Using C++ Classes in .NET Code

How to create a wrapper around SPx objects such that they can be accessed by code written in .NET languages.

Summary

The SPx library is a C++ class library that is typically used directly within native C++ applications.

For display clients written in .NET languages, the RDC application is available to decouple the scan conversion and display aspects from the rest of the client code. RDC then presents a network socket control interface that .NET clients can use to control its runtime operation.

However, RDC may not always provide the flexibility users require and there are numerous other classes and modules within the SPx library that .NET users may wish to access.

The solution in this case is to write a wrapper library that provides a bridge between the selected C++ classes and the client application code. The wrapper library provides a means of instantiating C++ objects from within a .NET application and presents a native interface to them.

This application note is written using C++/CLI as the example .NET language, but the principle is the same whichever native language is used. Equivalent constructs will be possible in any other .NET language.

Overview

The SPx library is "unmanaged" in the context of Microsoft Windows, meaning that it operates outside of the .NET Common Language Runtime (CLR) environment. Managed code, i.e. code that runs within the CLR environment, cannot freely use unmanaged code. Furthermore, managed code cannot use static libraries. Therefore a method is required of instantiating the unmanaged objects of the SPx library from the managed code, presenting it with a managed interface, within a dynamically-loaded library.

A C++ DLL therefore needs to be written that can be linked into and used by the CLR application. This dynamic library is the bridge between managed application code and the unmanaged SPx library. The dynamic library allows managed objects to be created, each of which instantiates an appropriate unmanaged SPx object and maintains a pointer to that unmanaged object. It is this pointer to the SPx object that provides the link between the managed world of the CLR code and the unmanaged world of SPx. The following diagram, in Figure 1, summarises this structure.

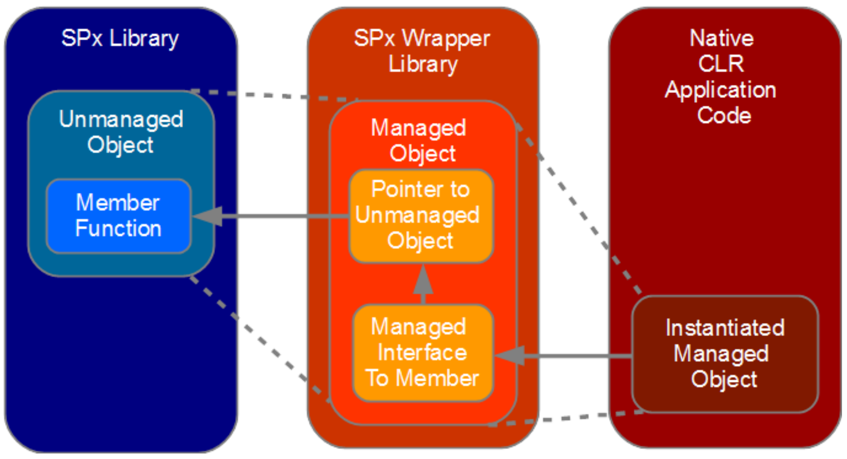
SPx Library
Unmanaged Object
Member Function
SPx Wrapper Library
Managed Object
Pointer to Unmanaged Object
Managed Interface To Member
Native CLR Application Code
Instantiated Managed Object

Figure 1: An overview of the structure of wrapper library approach

This approach works because the native CLR code is only presented with an interface that is allowed by the CLR.

The internals of the SPx library are hidden.

Each function of the SPx library object that needs to be invoked from the application code should be exposed to the CLR by writing an appropriate wrapper function.

The wrapper presents an acceptable interface to the CLR while using the pointer to the SPx object to access the member function.

Getting Started

A new wrapper library can be created by simply opening new project within Microsoft Visual Studio as a C++ CLR Class Library.

The project needs to be configured in the same way as any other SPx code project on Windows, as described in the SPx Programming Reference Manual, so the code needs to "#include" the appropriate SPx header file and the project should link against the static SPx library, plus other libraries as required.

The classes that are written within the new wrapper library need to be defined as reference classes ("class ref"), which means that they are managed classes that will be recognised by the CLR. For example, a simple wrapper around the `SPxRIB` class could be declared like this:

```
public ref class SPxRIB_CLR
{
public:
	// Constructor for the managed object
	SPxRIB_CLR(int size);
	// Destructor for the managed object
	~SPxRIB_CLR();
private:
	// Pointer to the unmanaged SPx object
	SPxRIB* m_SPxRIB;
};
```

The constructor, within the managed object, could then be written like this:

```
SPxRIB_CLR::SPxRIB_CLR(int size){

	// Instantiate the unmanaged object that we want to expose
	m_SPxRIB = new SPxRIB(size);
 }
```

The constructor for the managed object simply needs to use the C++ "new" keyword to create an instance of the normal SPx object as a member of the wrapper class. Now the managed CLR code can simply create an "`SPxRIB_CLR`" object and this will in turn create the underlying `SPxRIB` object.

