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

Format Identifier	Description	Usage	Supported Platforms
CF_TEXT	NULL-terminated ANSI text with CRLF line endings	Simple text transfer	All Windows versions
CF_OEMTEXT	Text data using the OEM character set	Primarily for interaction with MS-DOS programs	All Windows versions
CF_UNICODETEXT	Unicode text with CRLF line endings and NULL termination	Unicode text transfer	Windows NT and later
CF_LOCALE	Handle to a locale identifier	Indicates the locale associated with clipboard text	All Windows versions
CF_SYLK	Microsoft Symbolic Link format for data exchange	Used with Multiplan, Chart, and Excel	Primarily for legacy applications
CF_DIF	Data Interchange Format for spreadsheet data	Used with VisiCalc and similar applications	Primarily for legacy applications
CF_BITMAP	Device-dependent bitmap	Pixel data for images	All Windows versions
CF_DIB	Device-independent bitmap with information structure	Flexible bitmap transfer	All Windows versions
CF_PALETTE	Handle to a color palette	Used in conjunction with CF_DIB for defining colors	All Windows versions
CF_TIFF	Tag Image File Format for image data	Industry-standard image format	All Windows versions
CF_METAFILEPICT	Metafile picture based on older Windows metafile format	Legacy metafile format	All Windows versions
CF_ENHMETAFILE	Enhanced metafile format	Advanced metafile format	Windows 32-bit and later
CF_PENDATA	Used with pen extensions	Primarily for pen-based input	Platforms with pen support
CF_WAVE	Waveform audio file	Sound data transfer	All Windows versions
CF_RIFF	Multimedia data in Resource Interchange File Format	General multimedia data	Windows multimedia platforms
CF_HDROP	List of files used for drag-and-drop	Data transfer between applications	All Windows versions

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.

```
HGLOBAL GlobalAlloc(
    UINT uFlags,
    DWORD dwBytes
);
```

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.

```
HGLOBAL GlobalReAlloc(
    HGLOBAL hMem,
    DWORD dwBytes,
    UINT uFlags
);
```

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

```
DWORD GlobalSize(
    HGLOBAL hMem
);
```

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

```
BOOL GlobalFree(
    HGLOBAL hMem
);
```

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.

```
190  int* p;
191  GLOBALHANDLE hGlobal;
192
193  // Allocate memory
194  hGlobal = GlobalAlloc(GHND, 1024);
195
196  // Access the memory block
197  p = (int*)GlobalLock(hGlobal);
198
199  // ... Perform operations on the memory block ...
200
201  // Release the memory block
202  GlobalUnlock(hGlobal);
203
204  // Free the memory
205  GlobalFree(hGlobal);
```

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:

```
hGlobal = GlobalAlloc(GMEM_MOVEABLE | GMEM_SHARE, dataSize);
```

Clipboard memory management functions:

Function	Description	Code Example
`GlobalAlloc`	Allocates a global memory block.	`hGlobal = GlobalAlloc(GHND, 1024);`
`GlobalLock`	Locks a global memory block and returns a pointer to access it.	`p = (int *) GlobalLock(hGlobal);`
`GlobalUnlock`	Unlocks a previously locked global memory block.	`GlobalUnlock(hGlobal);`
`GlobalFree`	Frees a global memory block.	`GlobalFree(hGlobal);`
`GlobalHandle`	Retrieves the handle associated with a locked global memory block using its pointer.	`hGlobal = GlobalHandle(p);`

TEXT TRANSFER TO CLIPBOARD

Function Breakdown:

GlobalAlloc:

Allocate a memory block of sufficient size for the string.

```
hGlobal = GlobalAlloc(GHND | GMEM_SHARE, iLength + 1);
```

- 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.

```
pGlobal = GlobalLock(hGlobal);
```

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

String Copying:

Copy the string content into the allocated memory block.

```
for (i = 0; i < iLength; i++) {
    *pGlobal++ = *pString++;
}
```

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

GlobalUnlock:

Release the lock on the memory block.

```
GlobalUnlock(hGlobal);
```

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

Open/Close Clipboard:

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

```
OpenClipboard(hwnd);
EmptyClipboard();
CloseClipboard();
```

- 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.

```
SetClipboardData(CF_TEXT, hGlobal);
```

- 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:

```
bAvailable = IsClipboardFormatAvailable(CF_TEXT);
```

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.

```
OpenClipboard(hwnd);
```

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.

```
hGlobal = GetClipboardData(CF_TEXT);
```

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

```
CloseClipboard();
```

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.

```
pText = (char *) malloc(GlobalSize(hGlobal));
```

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

```
pGlobal = GlobalLock(hGlobal);
```

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.

```
strcpy(pText, pGlobal);
```

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

```
while (*pText++ = *pGlobal++) ;
```

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

```
GlobalUnlock(hGlobal);
```

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

```
CloseClipboard();
```

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…

Opening and Closing the Clipboard: A Deep Dive

This section delves deeper into the intricacies of opening and closing the clipboard in Windows applications.

