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.

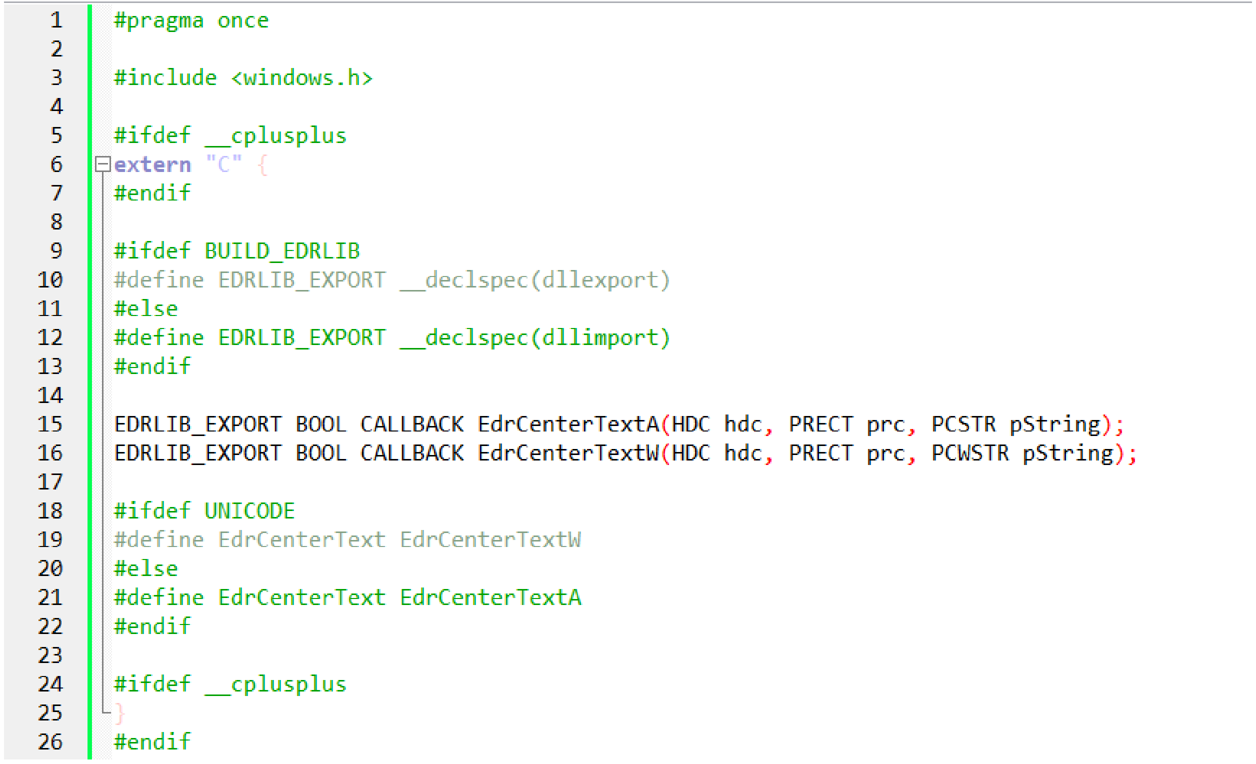
Stay Curious, Stay Engaged:

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



```
#pragma once

#include <windows.h>

#ifdef __cplusplus
extern "C" {
#endif

#ifdef BUILD_EDRLIB
#define EDRLIB_EXPORT __declspec(dllexport)
#else
#define EDRLIB_EXPORT __declspec(dllimport)
#endif

EDRLIB_EXPORT BOOL CALLBACK EdrCenterTextA(HDC hdc, PRECT prc, PCSTR pString);
EDRLIB_EXPORT BOOL CALLBACK EdrCenterTextW(HDC hdc, PRECT prc, PCWSTR pString);

#ifdef UNICODE
#define EdrCenterText EdrCenterTextW
#else
#define EdrCenterText EdrCenterTextA
#endif

#ifdef __cplusplus
}
#endif
```

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.

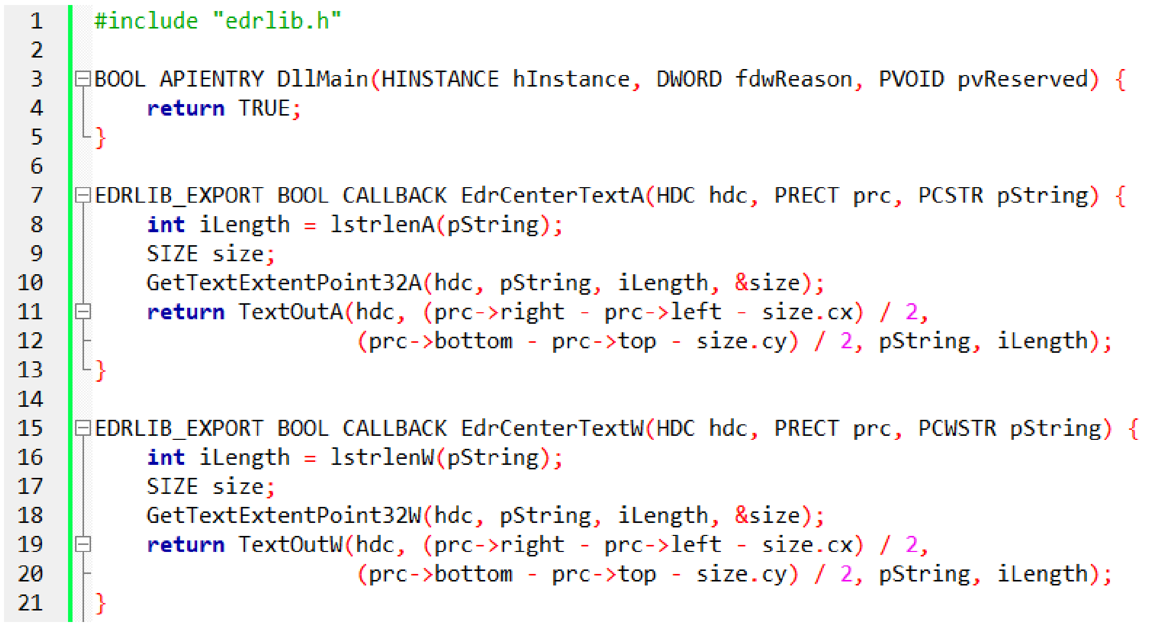
extern "C":

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



```c
#include "edrlib.h"

BOOL APIENTRY DllMain(HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved) {
    return TRUE;
}

EDRLIB_EXPORT BOOL CALLBACK EdrCenterTextA(HDC hdc, PRECT prc, PCSTR pString) {
    int iLength = lstrlenA(pString);
    SIZE size;
    GetTextExtentPoint32A(hdc, pString, iLength, &size);
    return TextOutA(hdc, (prc->right - prc->left - size.cx) / 2,
                    (prc->bottom - prc->top - size.cy) / 2, pString, iLength);
}

EDRLIB_EXPORT BOOL CALLBACK EdrCenterTextW(HDC hdc, PRECT prc, PCWSTR pString) {
    int iLength = lstrlenW(pString);
    SIZE size;
    GetTextExtentPoint32W(hdc, pString, iLength, &size);
    return TextOutW(hdc, (prc->right - prc->left - size.cx) / 2,
                    (prc->bottom - prc->top - size.cy) / 2, pString, iLength);
}
```

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.

