CHAPTER 12 – CLIPBOARD

Purpose: The Windows clipboard allows data transfer between programs.

Simple Mechanism: It requires minimal overhead for both data insertion and retrieval.

Clipboard Viewer: Windows 98 and NT include programs to show the current clipboard content.

Common Clipboard Interactions: Many programs have Cut/Copy/Paste functionality for data transfer.

* Cut/Copy: Transfers data (text, bitmap, metafile) from program to clipboard.
* Paste: Transfers data from clipboard to program based on supported formats.

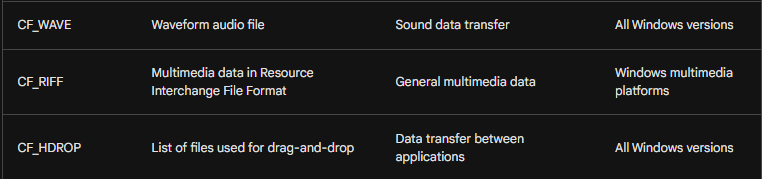
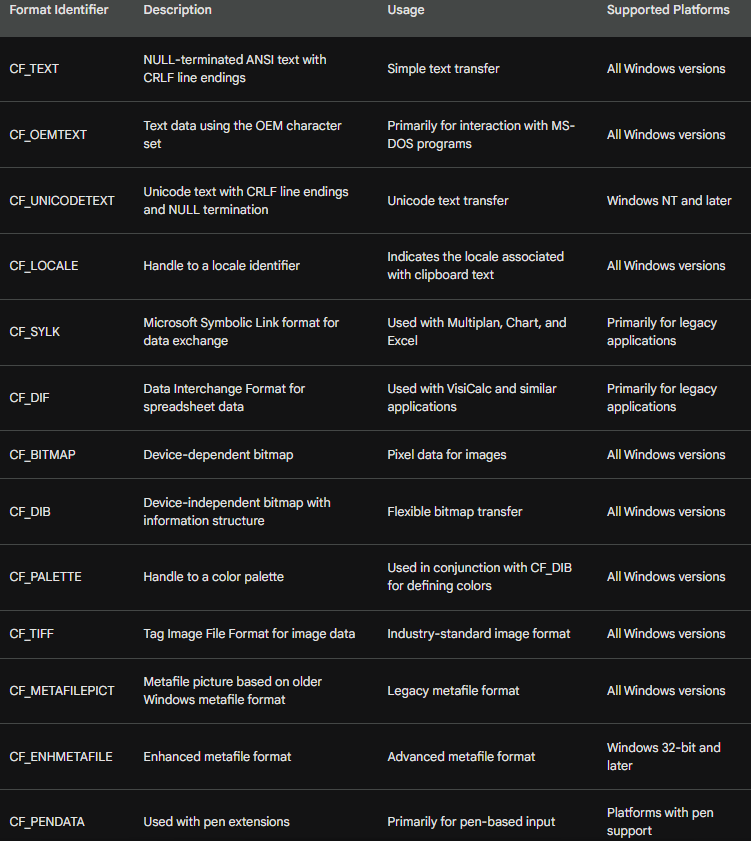
User Control: Programs should only access clipboard with explicit user instructions.

Data Persistence: Cut/Copy data remains in clipboard until next Cut/Copy.

Chapter Focus: Transferring text data to and from the clipboard.

Future Chapters: Clipboard usage with bitmaps (Chapters 14-16) and metafiles (Chapter 18).

Clipboard Data Formats: In-depth Breakdown



Memory Allocation for Clipboard

This section delves deeper into the memory allocation mechanisms used for clipboard operations in Windows, specifically focusing on the functions involved and their functionalities.

Global Memory Allocation:

When transferring data to the clipboard, programs need to allocate memory blocks using the Windows API, not the standard C malloc function.

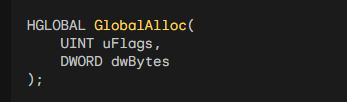
This is because the clipboard operates within the shared memory space accessible by various applications, requiring specific memory management mechanisms.

The GlobalAlloc function serves this purpose, taking two parameters:

* uiFlags: Optional flags specifying allocation behavior (e.g., fixed memory, zero initialization).
* dwSize: Size of the memory block to allocate in bytes.

The function returns a handle of type HGLOBAL, which represents the allocated memory block.

A NULL return value indicates insufficient memory for the requested size.



Important Flags:

GMEM\_FIXED: When used in uiFlags, the returned handle directly points to the allocated memory block, making it accessible as a pointer.

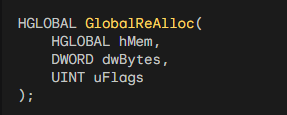
GMEM\_ZEROINIT: This flag initializes all bytes in the allocated memory to zero.

GPTR: A convenient flag combining GMEM\_FIXED and GMEM\_ZEROINIT for both direct access and zero initialization.

