**Dynamic Link Libraries**

We have now seen that memory management and file mapping are important and useful techniques in a wide class of programs. The OS itself also uses memory management, and DLLs are the most visible and important use because Windows applications use DLLs extensively. DLLs are also essential to higher-level technologies, such as COM, and many software components are provided as DLLs. The first step is to consider the different methods of constructing libraries of commonly used functions.

**Static and Dynamic Libraries**

The most direct way to construct a program is to gather the source code of all the functions, compile them, and link everything into a single executable image. Common functions, such as , can be put into a library to simplify the build process. This technique was used with all the sample programs presented so far, although there were only a few functions, most of them for error reporting. This monolithic, single-image model is simple, but it has several disadvantages.

• The executable image may be large, consuming disk space and physical memory at run time and requiring extra effort to manage and deliver to users.

• Each program update requires a rebuild of the complete program even if the changes are small or localized.

• Every program in the computer that uses the functions will have a copy of the functions, possibly different versions, in its executable image. This arrangement increases disk space usage and, perhaps more important, physical memory usage when several such programs are running concurrently.

• Distinct versions of the program, using different techniques, might be required to get the best performance in different environments.

**DLLs solve these and other problems quite neatly.**

• Library functions are not linked at build time. Rather, they are linked at program load time (implicit linking) or at run time (explicit linking). As a result, the program image can be much smaller because it does not include the library functions.

• DLLs can be used to create shared libraries. Multiple programs share a single library in the form of a DLL, and only a single copy is loaded into memory. All programs map the DLL code to their process address space, although each process has a distinct copy of the DLL’s global variables. For example, the function was used by nearly every example program; a single DLL implementation could be shared by all the programs.

• New versions or alternative implementations can be supported simply by supplying a new version of the DLL, and all programs that use the library can use the new version without modification.

• The library will run in the same processes as the calling program. DLLs, sometimes in limited form, are used in nearly every OS. For example, UNIX uses the term “shared libraries” for the same concept. Windows uses DLLs to implement the OS interfaces, among other things. The entire Windows API is supported by a DLL that invokes the Windows kernel for additional services. Multiple Windows processes can share DLL code, but the code, when called, runs as part of the calling process and thread. Therefore, the library will be able to use the resources of the calling process, such as file handles, and will use the calling thread’s stack. DLLs should therefore be written to be thread-safe.

**Implicit Linking**

Implicit or load-time linking is the easier of the two techniques. The required steps, using Microsoft Visual C++, are as follows.

1. The functions in a new DLL are collected and built as a DLL rather than, for example, a console application

2. The build process constructs a library file, which is a stub for the actual code and is linked into the calling program at build time, satisfying the function references. The file contains code that loads the DLL at program load time. It also contains a stub for each function, where the stub calls the DLL. This file should be placed in a common user library directory specified to the project.

3. The build process also constructs a file that contains the executable image. This file is typically placed in the same directory as the application that will use it, and the application loads the DLL during its initialization. The alternative search locations are described in the next section.

4. Take care to export the function interfaces in the DLL source

**Exporting and Importing Interfaces**

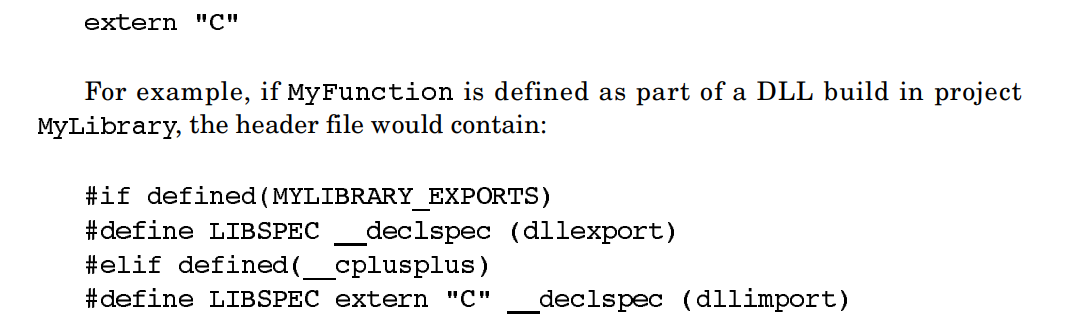
The most significant change required to put a function into a DLL is to declare it to be exportable (UNIX and some other systems do not require this explicit step). This is achieved either by using a file or, more simply, with Microsoft C/C++, by using the storage modifier as follows:

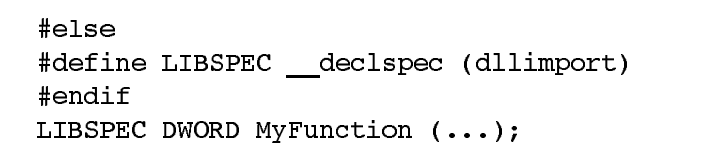


The build process will then create a file and a file. The file is the stub library that should be linked with the calling program to satisfy the external references and to create the actual links to the file at load time.

The calling or client program should declare that the function is to be imported by using the storage modifier. A standard technique is to write the include file by using a preprocessor variable created by appending the Microsoft Visual C++ project name, in uppercase letters, with .

One further definition is necessary. If the calling (importing) client program is written in C++, is defined, and it is necessary to specify the C calling convention, using:





Visual C/C++ automatically defines when invoking the compiler within the DLL project. A client project that uses the DLL does not define , so the function name is imported from the library.

When building the calling program, specify the file. When executing the calling program, ensure that the file is available to the calling program; this is frequently done by placing the file in the same directory as the executable. As mentioned previously, there is a set of DLL search rules that specify the order in which Windows searches for the specified file as well as for all other DLLs or executables that the specified file requires, stopping with the first instance located. The following default safe DLL search mode order is used for both explicit and implicit linking:

• The directory containing the loaded application.

• The system directory. You can determine this path with ; normally its value is .

• The 16-bit Windows system directory. There is no function to obtain this path, and it is obsolete for our purposes.

• The Windows directory ( ).

• The current directory.

• Directories specified by the environment variable, in the order in which they occur

**Explicit Linking**

Explicit or run-time linking requires the program to request specifically that a DLL be loaded or freed. Next, the program obtains the address of the required entry point and uses that address as the pointer in the function call. The function is not declared in the calling program; rather, you declare a variable as a pointer to a function. Therefore, there is no need for a library at link time. The three required functions are (or ), , and . Note: The function definitions show their 16-bit legacy through far pointers and different handle types.

The two functions to load a library are

