Using and extending the Orcas marshal_as library

Nish Nishant

12 Jul 2007 CPOL

This article covers basic marshal_as usage, as well as how to extend marshal_as to support additional type conversions

Introduction

The need to marshal between native and managed types is a very frequent scenario in mixed-mode programming. This is specially true when it comes to strings - when you've got MFC strings, COM strings, standard C++ strings and CLR strings and need to convert between those types. In fact that's what prompted me to write the StringConvertor class for managed-unmanaged string conversions. While I did realize that I had misspelled converter as *convertor* I decided to leave it like that so I got a unique class-name and that way I could avoid Google dilution (there are dozens of other StringConverter classes, specially Java based ones).

Anyway, in the Orcas release of Visual C++, the VC++ team have added a mixed-mode marshalling library which primarily consists of the `marshal_as` template function. While `marshal_as` is technically not limited to string conversions, that is its most important role as of now. Using it is very similar to using one of the C++ cast operators such as `static_cast` - though be aware that you are not really casting here, you are doing a conversion - which may or may not be a different thing from a mere cast.

Basic usage

Here's a code snippet that shows how to convert various native strings to `System::String`

Hide Copy Code

```
String^ clrString;

const char* pcszHello = "hello world";
clrString = marshal_as<String^>(pcszHello);

wchar_t* pwszHello = L"hello wide world";
clrString = marshal_as<String^>(pwszHello);

bstr_t bstrtHello("hello bstr_t world");
clrString = marshal_as<String^>(bstrtHello);

std::string stdHello = "hello from std::string";
clrString = marshal_as<String^>(stdHello);

CString mfcString("hello from CString");
clrString = marshal_as<String^>(mfcString);

CComBSTR atrBSTR(L"hello from CComBSTR");
clrString = marshal_as<String^>(atrBSTR);
```

The reverse conversion is also similar.

Hide Copy Code

```
String^ clrString = "Original System::String";

std::string stdHello = marshal_as<std::string>(clrString);
CString mfcString = marshal_as<CString>(clrString);
CComBSTR atrBSTR = marshal_as<CComBSTR>(clrString);
```

You cannot directly use `marshal_as` for converting a `String^` to a `const char*`, a `const wchar_t*` or a `BSTR` - because those conversions require the unmanaged resources to be freed after use. For those, you need to use a context object as shown below.

Hide Copy Code

```
marshal_context context;

const char* pcszHello = context.marshal_as<const char*>(clrString);
const wchar_t* pcwszHello = context.marshal_as<const wchar_t*>(clrString);
BSTR bstrString = context.marshal_as<BSTR>(clrString);

Console::WriteLine(context._clean_up_list.Count);
```

The context object keeps track of the allocated objects and frees them in its destructor. Internally it maintains a linked list of objects that are allocated, and the output of the `Console::WriteLine` in the above code will be 3 (as we have allocated three objects using the context object). Initially I found this a little annoying to do, but I couldn't think of any alternate solution that was more elegant and where we could avoid using the context object. In fact, in my

Hide Copy Code

```
StringConvertor
```

class I had internally used an `std::vector` to store all allocated objects so I could free them in the destructor.

Extending marshal_as functionality

While the built-in functionality only allows string conversions, it's also possible to extend `marshal_as` functionality to support other type conversions. As an example of doing this, I have written a sample extension that supports converting between the Windows Forms `Rectangle` structure and the Win32 `RECT` structure. For the sake of completion I have also added specializations for the MFC `CRect` wrapper (which is a thin wrapper around the `RECT` structure). Here's my extension (put into a separate header file) :-

Hide Shrink Copy Code

```
namespace msclr
{
    namespace interop
    {
        template<> System::Drawing::Rectangle
            marshal_as<System::Drawing::Rectangle, RECT> (
            const RECT& from)
        {
            return System::Drawing::Rectangle(from.left, from.top,
                from.right - from.left, from.bottom - from.top);
        }

        template<> System::Drawing::Rectangle marshal_as<
            System::Drawing::Rectangle, CRect> (
            const CRect& from)
        {
            return System::Drawing::Rectangle(from.left, from.top,
                from.Width(), from.Height());
        }
```

```
        template<> RECT marshal_as<RECT, System::Drawing::Rectangle>(
            const System::Drawing::Rectangle& from)
        {
            System::Drawing::Rectangle rectangle = from; //remove const
            RECT rect = {rectangle.Left, rectangle.Top,
                rectangle.Right, rectangle.Bottom};
            return rect;
        }

        template<> CRect marshal_as<CRect, System::Drawing::Rectangle>(
            const System::Drawing::Rectangle& from)
        {
            System::Drawing::Rectangle rectangle = from; //remove const
            return CRect (rectangle.Left, rectangle.Top,
                rectangle.Right, rectangle.Bottom);
        }
    }
}
```

Effectively I've added four specializations for the four conversions that I need. Note how I have put the conversions into the `msclr::interop` namespace. This is to allow me (and anyone else) to use these additional conversions the same way I'd be using `marshal_as` for the regular conversions.