Exclusive Access and Responsibility:

- Only one program can have the clipboard open at a time.
- OpenClipboard ensures data integrity by preventing changes while in use.
- It returns TRUE if successful and FALSE if another program holds the lock.

Importance of Prompt Opening and Closing:

- Minimizes the risk of conflicting applications accessing the clipboard.
- Promotes smooth operation and avoids potential data corruption.

Preemptive Multitasking and Potential Issues:

- Background processes might access the clipboard, altering its contents unexpectedly.
- Always check the clipboard data before assuming its state is unchanged.

Message Boxes and Clipboard Access:

- Using non-modal message boxes while the clipboard is open allows users to switch to other applications.
- This can lead to unexpected behavior and data inconsistencies.
- Consider using system modal message boxes or closing the clipboard before displaying them.

Dialog Boxes and Edit Fields:

- Edit fields in dialog boxes rely on the clipboard for cut-and-paste functionality.
- Leaving the clipboard open during dialog box interaction can lead to conflicts.
- Close the clipboard before displaying dialog boxes to prevent potential issues.

Unicode Support and Clipboard Conversions:

- Windows automatically handles text conversions between formats (CF_TEXT, CF_OEMTEXT, CF_UNICODETEXT).
- Programs can call SetClipboardData with their preferred format and GetClipboardData with their desired format.
- Windows will perform the necessary conversion in the background.

Program Implementation Recommendations:

- Use CF_UNICODETEXT if the UNICODE flag is defined, otherwise use CF_TEXT.
- This approach ensures compatibility with different Unicode configurations.
- The CLIPTEXT program demonstrates a practical implementation of format switching based on the UNICODE flag.

Maintaining proper control over the clipboard is crucial for ensuring data integrity and avoiding conflicts in Windows applications. By understanding the exclusive access mechanism, potential issues, and Unicode handling, you can write programs that interact with the clipboard reliably and efficiently.

CLIPTEXT PROGRAM

The CLIPTEXT program showcases the interaction with the Windows clipboard for transferring text data. It demonstrates operations like copying, pasting, clearing, and resetting text content. This analysis dives deeper into the program's structure and functionality.

Preprocessor Directives:

- Include necessary header files like windows.h and resource.h.
- Define UNICODE-specific text formats and default message strings.

Global Variables:

- pText: Pointer to the stored clipboard text.
- bEnable: Boolean flag for enabling menu items.
- hGlobal: Handle to the global memory block containing text.
- hdc: Device context handle for drawing text.
- pGlobal: Pointer to the memory block within the clipboard.

Window Procedure:

Handles various messages received by the application window.

- WM_CREATE: Initialises pText with default text.
- WM_INITMENUPOPUP: Enables/disables menu items based on clipboard content and pText availability.
- WM_COMMAND: Handles user actions like menu selections.
- IDM_EDIT_PASTE: Opens clipboard, retrieves text, updates internal storage, and invalidates the window for redraw.

- IDM_EDIT_CUT/COPY: Allocates memory, copies text to clipboard, updates internal storage, and invalidates the window (clear for cut).
- IDM_EDIT_CLEAR: Frees allocated memory and invalidates the window.
- IDM_EDIT_RESET: Restores default text and invalidates the window.
- WM_PAINT: Renders the stored text onto the client area.
- WM_DESTROY: Frees allocated memory and sends quit message.

Resource Files:

- CLIPTEXT.RC: Defines the menu structure and accelerator keys.
- CLIPTEXT.H: Provides symbolic IDs for menu items and accelerator keys.

Key Concepts:

Clipboard Access:

- OpenClipboard: Establishes access to the clipboard for reading/writing data.
- GetClipboardData: Retrieves the handle to the global memory block containing clipboard data.
- GlobalLock/Unlock: Locks/unlocks the memory block for accessing its content.
- EmptyClipboard: Clears the existing content of the clipboard.
- SetClipboardData: Places the provided memory block with data onto the clipboard.
- CloseClipboard: Relinquishes control of the clipboard after operations.

Memory Management:

- malloc/free: Allocate/deallocate memory for storing the clipboard text.
- GlobalAlloc/GlobalFree: Allocate/deallocate memory for the global memory block used by the clipboard.

Text Rendering:

- GetClientRect: Retrieves the dimensions of the client area for drawing.
- DrawText: Renders the text string onto the specified device context.

Menu and Accelerator Management:

- EnableMenuItem: Enables/disables menu items based on specific conditions.
- LoadAccelerators: Loads the accelerator table associated with the program.
- TranslateAccelerator: Handles keyboard shortcuts defined in the accelerator table.

