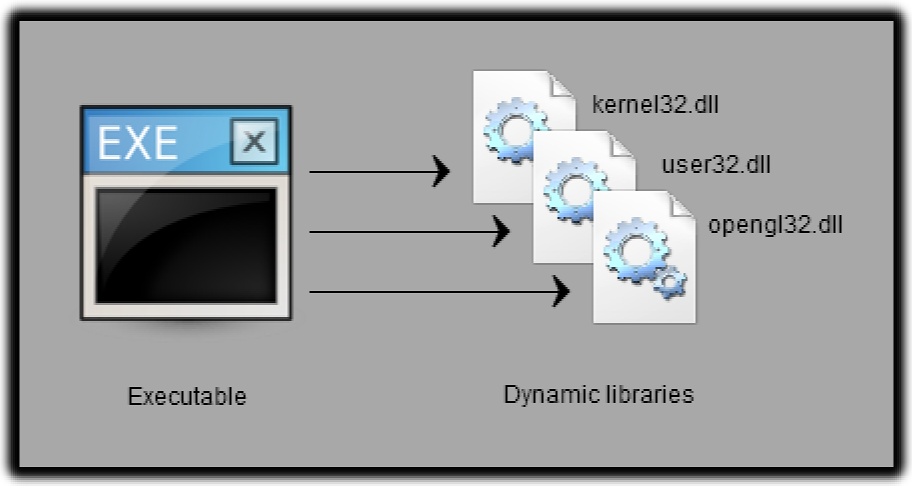
DLLS IN WINDOWS 🧩

1. What a DLL Really Is

A **Dynamic-Link Library (DLL)** is a file that contains **functions and resources** that are used by other programs.
It is **not** a standalone program and cannot run by itself.

DLLs exist to **share code and data** between multiple executables.



2. How DLLs Differ from Executables

- **EXE** → starts execution, owns the process.
- **DLL** → loaded into an existing process and provides services.

A DLL has:

- No WinMain
- No message loop
- No independent lifetime

It lives **only as long as the program using it**.

3. Dynamic Linking (What "Dynamic" Actually Means)

Dynamic linking happens **at runtime**, not at compile time.

Flow:

1. Program starts.
2. Windows loads required DLLs.
3. Function calls inside the program are resolved to DLL code.
4. Execution continues as if the code were local.

Result:

- One copy of code
- Used by many processes
- Loaded only when needed

4. DLLs Are the Fabric of Windows 🧠

Windows itself is built from DLLs.

Examples:

- KERNEL32.DLL → processes, memory, files
- USER32.DLL → windows, messages, input
- GDI32.DLL → graphics and drawing
- Device drivers
- Font engines

Writing a DLL is effectively **extending Windows**, not just writing helper code.

5. DLL File Extensions

- .DLL → standard and auto-loaded
- Other extensions → allowed, but must be loaded manually

Manual loading uses:

- LoadLibrary
- LoadLibraryEx

Rule: If Windows doesn't recognize the extension, **you must load it yourself**.

6. Why DLLs Exist (Real Advantages)

Code Reuse

- One implementation
- Many programs
- Less disk usage
- Less memory usage

Independent Updates

- Fix the DLL
- No need to recompile dependent programs
- As long as the interface stays stable

Shared Resources

- Fonts
- Icons
- Images
- Tables
- Data blobs

7. DLLs in Large Applications

DLLs shine when applications are **split into many programs**.

Example:

- Accounting software
- Multiple tools
- Shared calculations
- Shared data logic

Instead of duplicating code:

- Put common logic in ACCOUNT.DLL
- All tools use the same implementation

8. DLLs as Products

DLLs can be **commercial products**.

Example:

- GDI3.DLL with 3D drawing routines
- Licensed to many graphics applications
- Users install one copy
- All programs benefit

This is how **real software ecosystems** are built.

TYPES OF “LIBRARIES” IN WINDOWS ⚙️

The word *library* is overloaded. These are **not the same thing**.

---

9. Dynamic-Link Libraries (DLLs)

- Loaded at runtime
- Shared between processes
- Provide executable code and resources
- Central to Windows design

10. Object Libraries (.LIB)

- Statically linked
- Code becomes part of the EXE
- No runtime sharing
- Cannot be updated independently

Example:

- C runtime static libraries

Once linked, they are **part of the executable forever**.