Wrapping SPx Class Member Functions

It is important that the wrapped library interface presents only data types and objects that are acceptable in the managed domain. In general, it is safe to use fundamental C++ data types, such as "int", "char" and "double", within the wrapper interface.

Where an SPx constructor or member function takes a pointer to another SPx class as an input argument, the wrapped functions should in turn take a managed pointer (handle) to a wrapped class as its argument. For example, consider the case where the `SPxTestGenerator` object has been wrapped. The constructor for an `SPxTestGenerator` object requires an `SPxRIB` object to be passed as an argument. The constructor for the wrapped `SPxTestGenerator` class therefore needs to be provided with a handle to a wrapped version of the `SPxRIB` class. For example, the constructor for the wrapped `SPxTestGenerator` class may take this form:

```
SPxTestGenerator_CLR::SPxTestGenerator_CLR(SPxRIB_CLR^ buffer,
                                           int rangeSamples,
                                           double scanPeriod,
                                           int returnsPerSecond,
                                           int testPattern,
                                           int arg1);
```

Where "buffer" is a handle to a wrapped `SPxRIB` object.

It is good practice for wrapped classes to provide an accessor function, to allow indirect access to the underlying SPx object pointer. This is especially important in wrapped classes that may be used by other wrapped classes, which will then need to use the SPx object pointer. For example, the wrapped `SPxTestGenerator` constructor mentioned above will ultimately need to access the `SPxRIB` pointer, contained in the wrapped `SPxRIB`. The code for the wrapped `SPxTestGenerator` constructor might contain a line like this:

```
m_SPxTestGenerator = new SPxTestGenerator(buffer->getSPxPtr(),
                           rangeSamples,scanPeriod,returnsPerSecond,tes
                           tPattern,arg1);
```

Where "`getSPxPtr()`" is a member of the wrapped `SPxRIB` class that simply returns the pointer to the underlying `SPxRIB` object, for example:

```
SPxRIB SPxRIB_CLR::getSPxPtr(void){
      return m_SPxRIB;
}
```

Optional arguments in C++ function calls, i.e. arguments that are provided with a default value, are not supported under managed code. Where an SPx member function to be wrapped contains optional arguments, the simplest solution is to write overloaded wrapper functions explicitly, to cover the possible options that will be used in the managed code.

For example, two different constructors for an `SPxScFollowWinRaw` wrapper might be declared as:

```
SPxScFollowWinRaw_CLR(SPxScSourceLocal_CLR^ radarWindow, IntPtr win);
SPxScFollowWinRaw_CLR(SPxScSourceLocal_CLR^ radarWindow, IntPtr win,
                      SPxScDestDisplayWinRaw_CLR^ destDisplay); // Optional
                      argument "destDisplay" provided by overloaded function
```

Wrapping Macros and Enumerated Types

Macros that are provided within the SPx header files, such as "`SPX_PIM_OUTPUT`" (defined in SPxPIM.h) can be re-implemented in the wrapper library by writing them within an "abstract sealed" reference class. An abstract sealed class is essentially a static class that is not instantiated but is available as a global object. For example, the "`SPX_PIM_OUTPUT`" macro could be written as follows:

```
/* Macros */
public ref class MACRO abstract sealed
{
public:
      static int SPX_PIM_OUTPUT_CLR(int a){ return SPX_PIM_OUTPUT(a); };
};
```

The new, "MACRO" wrapper class can then be used in the managed application code like this:

```
        MACRO.SPX_PIM_OUTPUT_CLR(1)
```

Enumerated types, such as "`SPxPIMrangeCombine`" (defined in SPxPIM.h), can be wrapped by defining a corresponding enum class, for example:

```
 /* Enumerations */
 public enum class SPxPIMrangeCombine_CLR : Byte
 {
        SPX_PIM_RAN_SUBSAMPLE = 1,
        SPX_PIM_RAN_PEAK = 2,
        SPX_PIM_RAN_MIN = 3,
        SPX_PIM_RAN_SMOOTH = 4,
        SPX_PIM_RAN_LAST_ENTRY = 5
 };
```

The wrapped enum class can then be used in the application code like this:

```
        SPxPIMrangeCombine_CLR.PEAK
```

Wrapping SPx Processes

In order to use an SPx processing chain natively the "`SPxRunProcess`" class needs to be wrapped, along with the "`SPxProcess`" class. Furthermore, a means of initialising the SPx processing chain, i.e. calling the "`SPxInit`" function, from within the wrapper library is required. The call to the "`SPxInit`" function within the SPx library is what creates the standard process objects within the SPx framework.