Additional Notes:

- The program demonstrates handling both ANSI and Unicode text formats based on the UNICODE preprocessor flag.
- The CLIPTEXT.RC file defines the menu structure and keyboard shortcuts for Cut, Copy, Paste, Clear, and Reset functionalities.
- The CLIPTEXT.H file provides symbolic IDs for menu items and accelerator keys to improve code readability and maintainability.

The CLIPTEXT program provides a comprehensive example of interacting with the Windows clipboard for text data transfer. It demonstrates fundamental concepts like opening/closing the clipboard, retrieving/setting data, managing memory, drawing text, and handling user input through menus and keyboard shortcuts.

Deep Dive into CLIPTEXT and Clipboard Transformations

The CLIPTEXT program showcases the clipboard's ability to translate between Unicode and ANSI character sets. This is achieved through the #ifdef statement at the beginning:

```
265  #ifdef UNICODE
266  #define CF_TCHAR CF_UNICODETEXT
267  TCHAR szDefaultText[] = TEXT("Default Text - Unicode Version");
268  TCHAR szCaption[] = TEXT("Clipboard Text Transfers - Unicode Version");
269  #else
270  #define CF_TCHAR CF_TEXT
271  TCHAR szDefaultText[] = TEXT("Default Text - ANSI Version");
272  TCHAR szCaption[] = TEXT("Clipboard Text Transfers - ANSI Version");
273  #endif
```

This defines a generic text format CF_TCHAR that maps to either CF_UNICODETEXT for Unicode builds or CF_TEXT for ANSI builds. This ensures consistent behavior across both versions.

Clipboard Operations:

The program demonstrates basic clipboard operations like copying, pasting, clearing, and resetting text content. Here's a breakdown of the key functions:

Setting Clipboard Data:

```
275  OpenClipboard(hwnd);
276  EmptyClipboard();
277  hGlobal = GlobalAlloc(GHND | GMEM_SHARE, (lstrlen(pText) + 1) * sizeof(TCHAR));
278  pGlobal = GlobalLock(hGlobal);
279  lstrcpy(pGlobal, pText);
280  GlobalUnlock(hGlobal);
281  SetClipboardData(CF_TCHAR, hGlobal);
282  CloseClipboard();
```

- Open the clipboard.
- Clear existing content.
- Allocate memory for the text data.
- Lock the memory block for access.
- Copy the program's text to the clipboard memory.
- Unlock the memory block.
- Set the clipboard data with the specified format and memory handle.
- Close the clipboard.

Getting Clipboard Data:

```
285  OpenClipboard(hwnd);
286  hGlobal = GetClipboardData(CF_TCHAR);
287
288  if (hGlobal) {
289      pGlobal = GlobalLock(hGlobal);
290      // Access and process clipboard data
291      GlobalUnlock(hGlobal);
292  }
293  CloseClipboard();
```

- Open the clipboard.
- Retrieve the handle to the clipboard data using the desired format.
- If data exists, lock the memory block for access.
- Access and process the clipboard data within the memory block.
- Unlock the memory block.
- Close the clipboard.

Text Rendering and User Interaction:

- The program displays the stored text using DrawText during the WM_PAINT message.
- Menu and accelerator keys allow users to perform Cut, Copy, Paste, Clear, and Reset actions.
- The bEnable flag controls the availability of menu items based on the text presence.

Unicode and ANSI Conversions:

The program demonstrates how clipboard transfers trigger automatic conversions between Unicode and ANSI character sets. By running both versions and performing Copy/Paste operations, you can observe the conversion in action.

Beyond Simple Clipboard Use:

While the provided code demonstrates basic clipboard interactions, it's crucial to understand that the clipboard offers more advanced capabilities. You can:

- Monitor clipboard changes: Register a window to receive notifications when the clipboard content changes.
- Transfer custom data formats: Implement custom formats using RegisterClipboardFormat and handle them appropriately.
- Share data between applications: Use the clipboard as a communication mechanism between different programs.

USING MULTIPLE DATA ITEMS WITH THE CLIPBOARD

While the clipboard can only hold one item at a time, you can leverage it to store data in multiple formats simultaneously. This allows different programs to access the same information, interpreting it in their respective ways.

