Rudy's Delphi Corner

Using C++ objects in Delphi

Delphi is one of the greatest RAD tools on the market, but it in this currently C++-dominated world, it can sometimes be hard to find a Delphi or Pascal solution to your problem. There is, however, a chance you'll find a C++ class instead. This article describes several ways that enable you to use C++ classes from your Delphi code.

First the bad news: you won't be able to directly link the classes into your Delphi code. The Delphi linker can't link C++ object files into your application. You can link C object files, but that is the subject of another article. You'll need access to a C++ compiler that can create DLLs, and use the classes from the DLL.

Different object structures

The greatest difficulty in interfacing Delphi and C++ is, that their object structures differ. All Delphi classes descend from TObject, and are created on the heap. C++ classes are more like Delphi records with methods, and can be created statically or dynamically. They are similar to the legacy **object** types, carried over into Delphi from Borland and Turbo Pascal.

Even if the internal structure can be made similar, it is not safe to model the structure, especially since C++ classes can be the result of multiple inheritance and can have private data members that are hard to mimic or find out. Actually, mimicking the structure of C++ abstract classes was my first approach, but you would never be able to construct or Free such a class. They would not have the RTTI and special methods of Delphi classes either.

Exporting methods

Another problem is exporting methods from a DLL. There are basically two ways: the first one "flattens" the C++ class into a set of C functions, which all take an object as first parameter; the second one uses virtual, abstract methods that are laid out like in a COM-style interface.

Demo class

Say you have the following simple C++ object: a Console class, that uses the functions in *conio.h* to achieve simple console functionality. On creation, it saves the current screen and, on destruction, restores it again.

> *I know it is not a very useful class, since a C++Builder user would simply use conio.h directly, but it nicely demonstrates how you can use an existing C++ class and it is easy to implement as a demo class. And some of the functionality is not found in the conio.h of other compilers.*

```
enum TextColors
{
    tcBLACK,
    tcBLUE,
    tcGREEN,
    tcCYAN,
    tcRED,
    tcMAGENTA,
    tcBROWN,
    tcLIGHTGRAY,
    tcDARKGRAY,
    tcLIGHTBLUE,
    tcLIGHTGREEN,
    tcLIGHTCYAN,
    tcLIGHTRED,
    tcLIGHTMAGENTA,
    tcYELLOW,
    tcWHITE
};

class Console
{
public:
    Console();
    virtual ~Console();
    void reset();
    void clearScreen();
    void gotoXY(int x, int y);
    void textColor(TextColors newColor);
    void textAttribute(int newAttribute);
    void textBackground(int newBackground);
    int readKey();
    bool keyPressed();
    void write(const char *text);
    void writeLn(const char *text);

private:
    text_info oldState;
    char *screenBuffer;
};
```

This class has ten instance methods, a destructor and a constructor. I will now demonstrate two ways of using this C++ class from Delphi.

“Flattening” the object

To “flatten” a class, you export a simple C function for each method, as well as functions for the constructor and the destructor. The first parameter of the functions (except for the “constructor” function) should be a pointer to the object. To flatten the *Console* class, you would declare 12 functions like this:

```
#include <windows.h>;
#include "console.h"

typedef Console *ConsoleHandle;

// define a macro for the calling convention and export type
#define EXPORTCALL __declspec(dllexport) __stdcall

extern "C"
{

    ConsoleHandle EXPORTCALL NewConsole(void)
    {
        return new Console();
    }

    void EXPORTCALL DeleteConsole(ConsoleHandle handle)
    {
        delete handle;
    }

    void EXPORTCALL ConsoleReset(ConsoleHandle handle)
    {
        handle->reset();
    }

    void EXPORTCALL ConsoleClearScreen(ConsoleHandle handle)
    {
        handle->clearScreen();
    }

    void EXPORTCALL ConsoleGotoXY(ConsoleHandle handle,
        int x, int y)
    {
        handle->gotoXY(x, y);
    }

    void EXPORTCALL ConsoleTextColor(ConsoleHandle handle,
        TextColors newColor)
    {
        handle->textColor(newColor);
    }

    void EXPORTCALL ConsoleTextAttribute(ConsoleHandle handle,
        int newAttribute)
    {
        handle->textAttribute(newAttribute);
    }

    void EXPORTCALL ConsoleTextBackground(ConsoleHandle handle,
        int newBackground)
    {
        handle->textBackground(newBackground);
    }

    int EXPORTCALL ConsoleReadKey(ConsoleHandle handle)
    {
        return handle->readKey();
    }

    bool EXPORTCALL ConsoleKeyPressed(ConsoleHandle handle)
    {
        return handle->keyPressed();
    }

    void EXPORTCALL ConsoleWrite(ConsoleHandle handle,
        const char *text)
    {
        handle->write(text);
    }

    void EXPORTCALL ConsoleWriteLn(ConsoleHandle handle,
        const char *text)
    {
        handle->writeLn(text);
    }

} // extern "C"

#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void* lpReserved)
{
    return 1;
}
```