BUILDING THE DLL: UNVEILING THE TREASURE MAP AND THE TREASURE CHEST:

EDRLIB.LIB: This import library acts like a meticulously crafted treasure map, guiding the linker to the exact locations of functions and resources within the DLL. It's an essential tool for programs seeking to tap into the DLL's riches.

EDRLIB.DLL: This dynamic-link library is the treasure chest itself, holding the executable code that delivers valuable functionality to external programs. It's the embodiment of code reusability and modularity.

Navigating Text Encodings: Speaking Fluent ANSI and Unicode:

Separate Functions for Each Encoding: The DLL caters to both ANSI and Unicode text through thoughtful function pairs: EdrCenterTextA for ANSI and EdrCenterTextW for Unicode. It's like having a skilled translator who can seamlessly bridge different language worlds.

Choosing the Right Windows API Functions: The DLL carefully selects appropriate Windows API functions based on the text encoding, ensuring compatibility and accurate text handling. It's like using the right tools for the job, whether it's building a house with ANSI bricks or constructing a Unicode palace.

The UNICODE Switchboard: The #ifdef UNICODE block in EDRLIB.H acts as an intelligent switchboard, automatically directing calls to the correct text handling function based on the program's encoding settings. It's a clever mechanism that streamlines code maintenance and eliminates the need for manual configuration.

DllMain: The DLL's Gatekeeper and Life Cycle Manager:

Replacing WinMain: While WinMain is the heart of traditional Windows applications, DllMain takes on this central role in DLLs. It's the first function called when a DLL is loaded and the last function called when it's unloaded, making it responsible for essential initialization and termination tasks.

A Successful Start: In this particular DLL, DllMain simply returns TRUE, indicating successful initialization and a smooth start to the DLL's journey. It's like a green light signaling that the DLL is ready for action.

Exporting Functions: Opening Doors for Collaboration:

The EXPORT Macro: A Passport for Functions: The EXPORT macro, defined as extern "C" __declspec(dllexport), serves as a passport for functions, allowing them to cross borders and be used by external programs. It's a powerful tool for enabling collaboration and code sharing.

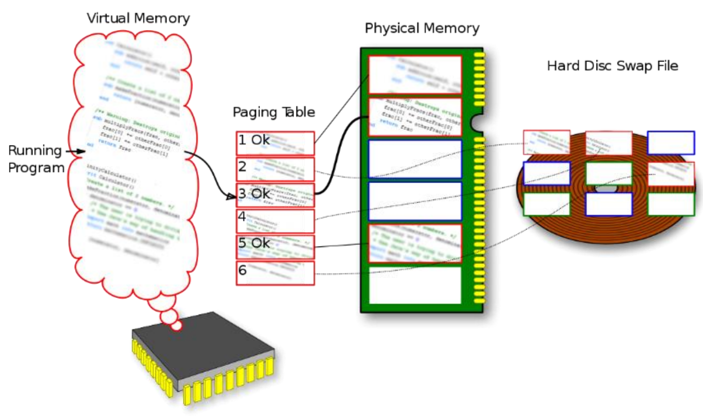
Function Visibility and Cross-Language Compatibility: The __declspec(dllexport) component makes functions visible within the DLL, ensuring they can be found and called by external programs. Meanwhile, extern "C" prevents C++ name mangling, promoting smooth communication between different programming languages. It's like ensuring everyone speaks the same language, fostering understanding and cooperation.

Key Takeaways: Building a Shared Code Universe:

- DLLs offer a potent path towards modularity and code reusability, but careful attention to text encoding is crucial to ensure broad compatibility and avoid communication glitches.
- DllMain serves as a pivotal gatekeeper and lifecycle manager, overseeing the DLL's smooth integration into the application landscape.
- The EXPORT macro acts as a bridge, opening up the DLL's functions to external programs and fostering a collaborative ecosystem of shared code resources.

EMBARKING ON THE DLL JOURNEY: INITIALIZATION AND BEYOND

When a DLL is loaded into a process's address space, a world of possibilities unfolds. This integration begins with the DLL_PROCESS_ATTACH event, signaling the genesis of DLL orchestration. During this stage, critical tasks such as memory allocation, resource acquisition, and setup of global variables take center stage. The successful execution of these steps sets the foundation for the smooth progression of the program.



To ensure a seamless transition, it is essential to prioritize efficiency and brevity within the DllMain function. This function acts as the entry point for the DLL and plays a crucial role in its initialization. By carefully crafting DllMain, developers can avoid potential performance bottlenecks that might hinder the overall system performance. By keeping the code concise and focused, unnecessary delays and resource consumption can be minimized.

Moreover, comprehensive error handling strategies should be implemented within DllMain to safeguard against unexpected issues. Error handling mechanisms, such as proper exception handling or appropriate return codes, can help detect and gracefully recover from errors during the DLL's integration process. By proactively addressing potential failure scenarios, developers can enhance the reliability and stability of the DLL.

Additionally, it is crucial to prioritize thread safety within DllMain to prevent race conditions and maintain harmony within the system. Since multiple threads can concurrently access the DLL during its integration, synchronization mechanisms, such as locks or critical sections, should be employed to ensure data consistency and integrity. By meticulously guarding against thread-related issues, developers can mitigate potential conflicts and maintain the overall stability and correctness of the DLL.

The Art of Graceful Departure: Cleanup and Closure

DLL_PROCESS_DETACH: The DLL's Final Curtain Call

As the DLL's journey within a process nears its end, meticulous cleanup ensures a graceful exit. Releasing resources, closing handles, and finalizing any remaining tasks are of utmost importance. This diligent attention to cleanup tasks prevents memory leaks and resource conflicts, fostering a well-maintained and stable system.

