CHAPTER 12: THE CLIPBOARD (THE GREAT MIDDLEMAN)

What is the Clipboard?

The Clipboard is a small slice of memory managed by Windows. Its only job is to hold data temporarily so you can move it from one place to another.

- **It's simple:** It doesn't take much code to use it.
- **It's persistent:** Once you "Copy" something, it stays there until you copy something else or turn off the computer.
- **It's visible:** You can actually open a "Clipboard Viewer" app in Windows to see exactly what is sitting in that memory right now.

The Three Main Actions

Most programs use the Clipboard through three classic commands:

1. **Copy:** You take a "photo" of the data and put it in the Clipboard. The original stays where it is.
2. **Cut:** You pick up the data, put it in the Clipboard, and **remove** it from the original program.
3. **Paste:** You ask the Clipboard, "What do you have for me?" and pull a copy of that data into your current program.

The "Rules of the Road"

To keep things organized, Windows follows a few strict rules:

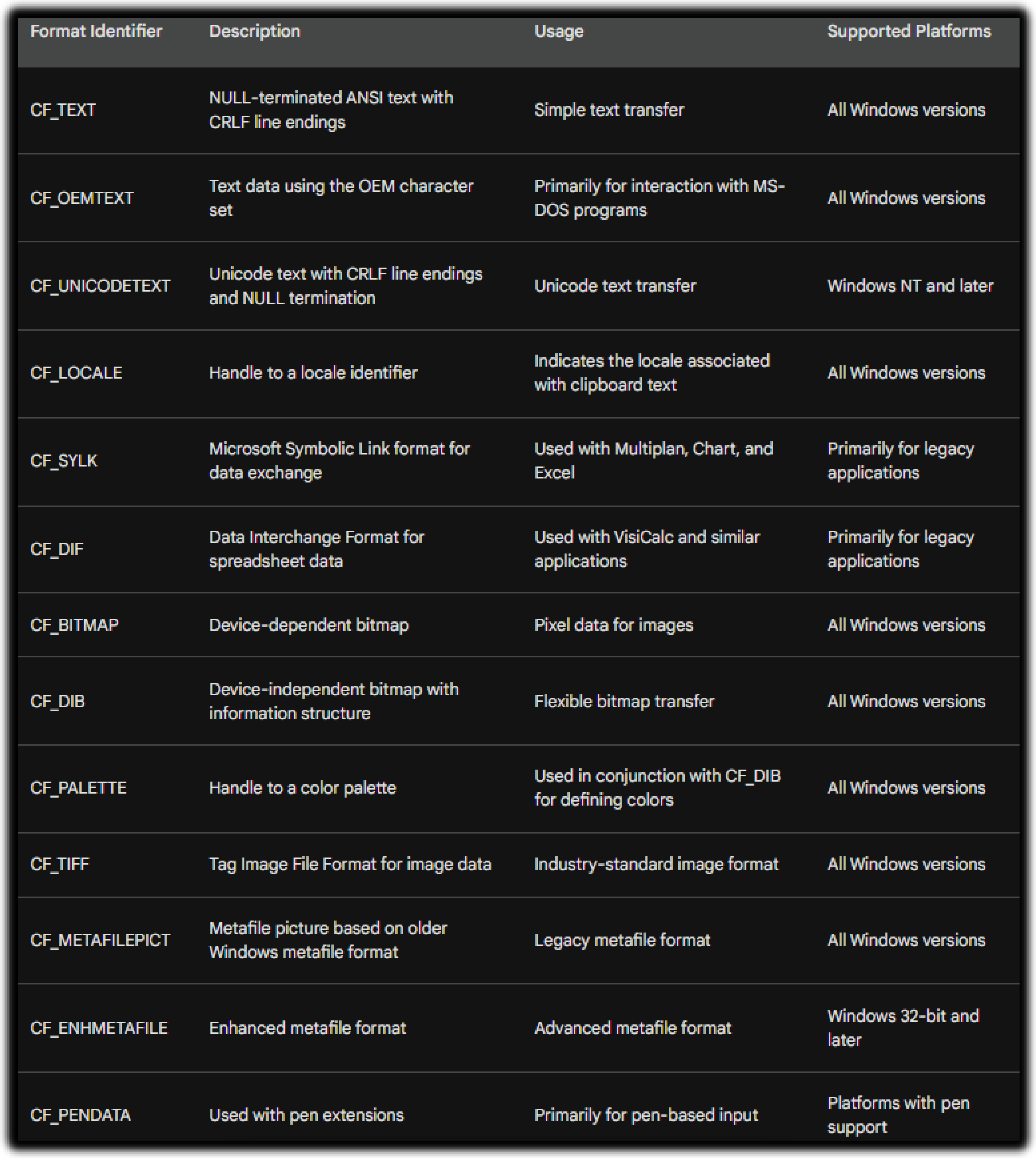
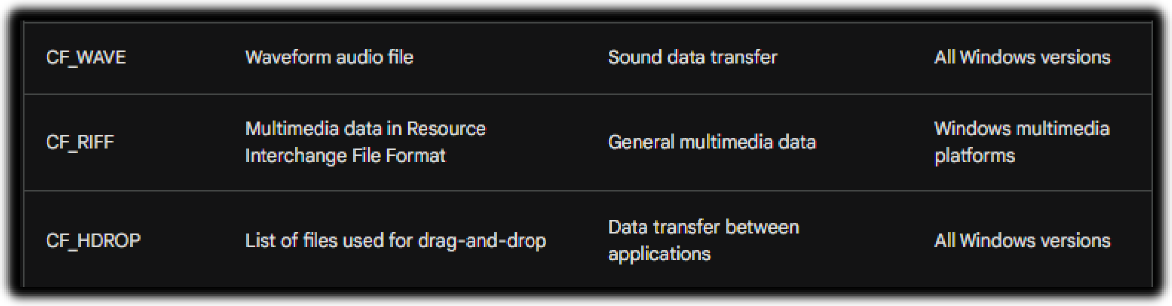
- **User is Boss:** A program should **never** touch the clipboard unless the user explicitly asks it to (like clicking "Paste"). You don't want programs changing your clipboard data behind your back!
- **Format Matching:** If you copy a picture, but try to paste it into a program that only handles text (like Notepad), the "Paste" button should be grayed out. The program checks the "Format" before it tries to grab the data.
- **Global Access:** Any program can talk to the Clipboard. This is why you can copy a URL from a web browser and paste it into a Word document.