Now you only have to compile this to a DLL, and your object can be used from Delphi in the same manner as any API call. The interface unit would look like this:

```
unit ConsoleFlat;

interface

uses
  SysUtils;

type
  // Can't use the class directly, so it is treated as an opaque handle.
  // THandle is guaranteed to have the right size, even on other platforms.
  ConsoleHandle = THandle;

  TTextColor = (
    tcBLACK,
    tcBLUE,
    tcGREEN,
    tcCYAN,
    tcRED,
    tcMAGENTA,
    tcBROWN,
    tcLIGHTGRAY,
    tcDARKGRAY,
    tcLIGHTBLUE,
    tcLIGHTGREEN,
    tcLIGHTCYAN,
    tcLIGHTRED,
    tcLIGHTMAGENTA,
    tcYELLOW,
    tcWHITE
  );

function NewConsole: ConsoleHandle; stdcall;
procedure DeleteConsole(handle: ConsoleHandle); stdcall;
procedure ConsoleReset(handle: ConsoleHandle); stdcall;
procedure ConsoleClearScreen(handle: ConsoleHandle); stdcall;
procedure ConsoleGotoXY(handle: ConsoleHandle; x, y: Integer); stdcall;
procedure ConsoleTextColor(handle: ConsoleHandle; newColor: TTextColor); stdcall;
procedure ConsoleTextAttribute(handle: ConsoleHandle; newAttribute: Integer); stdcall;
procedure ConsoleTextBackground(handle: ConsoleHandle; newBackground: TTextColor); stdcall;
function ConsoleReadKey(handle: ConsoleHandle): Integer; stdcall;
function ConsoleKeyPressed(handle: ConsoleHandle): Boolean; stdcall;
procedure ConsoleWrite(handle: ConsoleHandle; text: PAnsiChar); stdcall;
procedure ConsoleWriteLn(handle: ConsoleHandle; text: PAnsiChar); stdcall;

implementation

const
  DLLName = 'ConsoleFlat.dll';

function NewConsole; external DLLName;
procedure DeleteConsole; external DLLName;
procedure ConsoleReset; external DLLName;
procedure ConsoleClearScreen; external DLLName;
procedure ConsoleGotoXY; external DLLName;
procedure ConsoleTextColor; external DLLName;
procedure ConsoleTextAttribute; external DLLName;
procedure ConsoleTextBackground; external DLLName;
function ConsoleReadKey; external DLLName;
function ConsoleKeyPressed; external DLLName;
procedure ConsoleWrite; external DLLName;
procedure ConsoleWriteLn; external DLLName;

end.
```

This can then be used anyway you need. The disadvantage is, of course, that you are using functions, where the C++ user can use a class. So instead of

```
  Console := TConsole.Create;
  try
    Console.TextBackground(tcRED);
    Console.TextColor(tcYELLOW);
    Console.ClearScreen;
    Console.WriteLn('Yellow on red');
    Console.ReadKey;
  finally
    Console.Free;
  end;
```

you must do the following:

```
  Console := NewConsole;
  try
    ConsoleTextBackground(Console, tcRED);
    ConsoleTextColor(Console, tcYELLOW);
    ConsoleClearScreen(Console);
    ConsoleWriteLn(Console, 'Yellow on red');
    ConsoleReadKey(Console);
  finally
    DeleteConsole(Console);
  end;
```

This leads to the next way of using the class from Delphi, one which is a bit more convenient.