Navigating the Threads of Execution: A Delicate Dance

1. DLL_THREAD_ATTACH: Welcoming New Threads to the Performance

When new threads emerge within an attached process, DllMain gracefully receives notification. If necessary, thread-specific initialization and synchronization measures are undertaken to ensure seamless collaboration and proper functioning.

2. DLL_THREAD_DETACH: Bidding Farewell to Departing Threads

As threads depart, they are given a respectful send-off. However, caution must prevail. It is advisable to avoid using PostMessage due to the potential loss of messages. Additionally, vigilance regarding thread synchronization is crucial to prevent data corruption and unexpected behavior.

Additional Pearls of Wisdom:

1. Global Instance Handle: A Key for Resource Access

The hInstance parameter, often stored globally, unlocks the DLL's resource potential, enabling the usage of dialog boxes and other resources.

2. Multiple Process Encounters

DllMain encounters each process that loads the DLL independently, even if multiple instances of the same program exist within the system.

3. Thread Synchronization: A Delicate Art

When DllMain accesses shared resources, meticulous thread synchronization is necessary to safeguard against data corruption and ensure predictable behavior.

Conclusion: Mastering the Orchestration

By comprehending the nuances of DllMain and adhering to best practices, developers possess the ability to craft well-behaved and adaptable DLLs that seamlessly integrate into the dynamic tapestry of Windows applications. Through careful attention to initialization, cleanup, and thread management, DLLs contribute to a harmonious and efficient software ecosystem.

EDRTEST PROGRAM

A Journey of Collaboration and Flexibility:

EDRTEST.C Embarks on a Quest: This program embarks on a journey to demonstrate the power of dynamic-link libraries (DLLs) by integrating functions from EDRLIB.DLL, showcasing the benefits of modularity and extensibility in software development.

A Windows Application with a Clear Purpose:

The Foundation: EDRTEST.C adheres to the core structure of a Windows program, establishing a window, a message loop, and a window procedure (WndProc) to interact with the operating system and respond to user events.

The Spotlight: The WM_PAINT message handler within WndProc takes center stage, acting as the catalyst for calling upon the DLL's expertise.

A Partnership with a Dynamic Library:

Bridging the Gap: The program seamlessly integrates with EDRLIB.DLL by incorporating the necessary header file, edrlib.h, revealing the accessible functions within the DLL.

The Call for Text Expertise: When the window receives a WM_PAINT message, signaling a need to refresh its visual content, WndProc meticulously orchestrates a series of steps:

Obtaining the Canvas: It acquires a device context (hdc), the virtual canvas upon which graphical elements will be painted.

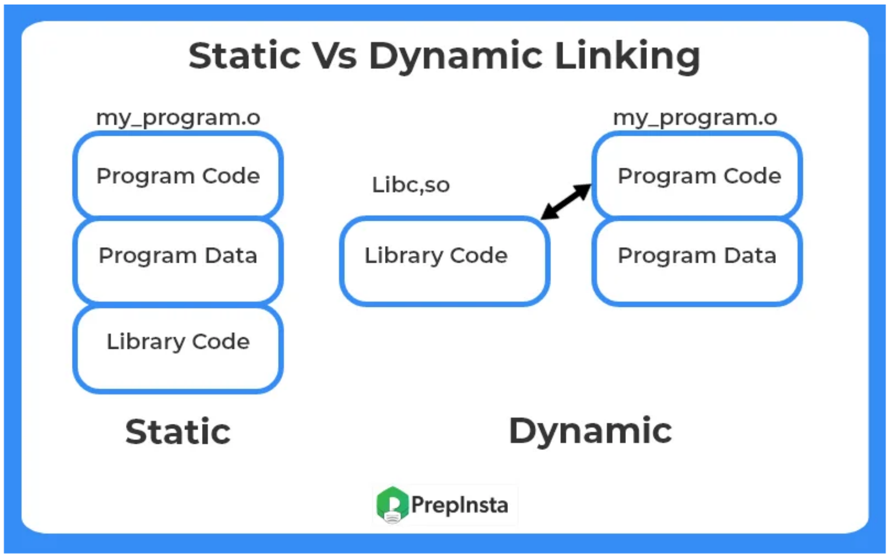
Measuring the Stage: It retrieves the dimensions of the window's client area (rect), defining the boundaries for text placement.

Extending the Invitation: It ventures beyond its own code and extends an invitation to the DLL's EdrCenterText function, passing along essential parameters:

- The device context (hdc), providing the means for drawing.
- A pointer to the RECT structure (prc), outlining the text's intended canvas.
- The text string itself (pString), awaiting its moment in the spotlight.
- Releasing the Canvas: It gracefully releases the device context, ensuring proper resource management and concluding the painting process.

Embracing Modularity, Embracing Potential:

- Code Reusability Redefined: EDRTEST.C embodies the concept of code reusability, showcasing how DLLs enable developers to share and leverage functionality across multiple programs, fostering efficient development and maintainability.
- A Symphony of Extensibility: The program gracefully demonstrates its ability to expand its own capabilities by incorporating external libraries, showcasing a world of potential for limitless growth and adaptation.
- A Path to Independent Updates: DLLs offer the flexibility to be updated independently, allowing for bug fixes, enhancements, or even complete replacement without necessitating a full recompilation of the main program, fostering a more agile and adaptable software ecosystem.



The provided program, EDRTEST.C, is an example program that utilizes the EDRLIB dynamic-link library (DLL). Let's break down the program into in-depth paragraphs:

Header Files and Declarations:

The program includes necessary header files, such as <windows.h>, and the header file for the EDRLIB DLL, "edrlib.h". It also declares the callback function WndProc and the WinMain entry point function.

Window Class Registration:

The program registers a window class by filling out a WNDCLASS structure. The WndProc function serves as the window procedure for handling messages associated with the application's window. The WNDCLASS structure specifies various attributes of the window, such as its style, background brush, icon, cursor, and class name. If the registration fails, an error message is displayed, indicating that the program requires Windows NT.

Window Creation and Display:

The program creates a window using the CreateWindow function. The window is given a title, dimensions, and window styles. The window is then shown and updated using the ShowWindow and UpdateWindow functions, respectively.

Message Loop:

The program enters a message loop using the GetMessage function. The loop retrieves messages from the message queue and dispatches them to the appropriate window procedure using the TranslateMessage and DispatchMessage functions. The loop continues until the WM_QUIT message is received.

Window Procedure:

The WndProc function handles messages sent to the program's window. In the case of the WM_PAINT message, the function begins painting by calling BeginPaint and obtains the client area's rectangle using GetClientRect. The EdrCenterText function from the EDRLIB DLL is then called to center the text "This string was displayed by a DLL" within the client area. Finally, the painting is ended using EndPaint.

