By CodeGuru Staff March 6, 2002

Environment: Visual C++

At one point in time, before COM, before ATL, programmers used ordinary .DLLs instead. You could do a lot with a .DLL. If you had several programs that used the same functions or other resources, you could save space by putting those resources in a .DLL. Putting code used by multiple programs in a single .DLL often saved maintenance time because the code was all in one place. Fixes and other modifications would only have to be done one time. If you had a program which needed to run different routines at different times, you could put those routines into .DLLs and have the application load the appropriate .DLL when it was needed. There were lots of good reasons to use .DLLs.

There are still a lot of good reasons to use .DLLs. They haven't gone away. Sure, whatever you can do with a .DLL, you can probably do with a COM object. Granted, there are a number of shortcomings to .DLLs, some of them serious, which is why we ended up with COM in the first place. Still, .DLLs remain a very useful tool. Compared to COM and ATL, they are much easier to make. Learning COM or ATL requires a serious investment of time and effort. Making a .DLL is relatively easy. Modifying one is easy too. If you know some C++ and MFC, you could be making .DLLs today.

This article will review the types of .DLLs you can make with MFC, including when to use each type and how to make them. In the next article there will be a discussion of the limitations of .DLLs (which led to the rise of COM and ATL), and how these can be partially avoided. In the third article, there will be more coding details and examples.

Different types of .DLLs

There are two kinds of .DLLs you can make using MFC: an MFC extension .DLL or a regular .DLL. Regular .DLLs in turn come in two varieties: dynamically linked or statically linked. Visual C++ also allows you to make a generic Win32 .DLL, but in this article I'm only going to discuss the MFC-based .DLL types.

MFC extension .DLLs

Every .DLL has some kind of interface. The interface is the set of the variables, pointers, functions or classes provided by the .DLL which you can access from the client program. They are the things that allow the client program to use the .DLL. An MFC extension .DLL can have a C++ style interface. That is, it can provide ("export") C++ functions and entire C++ classes to be used by the client application. The functions it exports can use C++ or MFC data types as parameters or as return values. When it exports a class, the client will be able to create objects of that class or derive new classes from it. Inside the .DLL, you can also use MFC and C++.

The MFC code library used by Visual C++ is stored in a .DLL. An MFC extension .DLL dynamically links to the MFC code library .DLL. The client application must also dynamically link to the MFC code library .DLL. As the years have gone by the MFC library has grown. As a result, there are a few different versions of the MFC code library .DLL out there. Both the client program and the extension .DLL must be built using the same version of MFC. Therefore, for an MFC extension .DLL to work, both the extension .DLL and the client program must dynamically link to the same MFC code library .DLL, and this .DLL must be available on the computer where the application is running.

Note: If you have an application which is statically linked to MFC, and you wish to modify it so that it can access functions from an extension .DLL, you can change the application to dynamically link to MFC. In Visual C++, select "Project | Settings" from the menu. On the "General" settings tab you can change your application to dynamically link to MFC.

MFC extension .DLLs are very small. You can build an extension .DLL which exports a few functions or small classes and has a size of 10-15 KB. Obviously, the size of your .DLL depends on how much code you store in it, but in general MFC extension .DLLs are relatively small and quick to load.

Regular .DLLs

The MFC extension .DLL only works with MFC client applications. If you need a .DLL that can be loaded and run by a wider range of Win32 programs, you should use a regular .DLL. The downside is that your .DLL and your client application cannot send each other pointers or references to MFC-derived classes and objects. If you export a function, it cannot use MFC data types in its parameters or return values. If you export a C++ class, it cannot be derived from MFC. You can still use MFC **inside** your .DLL, but not in your interface.

Your regular .DLL still needs to have access to the code in the MFC code library .DLL. You can dynamically link to this code or statically link. If you dynamically link, that means the MFC code your .DLL needs in order to function is not built into your .DLL. Your .DLL will get the code it needs from the MFC code library .DLL found on the client application's computer. If the right version of the MFC code library .DLL is not there, your .DLL won't run. Like the MFC extension .DLL, you get a small .DLL (because the .DLL doesn't include the MFC code), but you can only run if the client computer has the MFC code library .DLL.