11. Import Libraries (.LIB)

Import libraries are **not code containers**.

They are:

- Linker instructions
- Address maps
- Function name translators

Purpose:

- Tell the linker how to call into a DLL
- Enable clean dynamic linking at build time

They exist only to **bridge EXE ↔ DLL**.

12. When Each Is Used

- **DLL** → runtime execution
- **Object library** → compile-time inclusion
- **Import library** → compile-time setup for runtime DLL usage

This separation is intentional and powerful.

DLL LOADING MECHANISM 🔍

13. How Windows Finds a DLL

Windows searches in this order:

1. Program directory
2. Current working directory
3. System directory
4. Windows directory
5. Directories in PATH

Understanding this order is **critical** for debugging DLL issues.

PROJECT AND WORKSPACE STRUCTURE 🧱

14. Workspace vs Project

- **Workspace** → container for related projects
- **Project** → builds one thing (DLL or EXE)

Large systems always use **multiple projects**.

15. DLL Project Components

Header Files

- Define exported functions
- Define shared structures
- Act as the DLL's public contract

Source Files

- Actual implementation
- Internal logic
- Hidden details

Rule:

Headers describe *what* the DLL offers,
source files implement *how* it works.

MENTAL MODELS THAT MATTER 🧠

- DLL = shared toolbox
- EXE = worker using tools
- Import library = instruction manual
- Runtime linking = flexible and efficient
- Static linking = fixed and heavy

WHAT'S COMING NEXT 🚀

- Reading real DLL headers
- Understanding exported functions
- Seeing EXE ↔ DLL interaction
- Breaking and fixing linkage mistakes

This is **real Windows architecture**, not theory.

EDRLIB.H and EDRLIB.C

```
#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
```

```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);
}
```

EXPORTING FUNCTIONS FROM A DLL

1. #pragma once

#pragma once ensures a header file is included **only once** during compilation.

Why it exists:

- Prevents duplicate definitions
- Replaces classic #ifndef / #define include guards
- Simpler and less error-prone

Important:

- Not part of the C standard
- Fully supported by MSVC and most modern compilers
- Safe to use in Windows projects

Mental model:

One header, one inclusion — no accidents.

EXPORTING AND IMPORTING SYMBOLS 🔗

2. __declspec(dllexport) and __declspec(dllimport)

These are **Microsoft-specific** attributes that control **symbol visibility** across module boundaries.

- __declspec(dllexport)
  - ✓ Used when **building the DLL**
  - ✓ Marks functions or data as **exported**
  - ✓ Makes them visible to other modules
- __declspec(dllimport)
  - ✓ Used when **using the DLL**
  - ✓ Tells the compiler the symbol lives elsewhere
  - ✓ Enables efficient calling through the import table

Rule: Same header, two meanings — depends on who is compiling.

3. The EDRLIB_EXPORT Macro (Correct Pattern)

The macro switches behavior based on a build flag.

Logic:

- When building the DLL → export
- When consuming the DLL → import

This avoids:

- Duplicate headers
- Manual edits
- Mistakes across projects

Mental model:

One header rules both sides.

FUNCTION DECLARATIONS (ANSI vs UNICODE) 🧠

4. EdrCenterTextA and EdrCenterTextW

Windows uses a **dual-function pattern**:

- A → ANSI (8-bit)
- W → Wide / Unicode (UTF-16)

Why both exist:

- Backward compatibility
- Explicit encoding control
- Performance predictability

Each function:

- Receives a device context (HDC)
- Receives a rectangle (RECT)
- Receives a string
- Draws text centered in that rectangle

5. The EdrCenterText Macro

This macro selects the correct function automatically.

- UNICODE defined → EdrCenterTextW
- UNICODE not defined → EdrCenterTextA

Why this matters:

- Call sites stay clean
- Encoding choice becomes a **build decision**
- Same source works in both worlds

This is **classic WinAPI design**.

extern "C" — WHY IT MATTERS ⚠️

6. Preventing Name Mangling

When compiling with C++:

- Function names get mangled
- Symbol names change
- DLL exports break silently

extern "C":

- Forces C linkage
- Keeps symbol names predictable
- Allows C and C++ code to interoperate

Rule:

DLL boundaries hate name mangling.

DLL IMPLEMENTATION (EDRLIB.C) 🧱

7. DllMain — The Real Entry Point

DLLs don’t have WinMain.