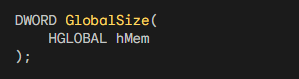
Additional Memory Management Functions:

GlobalReAlloc: This function resizes an existing memory block allocated with GlobalAlloc.

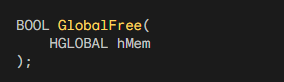
It takes the original handle, the new desired size, and optional flags like GMEM\_ZEROINIT for additional memory initialization.



GlobalSize: This function retrieves the size in bytes of a memory block allocated with GlobalAlloc.



GlobalFree: This function frees the memory associated with a given handle obtained from GlobalAlloc.



Key Points:

* Understanding these memory allocation functions is crucial for interacting effectively with the clipboard in Windows programs.
* These functions are part of the Windows API and coexist with the standard C library functions like malloc, but serve specific purposes for shared memory management within the operating system.
* Using the appropriate flags and functions ensures proper memory allocation, access, and release for clipboard operations.

Code Breakdown:

* GlobalAlloc: Allocates a memory block for clipboard data.
* GlobalReAlloc: Resizes an existing memory block allocated for clipboard data.
* GlobalSize: Retrieves the size of a memory block allocated for clipboard data.
* GlobalFree: Frees the memory block associated with clipboard data.

MOVABLE MEMORY FOR CLIPBOARD OPERATIONS

This section delves deeper into the concept of movable memory and its application in clipboard operations, particularly focusing on the 16-bit and 32-bit versions of Windows.

Early Windows and GMEM\_FIXED vs. GMEM\_MOVEABLE:

In 16-bit Windows, the GMEM\_FIXED flag was discouraged due to limitations in memory management.

Windows could not move fixed memory blocks in physical memory, potentially leading to performance issues.

The GMEM\_MOVEABLE flag was recommended for 16-bit applications to allow memory movement in virtual memory.

This approach enabled efficient memory management and avoided potential problems with fixed memory.

GMEM\_MOVEABLE in 32-bit Windows:

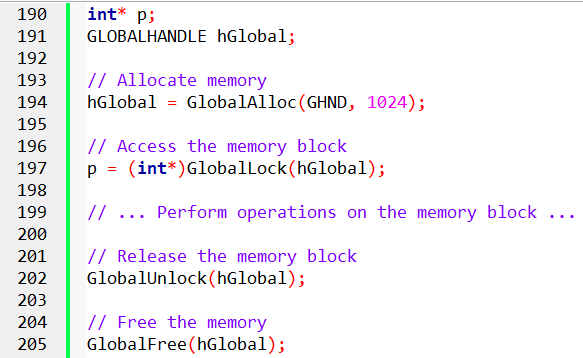
* With the introduction of 32-bit Windows, GMEM\_FIXED became more widely used as virtual addresses were employed.
* The operating system can now manage memory more efficiently with virtual address space, allowing for movement of fixed memory blocks without affecting program functionality.
* However, GMEM\_MOVEABLE still holds some value in specific scenarios.

Benefits of Movable Memory:

* Reduced virtual memory fragmentation: Frequent allocation and reallocation of memory can fragment the virtual memory space, potentially impacting performance.
* Efficient memory management: Movable memory allows Windows to optimize memory usage by relocating blocks without data copying, enhancing efficiency.

Using Movable Memory for Clipboard:

* When interacting with the clipboard, it is crucial to use movable memory due to potential sharing of memory blocks between applications.
* The GMEM\_MOVEABLE flag ensures that the clipboard memory can be accessed and manipulated by other programs without causing conflicts.
* Additionally, the GMEM\_SHARE flag should be used to explicitly allow sharing of the allocated memory block with other applications.



When accessing the memory block, calling GlobalLock translates the handle into a pointer and fixes the address in virtual memory while the block is locked.

Subsequently, calling GlobalUnlock allows Windows the flexibility to move the block in virtual memory.

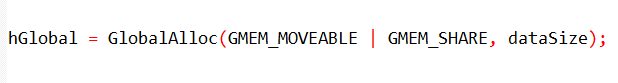
For optimal practice, it is recommended to lock and unlock the memory block within the scope of a single message.

To free the memory, use GlobalFree with the handle rather than the pointer. If you don't have access to the handle, you can retrieve it using GlobalHandle(p).

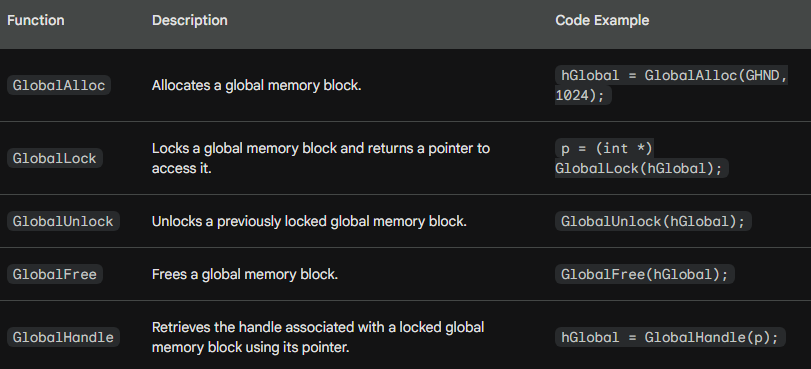
Locking a memory block multiple times increments a lock count, and each lock must have a corresponding unlock before the block is free to be moved.

