HowTo: Export C++ classes from a DLL

Alex Blekhman

- Download source - 11.1 KB

Contents

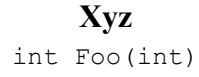
Introduction

Dynamic-Link libraries (DLL) are an integrated part of the Windows platform from its very beginning. DLLs allow encapsulation of a piece of functionality in a standalone module with an explicit list of C functions that are available for external users. In 1980's, when Windows DLLs were introduced to the world, the only viable option to speak to broad development audience was C language. So, naturally, Windows DLLs exposed their functionality as C functions and data. Internally, a DLL may be implemented in any language, but in order to be used from other languages and environments, a DLL interface should fall back to the lowest common denominator – the C language.

Using the C interface does not automatically mean that a developer should give up object oriented approach. Even the C interface can be used for true object oriented programming, though it may be a tedious way of doing things. Unsurprisingly, the second most used programming language in the world, namely C++, could not help but to fall prey to the temptation of a DLL. However, opposite to the C language, where the binary interface between a caller and a callee is well-defined and widely accepted, in the C++ world, there is no recognized application binary interface (ABI). In practice, it means that binary code that is generated by a C++ compiler is not compatible with other C++ compilers. Moreover, the binary code of the same C++ compiler may be incompatible with other versions of this compiler. All this makes exporting C++ classes from a DLL quite an adventure.

The purpose of this article is to show several methods of exporting C++ classes from a DLL module. The source code demonstrates different techniques of exporting the imaginary `Xyz` object. The `Xyz` object is very simple, and has only one method: `Foo`.

Here is the diagram of the object `Xyz`:



The implementation of the `Xyz` object is inside a DLL, which can be distributed to a wide range of clients. A user can access `Xyz` functionality by:

- Using pure C
- Using a regular C++ class
- Using an abstract C++ interface

The source code consists of two projects:

- **XyzLibrary** – a DLL library project
- **XyzExecutable** – a Win32 console program that uses "*XyzLibrary.dll*"

The **XyzLibrary** project exports its code with the following handy macro:

```
#if defined(XYZLIBRARY_EXPORT) // inside DLL
#   define XYZAPI   __declspec(dllexport)
#else // outside DLL
#   define XYZAPI   __declspec(dllimport)
#endif  // XYZLIBRARY_EXPORT
```

The `XYZLIBRARY_EXPORT` symbol is defined only for the **XyzLibrary** project, so the `XYZAPI` macro expands into `__declspec(dllexport)` for the DLL build and into `__declspec(dllimport)` for the client build.

C Language Approach

Handles

The classic C language approach to object oriented programming is the usage of opaque pointers, i.e., handles. A user calls a function that creates an object internally, and returns a handle to that object. Then, the user calls various functions that accept the handle as a parameter and performs all kinds of operations on the object. A good example of the handle usage is the Win32 windowing API that uses an `HWND` handle to represent a window. The imaginary `Xyz` object is exported via a C interface, like this:

```
typedef tagXYZHANDLE {} * XYZHANDLE;

XYZAPI XYZHANDLE APIENTRY GetXyz(VOID);

XYZAPI INT APIENTRY XyzFoo(XYZHANDLE handle, INT n);
XYZAPI VOID APIENTRY XyzRelease(XYZHANDLE handle);
```

Here is an example of how a client's C code might look like:

```
#include "XyzLibrary.h"

...


XYZHANDLE hXyz = GetXyz();
```

```
if(hXyz)
{

    XyzFoo(hXyz, 42);


    XyzRelease(hXyz);


    hXyz = NULL;
}
```

With this approach, a DLL must provide explicit functions for object creation and deletion.

Calling Conventions

It is important to remember to specify the calling convention for all exported functions. Omitted calling convention is a very common mistake that many beginners do. As long as the default client's calling convention matches that of the DLL, everything works. But, once the client changes its calling convention, it goes unnoticed by the developer until runtime crashes occur. The **XyzLibrary** project uses the `APIENTRY` macro, which is defined as `__stdcall` in the "*WinDef.h*" header file.

Exception Safety

No C++ exception is allowed to cross over the DLL boundary. Period. The C language knows nothing about C++ exceptions, and cannot handle them properly. If an object method needs to report an error, then a return code should be used.

Advantages

- A DLL can be used by the widest programming audience possible. Almost every modern programming language supports interoperability with plain C functions.
- C run-time libraries of a DLL and a client are independent of each other. Since resource acquisition and freeing happens entirely inside a DLL module, a client is not affected by a DLL's choice of CRT.