They have:

```
BOOL APIENTRY DllMain(...)
```

This function is called by Windows when:

- DLL is loaded
- DLL is unloaded
- Threads are created
- Threads are destroyed

In this DLL:

- DllMain returns TRUE
- No initialization logic
- Safe and minimal

That’s intentional.

IMPLEMENTING THE EXPORTED FUNCTIONS 🧠

8. EdrCenterTextA

Steps:

1. Compute string length (lstrlenA)
2. Measure text size (GetTextExtentPoint32A)
3. Compute centered position
4. Draw text (TextOutA)

Key idea:

Measure first, draw later — never guess.

9. EdrCenterTextW

Same logic as the ANSI version, but:

- Uses wide strings
- Uses W versions of APIs

This guarantees:

- Correct Unicode handling
- No encoding bugs
- Predictable output

BUILD OUTPUTS 🏗️

10. What the Build Produces

- EDRLIB.DLL → the actual executable code
- EDRLIB.LIB → the import library

Important distinction:

- .DLL → loaded at runtime
- .LIB → used by the linker

Mental model:

LIB = instructions
DLL = execution

TEXT ENCODING STRATEGY 🧬

11. Why ANSI + Unicode Still Exists

Windows evolved, not rebooted.

DLL design must:

- Support legacy code
- Support modern Unicode
- Avoid forcing decisions on users

The macro-based switch:

- Centralizes encoding choice
- Keeps APIs clean
- Avoids duplicated call logic

DllMain — LIFE CYCLE EVENTS ⚙️

12. DLL_PROCESS_ATTACH

Triggered when:

- A process loads the DLL

Typical tasks:

- Initialize global data
- Allocate resources
- Prepare internal state

Rules:

- Keep it fast
- No heavy work
- No blocking calls

13. DLL_PROCESS_DETACH

Triggered when:

- Process unloads the DLL

Responsibilities:

- Free memory
- Close handles
- Release resources

Failure here causes:

- Memory leaks
- Handle leaks
- Silent system instability

14. Thread Notifications

- DLL_THREAD_ATTACH
- DLL_THREAD_DETACH

These fire for **every thread** in the process.

Important warning:

- Avoid complex logic
- Avoid messaging APIs
- Avoid synchronization traps

Many real bugs live here.

CRITICAL RULES FOR DllMain ⚠️

15. What NOT to Do

Never:

- Call LoadLibrary
- Call FreeLibrary
- Create threads
- Perform blocking I/O
- Use complex synchronization

Reason:

Loader lock.

Violating this leads to:

- Deadlocks
- Random crashes
- Impossible-to-debug failures

ADDITIONAL PRACTICAL NOTES 🧠

16. hInstance

- Identifies the DLL module
- Needed for resource loading
- Often stored globally

17. Multiple Processes

- Each process gets its own instance
- Global variables are **per process**
- DLL code is shared, data is not

18. Thread Safety Reality

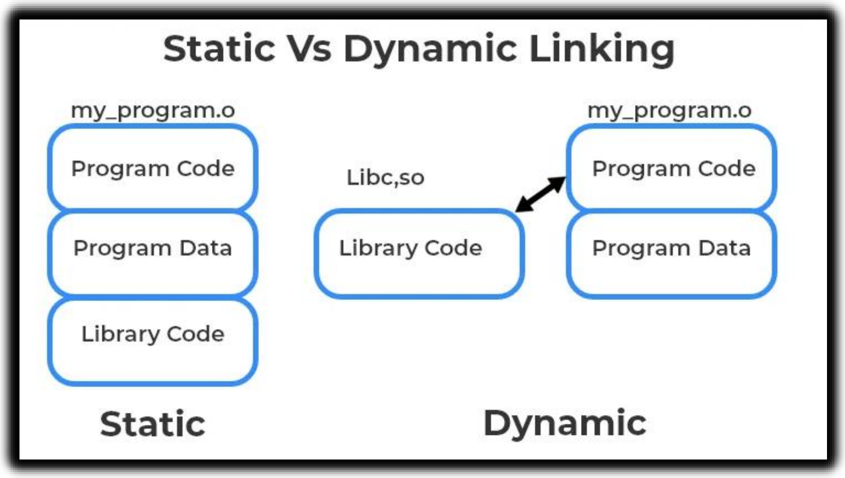
- DllMain can be called concurrently
- Shared state must be protected
- Minimal logic reduces risk

FINAL MENTAL MODEL ⚙️

- Headers define the **contract**
- Macros control visibility
- Import libraries guide linking
- DLLs execute at runtime
- DllMain is **not** a playground
- Unicode is the default reality
- Simplicity in loaders = stability

That's what I call **real DLL engineering**.