In 32-bit Windows, the primary reason for allocating a movable block is to prevent virtual memory fragmentation. When dealing with the clipboard, using movable memory is also advisable.

When allocating memory for the clipboard, it's recommended to use GlobalAlloc with both GMEM\_MOVEABLE and GMEM\_SHARE flags:



Clipboard memory management functions:

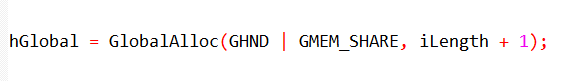


TEXT TRANSFER TO CLIPBOARD

Function Breakdown:

GlobalAlloc:

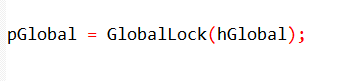
Allocate a memory block of sufficient size for the string.



* Allocates iLength + 1 bytes considering a potential null terminator.
* GHND: Flag for movable, zero-initialized memory.
* GMEM\_SHARE: Flag for sharing the memory block with other applications.

GlobalLock:

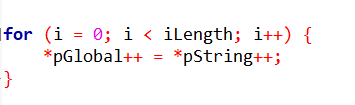
Obtain a pointer to the allocated memory block.



* Locks the memory block and returns a pointer to access its data.

String Copying:

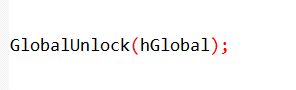
Copy the string content into the allocated memory block.



* Loops through the string, copying each character from pString to pGlobal and incrementing both pointers.

GlobalUnlock:

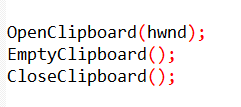
Release the lock on the memory block.



* Ensures other applications can access the memory block once finished copying.

Open/Close Clipboard:

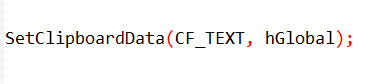
Open the clipboard, empty its content, and close it.



* OpenClipboard: Establishes access to the clipboard for the application.
* EmptyClipboard: Clears any existing content before adding new data.
* CloseClipboard: Releases access to the clipboard after data transfer.

SetClipboardData:

Transfer the memory block containing the string to the clipboard.



* Associates the hGlobal handle with the CF\_TEXT clipboard format.
* Transfers ownership of the memory block to the clipboard.

Important Points:

* Open and close the clipboard within the processing of a single message.
* Avoid keeping the clipboard open unnecessarily.
* Don't pass a locked memory block to the clipboard.
* After transferring data, treat the previously used global handle as invalid.
* Make additional copies or read the data from the clipboard for further usage.
* SetClipboardData also returns a handle for accessing the memory block temporarily.
* Remember to unlock this handle before closing the clipboard.

Additional Notes:

* This process demonstrates transferring a NULL-terminated ANSI string.
* Other clipboard formats like CF\_UNICODE and CF\_OEMTEXT exist for different character sets.
* The example assumes the existence of a valid hwnd representing the window handle.

GETTING TEXT FROM THE CLIPBOARD

Checking for Text Availability:

Before attempting to retrieve text from the clipboard, it's important to verify its presence in the desired format. You can use the IsClipboardFormatAvailable function to check specifically for the CF\_TEXT format:



This function returns TRUE if text data is present, enabling you to adjust your program's behavior accordingly.

Retrieving Text Data:

Open the Clipboard: Gaining access to the clipboard is crucial before attempting to extract any data.



Obtain Global Handle: This function retrieves the handle to the global memory block containing the text. If no text is available, hGlobal will be NULL.



Check for Null Handle: If GetClipboardData returns NULL, it means the clipboard doesn't contain text. In this case, close the clipboard:



Allocate Memory: Create a memory block within your program to store the copied text. Use GlobalSize to determine the size of the clipboard memory block and allocate the same size for your own.



Lock Clipboard Memory: Gain access to the data within the clipboard memory block.



Copy Data: You have two options for copying the data.

Using strcpy - This function copies the entire string from the clipboard memory to your program's memory.



Using a Loop: This loop iterates through both pointers, copying each character individually.



Unlock Clipboard Memory: Release access to the clipboard memory block.



Close Clipboard: Relinquish control of the clipboard after successfully retrieving the desired data.



Accessing Copied Text: The variable pText now points to your program's own copy of the clipboard text. You can freely use this data for further processing within your application.

Additional Notes:

* This process focuses on retrieving and copying ANSI text data.
* Alternative clipboard formats exist for different character sets and data types.
* The provided code snippet demonstrates two methods for data copying.
* Choose the method that best suits your coding style and preferences.

ClipText program chapter 12 folder for the code…