Disadvantages

- The responsibility of calling the right methods on the right instance of an object rests on the user of a DLL. For example, in the following code snippet, the compiler won't be able to catch the error:

  ```
  XYZHANDLE h = GetSomeOtherObject();


  XyzFoo(h, 42);
  ```

- Explicit function calls are required in order to create and destroy object instances. This is especially annoying for deletion of an instance. The client function must meticulously insert a call to `XyzRelease` at all points of exit from a function. If the developer forgets to call `XyzRelease`, then resources are leaked because the compiler doesn't help to track the lifetime of an object instance. Programming languages that support destructors or have a garbage collector may mitigate this problem by making a wrapper over the C interface.
- If object methods return or accept other objects as parameters, then the DLL author has to provide a proper C interface for these objects, too. The alternative is to fall back to the lowest common denominator, that is the C language, and use only built-in types (like `int`, `double`, `char*`, etc.) as return types and method parameters.

C++ Naive Approach: Exporting a Class

Almost every modern C++ compiler that exists on the Windows platform supports exporting a C++ class from a DLL. Exporting a C++ class is quite similar to exporting C functions. All that a developer is required to do is to use the `__declspec(dllexport/dllimport)` specifier before the class name if the whole class needs to be exported, or before the method declarations if only specific class methods need to be exported. Here is a code snippet:

```
class XYZAPI CXyz
{
public:
    int Foo(int n);
};

class CXyz
{
public:
    XYZAPI int Foo(int n);
};
```

There is no need to explicitly specify a calling convention for exporting classes or their methods. By default, the C++ compiler uses the `__thiscall` calling convention for class methods. However, due to different naming decoration schemes that are used by different compilers, the exported C++ class can only be used by the same compiler and by the same version of the compiler. Here is an example of a naming decoration that is applied by the MS Visual C++ compiler:

E	Ordinal ^	Hint	Function	Entry Point
C++	1 (0x0001)	0 (0x0000)	??4CXyz@@QAEAAV0@ABV0@@Z	0x0001101E
C++	2 (0x0002)	1 (0x0001)	?Foo@CXyz@@QAEHH@Z	0x0001114A

Notice how the decorated names are different from the original C++ names. Following is a screenshot of the same DLL module with name decoration deciphered by the Dependency Walker tool:

E	Ordinal ^	Hint	Function	Entry Point
C++	1 (0x0001)	0 (0x0000)	class CXyz & CXyz::operator=(class CXyz const &)	0x0001101E
C++	2 (0x0002)	1 (0x0001)	int CXyz::Foo(int)	0x0001114A

Only the MS Visual C++ compiler can use this DLL now. Both the DLL and the client code must be compiled with the same version of MS Visual C++ in order to ensure that the naming decoration scheme matches between the caller and the callee. Here is an example of a client code that uses the `Xyz` object:

```
#include "XyzLibrary.h"

...
CXyz xyz;
xyz.Foo(42);
```

As you can see, the usage of an exported class is pretty much the same as the usage of any other C++ class. Nothing special.

Important: Using a DLL that exports C++ classes should be considered no different than using a static library. All rules that apply to a static library that contains C++ code are fully applicable to a DLL that exports C++ classes.

What You See Is Not What You Get

A careful reader must have already noticed that the Dependency Walker tool showes an additional exported member, that is the `CXyz& CXyz::operator =(const CXyz&)` assignment operator. What we see

is our C++ money at work. According to the C++ Standard, every class has four special member functions:

- Default constructor
- Copy constructor
- Destructor
- Assignment operator (operator =)

If the author of a class does not declare and does not provide an implementation of these members, then the C++ compiler declares them, and generates an implicit default implementation. In the case of the `CXyz` class, the compiler decided that the default constructor, copy constructor, and the destructor are trivial enough, and optimized them out. However, the assignment operator survived optimization and got exported from a DLL.