EDRTEST PROGRAM



EDRTEST PROGRAM (USING A DLL IN PRACTICE) 🧪

1. What EDRTEST Is

EDRTEST.EXE is a **normal Win32 GUI program** whose only purpose is to **prove DLL usage works**.

It does not:

- Implement text-centering logic
- Know how drawing is done
- Care about encoding details

It **delegates** that responsibility to EDRLIB.DLL.

This is correct architecture.

2. Structure of the Program

EDRTEST follows the **standard Win32 skeleton**:

- WinMain
- Window class registration
- CreateWindow
- Message loop
- WndProc

Nothing special here — and that's intentional.

The focus is on **integration**, not novelty.

3. Including the DLL Interface

The program includes:

```
#include "edrlib.h"
```

This header:

- Exposes exported functions
- Hides implementation details
- Acts as a contract

Important:

EDRTEST never sees the DLL's source code.

That separation is the entire point.

4. Where the DLL Is Used (WM_PAINT)

The DLL is used **only** during painting.

Flow inside WM_PAINT:

1. Call BeginPaint → get HDC
2. Call GetClientRect → get drawable area
3. Call EdrCenterText
4. Call EndPaint

The DLL:

- Measures text
- Calculates centering
- Draws the output

The EXE:

- Just asks for results

This is **clean responsibility separation.**

5. Why This Design Matters

Benefits shown clearly:

- **Reusability** → any program can reuse EDRLIB.DLL
- **Replaceability** → DLL can change without recompiling EXE
- **Testability** → EXE tests the DLL in isolation
- **Simplicity** → EXE code stays small

This is how large Windows apps are structured.

6. Message Loop and Lifetime

The message loop:

- Uses GetMessage
- Dispatches messages
- Ends on WM_QUIT

The DLL:

- Is loaded automatically at startup
- Remains loaded for the process lifetime
- Is unloaded at program exit

The EXE never manually loads or unloads it.

7. Expected Output

When run:

- A window appears
- Title: *DLL Demonstration Program*
- Text appears centered
- Text rendering is done entirely by the DLL

If the DLL fails:

- Program may fail to start
- Or crash on call

This shows how **critical DLL correctness is**.

SHARED MEMORY IN DLLs ⚠

(Breaking Isolation on Purpose)

8. Default Reality: DLLs Are Isolated

Normally:

- Each process has its **own data**
- Global variables are **per process**
- DLL code is shared
- DLL memory is not

This is **safe by default.**

9. When Shared Memory Is Needed

Sometimes you want:

- Multiple programs
- Multiple instances
- Shared state
- Central coordination

Examples:

- Caches
- Logs
- String registries
- License tracking
- IPC helpers

This requires **explicit shared memory**.

STRLIB.DLL — A SHARED STRING STORE

10. Purpose of STRLIB

STRLIB:

- Stores up to 256 strings
- Capitalizes them
- Sorts them
- Shares them across processes

This is **not normal DLL behavior —** it's deliberate.

STRLIB.H (Header File)

```
// 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
}
```

11. Core Functions Provided

AddString

- Adds a string to shared memory
- Capitalizes internally
- Allows duplicates
- Fails if:
  - ✓ Empty string
  - ✓ Allocation failure
  - ✓ 256-string limit reached

Returns:

- TRUE → success
- FALSE → failure

DeleteString

- Removes the first matching string
- Does nothing if not found
- Safe operation

Returns:

- TRUE → deleted
- FALSE → failed or not found

GetStrings (Callback-Based)

This is the important one.

- STRLIB does **not** return a list
- Instead, it **calls back into the EXE**

Flow:

1. Program passes a callback function
2. DLL iterates its shared strings
3. DLL calls the callback once per string
4. Enumeration stops when callback returns FALSE

Why this design?

- No shared ownership
- No buffer guessing
- No memory ownership confusion

This is **professional API design**.

