By CodeGuru Staff March 6, 2002

Environment: Visual C++

As I discussed in the last article, .DLLs are a useful tool for any MFC programmer. They are subject to a number of important limitations, however, and anyone who is making .DLLs should be aware of these.

MFC Issues

This was discussed in the last article, but is worth mentioning again briefly. An MFC extension .DLL can only be used if the client application dynamically links to the same version of MFC, and the correct MFC code library .DLL is available on the client computer. A regular .DLL which dynamically links to MFC will only work if the correct MFC code library .DLL is available.

Compiler Incompatibility Issues

One of the biggest problems with C++ based .DLLs arises when a .DLL is built on one brand of compiler and called by an application built on another brand of compiler. Often it won't work without a great deal of effort.

ANSI sets the standards for the C and C++ languages. That is, it specifies the C and C++ functions and data types which should be supported by a compiler. It does not, however, provide a complete standard as to how these functions and data types should be implemented
on a binary level. As a result, compiler vendors are free to implement language features in their own proprietary ways.

Most C / C++ programmers know that different compilers handle data types differently. One compiler will allocate 2 bytes for an int and another will allocate 4 bytes. One will use 4 byte doubles and

another
will use 8 bytes. An even bigger difference with C++ compilers arises from their implementation of function and operator overloading.
The differences between compilers go far beyond this, however. The same C or
C++ code may be compiled very differently by different compilers. These differences may keep your .DLL from running with someone else's application.

Of course, if you are building an MFC extension .DLL, this is not an issue for you. MFC extension .DLLs are made with a Microsoft compiler. As discussed in the previous article, they can only be used by applications
that are dynamically linked to MFC. These applications are also made with
a Microsoft compiler.

Compiler incompatibility problems can be fixed by
inserting pragmas and other precompile instructions into your code, but this is
hard to do and unreliable. There will always be the chance that someone else is using a compiler that's still incompatible.

Recompiling

Let's say you built a .DLL that exports a class called CMyClass.
You provide a copy of the header file for CMyClass to be used by the client application. Suppose that a CMyClass object is 30 bytes in size.

Now let's suppose you modify the .DLL to change CMyClass. It still has the same public functions and member variables, but now CMyClass has an
additional private member variable, an int. So now, when you create an object of type CMyClass, it's 34 bytes in size. You send this new .DLL to your users and tell them to replace the old .DLL. Now you have a problem. The client application is expecting a 30 byte object, but your new .DLL is creating a 34 byte object. The client application is going to get an error.

Here's a similar problem. Suppose that instead of exporting CMyClass your .DLL exports several functions which use CMyClass references or pointers in their parameters or return values. You have provided a CMyClass header file which now resides in the client application. Again, if you change the size of the CMyClass object without rebuilding the client application, you will have problems.

At this point, the only way to fix the problem is to replace the CMyClass header file in the client application and recompile it. Once recompiled, the client application will start looking for a 34 byte object.

This is a serious problem. One of the goals of having a .DLL is to be able to modify and replace the .DLL without modifying the client application. However, if your .DLL is exporting classes or class objects, this may not be possible. You may have to recompile the client application. If you don't have the source code for the client application, you simply can't use the new .DLL.

Solutions

If there were a perfect fix for these problems, we might not have COM. Here are a few suggestions:

MFC extension .DLLs don't have compiler incompatibility problems. They simply can't be used by applications built on non-Microsoft compilers. As for regular .DLLs, you can avoid many compiler problems by only exporting C-style functions and using the extern "C" specifier. Exporting C++ classes and overloaded C++ functions leaves you much more
vulnerable to compiler incompatibility.

As for having to recompile your client application when you modify the .DLL, there are relatively easy ways to avoid the problem. I will describe two of them: (1) an interface class and (2) static functions used to create and destroy your exported class.

Using an Interface Class

The goal of an interface class is to separate the class you want to export and the interface to that class. The way to do this is to create a second class which will serve as the interface for the class you want to export. Then, even when the export class changes, you will not need to recompile the client application, because the interface class remains the same.