Adding Multiple Formats:

- Open the clipboard: Call OpenClipboard(hwnd) to acquire access.
- Empty existing content: Use EmptyClipboard() to clear any previous data.
- Set data for different formats: Call SetClipboardData multiple times, each with a specific format identifier and corresponding data handle. For example:

```
OpenClipboard(hwnd);
EmptyClipboard();
hGlobalText = GlobalAlloc(GHND | GMEM_SHARE, sizeof(text));
lstrcpy(text, "Sample Text");
SetClipboardData(CF_TEXT, hGlobalText);

hBitmap = CreateBitmap(...); // Create a bitmap object
SetClipboardData(CF_BITMAP, hBitmap);

hGlobalMFP = CreateMetaFile(...); // Create a metafile object
SetClipboardData(CF_METAFILEPICT, hGlobalMFP);
CloseClipboard();
```

Accessing Multiple Formats:

- Open the clipboard: Call OpenClipboard(hwnd) to acquire access.
- Enumerate available formats: Use EnumClipboardFormats(iFormat) in a loop, starting with iFormat = 0. This function returns a non-zero value for each available format.
- Retrieve specific format data: Use GetClipboardData(iFormat) to obtain the data handle associated with the desired format.
- Close the clipboard: Call CloseClipboard() to release access. Code Example:

```
OpenClipboard(hwnd);
iFormat = 0;
while (iFormat = EnumClipboardFormats(iFormat)) {
  switch (iFormat) {
    case CF_TEXT:
      hGlobalText = GetClipboardData(iFormat);
      if (hGlobalText) {
        // Access and process text data
        GlobalFree(hGlobalText);
      }
      break;
    case CF_BITMAP:
      hBitmap = GetClipboardData(iFormat);
      if (hBitmap) {
        // Handle bitmap object
        DeleteObject(hBitmap);
      }
      break;
    case CF_METAFILEPICT:
      hGlobalMFP = GetClipboardData(iFormat);
      if (hGlobalMFP) {
        // Process metafile object
        DeleteMetaFile(hGlobalMFP);
      }
      break;
  }
}
CloseClipboard();
```

Important Considerations:

- Avoid adding multiple data formats of the same type (e.g., multiple text formats).
- Windows automatically converts between certain formats (e.g., CF_TEXT, CF_OEMTEXT, CF_UNICODETEXT).
- Use CountClipboardFormats() to get the total number of formats currently stored.

By leveraging multiple formats in the clipboard, developers can provide more versatile data transfer capabilities within their applications. This allows other programs to access and interpret the information in the most suitable way for their specific needs.

DELAYED RENDERING FOR EFFICIENT CLIPBOARD MANAGEMENT

When dealing with large data items in the clipboard, traditional methods can lead to unnecessary memory usage. Delayed rendering provides a solution by deferring the creation of the actual data until it's requested by another program. This can significantly improve memory efficiency and conserve resources.

The Basic Approach:

Open and Empty Clipboard: Open the clipboard using OpenClipboard(hwnd) and clear any existing content with EmptyClipboard().

Use NULL for SetClipboardData: Instead of providing a handle to the data, use NULL in the SetClipboardData(iFormat, NULL) call for each desired format.

Handle WM_RENDERFORMAT: When another program attempts to retrieve the data through GetClipboardData, Windows sends a WM_RENDERFORMAT message to the "clipboard owner" (your program).

Process WM_RENDERFORMAT: Upon receiving WM_RENDERFORMAT, your program must create a global memory block containing the actual data for the requested format and call SetClipboardData with the appropriate format and handle.

Handle WM_DESTROYCLIPBOARD: When the clipboard is emptied using EmptyClipboard, Windows sends the WM_DESTROYCLIPBOARD message to the clipboard owner, indicating the information is no longer needed.

Handle WM_RENDERALLFORMATS (Optional): If your program terminates while owning the clipboard with unrendered data, it receives WM_RENDERALLFORMATS. This prompts you to open the clipboard, empty it, create and render all data formats, and close the clipboard.

Code Example:

```
case WM_RENDERFORMAT:
    // Create a global memory block for the requested format
    hGlobal = GlobalAlloc(GHND | GMEM_SHARE, ...);

    // Prepare and fill the data within the memory block
    // ...

    // Set the clipboard data for the specified format
    SetClipboardData(wParam, hGlobal);

    return 0;

case WM_RENDERALLFORMATS:
    // Open the clipboard for rendering all formats
    OpenClipboard(hwnd);

    // Empty the clipboard before processing formats
    EmptyClipboard();

    // Process each format and render data using SetClipboardData
    for (UINT iFormat = EnumClipboardFormats(0); iFormat; iFormat = EnumClipboardFormats(iFormat)) {
        // Create and fill data based on the format
        hGlobal = GlobalAlloc(GHND | GMEM_SHARE, ...);
        // ...
        // Set the clipboard data for the current format
        SetClipboardData(iFormat, hGlobal);
    }

    // Close the clipboard after rendering all formats
    CloseClipboard();
    return 0;

case WM_DESTROYCLIPBOARD:
    // Release any resources associated with unrendered data
    // ...
    return 0;
```

Additional Notes:

- This approach is particularly beneficial for large data transfers, reducing memory footprint.
- Combining WM_RENDERALLFORMATS and WM_RENDERFORMAT processing is possible for programs with a single data format.
- Processing WM_DESTROYCLIPBOARD is optional unless resource retention is cumbersome.