If you statically link to the MFC code library, your .DLL will incorporate within itself all the MFC code it needs. Thus, it will be a larger .DLL, but it won't be dependent on the client computer having the proper MFC code library .DLL. If you can't rely on the host computer having the right version of MFC available, this is the way to go. If your application users are all within your own company, and you have control over what versions of the MFC .DLLs are lurking on their computers, or if your installation program also loads the right MFC .DLL, this might not be an issue.

Building a .DLL

You can make an MFC-based .DLL with the App Wizard. Select “File | New”
from the menu. On the “Projects” tab, select “MFC AppWizard (.DLL).” Pick a name for your new project and click “OK.” On the next screen, you will have the choice to create an MFC extension .DLL, a regular .DLL “using shared MFC .DLL” (i.e., a regular .DLL dynamically linked to MFC), or a regular .DLL statically linked to MFC. Pick the one you want and click “Finish.”

App Wizard builds a .DLL which doesn’t do anything. The new .DLL will compile, but since it doesn’t export any classes or functions yet, it is still essentially useless. You now have two jobs: (1) add functionality to make your .DLL useful; and (2) modify your client application to use your .DLL.

Export a class

Once you’re done with the App Wizard, you can add classes to your .DLL by adding the .cpp and .h files from another project, or you can create them from scratch within your current project. To export a class, you add “__declspec(dllexport)” to the class declaration so it looks like this:

```
class __declspec(dllexport) CMyClass

{

   //class declaration goes here

};
```

If you are making an MFC extension .DLL, you can instead use the AFX_EXT_CLASS macro:

```
class AFX_EXT_CLASS CMyClass

{

   //class declaration goes here

};
```

There are other ways to export a class, but this is the easiest. If your exported class requires a resource which is located in the .DLL, for example a class derived from CDialog, the process is more involved. I'll cover this subject in tutorial #3.

Below I'll discuss what to do to the client application so that it can use your exported class.

Export variables, constants and objects

Instead of exporting a whole class, you can have your .DLL export a variable, constant or object. To export a variable or constant, you simply declare it like this:

```
__declspec(dllexport) int   MyInt;

__declspec(dllexport) extern const COLORREF MyColor =

                                        RGB(50,50,50);
```

When you want to export a constant, you must use the "extern" specifier.
Otherwise you will get a link error.

You can declare and export a class object in the exact same manner:

```
__declspec(dllexport) CRect MyRect(30, 30, 300, 300);
```

Note that you can only export a class object if the client application recognizes the class and has its header file. If you make a new class inside your .DLL, the client application won't recognize it without the header file.

When you export a variable or object, each client application which loads the .DLL will get its own copy. Thus, if two different applications are using the same .DLL, changes made by one application will not affect the other application.

It's important to remember that you can only export objects and variables which are of **global** scope within your .DLL. Local objects and variables cease to exist when they go out of scope. Thus, if your .DLL included the following, it wouldn't work.

```
MyFunction( )
{
     __declspec(dllexport) CSomeClass SomeObject;
     __declspec(dllexport) int SomeInt;
}
```

As soon as the object and variable go out of scope, they will cease to exist.

Export a function

Exporting functions is similar to exporting objects or variables.
You simply tack "_declspec(dllexport)" onto the beginning of your function prototype:

```
__declspec(dllexport) int SomeFunction(int);
```

If you are making an MFC regular .DLL which will be used by a client application written in C, your function declaration should look like this:

```
extern "C" __declspec(dllexport) int SomeFunction(int);
```

and your function definition should look like this:

```
extern "C" __declspec(dllexport) int SomeFunction(int x
{
     //do something
}
```

If you are building a regular .DLL which is **dynamically linked** to the MFC code library .DLL, you must insert the AFX_MANAGE_STATE macro as
the first line of any exported function. Thus, your function definition would look like this:

```
extern “C” __declspec(dllexport) int AddFive(int x)

{

     AFX_MANAGE_STATE(AfxGetStaticModuleState( ));

     return x + 5;

}
```

It doesn’t hurt to do this in every regular .DLL. If you you switch your .DLL to static linking, the macro will simply have no effect.

That’s all there is to exporting functions. Remember, only an MFC extension .DLL can export functions with MFC data types in the parameters or return value.