Now using these conversions would be quite trivial as show below.

Hide Copy Code

```
//To Rectangle

RECT rect = {10, 10, 110, 110};
System::Drawing::Rectangle rectangle =
    marshal_as<System::Drawing::Rectangle>(rect);

CRect mfcRect(20, 20, 220, 220);
rectangle = marshal_as<System::Drawing::Rectangle>(mfcRect);

//From Rectangle

RECT rectBack = marshal_as<RECT>(rectangle);

CRect mfcRectBack = marshal_as<CRect>(rectangle);
```

One important thing to be aware of is that in all four conversions I've added there is no need to handle context as there is no explicit need for memory deallocation. This may not always be the case as we'll see in the next section.

Extending marshal_as for objects needing context

When you have type conversions which require explicit resource deallocation, you have to handle it slightly differently. As an example I have written some code that'll extend `marshal_as` to support conversions between .NET `Font` objects and native Windows

Hide Copy Code

```
HFONT
```

structures

Hide Shrink Copy Code

```
namespace msclr
{
    namespace interop
    {
        template<> ref class context_node<System::Drawing::Font^, HFONT>
        : public context_node_base
        {
        private:
```

```
            System::Drawing::Font^ _font;

        public:
            context_node(System::Drawing::Font^% to, HFONT from)
            {
                to = _font = System::Drawing::Font::FromHfont((IntPtr)from);
            }

            ~context_node()
            {
                this->!context_node();
            }

        protected:
            !context_node()
            {
                delete _font;
            }
        };

        template<> ref class context_node<HFONT, System::Drawing::Font^>
        : public context_node_base
        {
        private:
            HFONT _hFont;

        public:
            context_node(HFONT& to, System::Drawing::Font^ from)
            {
                to = _hFont = (HFONT)from->ToHfont().ToPointer();
            }

            ~context_node()
            {
                this->!context_node();
            }

        protected:
            !context_node()
            {
                DeleteObject(_hFont);
            }
        };
    }
}
```

I've added two specializations of the `context_node` template class which is used by the `marshal_context` class to handle conversions that need a context. In the constructors I create the object that's requested and in the destructor/finalizer I free the resource. In the case of `HFONT`, I have called `DeleteObject` whereas for `Font`, I have called `delete` (which calls `Dispose`). I need not have done that for

Hide Copy Code

```
Font
```

as the `Font` finalizer would have come into play eventually, but for GDI objects it's preferable to free them as soon as they are not required as they tend to be rather heavy on the memory side.

Using these conversions is pretty similar to how `marshal_as` conversions are used for `const char*` or `BSTR` (where we use a context object too).

Hide Copy Code

```
HFONT hFont = CreateSampleFont();

//...

marshal_context context;
```

```
System::Drawing::Font^ font =
    context.marshal_as<System::Drawing::Font^>(hFont);
HFONT hFontCopy = context.marshal_as<HFONT>(font);

//...

DeleteObject(hFont);
```

That's it. Pretty straightforward to extend and to use.

History

- **July 12th 2007** - Article first published

License

Share

About the Author

Nish Nishant
United States United States

Nish Nishant is a Principal Software Architect based out of Columbus, Ohio. He has over 17 years of software industry experience in various roles including Lead Software Architect, Principal Software Engineer, and Product Manager. Nish was a Microsoft Visual C++ MVP between 2002 and 2015.

Nish is an industry acknowledged expert in the Microsoft technology stack. He authored C++/CLI in A...

show more

Comments and Discussions

You must Sign In to use this message board.

Search Comments

First Prev Next

const& warning

Dina Goldshtein **9-Feb-12 2:23**

Re: const& warning
Nish Nishant 9-Feb-12 6:35

Broken link
Henry Minute **23-Mar-11 4:00**

Re: Broken link
Nish Nishant 24-Mar-11 3:46

Nice Article
Majid Shahabfar **17-Jul-07 7:48**

Well done!
Bartosz Bien **13-Jul-07 7:27**

Re: Well done!
Nish Nishant 14-Jul-07 7:53

Hai
rilov **13-Jul-07 4:48**

Re: Hai
Nish Nishant 14-Jul-07 7:54

Re: Hai
Prakash Nadar 16-Jul-07 23:43

Re: Hai
rilov 18-Jul-07 14:56

Refresh 1

General General News Suggestion Question Question Bug Answer Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Layout: fixed | fluid

Web03 2.8.20201217.4