Using COM-style interfaces

In C++ on Windows, interfaces have a structure which is dictated by COM and can be used by many languages, including Delphi. As this diagram from my article on pointers shows, they are pointers that point to a pointer, which points to an array of procedural pointers, which represent the methods of the interface. In C++, they are defined as abstract classes or structs. The fact they are pure virtual classes means that each method is represented in the virtual method table, and that the first member of an instance such a class is a pointer to that method table. This is also the structure required for interfaces.

The C++ part

To make the *Console* class usable, you must first define the interface and then implement it, using the existing *Console* class for the functionality. A header file for an *IConsole* interface could look like this:

```
#ifndef ICONSOLE_H
#define ICONSOLE_H

#include <System.hpp>
#include "console.h"

struct __declspec(uuid("{9BBDA1A4-21E7-4D11-8F1C-E2AD13D2779C}"))
    IConsole : public System::IInterface
{
public:
    virtual void __stdcall Reset() = 0;
    virtual void __stdcall ClearScreen() = 0;
    virtual void __stdcall GotoXY(int x, int y) = 0;
    virtual void __stdcall TextColor(TextColors newColor) = 0;
    virtual void __stdcall TextAttribute(int newAttribute) = 0;
    virtual void __stdcall TextBackground(int newBackground) = 0;
    virtual int __stdcall ReadKey() = 0;
    virtual bool __stdcall KeyPressed() = 0;
    virtual void __stdcall Write(const char *text) = 0;
    virtual void __stdcall WriteLn(const char *text) = 0;
};

// The following class takes care of automatic reference counting and can be
// used instead of the naked interface.
typedef System::DelphiInterface<IConsole> _di_IConsole;

#ifdef ICONSOLEDLL_EXPORTS
#define ICONSOLEDLL_API __declspec(dllexport) __stdcall
#else
#define ICONSOLEDLL_API __declspec(dllimport) __stdcall
#endif

extern "C" _di_IConsole ICONSOLEDLL_API CreateConsole(void);

#endif
```

Instead of doing all this manually, which is not too hard, but can still be a pretty frustrating job if you are not very familiar with C++, you can declare the interface in Delphi first *(see further in this text) and then use that file in a simple C++ project, so a .hpp file will be generated with the same base name as the .pas file. This contains all the necessary declarations in C++ syntax.*

The .hpp file mentioned above is generated by the Delphi compiler that comes with C++Builder. Alternatively to creating a new C++Builder project and adding the Delphi unit to it, you can also compile the unit on the command line, using the `-JPHNE` *option, which makes the Delphi compiler generate all necessary files for C++Builder. It is done like this:*

```
dcc32 -JPHNE IConsoleDLL.pas
```

Each method is pure virtual (In Delphi, this is called *abstract*), which is indicated by `= 0` after each method declaration. It is usual to use the *__stdcall* calling convention, so that is what I did too, although if the class is only to be used in Delphi, you can use *__fastcall* too, which is equivalent to Delphi's default *register* calling convention.

If you want to compile with a different compiler, __stdcall is the best choice. Their __fastcall is very likely not compatible (it uses different registers), and some compilers even have a very different default calling convention

> *(e.g. __thiscall) for instance methods. Using __stdcall avoids any problems.*

The GUID for the interface, in this case `__declspec(uuid(”{9BBDA1A4-21E7-4D11-8F1C-E2AD13D2779C}”))` can be generated directly in the C++Builder editor, with the key combination `Shift+Ctrl+G`.

To make the interface usable, and because you can't use C++ classes directly, for instance to call their constructor to create an instance, there must be a function that generates such an interface for you. This is what the *CreateConsole()* function does.