Window Destruction:

If the program receives the WM_DESTROY message, it posts a quit message to the message queue using PostQuitMessage, which causes the message loop to exit.

Output:

Based on the provided code, the program should display a window titled "DLL Demonstration Program" with the text "This string was displayed by a DLL" centered within the client area of the window. The EdrCenterText function from the EDRLIB DLL is responsible for centering the text.

When the program is executed, the window will be shown, and the text will be painted in the client area. The exact appearance may depend on factors such as the window's size, the font used, and the system's default settings for window appearance. However, the expected behavior is that the text will be centered horizontally and vertically within the window's client area.

SHARED MEMORY IN DLLS: BREAKING ISOLATION FOR COLLABORATION

While Windows generally isolates applications using the same DLLs, certain scenarios necessitate shared memory, enabling communication and data exchange between processes. STRLIB and STRPROG demonstrate this concept:

STRLIB.DLL: A Shared Memory Haven for Strings

Purpose: Stores and sorts up to 256 capitalized character strings in shared memory, accessible to multiple applications or instances of the same application.

AddString Function:

- Adds a capitalized string to STRLIB's shared memory.
- Takes a pointer (pStringIn) to the input string.
- If the string already exists, it adds another copy.
- Returns TRUE (nonzero) on success, FALSE (0) otherwise.
- Possible reasons for FALSE: empty string, memory allocation failure, or reaching the limit of 256 strings.

DeleteString Function:

- Deletes a string from STRLIB's shared memory.
- Takes a pointer (pStringIn) to the string to be deleted.
- If multiple matching strings exist, only the first one is removed.
- Returns TRUE (nonzero) on success, FALSE (0) otherwise.
- FALSE could indicate an empty string or the absence of a matching string.

GetStrings Function:

- Enumerates the strings stored in STRLIB's shared memory using a callback function.
- Takes a callback function pointer (pfnGetStrCallBack) and a user-defined parameter (pParam).
- The callback function (GetStrCallBack) is defined by the calling program.
- Calls the callback once for each string until it returns FALSE or all strings are processed.
- Returns the number of strings processed.

Unicode Support:

- Internally, STRLIB stores all strings in Unicode.
- Supports both Unicode and non-Unicode applications.
- Provides separate A (ANSI) and W (Wide) versions of functions (e.g., AddStringA, DeleteStringA) to handle the character set conversion.

Below is a concise and structured code snippet for the given scenario. Please note that this is a simplified example and doesn't include error handling or detailed implementation. It's intended to provide a basic understanding of the structure.

STRLIB.H (Header File)

```c
// STRLIB.H

#ifdef __cplusplus
#define EXPORT extern "C" __declspec (dllexport)
#else
#define EXPORT __declspec (dllexport)
#endif

// Function to add a string to shared memory
EXPORT BOOL CALLBACK AddString(const char* pStringIn);

// Function to delete a string from shared memory
EXPORT BOOL CALLBACK DeleteString(const char* pStringIn);

// Callback function for enumerating strings
EXPORT int CALLBACK GetStrings(BOOL(CALLBACK* pfnGetStrCallBack)(const char*, void*), void* pParam);
```

STRLIB.C (Implementation File)

```c
// STRLIB.C

#include <windows.h>
#include "STRLIB.h"

#define MAX_STRINGS 256

// Shared memory to store strings
char g_Strings[MAX_STRINGS][256];  // Assuming each string has a maximum length of 255 characters
int g_NumStrings = 0;

BOOL CALLBACK AddString(const char* pStringIn) {
    // Implementation to add a string to shared memory
    // Check for duplicates, allocate memory, and add the string
    // ...

    return TRUE; // Return TRUE on success, FALSE on failure
}

BOOL CALLBACK DeleteString(const char* pStringIn) {
    // Implementation to delete a string from shared memory
    // Find the string, remove it, and adjust the array
    // ...

    return TRUE; // Return TRUE on success, FALSE on failure
}

int CALLBACK GetStrings(BOOL(CALLBACK* pfnGetStrCallBack)(const char*, void*), void* pParam) {
    // Implementation to enumerate strings using the callback function
    // Call pfnGetStrCallBack for each string
    // ...

    return g_NumStrings; // Return the total number of strings processed
}
```

STRPROG.C (Test Program)

```c
// STRPROG.C

#include <stdio.h>
#include <windows.h>
#include "STRLIB.h"

int main() {
    // Test program (STRPROG) using the STRLIB functions
    // ...

    return 0;
}
```

Shared Memory in DLLs recap:

In Windows, applications that use the same dynamic-link libraries (DLLs) are typically isolated from each other. However, there are scenarios where you might want to create a DLL that contains shared memory, allowing multiple applications or instances of the same application to access and modify that memory. This can be achieved using memory-mapped files.

Let's examine how shared memory works with a program called STRPROG (String Program) and a dynamic-link library called STRLIB (String Library). STRLIB offers three exported functions that STRPROG can call, and one of these functions utilizes a callback function defined in STRPROG.

STRLIB is a DLL module that stores and sorts up to 256-character strings. The strings are capitalized and maintained using shared memory within STRLIB. STRPROG can use the three functions provided by STRLIB to add strings, delete strings, and retrieve all the current strings stored in STRLIB.

In the STRPROG test program, there are two menu items (Enter and Delete) that open dialog boxes to add and delete strings. STRPROG then lists all the current strings in its client area, which it obtains from STRLIB.

One of the functions defined in STRLIB is AddString, which adds a string to STRLIB's shared memory. The function takes a pointer to the string as an argument and capitalizes the string within the AddString function. If an identical string already exists in STRLIB's list of strings, this function adds another copy of the string.

AddString returns TRUE (nonzero) if it is successful and FALSE (0) otherwise. A FALSE return value can indicate a string length of 0, failure to allocate memory for the string, or reaching the maximum limit of 256 stored strings.

Another function in STRLIB is DeleteString, which removes a string from STRLIB's shared memory. Like AddString, it takes a pointer to the string as an argument. If multiple strings match the given string, only the first match is removed. DeleteString returns TRUE if it is successful and FALSE otherwise. A FALSE return value indicates a string length of 0 or the inability to find a matching string.

The third function in STRLIB, GetStrings, uses a callback function located in the calling program (STRPROG) to enumerate the strings currently stored in STRLIB's shared memory. GetStrings takes the pointer to the callback function (pfnGetStrCallBack) and a programmer-defined data pointer (pParam) as arguments.

