Vb.Net

```
GetArray(emptyarray2d, ptr, size1, size2, size3)
GetJagged(emptyjagged2d, ptr, size1, size2, size3)
```

Und weiter geht's mit den Create-Methoden.

Vb.Net

```
Dim ptr = CreateRngByt2dj(size1, size2)
Dim jagged2d = GetJagged(Of Byte)(ptr, size1, size2)

<DllImport(DllName, EntryPoint:="create_rng_byt_2dj",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function CreateRngByt2dj(size1 As Int32, size2 As Int32) As IntPtr
End Function
Dim ptr = CreateRngByt2d(size1, size2)
Dim array2d = GetArray(Of Byte)(ptr, size1, size2)

<DllImport(DllName, EntryPoint:="create_rng_byt_2d",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function CreateRngByt2d(size1 As Int32, size2 As Int32) As IntPtr
End Function
```

C++

```
typedef int i32;
typedef unsigned char b8;

EXP32 b8 *    APIENTRY create_rng_byt_2d(i32 szbuffer1, i32 szbuffer2);
EXP32 b8 *    APIENTRY create_rng_byt_3d(i32 szbuffer1, i32 szbuffer2, i32 szbuffer3);

EXP32 b8 **   APIENTRY create_rng_byt_2dj(i32 szbuffer1, i32 szbuffer2);
EXP32 b8 ***  APIENTRY create_rng_byt_3dj(i32 szbuffer1, i32 szbuffer2, i32 szbuffer3);
```

Wird der Speicherbereich in C++ alloziert, so müssen nur die entsprechenden Size-Grössen als Parameter übergeben werden. Auch hier stehen dafür entsprechende Puffer-Variablen in der Dll zur Verfügung, die später mit der Methode `dispose_buffer()` wieder zurückgesetzt werden können.

Interessant an den Methoden sind die Rückgaben. Es handelt sich auch wieder um IntPtr die .Net-seitig jedoch unterschiedlich interpretiert werden müssen. Bei Jagged-Arrays kann es je nachdem ob es sich um eine 2D- oder 3D-Array handelt, um ein Double- oder Triple-Pointer handeln. Bei Multi-Array in diesem Fall um einen normalen Pointer.

String und String-Array

Vb.Net

```
<DllImport(DllName, EntryPoint:="create_rng_str",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function CreateRngStr(size As Int32) As IntPtr
End Function

<DllImport(DllName, EntryPoint:="fill_rng_str",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function FillRngStr(ptr As IntPtr, size As Int32) As Int32
End Function

<DllImport(DllName, EntryPoint:="fill_rng_str_list",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function FillRngStrList(ptr As IntPtr, size As Int32, sizex() As Int32) As Int32
End Function

<DllImport(DllName, EntryPoint:="create_rng_str_list",
           CallingConvention:=CallingConvention.StdCall)>
Friend Function CreateRngStrList(size As Int32, _min As Int32, _max As Int32, sizex() As Int32) As
IntPtr
End Function
```

C++

```
typedef int i32;
typedef unsigned char b8;

EXP32 b8 *    APIENTRY create_rng_str(i32 szbuffer);
EXP32 i32     APIENTRY fill_rng_str(b8 * buffer, i32 szbuffer);

EXP32 i32     APIENTRY fill_rng_str_list(b8 ** buffer, i32 szbuffer, const i32 *szxbuffer);
EXP32 b8 **   APIENTRY create_rng_str_list(i32 szbuffer, i32 _min, i32 _max, i32 *szxbuffer);
```

Strings müssen generell für die Übergabe an C++ speziell behandelt werden. Strings beinhalten in .Net immer UniCodes und können sofern die String-Encoding bekannt sind auch direkt übergeben werden. Dafür besitzt das Attribut `DllImport` das Subattribut `CharSet` das angegeben werden kann. Wird das Subattribut bei direkter String-Übergabe nicht angegeben, wird `CharSet.None` angenommen was dem Default `CharSet.Ansi` entspricht.

In meinem Beispiel werden die String als Byte behandelt, was wiederum Interop-Marshalling voraussetzt. Hat den Vorteil das Marshalling zu IntPtr einfacher und meiner Meinung nach auch logischer umsetzbar ist. C++-seitig werden die IntPtr als unsigned char (Double-) Pointer entgegen genommen oder zurückgegeben.

In der Methode `fill_rng_str_list` erkennt man, dass `szxbuffer` konstant ist. Das heisst die Variable `buffer` ist eine leere String-Liste mit den in `szxbuffer` Size-Grössen. `szxbuffer` ist also auch eine Array, die jedoch durch `const` nicht verändert werden kann. Anders in der Methode `create_rng_str_list`, wo über die Variable `szxbuffer` eine leere Array übergeben wird, und die auf C++ Seite gefüllt wird, und wieder zurückgegeben wird an .Net.

Mit freundlichen Grüssen

exc-jdbi

www.vb-paradise.de/index.php/User/23082-exc-jdbi