The implementation of the *IConsole* interface is simply a class with the proper base class, in this case the well known *TInterfacedObject*, which is also often used in Delphi, and as second base the interface. A header file should look like this:

```
#pragma once

#include ”iconsole.h”
#include ”console.h”

class TConsole : public TInterfacedObject, IConsole
{
public:

    // __fastcall is required for Delphi-derived classes

    __fastcall TConsole() : Console() { }

    virtual __fastcall ~TConsole() { }

    // __stdcall for the implementation of the COM-style interface methods

    void __stdcall Reset()
    {
        FConsole.reset();
    }

    void __stdcall ClearScreen()
    {
        FConsole.clearScreen();
    }

    void __stdcall GotoXY(int x, int y)
    {
        FConsole.gotoXY(x, y);
    }

    void __stdcall TextColor(TextColors newColor)
    {
        FConsole.textColor(newColor);
    }

    void __stdcall TextAttribute(int newAttribute)
    {
        FConsole.textAttribute(newAttribute);
    }

    void __stdcall TextBackground(int newBackground)
    {
        FConsole.textBackground(newBackground);
    }

    int __stdcall ReadKey()
    {
        return FConsole.readKey();
    }

    bool __stdcall KeyPressed()
    {
        return FConsole.keyPressed();
    }

    void __stdcall Write(const char *text)
    {
        FConsole.write(text);
    }

    void __stdcall WriteLn(const char *text)
    {
        FConsole.writeLn(text);
    }

    // Methods of IInterface/IUnknown have to be reimplemented and forwarded.

    HRESULT __stdcall QueryInterface(const GUID& IID, void **Obj)
    {
        return GetInterface(IID, Obj) ? S_OK : E_NOINTERFACE;
    }

    ULONG __stdcall AddRef()
    {
        return TInterfacedObject::_AddRef();
    }

    ULONG __stdcall Release()
    {
        return TInterfacedObject::_Release();
    }

private:
    Console FConsole;
};

#endif
```

If you intend to use another compiler than C++Builder, then you won't be able to descend from TInterfacedObject. And it is very likely that its conio.h doesn't have the functionality of the Borland/Embarcadero version. I will not show the code for the functionality here, but it can be found in the Downloads.

But for TConsole, this means you will have to do something like:

```
#pragma once

#include "stdafx.h"
#include "iconsole.h"
#include "console.h"

class TConsole : public IConsole
{
public:

    TConsole() : FConsole(), FRefCount(0) { }

    // Rest of functions same as above, left out for brevity.

    // Methods of IUnknown

    HRESULT __stdcall QueryInterface(REFIID riid, void **ppvObject)
    {
        if (IsEqualGUID(riid, __uuidof(IConsole))
        {
            *ppvObject = (void *)this;
            return S_OK;
        }
        else
        {
            *ppvObject = NULL;
            return E_NOINTERFACE;
        }
    }

    ULONG __stdcall AddRef()
    {
        return InterlockedIncrement(&FRefCount);
    }

    ULONG __stdcall Release()
    {
        ULONG result = InterlockedDecrement(&FRefCount);
        if (!result)
            delete this;
        return result;
    }

private:
    Console FConsole;
    ULONG FRefCount;
};
```

In other words, you will have to implement reference counting and QueryInterface() on your own. The implementation above should do.

As you can see, there is no need to declare each of the implementing functions `virtual`, as in C++, this is automatically so, if they are virtual in the base class. But the *IInterface*/*IUnknown* methods must be implemented again, see the source code.

I could have used public or private inheritance of *Console* to implement the methods, but I used aggregation instead. This way, the inheritance tree is a little easier to understand, especially for a Delphi user like me.

Now the DLL must be created. I used the DLL wizard by selecting menu *New* → *Other...* → *C++Builder Projects* → *Dynamic-link library*. I added the implementation of the generator (or factory) function for the interface:

```
#include <windows.h>
#define ICONSOLEDLL_EXPORT
#include "iconsole.h"
#include "iconsoleimpl.h"

_di_IConsole ICONSOLEDLL_API CreateConsole()
{
    return (IConsole *)(new TConsole());
}

int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void* lpReserved)
{
    return 1;
}
```

The files that must be generated are a little different for Visual C++™ or other Windows compilers. In the Downloads you can find a simple VC++ 2010 example.