Important: Marking the class as exported with the `__declspec(dllexport)` specifier tells the compiler to attempt to export everything that is related to the class. It includes all class data members, all class member functions (either explicitly declared, or implicitly generated by the compiler), all base classes of the class, and all their members. Consider:

```
class Base
{
    ...
};

class Data
{
    ...
};

class __declspec(dllexport) Derived :
    public Base
{
    ...

private:
    Data m_data;    };
```

In the above code snippet, the compiler will warn you about the not exported base class and the not exported class of the data member. So, in order to export a C++ class successfully, a developer is required to export all the relevant base classes and all the classes that are used for the definition of the data members. This snowball exporting requirement is a significant drawback. That is why, for instance, it is very hard and tiresome to export classes that are derived from STL templates or to use STL templates as data members. An instantiation of an STL container like `std::map<>`, for example, may require tens of additional internal classes to be exported.

Exception Safety

An exported C++ class may throw an exception without any problem. Because of the fact that the same version of the same C++ compiler is used both by a DLL and its client, C++ exceptions are thrown and caught across DLL boundaries as if there were no boundaries at all. Remember, using a DLL that exports C++ code is the same as using a static library with the same code.

Advantages

- An exported C++ class can be used in the same way as any other C++ class.
- An exception that is thrown inside a DLL can be caught by the client without any problem.
- When only small changes are made in a DLL module, no rebuild is required for other modules. This can be very beneficial for big projects where huge amounts of code are involved.

- Separating logical modules in a big project into DLL modules may be seen as the first step towards true module separation. Overall, it is a rewarding activity that improves the modularity of a project.

Disadvantages

- Exporting C++ classes from a DLL does not prevent very tight coupling between an object and its user. The DLL should be seen as a static library with respect to code dependencies.
- Both client code and a DLL must link dynamically with the same version of CRT. It is necessary in order to enable correct bookkeeping of CRT resources between the modules. If a client and DLL link to different versions of CRT, or link with CRT statically, then resources that have been acquired in one instance of the CRT will have been freed in a different instance of the CRT. It will corrupt the internal state of the CRT instance that attempts to operate on foreign resources, and most likely will lead to crash.
- Both the client code and the DLL must agree on the exception handling/propagating model, and use the same compiler settings with respect to C++ exceptions.
- Exporting a C++ class requires exporting everything that is related to this class: all its base classes, all classes that are used for the definition of data members, etc.

C++ Mature Approach: Using an Abstract Interface

A C++ abstract interface (i.e., a C++ class that contains only pure virtual methods and no data members) tries to get the best of both worlds: a compiler independent clean interface to an object, and a convenient object oriented way of method calls. All that is required to do is to provide a header file with an interface declaration and implement a factory function that will return the newly created object instances. Only the factory function has to be declared with the `__declspec(dllexport/dllimport)` specifier. The interface does not require any additional specifiers.

```
struct IXyz
{
    virtual int Foo(int n) = 0;
    virtual void Release() = 0;
};

extern "C" XYZAPI IXyz* APIENTRY GetXyz();
```

In the above code snippet, the factory function `GetXyz` is declared as `extern "C"`. It is required in order to prevent the mangling of the function name. So, this function is exposed as a regular C function, and can be easily recognized by any C-compatible compiler. This is how the client code looks like, when using an abstract interface:

```
#include "XyzLibrary.h"

...
IXyz* pXyz = ::GetXyz();

if(pXyz)
{
    pXyz->Foo(42);

    pXyz->Release();
    pXyz = NULL;
}
```