The callback function, GetStrCallBack, must be defined in STRPROG and receives each string through its PSTR parameter. GetStrings calls GetStrCallBack once for each string or until the callback function returns FALSE. GetStrings returns the total number of strings passed to the callback function.

Supporting both Unicode and non-Unicode applications adds complexity to STRLIB. It provides both A (ANSI) and W (wide character) versions of its functions. Internally, STRLIB stores all strings in Unicode format. When a non-Unicode program uses STRLIB (calling AddStringA, DeleteStringA, and GetStringsA), the strings are converted to and from Unicode as needed.

In the STRPROG and STRLIB projects, which are part of the STRPROG workspace, there are two necessary files to create the STRLIB.DLL module. These files are assembled similarly to the EDRTEST workspace, as shown in Figure 21-3 of the original material.

Overall, shared memory in DLLs allows multiple applications or instances of the same application to access and modify a common memory space. By utilizing memory-mapped files, STRPROG and STRLIB demonstrate how shared memory can be effectively implemented and utilized for string management.

STRLIB LIBRARY PROGRAM

STRLIB is a library that provides string management functionality through shared memory. It consists of a header file (STRLIB.H) and a source file (STRLIB.C), which work together to implement the desired features.

The header file (STRLIB.H) defines constants for the maximum string count and length. These constants provide constraints for the number of strings that can be stored and the maximum length allowed for each string.

Additionally, the header file declares a callback function type for string retrieval. This callback function is used by the calling program (such as STRPROG) to receive the strings stored in STRLIB's shared memory. The callback function must be defined in the calling program and is responsible for processing each string passed to it.

The source file (STRLIB.C) includes the necessary headers, specifically windows.h and wchar.h, which provide the required functionality for working with shared memory and Unicode strings.

Within STRLIB.C, a shared memory section is created using the #pragma data_seg directive. This directive allows the definition of shared data that can be accessed by multiple instances of the DLL. In this case, the shared data includes two variables: iTotal and szStrings.

The iTotal variable keeps track of the current string count in the shared memory. It is updated whenever a string is added or deleted from the szStrings array.

The szStrings variable is a 2D array that stores the strings. It has a maximum capacity of 256 strings, and each string can have a maximum length of 63 characters. Notably, the strings are internally stored as Unicode to support both Unicode and non-Unicode applications.

To ensure proper linking of the shared memory section, the #pragma comment(linker,"/SECTION:shared,RWS") directive is used. This directive instructs the linker to link the shared memory section named "shared" with read, write, and shared permissions.

The exported functions provided by STRLIB are implemented in this source file:

AddString(A/W) function:

- This function adds a string to the shared memory.
- It takes an input string as a parameter and capitalizes it.
- The function then adds the capitalized string to the szStrings array in alphabetical order.
- It handles duplicates by allowing multiple copies of the same string.
- The function returns a success/failure status to indicate whether the string was added successfully.

DeleteString(A/W) function:

- This function removes the first matching string from the shared memory.
- It takes an input string as a parameter and searches for a match in the szStrings array.
- If a match is found, the function removes the first occurrence of the matching string.
- The function returns a success/failure status to indicate whether the string was deleted successfully.

GetStrings(A/W) function:

- This function retrieves the stored strings from the shared memory and passes them to a callback function provided by the calling program.
- It takes the pointer to the callback function and a programmer-defined data pointer as parameters.
- The function iterates through the szStrings array and calls the callback function for each stored string.
- The iteration continues until all strings have been processed or until the callback function returns FALSE.
- The function returns the total number of strings processed.

These functions have both ANSI (A) and Unicode (W) versions, allowing compatibility with different character encodings. The appropriate version is automatically selected based on the UNICODE define.

Shared Memory and Unicode Handling in STRLIB:

Shared Memory:

To enable shared memory functionality, STRLIB utilizes #pragma directives and a linker option. The #pragma data_seg directive is used in the STRLIB.C source file to define a shared memory section.

This shared memory section, named "shared," contains the necessary variables for storing and managing strings. By linking the shared memory section using the #pragma comment (linker,"/SECTION:shared, RWS") directive, the linker ensures that the shared memory is accessible to multiple processes using STRLIB.DLL.

This shared memory allows different instances of the calling program (e.g., STRPROG) to access and modify the same strings.

Unicode Support:

STRLIB internally stores strings as Unicode to maintain consistency across different character encodings.

It uses the wchar.h header to work with wide characters and provide Unicode support. To accommodate both ANSI and Unicode applications, STRLIB provides dual-versioned functions with A (ANSI) and W (Unicode) suffixes.

For example, the AddString function has both AddStringA and AddStringW versions. The appropriate function version is automatically selected based on the UNICODE define.

String Conversions:

When a non-Unicode application calls the A version of a function, such as AddStringA or GetStringsA, STRLIB performs the necessary conversions between ANSI and Unicode.

It converts the input string from ANSI to Unicode before processing it internally. Similarly, when retrieving strings with GetStringsA, STRLIB converts the stored Unicode strings back to ANSI before passing them to the callback function.

These conversions ensure seamless integration with both ANSI and Unicode applications.

Callback Function:

STRLIB's GetStrings function allows the calling program to retrieve strings stored in the shared memory one by one.

It takes a callback function pointer (GETSTRCB) and a parameter (pParam) as arguments.

The callback function, defined in the calling program, receives each string through its parameter (PSTR).

The GetStrings function iterates through the stored strings and calls the callback function for each string until all strings have been processed or until the callback function returns FALSE.

This callback mechanism enables the calling program to process the strings in a customized manner, providing flexibility and extensibility.

DLL Exports and DllMain:

- To make the functions within STRLIB.DLL accessible to other programs, the EXPORT macro is used to mark the desired functions for export.
- This allows other programs to import and utilize these functions as needed.
- The DllMain function is a required function that is called by the system when the DLL is loaded or unloaded.
- In the case of STRLIB, the DllMain function simply returns TRUE, indicating successful initialization.
- While this implementation does not involve any specific processing within DllMain, it can be extended to perform additional initialization or cleanup tasks if required.

Key Points:

- STRLIB demonstrates the usage of shared memory within a DLL to facilitate inter-process communication.
- It provides support for both ANSI and Unicode applications through dual-versioned functions.
- String conversions between ANSI and Unicode are handled to ensure compatibility with different character encodings.
- The callback function mechanism allows the calling program to receive and process strings retrieved from the shared memory in a customizable manner.
- The #pragma directives and linker options play a crucial role in establishing and linking the shared memory section, enabling efficient data sharing among multiple processes using STRLIB.DLL.