Export a pointer

Exporting an uninitialized pointer is simple. You do it the same way you export a variable or object:

```
__declspec(dllexport) int* SomeInt;
```

You can also export an initialized object this way:

```
__declspec(dllexport) CSomeClass* SomePointer =

                                        new CSomeClass;
```

Of course, if you declare and initialize your pointer you need to find a place to delete it.

In an extension .DLL, you will find a function called DllMain( ). This function gets called when the client program attaches your .DLL and again when it detaches. So here's one possible way to handle your pointers in an extension .DLL:

```
#include "SomeClass.h"
_declspec(dllexport) CSomeClass* SomePointer = new CSc


DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOI

{

    if (dwReason == DLL_PROCESS_ATTACH)

    {


    }

    else if (dwReason == DLL_PROCESS_DETACH)

    {
```

```
        delete SomePointer;

    }

}
```

A regular .DLL looks more like an ordinary MFC executable. It has an object
derived from CWinApp to handle opening and closing your .DLL. You can use the
class wizard to add an InitInstance( ) function and an ExitInstance( ) function.

```
int CMyDllApp::ExitInstance()

{

     delete SomePointer;

     return CWinApp::ExitInstance();

}
```

Using the .DLL in a client application

A .DLL can't run on its own. It requires a client application to load it and use its interface. Making a client application that can do so is not difficult.

When you compile your .DLL, the compiler creates two important files: the .DLL file and the .lib file. Your client application needs both of these. You must copy them into the project folder of your client application. Note that the .DLL and .lib files that are created when

you build in Debug are different that those built when you build in Release.
When you are building your client application in Debug, you need the Debug versions of the .DLL and .lib files, and when you are building in Release you need the Release .DLL and .lib. The easiest way to handle this is to put the Debug .DLL and .lib files in your client application's Debug folder and the Release .DLL and .lib in the Release folder.

The next step is to go into your client project settings and tell the linker to look for your .lib file. You must tell the linker the name of your .lib file and where it can be found. To do this, open the project settings, go to the "Link" tab and enter your file name and path in the "Object/library modules" box. It should look something like this:

In addition to the .DLL and .lib files, your client application needs a header file for the imported classes, functions, objects and variables. When we were exporting, we added "__declspec(dllexport)" to our declarations.
Now when we are importing, we will add "__declspec(dllimport)." So if we
wanted to import the variable, object and function used in our previous examples, our header file would contain the following:

```
__declspec(dllimport) int SomeFunction(int);

__declspec(dllexport) CSomeClass SomeObject;

__declspec(dllexport) int SomeInt;
```

Remember, if you used the **extern "C"** specifier in the .DLL, you must also use it in the client application:

```
extern "C" __declspec(dllimport) int SomeFunction(int);
```

To make things more readable, we might write it like this instead:

```
#define DLLIMPORT __declspec(dllimport)
DLLIMPORT int SomeFunction(int);

DLLIMPORT CSomeClass SomeObject;

DLLIMPORT int SomeInt;
```

Now that you have declared your object, variable and function in a header file inside your client application, they are available for use.

To import an entire class, you must copy the entire .h header file into the client application. The .DLL and the client application will thus have identical header files for the exported class, except one will say "class __declspec(dllexport) CMyClass" and one will say "class __declspec(dllimport) CMyClass". If you are making an MFC extension .DLL, you could instead say "class AFX_EXT_CLASS CMyClass" in both places.

Once you are done building your client application and you're ready to turn it over to the actual users, you should give them your Release executable and the Release .DLL. You do not need to give the users the

.lib file. The .DLL can go in the same directory as the executable, or it can go in the Windows System directory. As discussed above, you may also have to provide your users with the correct MFC code library .DLL. This .DLL was loaded onto your computer when you installed Visual C++. Your users, however, may not have it. It does not come standard with Windows.

A Word of Caution

This article should provide you with enough information to start building your own .DLLs. A word of caution is needed, however. As mentioned at the beginning of this article, there are several serious shortcomings to .DLLs. These shortcomings are the reason that we now have COM and ATL.
There are two main problems. First, a .DLL built with one brand of compiler may not be compatible with a client application built with a different compiler. Second, when you modify the .DLL, you may have to recompile the client application, even though you aren't changing any code in the client application. You may still have to copy in a new .DLL and .lib file and recompile.

There are ways to avoid this problem under some circumstances. I'll discuss the problem in more detail in the next article.