12. Unicode Strategy

Internally:

- STRLIB stores everything in **Unicode**

Externally:

- Provides A and W APIs
- Converts when needed

Result:

- One internal representation
- Multiple consumer types
- Clean separation

This avoids data corruption and encoding bugs.

STRPROG.EXE — USING SHARED MEMORY 🧪

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;
}
```

13. What STRPROG Does

STRPROG:

- Has menus (Enter / Delete)
- Shows strings in its client area
- Uses STRLIB for all storage

It owns:

- UI
- Input
- Display

STRLIB owns:

- Data
- Sorting
- Storage
- Sharing

Correct division.

14. Callback in Practice

STRPROG defines:

- A callback function
- Receives strings one-by-one
- Displays them

STRLIB:

- Knows nothing about UI
- Knows nothing about windows
- Only knows how to enumerate

This keeps the DLL **generic and reusable**.

HOW SHARED MEMORY IS DONE 🔐

15. Memory-Mapped Files

Shared memory in Windows uses:

- File mapping objects
- Named mappings
- Same name → same memory

All processes:

- Map the same memory region
- See the same data
- Must synchronize access

This is powerful — and dangerous.

16. Synchronization Is Mandatory

Shared memory requires:

- Mutexes
- Critical sections
- Careful design

Without synchronization:

- Data corruption
- Random crashes
- Undefined behavior

Rule:

Shared memory without locks is a time bomb.

FINAL MENTAL MODELS 🧠

17. EDRTEST Lesson

- EXE = UI + coordination
- DLL = logic + capability
- WM_PAINT = integration point
- Headers = contracts

18. STRLIB Lesson

- DLLs can share memory — but only intentionally
- Callbacks avoid ownership problems
- Unicode-first design is mandatory
- Synchronization is non-negotiable

19. Big Picture

- DLLs scale software
- Shared memory scales systems
- Bad DLLs crash everything
- Good DLLs disappear into reliability

This is **real Windows engineering**, not tutorial fluff.
That's why the former chapters were a necessary foundation.

STRLIB LIBRARY PROGRAM

1. Purpose and Structure

STRLIB is a DLL that provides **string management through shared memory**.
Multiple processes can access and modify the same string list.

It consists of:

- **STRLIB.H** → interface and contracts
- **STRLIB.C** → implementation and shared memory logic

2. STRLIB.H (Header Design)

1. Constants
    - Maximum number of strings
    - Maximum length per string
      These define the fixed size of the shared memory.
2. Callback Function Type
    - Declares a callback used for string retrieval
    - The calling program (e.g. STRPROG) must implement it
    - STRLIB does not decide *what to do* with strings — it only delivers them
3. Design Intent
    - STRLIB manages data
    - The caller manages presentation and behavior

This separation is deliberate.

3. STRLIB.C (Implementation Core)

1. Required Headers
    - windows.h → shared memory, DLL behavior
    - wchar.h → Unicode string handling
2. Shared Memory Section
    - Created using #pragma data_seg("shared")
    - This section is shared across all processes loading the DLL
3. Shared Variables
    a) **iTotal**
    - Tracks how many strings are stored
    - Updated on add/delete operations

    b) **szStrings**
    - 2D array of strings
    - Capacity: 256 strings
    - Max length: 63 characters
    - Stored internally as Unicode
4. Linker Configuration
    - /SECTION:shared,RWS
    - Ensures the section is:
        - ✓ Readable
        - ✓ Writable
        - ✓ Shared across processes

Without this, the memory would not actually be shared.

EXPORTED FUNCTIONS 🛠️

1. AddString (A/W)

1. Behavior
    - Accepts an input string
    - Converts it to uppercase
    - Inserts it into the array in alphabetical order
2. Duplicate Handling
    - Duplicates are allowed
    - No uniqueness enforcement
3. Result
    - Returns success or failure
    - Failure usually means capacity limits

2. DeleteString (A/W)

1. Behavior
    - Searches for the first matching string
    - Removes only the first occurrence
2. Design Choice
    - Predictable deletion
    - No mass removal side effects
3. Result
    - Returns success or failure

3. GetStrings (A/W)

1. Core Mechanism
    - Iterates through shared memory
    - Calls a callback for each string
2. Control Flow
    - Stops when:
        - ✓ All strings are processed, or
        - ✓ The callback returns FALSE
3. Return Value