Delayed rendering offers a powerful technique for managing large clipboard data efficiently. By understanding the process and implementing it correctly, developers can significantly improve memory usage and enhance their program's performance.

PRIVATE DATA FORMATS: SHARING BEYOND STANDARD FORMATS

While the Windows clipboard provides a set of standard formats for data transfer, sometimes you need to share information that only your program understands. This is where private data formats come into play.

Why Use Private Data Formats?

- Sharing between your program: Use private formats to transfer data between different instances of your program. This allows them to share information that other programs wouldn't understand.
- Extended data representation: Store information beyond the capabilities of standard formats. For example, word processors use private formats to store text with font and formatting information.

Types of Private Formats:

- DSP Formats: These formats (CF_DSPTEXT, CF_DSPBITMAP, CF_DSPMETAFILEPICT, CF_DSPENHMETAFILE) allow your program to store data in a standard format but with a private interpretation. This enables the Windows clipboard viewer to display the data, though other programs won't understand the specific details.
- CF_OWNERDISPLAY: This format sets the global memory handle to NULL and signals your program's responsibility for displaying the clipboard content. This approach is often used by word processors to render formatted text in the clipboard viewer.

Identifying Private Data Sources:

- GetClipboardOwner: Use this function to retrieve the handle of the window that owns the clipboard data.
- GetClassName: Compare the window class name with your program's class name. If they match, the data likely originated from another instance of your program.

Code Examples:

Using DSP Format:

```
410  // Store formatted text in DSP format
411  OpenClipboard(hwnd);
412  EmptyClipboard();
413
414  // Calculate the size of the text buffer
415  const char* formattedText = "Sample Text with Private Formatting";
416  size_t textLength = lstrlen(formattedText) + 1; // Include null terminator
417
418  // Allocate global memory for formatted text
419  HGLOBAL hGlobalText = GlobalAlloc(GHND | GMEM_SHARE, textLength);
420  if (hGlobalText != NULL) {
421      // Copy the formatted text into the global memory
422      char* pText = (char*)GlobalLock(hGlobalText);
423      if (pText != NULL) {
424          lstrcpy(pText, formattedText);
425          GlobalUnlock(hGlobalText);
426
427          // Set the clipboard data in DSP format
428          SetClipboardData(CF_DSPTEXT, hGlobalText);
429      } else {
430          // Handle memory lock failure
431          GlobalFree(hGlobalText);
432      }
433  }
434
435  CloseClipboard();
436
437  // Retrieve and process DSP format data
438  OpenClipboard(hwnd);
439  hGlobalText = GetClipboardData(CF_DSPTEXT);
440  if (hGlobalText != NULL) {
441      // Access and process formatted text based on your program's logic
442
443      // Free the global memory after processing
444      GlobalFree(hGlobalText);
445  }
446
447  CloseClipboard();
```

Using CF_OWNERDISPLAY:

```
450  // Clear clipboard and set up for custom rendering
451  OpenClipboard(hwnd);
452  EmptyClipboard();
453  SetClipboardData(CF_OWNERDISPLAY, NULL);
454  CloseClipboard();
455
456  // Custom function to draw clipboard content based on program data
457  void DrawClipboardContent(HDC hdc) {
458      // Render private data onto the provided device context
459  }
460
461  // Handle clipboard drawing message
462  case WM_DRAWCLIPBOARD:
463      DrawClipboardContent(wParam);
464      return 0;
```

Private data formats offer a powerful mechanism for sharing information beyond the standard clipboard capabilities.

Processing Messages for CF_OWNERDISPLAY

When using the CF_OWNERDISPLAY format, the clipboard owner (your program) receives several additional messages from both the clipboard viewer and Windows:

**Message	Purpose	wParam	lParam**
WM_ASKCBFORMATNAME	Retrieve format name	Maximum character count for buffer	Pointer to buffer for format name
WM_SIZECLIPBOARD	Update on viewer size change	Handle to clipboard viewer	Pointer to RECT structure with new size
WM_PAINTCLIPBOARD	Request to update viewer content	Handle to clipboard viewer	Global handle to PAINTSTRUCT
WM_HSCROLLCLIPBOARD	Horizontal scroll event	Handle to clipboard viewer	Scroll request and thumb position (optional)
WM_VSCROLLCLIPBOARD	Vertical scroll event	Handle to clipboard viewer	Scroll request and thumb position (optional)

Handling the Messages:

WM_ASKCBFORMATNAME: This message requests the name of the private format. The owner must copy the name into the provided buffer, using a maximum length of wParam.

WM_SIZECLIPBOARD: This message informs the owner of the viewer's resized client area. The owner can use the information to adjust its drawing calculations.