The Delphi part

```
unit IConsoleDLL;

interface

type
  TTextColor = (
    tcBLACK, tcBLUE, tcGREEN, tcCYAN, tcRED, tcMAGENTA, tcBROWN, tcLIGHTGRAY,
    tcDARKGRAY, tcLIGHTBLUE, tcLIGHTGREEN, tcLIGHTCYAN, tcLIGHTRED,
    tcLIGHTMAGENTA, tcYELLOW, tcWHITE
  );

  IConsole = interface
  ['{9BBDA1A4-21E7-4D11-8F1C-E2AD13D2779C}']
    procedure Reset; stdcall;
    procedure ClearScreen; stdcall;
    procedure GotoXY(x, y : Integer); stdcall;
    procedure TextColor(newColor: TTextColor); stdcall;
    procedure TextAttribute(newAttribute: Integer); stdcall;
    procedure TextBackground(newBackground: TTextColor); stdcall;
    function ReadKey: Integer; stdcall;
    function KeyPressed: Boolean; stdcall;
    procedure Write(text: PAnsiChar); stdcall;
    procedure WriteLn(text: PAnsiChar); stdcall;
  end;

function CreateConsole: IConsole; stdcall;

implementation

function CreateConsole; external 'IConsoleDLL.dll' name 'CreateConsole';

end.
```

Note that the GUID must be the same as defined in C++. I think it is obvious how the rest of the C++ declaration is converted.

The declaration of the rather ugly *_di_IConsole*, which is the original return value of the *CreateConsole()* function in C++, is fortunately not needed in Delphi. Also, in C++, an interface is returned as pointer, i.e. `IConsole *`, and not as `IConsole`, while in Delphi, interfaces are reference types, so they are not treated as pointers. Behind the scenes, Delphi reference types like objects or interfaces *are* pointers, so they are equivalent to the C++ pointer types.

The only function you import from the DLL is *CreateConsole*. The source in the *implementation* section shows how this can be done.

Using the interface

To use the interface in Delphi, you can do something like:

```
var
  Con: IConsole;
begin
  Con := CreateConsole;
  Con.Write('press any key...');
  Con.ReadKey;
  Con.TextBackground(tcRED);
  Con.ClearScreen;
  Con.TextColor(tcYELLOW);
  Con.WriteLn('Text in Yellow');
  Con.TextColor(tcLIGHTRED);
  Con.WriteLn('Text in LightRed');
  Con.TextColor(tcLIGHTGREEN);
  Con.WriteLn('Text in LightGreen');
  Con.Write('press any key...');
  Con.ReadKey;
  Con.Reset;
  Con.WriteLn('Normal settings again, and screen restored.');
  Con.Write('press any key...');
  Con.ReadKey;
end;
```

Note that in the code above, there is no need to code a *try — finally* construct around the code that uses the interface, since the lifetime of interfaces is managed automatically by the Delphi runtime, and there is already an implicit, hidden *try — finally* around the code.

> *You could use the interface in C++Builder too, but in C++, it is a lot easier to use the original Console class, so I did not try this.*

Slippery When Wet

There are a few things to be considered when you want to use a C++ class. Irrespective of which method you use to make it available, you will have to take care of certain matters. I will briefly discuss them here.

Exceptions

No matter what you do, *never* let exceptions escape the DLL, as this is very likely going to cause trouble. If the DLL was written in C++Builder and the only consumer is the same version of Delphi or C++Builder, things *might* work out. Otherwise, the Delphi code will very likely not be able to handle the exception properly.

So if there is a chance you get an exception, handle it inside your wrapper (flat function or interface method) and pass any errors back as return values. One mechanism is returning a *HRESULT* with the error information and providing any real result, e.g. the result of *readKey()*, as last parameter. This is how safecall works.

So again, just in case I was not clear enough:

NEVER LET EXCEPTIONS ESCAPE A DLL!

Use return values (for instance *HRESULT*s or booleans) to indicate errors. The *safecall* pattern is the preferred way in COM.

Returning interfaces

In Delphi, interfaces are not returned like normal pointers. Instead, they are passed as last `var` parameter. So, in Delphi, code that looks like:

```
function CreateConsole: IConsole; stdcall;
```

is in fact compiled as:

```
procedure CreateConsole(var Result: IConsole); stdcall;
```

Originally, in my sources, which were partly based on the *.hpp* output of compiling my Delphi unit with `-JPHNE` settings, I had:

```
typedef System::DelphiInterface<IConsole> _di_IConsole;

...

extern "C" _di_IConsole ICONSOLEDLL_API CreateConsole(void);
```

The *System::DelphiInterface<IConsole>* template class does something comparable to the runtime system of Delphi, it automatically manages the reference counting for the *IConsole* interface.