    - Number of strings actually processed

STRLIB never paints, prints, or logs. It **enumerates only**.

SHARED MEMORY MODEL 🧠

1. Why Shared Memory

- Windows isolates process memory by default
- Shared memory allows:
    - ✓ Zero-copy data access
    - ✓ Immediate visibility across processes

2. How STRLIB Achieves It

1. Defines a shared section
2. Places data inside it
3. Marks it shared at link time
4. All processes loading STRLIB map the same memory

No pipes.
No sockets.
No IPC ceremony.

3. Memory Layout

- iTotal → 4 bytes
- szStrings → fixed-size buffer
- Total size ≈ 32 KB

Static, predictable, fast.

4. Alternative Approach

- File Mapping Objects
- Useful when:
  - ✓ Size must be dynamic
  - ✓ Data lifetime must outlive the DLL

STRLIB uses the **simpler, static model** intentionally.

UNICODE HANDLING

Internal Rule - All strings are stored as **Unicode.**

1. Dual API Design

- A functions → ANSI callers
- W functions → Unicode callers
- UNICODE macro selects automatically

2. Conversion Logic

1. ANSI caller
   - ✓ Convert ANSI → Unicode
   - ✓ Process internally
   - ✓ Convert back when returning data
2. Unicode caller
   - ✓ No conversion required

This guarantees consistency and compatibility.

CALLBACK DESIGN 🔁

1. Purpose

- STRLIB does not own presentation logic.
- The caller decides how strings are used.

2. Callback Flow

- Caller passes:
    - ✓ Callback function pointer
    - ✓ User-defined parameter
- STRLIB:
    - ✓ Calls callback for each string
    - ✓ Stops if callback returns FALSE

3. Advantage

- Flexible
- Extensible
- No coupling between DLL and UI

DLL EXPORTS AND LIFECYCLE ⚙️

1. Exporting Functions

- Export macro marks functions as public
- Allows external programs to link and call them

2. DllMain

- Called on load/unload
- STRLIB’s implementation:
  - ✓ Does nothing
  - ✓ Returns TRUE

This is acceptable because:

- Shared memory is static
- No per-process initialization required

STRPROG PROGRAM

1. Role

STRPROG is a **demonstration client** for STRLIB.

It shows:

- Shared memory in action
- Multi-instance synchronization
- DLL-based architecture

2. Program Structure

1. Registers a window class
2. Creates the main window
3. Enters a standard message loop
4. Delegates logic to WndProc

3. Message Handling

1. WM_CREATE
   - ✓ Registers a custom message for data changes
2. WM_COMMAND
   - ✓ Enter string → AddString
   - ✓ Delete string → DeleteString
   - ✓ Broadcasts update message
3. WM_PAINT
   - ✓ Calls GetStrings
   - ✓ Callback renders strings using TextOut
4. WM_DESTROY
   - ✓ Posts WM_QUIT

4. Dialog Boxes

- EnterDlg → input new string
- DeleteDlg → input string to remove
- Handled via dialog procedures
- Defined in resource file

MULTI-INSTANCE SYNCHRONIZATION 🔔

1. Shared Data

- All instances read/write the same memory

2. Notification Model

- Custom registered message
- Broadcast to all STRPROG windows
- Forces repaint and refresh

Shared memory gives the data.
Messages give the awareness.

DLL MESSAGE AND RESOURCE RULES 📦

1. Message Queues

- DLLs have no message queues
- They operate through the caller's queue

2. Resource Loading

- DLL instance handle → DLL resources
- App instance handle → App resources

3. Windows and Classes

- Windows created by DLLs still deliver messages to the caller's queue
- Best practice: Use caller's instance handle unless there's a strong reason not to

4. Modal Dialogs

- Modal dialogs run their own loop
- DLLs can safely create them
- Parent window can be NULL

KEY TAKEAWAYS 🧠

1. STRLIB
    - Shared memory DLL
    - Unicode-first design
    - Callback-based enumeration
2. STRPROG
    - Thin UI client
    - Demonstrates real inter-process sharing
3. Architecture Lessons
    - Shared memory beats IPC for speed
    - DLLs are extensions, not applications
    - Separation of data, logic, and UI is non-negotiable

This is **real WinAPI thinking**, not academic filler.

DYNAMIC LINKING FLEXIBILITY ⚡

1. Concept

Instead of linking a library at compile-time (static dynamic linking), you can **load a library at runtime**.

This is essential when:

- Library names are unknown at compile time
- Functionality depends on runtime conditions
- You're building **plugins or modular components**

2. Example: Rectangle in GDI32

Static linking:

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom); // Needs GDI32.LIB at compile-time
```