WM_PAINTCLIPBOARD: This message instructs the owner to update the viewer's content. The owner obtains the device context from the lParam structure and uses it to render the private data.

WM_HSCROLLCLIPBOARD/WM_VSCROLLCLIPBOARD: These messages indicate user interaction with the viewer's scrollbars. The owner can use the information to update the displayed content based on the scroll position.

Code Example:

```
// Handle messages for CF_OWNERDISPLAY format
case WM_ASKCBFORMATNAME:
    // Provide the custom format name
    lstrcpy((LPTSTR)lParam, "MyPrivateFormat");
    return lstrlen("MyPrivateFormat");

case WM_SIZECLIPBOARD:
    // Update internal data based on new size
    return 0;

case WM_PAINTCLIPBOARD:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint((HWND)wParam, &ps);
        // Render private data onto the device context
        EndPaint((HWND)wParam, &ps);
    }
    return 0;

case WM_HSCROLLCLIPBOARD:
    // Update display based on horizontal scroll position
    return 0;

case WM_VSCROLLCLIPBOARD:
    // Update display based on vertical scroll position
    return 0;
```

Benefits of CF_OWNERDISPLAY:

Enhanced user experience: Users see formatted data in the clipboard viewer, providing visual feedback for copied content.

Custom format representation: Enables programs to share data in proprietary formats beyond the standard ones.

Challenges of CF_OWNERDISPLAY:

Increased complexity: Requires handling several additional messages and managing the rendering process.

Compatibility considerations: Other programs won't understand the private format, limiting its interoperability.

Alternative Approaches:

DSP Formats: Use existing standard formats with private interpretation for display in the viewer.

Registered Formats: Register a custom format name with Windows, allowing other programs to access the data.

PUBLICLY SHARING CUSTOM CLIPBOARD FORMATS

In the vector-drawing program example, the program copies data in three formats: bitmap, metafile, and its own registered format. This section details registering the custom format and sharing its details.

Registering Custom Format:

- Define a unique format name string (e.g., "MyVectorDrawingFormat").
- Use iFormat = RegisterClipboardFormat(szFormatName) to register the format with Windows.
- This function returns an integer iFormat between 0xC000 and 0xFFFF, representing your custom format identifier.

Sharing Format Information:

- Publicly disclose the format name: This allows other developers to implement their programs to read and write data in your custom format.
- Document the data format: Provide detailed information about the data structure, including element types, sizes, and meanings.
- Consider open-sourcing the format: This encourages broader adoption and avoids vendor lock-in.

Benefits of Publicly Shared Formats:

- Increased interoperability: Other programs can exchange data with your program using the custom format.
- Enhanced functionality: Both programs can access richer data beyond standard formats.
- Community collaboration: Developers can contribute to improving and evolving the format.

Example Code:

```
500  // Registering the custom format
501  const char* szFormatName = "MyVectorDrawingFormat";
502  UINT iFormat = RegisterClipboardFormat(szFormatName);
503
504  // Accessing the format name from another program
505  char szBuffer[256];
506  if (GetClipboardFormatName(iFormat, szBuffer, sizeof(szBuffer)) > 0) {
507      printf("Custom format name: %s\n", szBuffer);
508  } else {
509      printf("Unable to retrieve format name.\n");
510  }
```

Becoming a Clipboard Viewer: Understanding the Chain

In Windows, any program that wants to be notified of changes in the clipboard content can become a "clipboard viewer." While Windows includes its own built-in viewer, you can write your custom program to monitor and react to clipboard updates.

The Clipboard Viewer Chain:

Multiple clipboard viewers can coexist in Windows, but only one is considered the "current" viewer. This viewer receives messages from Windows whenever the clipboard content changes. However, to ensure all interested programs are notified, a "clipboard viewer chain" exists.

This chain acts like a linked list, where each viewer points to the next one in line. When a program becomes the current viewer, it receives the window handle of the previous viewer, establishing the chain link.

Responsibilities of a Clipboard Viewer:

Handle Clipboard Messages: Windows sends specific messages to the current viewer when the clipboard changes. These messages include:

- WM_DRAWCLIPBOARD: Informs the viewer to update its display with the new content.
- WM_CHANGECBCHAIN: Notifies the viewer that another program is joining or leaving the chain.
- WM_DESTROYCLIPBOARD: Indicates the clipboard is being emptied.

Maintain the Chain: The viewer must forward received messages to the next program in the chain using the SetClipboardViewer and ChangeClipboardChain functions.

Become the Current Viewer: To start receiving notifications, the program must call SetClipboardViewer with its own window handle.

Becoming the Current Viewer:

- Call SetClipboardViewer(hwnd), where hwnd is your program's window handle.
- Windows stores your program as the current viewer and gives you the previous viewer's handle.
- Update your internal state to keep track of the previous viewer.