The fact that the template class is marked *RTL_DELPHIRETURN* in its declaration means, at least in C++Builder, that it is returned as last parameter, just like interfaces in Delphi, i.e. as if it were declared as:

```
extern "C" void ICONSOLEDLL_API CreateConsole(_di_IConsole *Result);
```

Since this is the same way Delphi returns interfaces, this worked out fine.

But, because I had no need for auto-reference-counting code (i.e. *System::DelphiInterface<IConsole>*) on the C++ side and to stay more in line with the code I had to write for Visual C++™, I changed this to:

```
extern "C" IConsole * __stdcall CreateConsole(void);
```

But this had the consequence that now, the `IConsole *` function return type was treated like a normal pointer (unlike the *DelphiInterface* template class) and therefore returned from the function in the `EAX` register, and not as extra parameter. This was not equivalent to the Delphi code anymore, so calling *CreateConsole* did not return the expected result.

I decided to do what I discussed in the previous section and introduced the *safecall* pattern in all sources. So in C++Builder, this became:

```
HRESULT ICONSOLEDLL_API CreateConsole(IConsole **console)
{
    try
    {
        *console = new TConsole();
    }
    catch (...)
    {
        return E_NOINTERFACE;
    }
    (*console)->AddRef();
    return S_OK;
}
```

The code for Visual C++ was very similar. The Delphi code became:

```
function CreateConsole: IConsole; safecall;
```

As you can see, that only required replacing *stdcall* with *safecall*.

So, do not forget that in Delphi, interfaces are not passed back to the caller as function result, they are passed back as `var` parameters. The C++ code should reflect this, or the code for both languages should be adjusted, like I did in my example.

In C++Builder, you have the luxury of the *Delphi::SystemInterface*<> template, which is passed to the user the same way as an interface in Delphi, and which will do automatic reference counting when necessary, e.g. when it is assigned to a reference parameter. With other compilers you do not have that luxury (although they may have other ways to deal with COM, like ATL), so you will have to adapt your code. The *safecall* pattern, even if it is probably not as nicely and actively supported as in Delphi, is still a good way to reach your goal.

Moritz showed me an alternative he wrote that makes writing and using the code equally easy in every language. Get the safecall subset of his UCL library and use that to make safecall code a litle easier.

It contains code that emulates Delphi's DelphiInterface<> and TInterfacedObject for every C++ compiler. Code can then look like:

```
// iconsole.h

#include <ucl/stl/intfptr.hpp>

struct __declspec(uuid("{9BBDA1A4-21E7-4D11-8F1C-E2AD13D2779C}"))
    IConsole : public IUnknown
{
public:
    ...
};

typedef ucl::stl::InterfacePtr<IConsole> IConsolePtr;
```

```
// tconsole.h

#include <ucl/stl/intfimpl.hpp>
#include "iconsole.h"
#include "console.h"

class TConsole : public ucl::stl::InterfacedObject<IConsole>
{
    // No need to manually implement QueryInterface(), AddRef() and
    // Release() anymore
    ...
}
```

CreateConsole() can then be as easy as:

```
#include <ucl/module/safecall.hpp>

HRESULT ICONSOLEDLL_API CreateConsole(IConsolePtr *console)
{
    UCL_SAFECALL_BEGIN
    *console = new TConsole;
    UCL_SAFECALL_END
}
```

UCL_SAFECALL_BEGIN and UCL_SAFECALL_END are macros that wrap the code in an exception handler frame which saves the exception error message with SetErrorInfo(), so the safecall handler can get it back when the HRESULT indicates an error.

Parameters and return values

This is probably common sense, but I'd like to repeat it anyway.

In C++, there are many features available to the programmer, and the C++ class you want to make available may use any of, e.g. other C++ classes or structs with virtual methods, single or multiple inheritance, templates, C++'s basic RTTI, `__typeid()`, etc. etc. The internal structure of these features is often intimately tied to one compiler and one version.

That is why none of such compiler-specific features or types should show up in the interface of the DLL, i.e. neither in the parameter list of exported functions or methods, nor as return types. So, for instance, a method or function that expects or returns a C++ `string` (or `std::string`) or any other C++-specific type, or even another C++ class, is absolutely taboo.

Parameters and return types should only be simple types that can be shared across several languages, i.e. scalar types — like integers, enums, floating point types, pointers, chars and booleans — or plain structs (no *virtual* methods, no inheritance), arrays or pointers with such types as base.