Runtime linking:

a) Typedef the function pointer:

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

b) Declare library handle and pointer:

```
HINSTANCE hLibrary;
PFNRECT pfnRectangle;
```

c) Load library and get function address:

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

d) Call through pointer:

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

e) Unload when done:

```
FreeLibrary(hLibrary);
```

3. Steps in Short

1. Define a **function pointer type**
2. Declare **library handle + pointer**
3. Load with **LoadLibrary**
4. Retrieve function with **GetProcAddress**
5. Call the function
6. Release library with **FreeLibrary**

4. Use Cases

- **Unknown library names at runtime**
- **Conditional functionality** (user settings, config files)
- **Plugin architectures** (load/unload modules on demand)

5. Internal Mechanisms

Reference counting:

- LoadLibrary increments count
- FreeLibrary decrements count
- Program exit decrements count automatically
- When count = 0 → memory is freed

Memory management:

- Library memory stays until reference count hits 0
- Ensures active users are unaffected

6. Benefits

- **Flexibility** → Decide which libraries to use at runtime
- **Reduced dependencies** → Only load what you need
- **Modular design** → Swap or update components dynamically

Resource-Only Libraries (ROLs)

1. Purpose

- Contain **only resources** (bitmaps, icons, dialogs, strings, menus)
- No exported functions
- Reusable across multiple programs

2. Common Uses

- **Bitmaps** → Multiple resolutions
- **Icons, menus, dialogs, strings** → Shared UI elements
- **Language packs** → Centralized localization

3. Creation

1. **Resource script (.RC)**
2. **Compile with RC.EXE** → .RES file
3. **Link .RES into DLL** → ROL

Example: BITLIB.DLL

- Nine bitmaps: BITMAP1.BMP → BITMAP9.BMP
- Defined in BITLIB.RC

```
BITMAP DISCARDABLE "bitmap1.bmp"
...
BITMAP DISCARDABLE "bitmap9.bmp"
```

DllMain is minimal:

```
#include <windows.h>
int WINAPI DllMain(HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)
{
    return TRUE;
}
```

Build independently, no .LIB file needed

4. Using ROLs in Programs

- Example: SHOWBIT
  - ✓ Loads BITLIB.DLL at runtime
  - ✓ Reads bitmaps with LoadBitmap(hLibrary, MAKEINTRESOURCE(id))
  - ✓ Draws them in window
  - ✓ Allows cycling with keyboard

Key points:

- InvalidateRect triggers repaint
- Bitmaps drawn via DrawBitmap:
  - ✓ Create compatible DC
  - ✓ Select bitmap
  - ✓ BitBlt to window DC
  - ✓ Cleanup memory DC
- Free library on WM_DESTROY

5. Benefits

- **Modular resource management**
- **Customizable resources** (display resolutions, language)
- **Simplified distribution** → Update resources without touching code
- **Shared usage** → Multiple instances of the program use the same DLL

SHOWBIT PROGRAM Overview 💻

Purpose: Load & display bitmaps from BITLIB.DLL

Flow:

- Register window, create main window
- Enter message loop
- Handle messages:
    1. WM_CREATE → Load BITLIB.DLL
    2. WM_CHAR → Cycle bitmaps
    3. WM_PAINT → Draw current bitmap
    4. WM_DESTROY → FreeLibrary + exit

DrawBitmap helper:

- ✓ Memory DC + BitBlt
- ✓ Copies bitmap to window
- ✓ Cleans up DC

Notes on ROL loading:

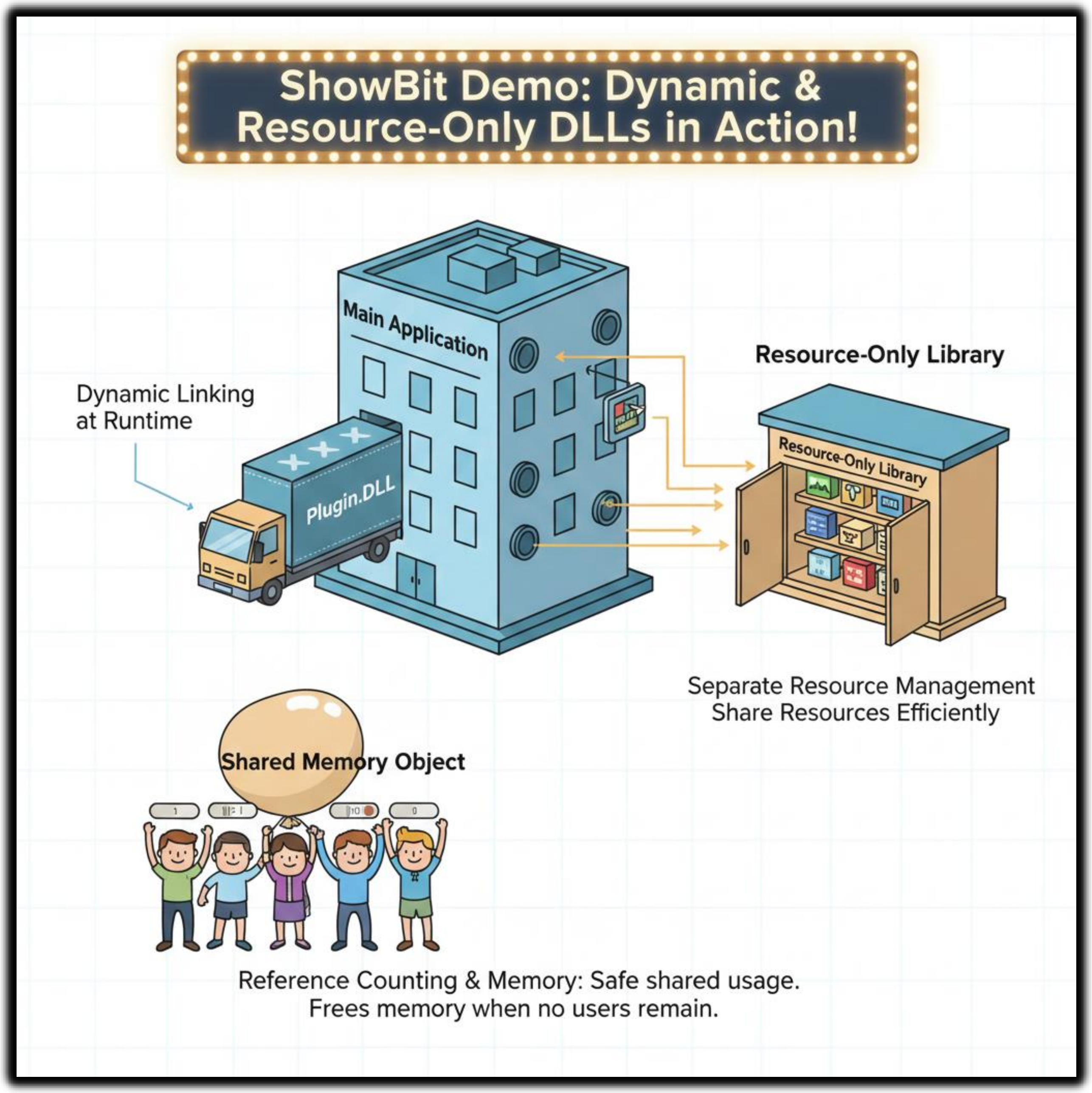
- If DLL not found → Windows searches PATH
- Resources are shared among multiple instances
- Last instance exit → BITLIB memory freed

Takeaways

1. **Dynamic linking at runtime**
    - Loads DLLs as needed
    - Supports plugin/modular architectures
2. **Resource-only libraries**
    - Separate resource management
    - Share resources efficiently
3. **Reference counting & memory**
    - Safe shared usage
    - Frees memory when no users remain
4. **SHOWBIT demo**
    - Practical example of dynamic and resource-only DLL usage

---

Next up is **chapter 22 – sounds**, so you can literally step into **multimedia, audio handling, and WinAPI sound messages.** 🎵


ShowBit Demo: Dynamic & Resource-Only DLLs in Action!
Main Application
Dynamic Linking at Runtime
Plugin.DLL
Resource-Only Library
Resource-Only Library
Separate Resource Management
Share Resources Efficiently
Shared Memory Object
Reference Counting & Memory: Safe shared usage.
Frees memory when no users remain.