One way to initialise the SPx processing chain within the wrapper library is to create another abstract sealed class, which calls the "`SPxInit`" function within its constructor, as well as creating each of the wrapped process objects. An abstract sealed class constructor is called once, early on at runtime, so this approach is an efficient way of initialising the SPx processes. The constructor for such a class may look like this:

```
 SPxProcessInit_CLR::SPxProcessInit_CLR(){
        SPxInit();
        SPxProScanConv_CLR = gcnew SPxProcess_CLR;
        SPxProScanConv_CLR->m_SPxProcess = SPxProScanConv;
 }
```

In this case, the process initialiser class is creating a wrapped version the SPx scan conversion process. Any other SPx processes may simply be created by extending this class in a similar fashion.

The wrapper for SPx processes is simply a means of storing and accessing a pointer to the underlying SPx process. The constructor for the class does not actually have to do anything.

The code for wrapping the "`SPxProcess`" class can therefore simply take this form:

```
public ref class SPxProcess_CLR
{
public:
        SPxProcess_CLR(){};
        SPxProcess* m_SPxProcess;
};
```

The wrapper for the "`SPxRunProcess`" class, which is what actually invokes an SPx process within a processing chain, needs to accept several wrapped SPx objects as inputs. A constructor for a wrapped "`SPxRunProcess`" object to invoke a scan conversion process could take the form:

```
SPxRunProcess_CLR::SPxRunProcess_CLR(SPxProcess_CLR^ proc, SPxRunProcess_CLR^ inputProc,
                                     SPxPIM_CLR^ pimA, SPxScSourceLocal_CLR^ scProc){
       /* Instantiate the unmanaged object that we want to expose */
       m_SPxRunProcess = new SPxRunProcess((proc != nullptr) ? proc->m_SPxProcess : NULL,
               (inputProc!=nullptr) ? inputProc->m_SPxRunProcess : NULL,
               (pimA != nullptr) ? pimA->m_SPxPIM : NULL,
               (scProc != nullptr) ? scProc->m_SPxScSourceLocal : NULL);
}
```

Note that where the wrapped class input arguments are handles to other wrapped objects it's good-practice to check for null pointers ("`nullptr`"), rather than simply passing the input argument through to the underlying SPx call. This is because it is invalid to attempt to access members of an object via a `nullptr` reference. It also means that the parameter checking within the SPx library can be utilised, rather than the code failing within the wrapper library if called incorrectly from the .NET application code.

Type Equivalence

Fundamental C++ data types, such as "int", "char" and "double", may be used within the wrapper library code interface. Equivalent .NET framework data types are well documented on the Internet for example here:

https://msdn.microsoft.com/en-us/library/0wf2yk2k.aspx.

A detailed discussion of them is therefore not repeated here.

It is advisable to add the following line into the C++/CLI code, to provide access to the .NET framework types, without qualifying them with the word "System":

```
        using namespace System;
```

For example, the type "Byte" may then be written instead of "`System::Byte`" within the wrapper code.

The "HWND" data type, representing a Windows window handle, cannot be used within the managed interface presented by the wrapper library. However, an HWND is essentially a pointer to an integer so it's safe to use the "`System::IntPtr`" in the interface and cast this to an HWND within the implementation code. For example, consider the `SPxScFollowWinRaw` object, whose constructor has the prototype:

```
SPxScFollowWinRaw::SPxScFollowWinRaw(SPxSc *radarWindow, HWND *win,
                                       SPxScDestDisplayWinRaw *destDisplay = 0)
```

A wrapper for this constructor could be defined thus:

```
SPxScFollowWinRaw_CLR::SPxScFollowWinRaw_CLR(SPxScSourceLocal_CLR^ radarWindow, IntPtr win,
                                             SPxScDestDisplayWinRaw_CLR^ destDisplay){

        /* Instantiate the unmanaged object that we want to expose */
        m_SPxScFollowWinRaw = new SPxScFollowWinRaw(radarWindow->m_SPxScSourceLocal,
                                (HWND)win.ToPointer(), destDisplay->m_SPxScDestDisplayWinRaw);
}
```

The `System::IntPtr` presented by the managed interface is cast to an HWND, via the `System::IntPtr::toPointer()` method, before being passed to the original SPx function.

Further Assistance

This document is intended for guidance only. Writing a wrapper library can be a complicated exercise and may vary depending on the language being used for both the wrapper and the native application.

Please consult Cambridge Pixel if further assistance is required.

< End of document >