Here's an example of how that would work. Suppose you want to export CMyClass. CMyClass has two public functions, int FunctionA(int) and int FunctionB(int). If I simply export CMyClass, I'll have to recompile the client application every time I add a new variable. Instead, I'll create and export an interface class, CMyInterface. CMyInterface will have a pointer to a CMyClass object. Here's the header file for CMyInterface as it looks inside the .DLL:

```
#include "MyClass.h"
class __declspec(dllexport) CMyInterface

{

    //private pointer to CMyClass object

    CMyClass *m_pMyClass;


    CMyInterface( );

    ~CMyInterface( );


    public:

    int FunctionA(int);
```

```
    int FunctionB(int);

};
```

Inside the client application, the header file will look slightly different. The #include will be gone. After all, you can't include MyClass.h, because the client application doesn't have a copy of it. Instead, you will use a forward declaration of CMyClass. This will allow you to compile even without the CMyClass header file:

```
class __declspec(dllimport) CMyInterface

{

     //Forward declaration of CMyClass

     class CMyClass;
   CMyClass *m_pMyClass;


   CMyInterface( );

   ~CMyInterface( );


   public:

   int FunctionA(int);

   int FunctionB(int);

};
```

Inside the .DLL, you implement CMyInterface as follows:

```
CMyInterface::CMyInterface( )

{

     m_pMyClass = new CMyClass;

}
CMyInterface::~CMyInterface( )

{

    delete m_pMyClass;

}


int CMyInterface::FunctionA( )

{

    return m_pMyClass->FunctionA( );

}


int CMyInterface::FunctionB( )

{

    return m_pMyClass->FunctionB( );

}
```

Thus, for every public function in CMyClass, CMyInterface will provide its own corresponding function. The client application has no contact

with CMyClass. If it wants to call CMyClass::FunctionA, it instead calls CMyInterface::FunctionA. The interface class then uses its pointer to call CMyClass. With this arrangement, you're free to modify with CMyClass. It doesn't matter if the size of the CMyClass object changes. The size of the CMyInterface object will remain the same. If you add a private member variable to CMyClass, the size of CMyInterface will remain
unchanged. If you add a public member variable to CMyClass, you can add
"getter" and "setter" functions for the new variable in CMyInterface without fear. Adding new functions to CMyInterface will not cause recompile problems.

Creating a separate interface class avoids some compiler incompatibility problems and most recompile problems. As long as the interface class doesn't change, there should be no need to recompile. There are still two relatively minor problems with this solution. First, for every public function and member variable in CMyClass, you must create a corresponding
function or variable in CMyInterface. In the example there are only two functions, so that's easy. If CMyClass had hundreds of functions and variables, this would be a more tedious, error-prone process.
Second, you are increasing the amount of processing that must be done.
The client application no longer calls CMyClass directly. Instead, it calls a CMyInterface function which calls CMyClass. If this is a function that will be called thousands of times by the client application, the extra processing time may begin to add up.

Static Functions

A different way to avoid having to recompile uses static functions to create and destroy the exported class. This solution was sent to me by Ran Wainstain in a comment to my previous article.

When you are creating a class which you intend to export, you add two public, static functions, CreateMe( ) and DestroyMe( ):

```
class __declspec(dllexport) CMyClass

{

    CMyClass( );

    ~CMyClass( );
  public:

  static CMyClass* CreateMe( );

  static void DestroyMe(CMyClass *ptr);

}
```

CreateMe( ) and DestroyMe( ) are implemented as follows:

```
CMyClass* CMyClass::CreateMe( )

{

     return new CMyClass;

}
void CMyClass::DestroyMe(CMyClass *ptr)

{

   delete ptr;

}
```

You export CMyClass as you would any other class. In the client application, you must be sure to use the CreateMe( ) and DestroyMe( ) functions. When you want to create a CMyClass object, you don't declare it in the usual fashion, i.e.:

```
CMyClass x;
```

Instead, you do this:

```
CMyClass *ptr = CMyClass::CreateMe( );
```

When you are done, you must remember to delete the object:

```
CMyClass::DeleteMe(ptr);
```

Using this technique, you can modify the size of CMyClass without having to recompile the client application.

Conclusion

This article is not a complete review of every issue concerning .DLLs, nor
does it cover every possible solution. Good discussions of these issues can be found in Inside COM by Dale Rogerson and Essential COM
by Don Box. For a more detailed understanding of the issue, that's where
I would go. This article should at least serve to make you aware of the biggest issues and possible fixes and get you started on your way.