DLL’s IN WINDOWS

Intriguing Introduction: Unveiling the Magic of Dynamic-Link Libraries (DLLs)

In the ever-evolving world of Windows programming, we've conquered the art of crafting standalone programs. Now, it's time to delve into the fascinating realm of dynamic-link libraries (DLLs), one of the cornerstones of Windows architecture. These enigmatic modules, often referred to as shared libraries or simply dylibs, play a crucial role in the intricate dance of Windows applications.



Imagine a vast library filled with countless volumes, each one holding specific functionalities that various programs can draw upon. That's essentially what DLLs are! They store reusable code and resources, acting as shared building blocks for multiple programs. This not only reduces redundancy but also boosts efficiency, since programs don't need to carry duplicate copies of the same functionality.



Most of those seemingly endless Windows files you encounter? A significant portion are either full-fledged programs or these dynamic powerhouses masquerading as DLLs. Now, buckle up, fellow coder, because we're about to embark on a journey into the exciting world of DLL creation!

While fundamental programming principles remain the backbone of both program and DLL development, subtle yet crucial differences emerge. We'll dive into these distinctions, equipping you with the knowledge and tools to craft your own dynamic-link masterpieces.



So, get ready to:

* Unravel the mysteries of DLL structure and function exports.
* Master the art of interfacing with programs through well-defined APIs.
* Grasp the intricacies of memory management and dynamic loading.
* Navigate the nuances of thread safety and resource sharing.

By the end of this chapter, you'll be well on your way to becoming a dynamic-link demigod, wielding the power of these versatile modules to enhance your Windows programming repertoire. So, let's embark on this adventure together and unlock the magic of DLLs!

UNLOCKING THE POWER OF SHARED LIBRARIES:

Distinct from Executables: DLLs aren't standalone programs. They're separate files containing functions and resources that can be called upon by programs and other DLLs.

Dynamic Linking at Runtime: Unlike static linking during program development, dynamic linking occurs when a program actually runs. Windows links function calls in the program to the corresponding functions within the DLL, enabling seamless code sharing.

The Fabric of Windows:

Core System Components: KERNEL32.DLL, USER32.DLL, GDI32.DLL, device drivers, and font files are examples of DLLs that form the foundation of Windows functionality.

Extension to Windows: Creating a DLL is essentially extending Windows’ capabilities, offering reusable code and resources to other programs.

Flexible File Extensions:

Standard .DLL: While DLLs can bear various extensions, .DLL is the most common.

Automatic Loading: Windows automatically loads DLLs with the .DLL extension. Other extensions require explicit loading using LoadLibrary or LoadLibraryEx functions.

Advantages of DLLs:

Code Reusability: Multiple programs can leverage the same DLL, reducing code duplication, conserving disk space, and streamlining memory usage.

Modular Updates: Changes to DLLs can be made independently without relinking dependent programs, simplifying maintenance and updates.

Sharing Resources: DLLs foster efficient resource sharing, including fonts, icons, images, and other non-executable data.

Ideal for Large Applications:

Common Routines: In extensive applications with multiple programs, DLLs house frequently used functions, promoting code efficiency and maintainability.

Accounting Example: An accounting package with many programs could benefit from sharing common routines within an ACCOUNT.DLL, reducing redundancy and enhancing update management.

Creating Viable Products:

Independent Products: DLLs can be standalone products licensed for inclusion in other programs, expanding their reach.

3D Drawing Example: A GDI3.DLL containing 3D drawing routines could be licensed to multiple graphics programs, ensuring users only need a single copy of the DLL.

DISSECTING THE LIBRARY LANDSCAPE:

Conquering Confusion: The term "library" wears multiple hats in Windows programming, each with a distinct role. Let's dive into their unique contributions.

Dynamic-Link Libraries (DLLs): Sharing the Wealth of Code and Resources

Building Blocks of Windows: DLLs are like LEGO® bricks, forming the core of Windows functionality and enabling versatile code and resource sharing.

Shared Resources for All: Imagine a community library where programs can borrow functions and resources, fostering efficiency and collaboration.

Dynamic Partnerships: DLLs join programs at runtime, forming adaptable alliances that streamline code execution and reduce redundancy.

Object Libraries (.LIB Files): Statically Linking for Integration

Becoming One with the Program: Think of object libraries as essential ingredients blended directly into a program's executable, ensuring their permanent presence.

The Foundation Beneath: They provide foundational code that supports essential tasks, like the C runtime library (LIBC.LIB) offering core functionality.

Import Libraries: The Architects of Dynamic Linking

Blueprints for Collaboration: Import libraries act as skilled architects, providing the linker with detailed plans for constructing connections between programs and DLLs.

Bridging the Gap: They facilitate communication and cooperation, translating function calls within programs into specific instructions for DLL interactions.

Key Differences and Usage: Understanding the Timing and Purpose

DLLs: Runtime Collaborators: Like trusted colleagues, DLLs join programs in action, providing their expertise at the moment of need.