Participating in the Chain:

- Process the received messages in your window procedure, like:
  - Draw the new clipboard content on receiving WM_DRAWCLIPBOARD.

- Update the chain by calling ChangeClipboardChain when receiving WM_CHANGECBCHAIN.
- Forward the messages to the next viewer using ChangeClipboardChain.

Example Code:

```
515  // Become the current clipboard viewer
516  hwnd = SetClipboardViewer(hwnd);
517
518  // Process clipboard messages
519  case WM_DRAWCLIPBOARD:
520      // Update your display with the new content
521      // ...
522
523      // Forward the message to the next viewer in the chain
524      SendMessage(hWndNextViewer, WM_DRAWCLIPBOARD, wParam, lParam);
525      break;
526
527  case WM_CHANGECBCHAIN:
528      // Update the chain based on the new viewer
529      ChangeClipboardChain(hwnd, (HWND)wParam);
530      break;
531
532  case WM_DESTROYCLIPBOARD:
533      // Clean up and remove yourself from the chain
534      ChangeClipboardChain(hwnd, NULL);
535      break;
```

Clipboard Viewer Chain: In-Depth Analysis

The clipboard viewer chain is a mechanism in Windows that allows multiple programs to be notified of changes to the clipboard content. This section provides a detailed breakdown of the functions and messages involved in this process.

Becoming a Viewer:

- SetClipboardViewer: This function registers your program as the current clipboard viewer and returns the handle of the previous viewer.

- Saving the Handle: Store the previous viewer's handle in a static variable (e.g., hwndNextViewer).

```
static HWND hwndNextViewer;

// During WM_CREATE message
hwndNextViewer = SetClipboardViewer(hwnd);
```

Handling WM_DRAWCLIPBOARD:

- Forwarding the Message: Unless hwndNextViewer is NULL, use SendMessage to pass the message to the next viewer in the chain.
- Invalidating Client Area: Update your program's display by invalidating its client area.

```
case WM_DRAWCLIPBOARD:
  if (hwndNextViewer) {
    SendMessage(hwndNextViewer, message, wParam, lParam);
  }
  InvalidateRect(hwnd, NULL, TRUE);
  return 0;
```

Processing WM_PAINT:

- Reading Clipboard Data: Use OpenClipboard, GetClipboardData, and CloseClipboard to access the current clipboard content.
- Update Display: Update your program's display based on the retrieved data.

Leaving the Chain:

- ChangeClipboardChain: Use this function to remove your program from the chain by providing your own handle and the next viewer's handle.
- Windows sends WM_CHANGECBCHAIN: When you leave the chain, Windows notifies the current viewer with this message.

```
556   case WM_DESTROY:
557     ChangeClipboardChain(hwnd, hwndNextViewer);
558     PostQuitMessage(0);
559     return 0;
```

Processing WM_CHANGECBCHAIN:

- Checking for Self Removal: If wParam matches your hwndNextViewer, update it with the lParam value (next viewer).
- Forwarding the Message: If wParam doesn't match and hwndNextViewer isn't NULL, forward the message to the next viewer.

```
562   case WM_CHANGECBCHAIN:
563     if ((HWND)wParam == hwndNextViewer) {
564       hwndNextViewer = (HWND)lParam;
565     } else if (hwndNextViewer) {
566       SendMessage(hwndNextViewer, message, wParam, lParam);
567     }
568     return 0;
```

Additional Functions:

- GetClipboardViewer: Retrieves the handle of the current clipboard viewer.
- EmptyClipboard: Empties the clipboard content.

Example Scenario:

- Initially, the chain is empty.
- Program hwnd1 joins the chain and becomes the current viewer. hwndNextViewer is NULL.
- Program hwnd2 joins the chain and becomes the current viewer. hwndNextViewer is set to hwnd1.