In summary, STRLIB is a library that offers string management capabilities through shared memory. It provides functions to add, delete, and retrieve strings stored in shared memory. The library supports both Unicode and non-Unicode applications, internally storing strings as Unicode. With the help of the callback function, the calling program can efficiently access and process the strings stored in STRLIB's shared memory.

STRPROG PROGRAM

Key Features:

Shared String Management: STRPROG empowers multiple users to create, maintain, and view a collaborative list of strings in real-time.

Collaborative Editing: Users can add and delete strings from the shared list, with changes instantly reflected across all open instances of STRPROG.

Shared Memory Foundation: This seamless collaboration is enabled by leveraging shared memory, a memory region accessible to multiple processes, ensuring data consistency.

DLL Interaction: STRPROG interacts with a dynamic-link library (DLL) named STRLIB, which handles the intricacies of shared memory management and string operations.

User-Friendly Interface: The program provides an intuitive interface for managing the string list, accessible through menu options and dialog boxes.

Efficient Communication: It employs custom messages to notify other STRPROG windows about updates, ensuring timely synchronization and a synchronized experience.

The STRPROG program is a simple application that demonstrates the usage of the STRLIB dynamic-link library. It allows users to enter and delete strings and displays the strings in its client area.

- The program consists of a main window procedure (WndProc) and several supporting functions. It includes the "strlib.h" header file, which defines the functions provided by the STRLIB library.
- When the program starts, it registers a window class and creates the main window. The main window's class name is "StrProg," and the window title is "DLL Demonstration Program." The window class specifies that the window should have a white background and handle horizontal and vertical window resizing.
- The program enters a message loop, where it retrieves messages from the message queue and dispatches them to the appropriate window procedure. This loop continues until a WM_QUIT message is received.
- The main window procedure (WndProc) handles various messages to perform the program's functionality. Here are some of the messages it handles:
- WM_CREATE: This message is sent when the window is created. In response to this message, the window procedure retrieves the instance handle and initializes some variables. It also registers a custom message (iDataChangeMsg) that will be used to notify instances of data changes.
- WM_COMMAND: This message is sent when a menu item or accelerator key is selected. The window procedure handles two menu items: IDM_ENTER and IDM_DELETE. When IDM_ENTER is selected, the program displays a dialog box (EnterDlg) that allows the user to enter a string. If the user clicks OK, the entered string is added using the AddString function provided by the STRLIB library.
- If the operation is successful, a custom message (iDataChangeMsg) is broadcasted to notify other instances of the program to update their data. IDM_DELETE works similarly but displays a different dialog box (DeleteDlg) and uses the DeleteString function to remove a string.
- WM_SIZE: This message is sent when the window is resized. The window procedure updates the client area's dimensions based on the new size.
- WM_PAINT: This message is sent when the window's client area needs to be repainted. In response to this message, the window procedure retrieves a device context (DC), sets up some parameters for displaying strings, and calls the GetStrings function provided by the STRLIB library. GetStrings enumerates the stored strings and calls the GetStrCallBack function for each string. The GetStrCallBack function uses the TextOut function to display the enumerated strings in the client area.
- WM_DESTROY: This message is sent when the window is being destroyed. The window procedure posts a WM_QUIT message to exit the message loop.

- The program also includes two dialog procedures (DlgProc) for handling the EnterDlg and DeleteDlg dialog boxes. These dialog procedures handle the initialization of the dialog boxes and process user actions such as button clicks.
- Additionally, the program includes a resource file (StrProg.rc) that defines the dialog boxes and menu items used by the program.
- One interesting aspect of the program is that it uses shared memory provided by the STRLIB library to store the character strings and their pointers. This allows multiple instances of the STRPROG program to share the same data. When a string is added or deleted, a custom message (iDataChangeMsg) is broadcasted to notify other instances of the program to update their data and repaint their windows.

Overall, the STRPROG program provides a basic example of using a dynamic-link library (STRLIB) to manage and share data between multiple instances of an application.

Enabling Data Sharing:

Windows typically isolates a process's memory, preventing direct access by other processes.

STRPROG circumvents this by strategically utilizing a shared memory section, allowing multiple instances to collaborate on a shared data set.

Creating the Shared Memory Section:

STRLIB establishes a special memory region using #pragma data_seg ("shared").

It places two crucial variables within this section:

iTotal: An integer tracking the current number of strings.

szStrings: A 2D array storing up to 256 strings, each with a maximum length of 63 characters.

The section is marked with #pragma data_seg ().

Linker Configuration:

The linker must be informed about this shared section to ensure its proper handling.

This is achieved either through a linker option in the project settings or directly within the STRLIB.C source code using #pragma comment(linker,"/SECTION:shared,RWS").

The RWS attribute designates the section as having read, write, and shared access permissions.

Shared Memory Mechanics:

All instances of STRPROG that load STRLIB gain access to this shared memory section.

They can directly read and modify the iTotal variable and the szStrings array, enabling seamless collaboration.

This shared memory paradigm eliminates the need for complex interprocess communication mechanisms like pipes or sockets, streamlining data sharing.

Memory Allocation and Size:

The shared memory section's size is predetermined based on the defined constants for maximum strings and string length.

In this case, it occupies 32,768 bytes (4 bytes for iTotal and 128 bytes for each of the 256 string pointers).

Alternative Approach:

For scenarios demanding dynamically allocated shared memory, file mapping objects offer greater flexibility.

These objects allow processes to create shared memory regions of varying sizes at runtime, providing adaptability.

Key Advantages:

- Real-time Updates: Changes made in one STRPROG instance are immediately reflected in others due to the shared memory model.
- Efficiency: Shared memory surpasses traditional interprocess communication mechanisms in terms of speed and resource utilization.
- Simplicity: Implementation using #pragma directives and linker configuration is relatively straightforward.

MESSAGE HANDLING IN DLLS:

No Direct Message Queue: DLLs don't possess their own message queues for receiving messages.

Message Retrieval on Behalf of Caller: DLLs can utilize GetMessage and PeekMessage to retrieve messages from the calling program's queue, acting as its proxy.

General Rule for Windows Functions: Most Windows functions invoked by a DLL function on behalf of the calling program.

Resource Loading:

Flexibility in Resource Source: DLLs can load resources (icons, strings, bitmaps) from either their own file or the calling program's file.

Instance Handle Determines Source:

Using the DLL's instance handle loads resources from the DLL file.

Using the calling program's instance handle loads resources from the program's file.

Window Management:

Tricky Handling of Window Classes and Windows: Registering window classes and creating windows within DLLs requires careful attention to instance handles.

Message Queue Considerations: While a DLL can use its own instance handle for window creation, resulting messages still flow through the calling program's message queue.