Object and Import Libraries: The Development Team: These libraries contribute during the initial development phase, setting the stage for successful runtime partnerships.

DLL Loading: Searching for the Right Partner

Windows Plays Matchmaker: Windows diligently searches for required DLLs, following a specific path to ensure programs can connect with their necessary collaborators:

* Program's home directory.
* Current working directory.
* Windows system directory.
* Windows directory.
* Directories listed in the PATH variable.

Mastering Workspaces and Projects: A Foundation for Organized Development

Think of workspaces as spacious studios where multiple projects can coexist harmoniously. They offer a bird's-eye view of your development landscape, ensuring a well-coordinated workflow.

Projects, on the other hand, are like focused workshops dedicated to crafting specific components, whether they be dynamic-link libraries or standalone applications.

Key Considerations for DLL Project Setup:

Directory Structure: Placing both the workspace and project within the same root directory streamlines file management and simplifies the development process. It's akin to having all your tools and materials within arm's reach in a well-organized workspace.

Header Files: These blueprints of the DLL act as communication bridges, defining functions and structures that other programs can interact with. They ensure a common understanding of the DLL's capabilities, fostering seamless collaboration.

Source Files: Containing the actual code implementation, these files are where the magic happens. They bring the DLL's functionality to life, empowering other programs to leverage its capabilities.

Analogies for Enhanced Understanding:

Think of DLLs as shared toolboxes: Multiple programs can access the same tools (functions and resources) within a DLL, promoting efficiency and code reusability.

Imagine workspaces as construction sites: They provide the overarching structure for managing multiple projects, ensuring a cohesive development process.

Projects are like individual buildings: Each project focuses on a specific component, contributing to the overall construction of a robust software application.

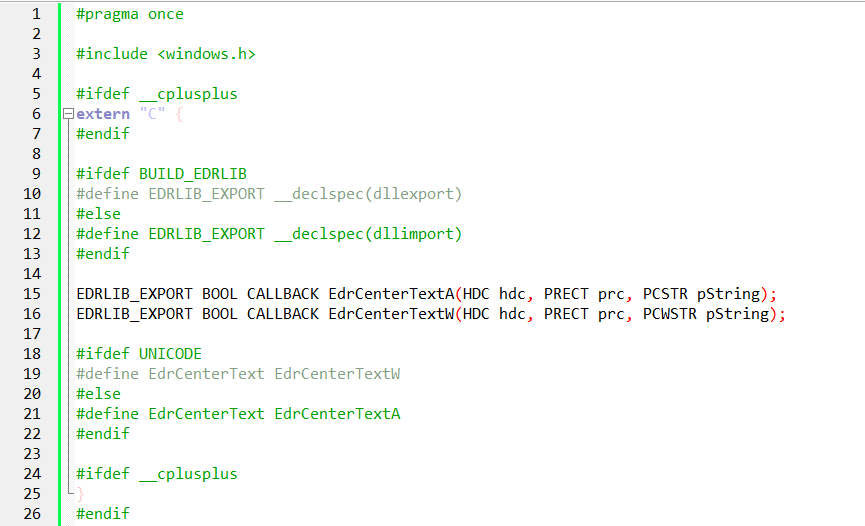
Stay Curious, Stay Engaged:

As we delve deeper into the contents of EDRLIB.H and EDRLIB.C, we'll uncover the intricate details that shape the DLL's functionality.

The creation of the test application EDRTEST.EXE will reveal the dynamic interplay between the DLL and the program, showcasing the power of shared code and resources.

Embrace the journey of discovering the multifaceted world of DLLs, and prepare for even more exciting concepts and techniques as we progress!

EDRLIB.H



The code snippet you provided showcases the usage of the #pragma once directive and the \_\_declspec(dllexport) and \_\_declspec(dllimport) attributes in C/C++ code.

#pragma once:

The #pragma once directive is a non-standard but widely supported preprocessor directive that ensures a header file is included only once during compilation, regardless of how many times it is referenced. It acts as an include guard, preventing multiple inclusions of the same header file, which can cause compilation errors due to duplicate definitions.

\_\_declspec(dllexport) and \_\_declspec(dllimport):

These attributes are Microsoft-specific and are used for exporting and importing functions and data from dynamic-link libraries (DLLs) in Windows.

* \_\_declspec(dllexport) is used to specify that a function or data item should be exported from a DLL. When a DLL is built with this attribute, the functions or data can be accessed by other modules (exe files or other DLLs) that link against the DLL.
* \_\_declspec(dllimport) is used to specify that a function or data item is imported from a DLL. When a module (exe file or DLL) is built with this attribute, it indicates that the functions or data are defined in an external DLL and should be resolved at runtime.