More on what you can and cannot export from a DLL can be found in my article DLL dos and don'ts.

Your wrapper will have to contain code necessary to convert any C++-specific features thus that they can handle the simple types. So if your original function expects a `std::string`, your wrapper takes a `char *`, turns it into a `std::string` and passes that to the original method. A string that is returned goes the other way around: your wrapper expects as parameters a pointer to a buffer and a size and fills the buffer with the contents of the internally used `std::string`. More on converting types in Pitfalls of Converting. That is not exactly a set of instructions, but it will give you a feeling for the kind of parameters you can use.

As done in the flat function solution, you can pass items like classes or structs with items that are compiler-specific around to other functions by simply treating them as opaque types, i.e. types of which you don't know or need to know their internal structure. The user doesn't have to know the structure, as he or she only passes them around. Your code knows the internal structure and can make use of it, where necessary.

Differences in name decoration

When I compiled my example DLL in Visual C++ and started my Delphi program that ought to call it, I got a message that the procedure entry point *CreateConsole* could not be found in the DLL. After pulling some hair, I found out that in Visual C++, if you are using the `__declspec(dllexport)` directive, combined with `__stdcall`, the function, even when it is declared as `extern "C"`, is exported as `_CreateConsole@4`.

There are two ways to deal with this:

- You can add a Module Definition (.def) file to your project somewhere in the project options for the linker. This will enable you to export the name as `CreateConsole`, but I find it quite a hassle, especially if you already used `__declspec(dllexport)`. It is however the "more proper" way to do things, since that way, you avoid publishing a DLL with underscored exported names.

- You can change the import name in your DLL interface unit:

  ```
  function CreateConsole; external 'IConsoleDLLVCpp.dll' name '_CreateConsole@4';
  ```

  That is what I would prefer to do, even if it is not very elegant.

Note that how the name is decorated depends on your compiler. This is discussed nicely on Wu Yongwei's website. But I think the easiest solution is to look into the DLL and see what the name of the exported function is. This can be done using a tool like Dependency Walker, which is a tool every serious Windows developer should have or at least know, in my opinion.

> *In my demo VC++ project, I use a .def file, simply to see how this works and because it generates a "cleaner" interface. The .def file can be added to the project in the project options:*
>
> Project → Properties → Configuration Properties → Linker → Input → Definition File: *IConsoleDLLVCpp.def*
>
> *The definition file is quite simple (you can create it as text file and then save it as .def file). Mine looks like this:*
>
> ```
> LIBRARY IConsoleDLLVCpp.dll
>
> EXPORTS
>     CreateConsole
> ```

Conclusion

Although it is often said that C++ classes can't be used in Delphi, this is only partly true, as this article demonstrates. But which of the two ways of importing C++ classes is preferrable?

The second way, using interfaces, is of course much more convenient to use. It is as if your interface was written in Delphi. But it is a lot more work on the C++ side: you must declare and implement the interface and the "factory" function for it. There are quite a lot more source files involved.

The "flat" variety is less convenient, but has one level of indirection less (since calling virtual functions, which is what you do when you call methods of an interface — see my article on pointers — is also an extra level of indirection), and that makes it a bit faster. If you need an interface or a class, you can always wrap the "flat" functions in Delphi.

In my opinion, both solutions are useful and doable ways of making C++ classes available. Each has its pros and its contras. Which one you choose depends on your preferences and your goals.

The C++ source code for both DLLs and small demo programs in Delphi can be downloaded from the Downloads page.

Rudy Velthuis

Standard Disclaimer for External Links

These links are being provided as a convenience and for informational purposes only; they do not constitute an endorsement or an approval of any of the products, services or opinions of the corporation or organization or individual. I bear no responsibility for the accuracy, legality or content of the external site or for that of subsequent links. Contact the external site for answers to questions regarding its content.

Disclaimer and Copyright

The coding examples presented here are for illustration purposes only. The author takes no responsibility for end-user use. All content herein is copyrighted by Rudy Velthuis, and may not be reproduced in any form without the author's permission. Source code written by Rudy Velthuis presented as download is subject to the license in the files.

Last update: Feb. 20, 2019

Back to top