Recommended Approach: It's generally advisable to employ the calling program's instance handle for window-related actions within DLLs.

Modal Dialog Boxes in DLLs:

Independent Message Handling: Modal dialog boxes manage their own message loops, operating outside the main message loop of the calling program.

DLL-Initiated Creation: DLLs can seamlessly create modal dialog boxes using DialogBox.

Instance Handle and Parent Window:

The DLL's instance handle can be used for dialog creation.

The hwndParent argument can be set to NULL, indicating no specific parent window.

Key Takeaways:

- DLLs function as extensions of calling programs, often working on their behalf.
- Understanding message handling, resource loading, and window management nuances is crucial for effective DLL development.
- Modal dialog boxes offer a straightforward mechanism for user interaction within DLLs.

Dynamic Linking Flexibility:

In certain scenarios, you might want to link a program with a library module dynamically while the program is running, rather than relying on the default dynamic linking during program loading. This approach allows for flexibility when the library name or functionality is determined at runtime.

For example, consider the typical way of calling the Rectangle function from the GDI32 library:

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom);
```

Conventional Approach: Calling Rectangle(hdc, xLeft, yTop, xRight, yBottom) directly relies on linking with GDI32.LIB during compilation.

Here's an alternative method using runtime dynamic linking. First, define a function type for Rectangle using typedef:

```
typedef BOOL (WINAPI * PFNRECT) (HDC, int, int, int, int);
```

Next, declare variables for the library handle and the function pointer:

```
HANDLE hLibrary;
PFNRECT pfnRectangle;
```

Now, set hLibrary to the handle of the library and pfnRectangle to the address of the Rectangle function:

```
hLibrary = LoadLibrary(TEXT("GDI32.DLL"));
pfnRectangle = (PFNRECT) GetProcAddress(hLibrary, TEXT("Rectangle"));
```

The LoadLibrary function returns NULL if the library file cannot be found or if an error occurs during loading. After calling the function through the function pointer, free the library:

```
pfnRectangle(hdc, xLeft, yTop, xRight, yBottom);
FreeLibrary(hLibrary);
```

Runtime Linking Steps in short:

- Typedef for Function Pointer: Define PFNRECT to represent a function matching Rectangle's signature.
- Variable Declaration: Declare hLibrary to hold the library handle and pfnRectangle for the function pointer.
- Loading the Library: Use LoadLibrary("GDI32.DLL") to load the library, obtaining its handle.
- Retrieving Function Address: Employ GetProcAddress(hLibrary, "Rectangle") to extract Rectangle's address, storing it in pfnRectangle.
- Function Call: Invoke pfnRectangle(hdc, xLeft, yTop, xRight, yBottom) to execute Rectangle.
- Releasing the Library: Call FreeLibrary(hLibrary) to unload the library when no longer needed.

Key Applications:

Unknown Library Names at Runtime: Crucial for scenarios where specific library requirements aren't determined until the program executes.

Conditional Functionality: Enables loading and using libraries based on user preferences or runtime conditions.

Plugin-Based Architectures: Fosters modular designs where components can be loaded and unloaded as needed.

Internal Mechanisms:

Reference Counting:

Windows employs reference counting to keep track of the usage of library modules. When a program loads a library using LoadLibrary, the reference count for that library is incremented. Similarly, when a program or instance using the library terminates or calls FreeLibrary, the reference count is decremented.

- LoadLibrary Increments the Count: When a program loads a library, the reference count is increased. This ensures that Windows knows how many instances are currently utilizing the library.
- FreeLibrary Decrements the Count: When a program or instance finishes using the library, the reference count is decremented. This occurs when FreeLibrary is called.
- Program Termination Decreases the Count: When a program terminates, whether gracefully or abruptly, the reference count is also decremented. This ensures that the reference count accurately reflects the number of active users of the library.

Memory Management:

- Windows manages the memory associated with loaded libraries based on reference counting. The critical point is when the reference count reaches zero. At this stage, Windows can safely unload the library from memory without affecting any active users.
- When Reference Count Reaches 0: Once the reference count drops to zero, Windows recognizes that the library is no longer in use. Consequently, it can unload the library from memory. This process conserves system resources by releasing the memory previously allocated to the library.

Benefits:

Enhanced Flexibility:

Runtime dynamic linking provides enhanced flexibility by allowing programs to link with libraries during execution rather than at compile time. This flexibility is particularly valuable when the specifics of library usage are determined dynamically during the program's runtime.

Reduced Dependencies:

Programs can potentially avoid linking with unnecessary libraries at compile time. This helps minimize the program's footprint by only including the necessary libraries when needed, reducing dependencies and improving efficiency.

Modular Design:

Runtime dynamic linking promotes a more modular architecture. Programs can dynamically incorporate or remove components (libraries) based on runtime conditions. This modular design facilitates flexibility in adapting to changing requirements and promotes efficient component management.

RESOURCE-ONLY LIBRARIES (ROLS):

Purpose:

ROLs serve as containers for resources that can be shared across multiple Windows programs, fostering code reusability and efficient resource management.

They don't contain functional code or exported functions, focusing exclusively on resource storage and access.

Common Use Cases:

Bitmaps: Storing different sets of bitmaps for various display resolutions, enhancing adaptability.

Icons, Dialogs, Strings, Menus: Centralizing resources used by multiple programs, promoting consistency and updates.

Language-Specific Resources: Encapsulating localized content for different languages, streamlining language support.

Creation Process:

Resource Script (.RC): Define the resources to be included using a resource script file.

Compilation: Compile the resource script using a resource compiler (e.g., RC.EXE) to generate a resource file (.RES).

Linking: Link the resource file with a linker to create the ROL (.DLL).

Example (BITLIB.DLL):

Contains nine bitmaps (BITMAP1.BMP to BITMAP9.BMP).

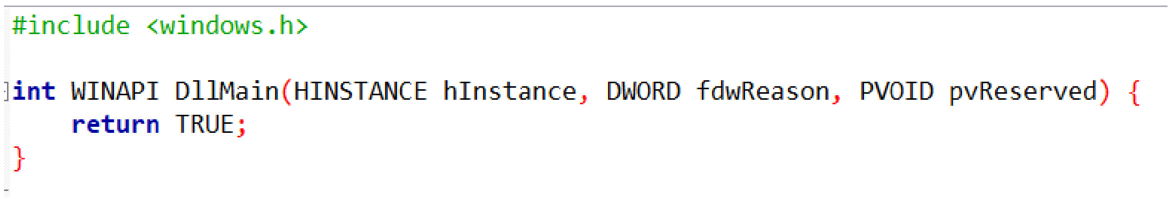
BITLIB.RC lists the bitmaps and assigns them numeric IDs (1-9).

Programs access resources using functions like LoadBitmap and LoadIcon, specifying the ROL name and resource ID.