EDRLIB\_EXPORT:

``EDRLIB\_EXPORT is a preprocessor macro that is defined based on whether theBUILD\_EDRLIBmacro is defined. IfBUILD\_EDRLIBis defined, EDRLIB\_EXPORTis set to\_\_declspec(dllexport), indicating that the associated functions or data items are being exported from the DLL. If BUILD\_EDRLIBis not defined, EDRLIB\_EXPORTis set to\_\_declspec(dllimport)`, indicating that the associated functions or data items are being imported from the DLL.

Function declarations:

The code snippet declares two functions: EdrCenterTextA and EdrCenterTextW. The A and W suffixes indicate that these functions have ASCII and wide-character (Unicode) versions, respectively. The functions take an HDC (handle to a device context), a pointer to a RECT structure, and a string as arguments.

EdrCenterText macro:

The EdrCenterText macro is defined based on whether the UNICODE macro is defined. If UNICODE is defined, the macro expands to EdrCenterTextW, which refers to the wide-character version of the function. If UNICODE is not defined, the macro expands to EdrCenterTextA, which refers to the ASCII version of the function.

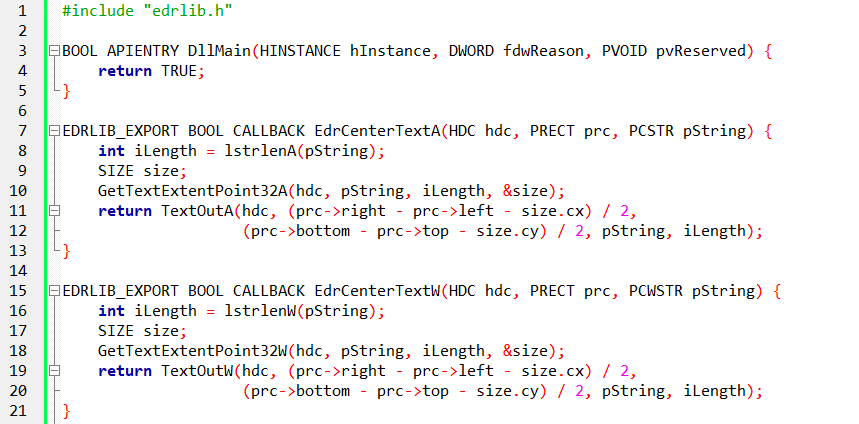
extern "C":

The extern "C" block is used to ensure that the function declarations have C linkage. This is important when the code is compiled with a C++ compiler to prevent name mangling, which allows the functions to be called from C code without any issues.

In summary, the code snippet provides a mechanism for exporting and importing functions from a DLL using the \_\_declspec(dllexport) and \_\_declspec(dllimport) attributes. The EDRLIB\_EXPORT macro is used to conditionally apply the appropriate attribute based on the presence of the BUILD\_EDRLIB macro.

The EdrCenterText macro allows for using either the ASCII or wide-character version of the function based on the UNICODE macro. The extern "C" block ensures that the function declarations have C linkage, enabling their use in both C and C++ code.

EDRLIB.C



The code you provided is an implementation of the functions declared in the edrlib.h header file. It defines the behavior of the EdrCenterTextA and EdrCenterTextW functions. Let's break down the functioning of this code:

*BOOL APIENTRY DllMain*

This is the entry point function for the DLL. It is called by the operating system when certain events occur, such as the loading or unloading of the DLL. In this code, the DllMain function is a placeholder and returns TRUE without performing any specific actions. Depending on the requirements of your DLL, you can customize this function to handle initialization, cleanup, or other tasks.

*EDRLIB\_EXPORT BOOL CALLBACK EdrCenterTextA*

This function is the implementation of the ASCII version of EdrCenterText. It takes an HDC (handle to a device context), a pointer to a RECT structure, and a null-terminated ASCII string as arguments. Within the function:

*lstrlenA(pString) calculates the length of the input string.*

GetTextExtentPoint32A determines the dimensions (width and height) of the text when drawn with the specified device context and font. The result is stored in the size structure.

TextOutA outputs the text to the specified device context. It calculates the position for centering the text within the given rectangle (prc) based on the text dimensions (size).

*EDRLIB\_EXPORT BOOL CALLBACK EdrCenterTextW*

This function is the implementation of the wide-character (Unicode) version of EdrCenterText. It takes an HDC, a pointer to a RECT structure, and a null-terminated wide-character string as arguments. The function logic is similar to EdrCenterTextA, but it operates on wide-character strings using the corresponding wide-character Win32 API functions (lstrlenW, GetTextExtentPoint32W, TextOutW).

The EDRLIB\_EXPORT macro ensures that these functions are properly exported from the DLL when building with BUILD\_EDRLIB defined. It applies the \_\_declspec(dllexport) attribute to indicate that the functions should be accessible from other modules that link against the DLL.

Overall, this code provides an implementation for centering text within a rectangle using either ASCII or wide-character strings, depending on the function called. It utilizes Win32 API functions to calculate the dimensions of the text and position it at the center of the given rectangle.