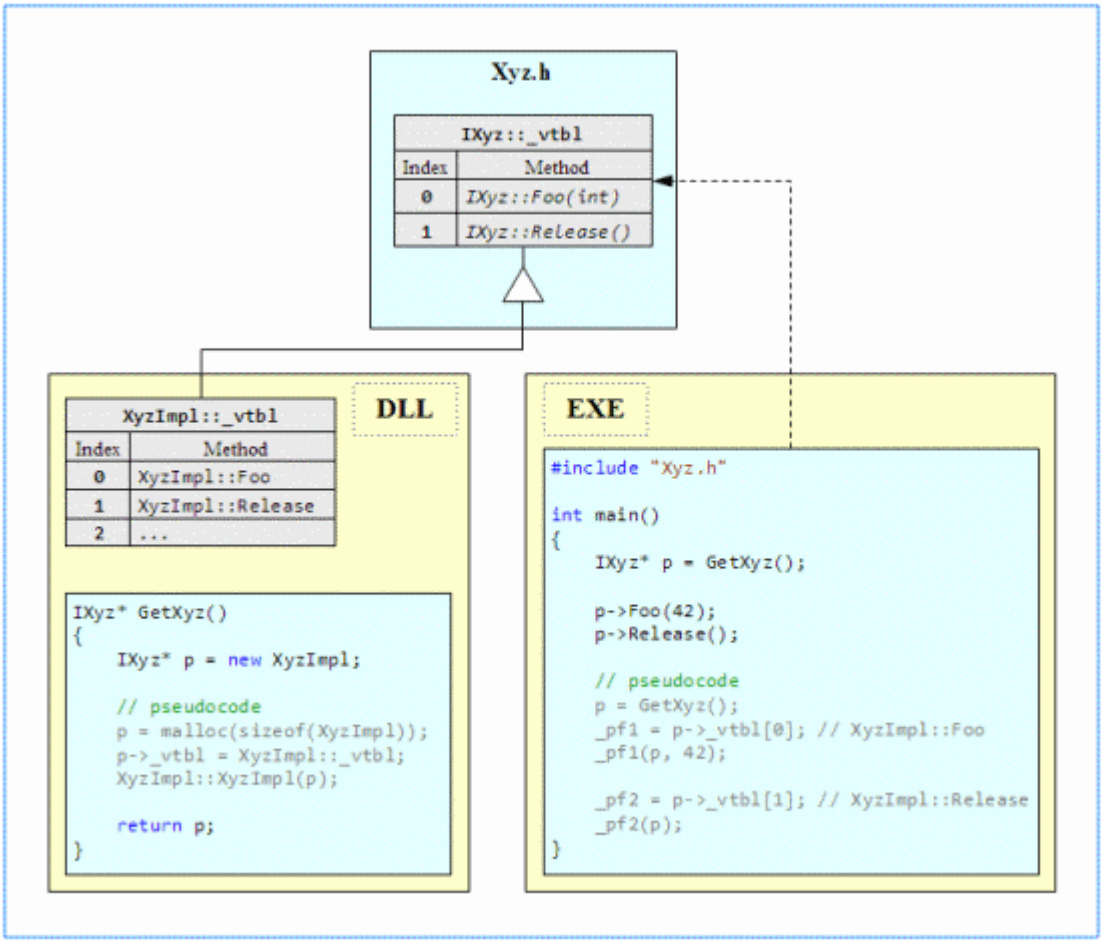
C++ does not provide a special notion for an interface as other programming languages do (for example, C# or Java). But it does not mean that C++ cannot declare and implement interfaces. The common approach to make a C++ interface is to declare an abstract class without any data members. Then, another separate class inherits from the interface and implements interface methods, but the implementation is hidden from the interface clients. The interface client neither knows nor cares about how the interface is

implemented. All it knows is which methods are available and what they do.

How This Works

The idea behind this approach is very simple. A member-less C++ class that consisting of pure virtual methods only is nothing more than a virtual table, i.e., an array of function pointers. This array of function pointers is filled within a DLL with whatever an author deems necessary to fill. Then, this array of pointers is used outside of a DLL to call the actual implementation. Bellow is the diagram that illustrates the `IXyz` interface usage.

Click on the image to view the full sized diagram in a new window:



The above diagram shows the `IXyz` interface that is used both by the DLL and the EXE modules. Inside the DLL module, the `XyzImpl` class inherits from the `IXyz` interface, and implements its methods. Method calls in the EXE module invoke the actual implementation in the DLL module via a virtual table.

Why This Works With Other Compilers

The short explanation is: because COM technology works with other compilers. Now, for the long explanation. Actually, using a member-less abstract class as an interface between modules is exactly what COM does in order to expose COM interfaces. The notion of a virtual table, as we know it in the C++ language, fits nicely into the specification of the COM standard. This is not a coincidence. The C++ language, being the mainstream development language for at least over a decade now, has been used extensively with COM programming. It is thanks to natural support for object oriented programming in the C++ language. It is not surprising at all that Microsoft has considered the C++ language as the main heavy-duty instrument for industrial COM development. Being the owner of the COM technology, Microsoft has ensured that the COM binary standard and their own C++ object model implementation in the Visual C++ compiler do match, with as little overhead as possible.

No wonder that other C++ compiler vendors jumped on the bandwagon and implemented the virtual table layout in their compilers in the same way as Microsoft did. After all, everybody wanted to support COM technology, and to be compatible with the existing solution from Microsoft. A hypothetical C++ compiler that fails to support COM efficiently is doomed to oblivion in the Windows market. That is why ,nowadays, exposing a C++ class from a DLL via an abstract interface will work reliably with every decent C++ compiler on the Windows platform.