```
1 BITMAP DISCARDABLE "bitmap1.bmp"
2 BITMAP DISCARDABLE "bitmap2.bmp"
...
9 BITMAP DISCARDABLE "bitmap9.bmp"
```

DLL Code (BITLIB.C):

The DllMain function is included as the entry point for the DLL.



```c
#include <windows.h>

int WINAPI DllMain(HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved) {
    return TRUE;
}
```

Building the Project:

Build the BITLIB project independently from the SHOWBIT project, as resource-only libraries don't require the creation of LIB files since they don't export functions.

Using Resource-Only Library in SHOWBIT:

Project Setup:

- Create the SHOWBIT project in a separate workspace.
- Don't set BITLIB as a dependency of SHOWBIT to avoid the creation of BITLIB.LIB

SHOWBIT Code (SHOWBIT.C):

- Read the bitmap resources from BITLIB and display them in the client area.
- Allow cycling through bitmaps by pressing a key on the keyboard.

Benefits of Resource-Only Libraries:

- Modular Resource Management: Resource-only libraries promote modular resource management, allowing developers to organize and distribute resources independently of executable code.
- Efficient Resource Customization: Different sets of resources, customized for specific scenarios (e.g., display resolutions), can be efficiently managed within separate resource-only libraries.
- Simplified Distribution: Resources can be distributed as separate libraries, making it easier to update or add new resources without modifying the executable code.

Resource-only libraries are a powerful tool for managing and organizing resources in a modular and efficient manner, enhancing the flexibility and maintainability of Windows applications.

SHOWBIT PROGRAM

The code provided is an implementation of the SHOWBIT program, which displays bitmaps using the BITLIB dynamic-link library. Let's go through the functionality of the code in more detail:

The code starts with the inclusion of necessary header files and the definition of the WndProc function, which is the window procedure for the program.

The WinMain function is the entry point of the program. It registers a window class with the RegisterClass function and creates the main window using the CreateWindow function. The window class has a white background and handles horizontal and vertical window resizing.

The program enters a message loop using the GetMessage function. Inside the message loop, it translates and dispatches messages using the TranslateMessage and DispatchMessage functions.

The WndProc function handles various messages to perform the program's functionality. Here are some of the messages it handles:

- WM_CREATE: This message is sent when the window is created. In response to this message, the function loads the BITLIB.DLL using the LoadLibrary function.
- WM_CHAR: This message is sent when a character is typed. The function increments the variable iCurrent and triggers a repaint of the window by calling InvalidateRect.
- WM_PAINT: This message is sent when the window's client area needs to be repainted. In response to this message, the function creates a device context (hdc) using BeginPaint and loads the bitmap specified by iCurrent using the LoadBitmap function from the BITLIB.DLL. It then calls the DrawBitmap function to draw the loaded bitmap on the device context. Finally, it cleans up by calling EndPaint.
- WM_DESTROY: This message is sent when the window is being destroyed. In response to this message, the function frees the loaded library using FreeLibrary and posts a quit message to exit the message loop.

The DrawBitmap function is a helper function that draws a bitmap on a specified device context (hdc) at the given coordinates (xStart, yStart). It creates a compatible device context (hMemDC) and selects the bitmap into it. It then uses BitBlt to copy the bitmap from the memory device context to the specified device context. Finally, it cleans up by deleting the memory device context.

- The function starts by creating a compatible device context (hMemDC) using the CreateCompatibleDC function. A compatible device context is a memory-based device context that matches the attributes of the target device context (hdc) it will be drawn on.
- The SelectObject function is then used to select the bitmap (hBitmap) into the memory device context (hMemDC). This allows subsequent drawing operations to be performed on the selected bitmap.
- The GetObject function retrieves information about the bitmap (hBitmap) and stores it in the BITMAP structure (bm). This includes the width, height, and other attributes of the bitmap.
- The pt structure is assigned the dimensions of the bitmap (bm.bmWidth and bm.bmHeight). These dimensions will be used as the width and height parameters for the BitBlt function, specifying the size of the area to be copied.
- The BitBlt function is then called to perform the actual bitmap drawing. It copies the bitmap from the memory device context (hMemDC) to the specified device context (hdc), starting at the coordinates (xStart, yStart) and using the dimensions specified in the pt structure. The SRCCOPY parameter indicates that the source pixels should be directly copied to the destination without any blending or composition.
- Finally, the memory device context (hMemDC) is cleaned up by calling the DeleteDC function. This releases the resources associated with the memory device context.

The provided code demonstrates a simple program that loads and displays bitmaps using a dynamic-link library (BITLIB.DLL). It responds to key presses to cycle through different bitmaps and updates the display accordingly.

The code snippet mentions that if the BITLIB.DLL is not found in the same directory as SHOWBIT.EXE, Windows will search for it as discussed earlier in the chapter. This refers to the standard search strategy used by Windows to locate DLL files, which includes searching in the application's directory, the system directory, and the directories listed in the system's PATH environment variable.

The LoadBitmap function is used to obtain a handle to a bitmap from the BITLIB.DLL. It takes the library handle (hLibrary) and the number of the bitmap (iCurrent) as parameters.

The MAKEINTRESOURCE macro is used to convert the numeric resource ID into a resource name that can be recognized by the LoadBitmap function.

The LoadBitmap function returns an error if the specified bitmap is not valid or if there is not enough memory to load the bitmap. This indicates a potential failure in loading the desired bitmap.

During the processing of the WM_DESTROY message, the FreeLibrary function is called to release the loaded library (BITLIB.DLL). This frees the resources associated with the library and decrements its reference count.

The code mentions that when the last instance of SHOWBIT terminates, the reference count of BITLIB.DLL drops to 0 and the memory it occupies is freed. This implies that the BITLIB.DLL library is designed to be shared across multiple instances of the SHOWBIT program, and the library's resources will be freed only when there are no more instances actively using it.

The code suggests that SHOWBIT could be used as a simple "clip art" program. It loads precreated bitmaps from the library and displays them in the program's window. These bitmaps could potentially be copied to the clipboard for use by other programs, although the code provided does not contain clipboard-related functionality.

Overall, the code demonstrates a basic implementation of a program that loads and displays bitmaps from a DLL file. It provides a starting point for building a more comprehensive application that involves working with bitmaps and libraries.

End of chapter 21 DLL's… Let's go listen to some music in chapter 22