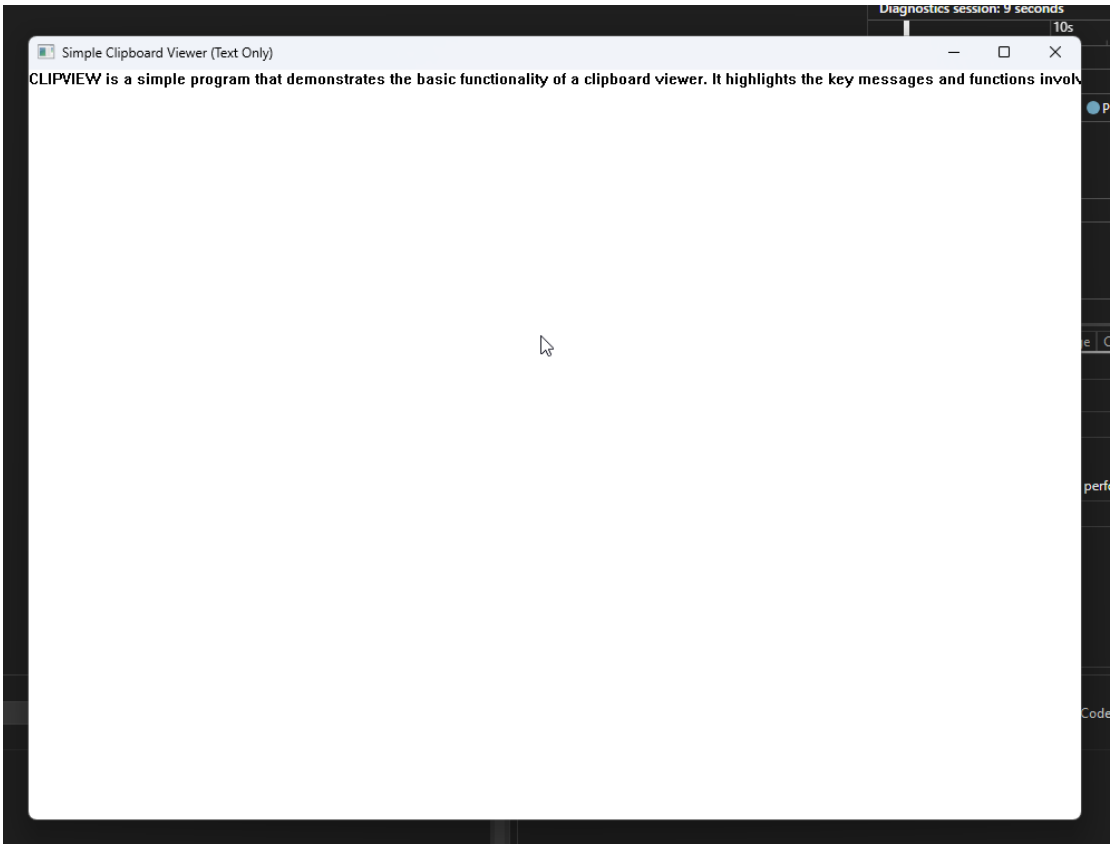
- Program hwnd3 joins the chain and becomes the current viewer. hwndNextViewer is set to hwnd2.
- Program hwnd2 leaves the chain. hwnd3 updates its hwndNextViewer to point to hwnd1.

Conclusion:

Understanding the clipboard viewer chain and its associated functions is crucial for programs that want to react to changes in the clipboard content.

By implementing the proper message handling and chain management, your programs can effectively integrate with the clipboard and perform relevant actions based on the data available.

ClipView program in chapter 12 folder for the program code…



CLIPVIEW: A Simple Clipboard Viewer Program

CLIPVIEW is a simple program that demonstrates the basic functionality of a clipboard viewer. It highlights the key messages and functions involved in the process, focusing specifically on displaying the CF_TEXT format.

Message Handling:

- WM_CREATE: Registers the program as the current clipboard viewer and stores the previous viewer's handle in hwndNextViewer.
- WM_CHANGECBCHAIN: Updates the hwndNextViewer value when another viewer joins or leaves the chain.
- WM_DRAWCLIPBOARD: Forwards the message to the next viewer and invalidates the program's client area, triggering a repaint.
- WM_PAINT: Retrieves the CF_TEXT data from the clipboard and uses DrawText to display it in the client area.
- WM_DESTROY: Removes the program from the chain and terminates the application.

Data Retrieval:

- CLIPVIEW uses OpenClipboard and GetClipboardData to access the clipboard content.
- It checks for the CF_TEXT format. If available, it obtains the global memory handle and locks it.
- The locked memory is then used with DrawText to render the text on the program's window.
- After displaying the content, the program unlocks the memory and closes the clipboard.

Limitations:

- CLIPVIEW only handles the CF_TEXT format, demonstrating a basic implementation.
- More advanced viewers may need to handle additional formats and utilize functions like EnumClipboardFormats and GetClipboardFormatName to display names and data for non-standard formats.
- For formats like CF_OWNERDISPLAY, the viewer must send specific messages (WM_PAINTCLIPBOARD, WM_VSCROLLCLIPBOARD, WM_SIZECLIPBOARD, WM_HSCROLLCLIPBOARD) to the clipboard owner for proper display.

Obtaining Clipboard Owner:

- To communicate with the owner of a custom format, CLIPVIEW needs the owner's window handle.
- This is achieved using the GetClipboardOwner function.
- Once the handle is obtained, the viewer can send the necessary messages to the owner for updating the content.

Key Points:

- CLIPVIEW provides a foundational example of a clipboard viewer.
- Understanding its message handling and data retrieval demonstrates the core principles of clipboard interaction.
- For more complex scenarios involving diverse formats and owner interaction, additional functionalities need to be implemented.

End of chapter 12…