Using a Smart Pointer

In order to ensure proper resource release, an abstract interface provides an additional method for the disposal of an instance. Calling this method manually can be tedious and error prone. We all know how common this error is in the C world where the developer has to remember to free the resources with an explicit function call. That's why typical C++ code uses RAII idiom generously with the help of smart pointers. The **XyzExecutable** project uses the `AutoClosePtr` template, which is provided with the example. The `AutoClosePtr` template is the simplest implementation of a smart pointer that calls an arbitrary method of a class to destroy an instance instead of `operator delete`. Here is a code snippet that demonstrates the usage of a smart pointer with the `IXyz` interface:

```
#include "XyzLibrary.h"
#include "AutoClosePtr.h"

...
typedef AutoClosePtr<IXyz, void, &IXyz::Release> IXyzPtr;

IXyzPtr ptrXyz(::GetXyz());

if(ptrXyz)
{
    ptrXyz->Foo(42);
}
```

Using a smart pointer will ensure that the `Xyz` object is properly released, no matter what. A function can exit prematurely because of an error or an internal exception, but the C++ language guarantees that destructors of all local objects will be called upon the exit.

Using Standard C++ Smart Pointers

Recent versions of MS Visual C++ provide smart pointers with the Standard C++ library. Here is the example of using `Xyz` object with `std::shared_ptr` class:

```
#include "XyzLibrary.h"

#include <memory>
#include <functional>

...

typedef std::shared_ptr<IXyz> IXyzPtr;

IXyzPtr ptrXyz(::GetXyz(), std::mem_fn(&IXyz::Release));

if(ptrXyz)
{
    ptrXyz->Foo(42);
}
```

Exception Safety

In the same way as a COM interface is not allowed to leak any internal exception, the abstract C++ interface cannot let any internal exception to break through DLL boundaries. Class methods should use return codes to indicate an error. The implementation for handling C++ exceptions is very specific to each compiler, and cannot be shared. So, in this respect, an abstract C++ interface should behave as a plain C function.

Advantages

- An exported C++ class can be used via an abstract interface, with any C++ compiler.
- C run-time libraries of a DLL and a client are independent of each other. Since resource acquisition and freeing happens entirely inside a DLL module, a client is not affected by a DLL's choice of CRT.
- True module separation is achieved. The resulting DLL module can be redesigned and rebuilt without affecting the rest of the project.
- A DLL module can be easily converted to a full-fledged COM module, if required.

Disadvantages

- An explicit function call is required to create a new object instance and to delete it. A smart pointer can spare a developer of the latter call, though.
- An abstract interface method cannot return or accept a regular C++ object as a parameter. It has be either a built-in type (like `int`, `double`, `char*`, etc.) or another abstract interface. It is the same limitation as for COM interfaces.

What About STL Template Classes?

The Standard C++ Library containers (like `vector`, `list`, or `map`) and other templates were not designed with DLL modules in mind. The C++ Standard is silent about DLLs because this is a platform specific technology, and it is not necessarily present on other platforms where the C++ language is used. Currently, the MS Visual C++ compiler can export and import instantiations of STL classes which a developer explicitly marks with the `__declspec(dllexport/dllimport)` specifier. The compiler emits a couple of nasty warnings, but it works. However, one must remember that exporting STL template instantiations is in no way different from exporting regular C++ classes, with all accompanying limitations. So, there is nothing special about STL in that respect.

Summary

The article discussed different methods of exporting a C++ object from a DLL module. Detailed description is given of the advantages and disadvantages for each method. Exception safety considerations are outlined. The following conclusions are made:

- Exporting an object as a set of plain C functions has an advantage of being compatible with the widest range of development environments and programming languages. However, a DLL user is required to use outdated C techniques or to provide additional wrappers over the C interface in order to use modern programming paradigms.
- Exporting a regular C++ class is no different than providing a separate static library with the C++ code. The usage is very simple and familiar; however, there is a tight coupling between the DLL and its client. The same version of the same C++ compiler must be used, both for the DLL and its client.
- Declaring an abstract member-less class and implementing it inside a DLL module is the best approach to export C++ objects, so far. This method provides a clean, well-defined object oriented interface between the DLL and its client. Such a DLL can be used with any modern C++ compiler on the Windows platform. The usage of an interface in conjunction with smart pointers is almost as

easy as the usage of an exported C++ class.

The C++ programming language is a powerful, versatile, and flexible development instrument.