What’s Coming in this Chapter?

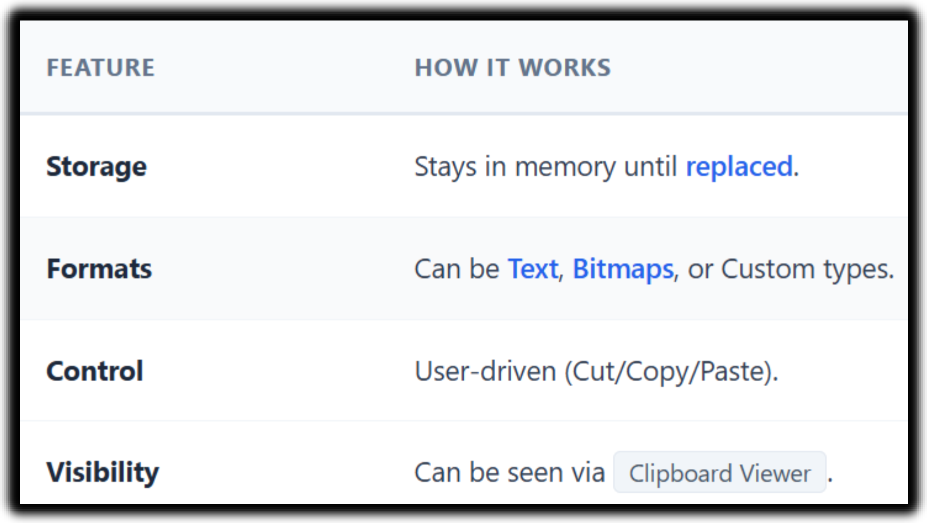
While the clipboard can hold complex things like photos (Bitmaps) or high-quality drawings (Metafiles), this chapter focuses on the most basic building block: **Text**.

- How to "Open" the clipboard.
- How to "Empty" it to make room for new stuff.
- How to "Give" it text and "Take" text back.

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



FEATURE	HOW IT WORKS
Storage	Stays in memory until replaced.
Formats	Can be Text, Bitmaps, or Custom types.
Control	User-driven (Cut/Copy/Paste).
Visibility	Can be seen via `Clipboard Viewer`.

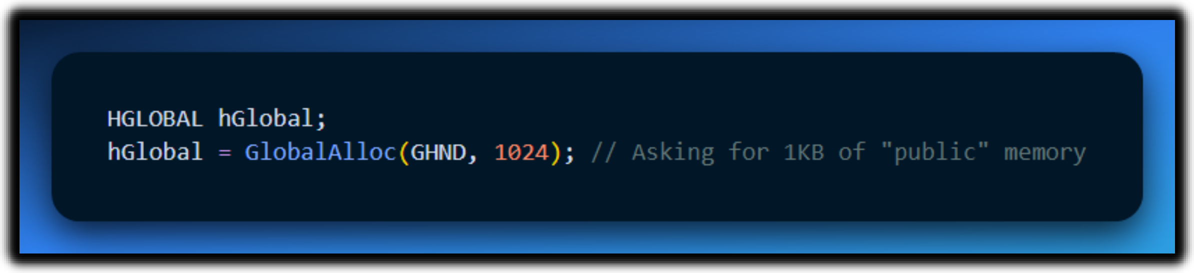
Why Global Memory?

Imagine your program's memory is like your private house. If you put data there, other programs (like Chrome or Word) can't get in to see it. **Global Memory** is like a public storage unit. Windows gives you a "key" (called a **Handle**) that you can hand off to other programs so they can access the data.

1. The Main Tool: GlobalAlloc

To get a piece of this public memory, you use the function GlobalAlloc. It takes two pieces of information:

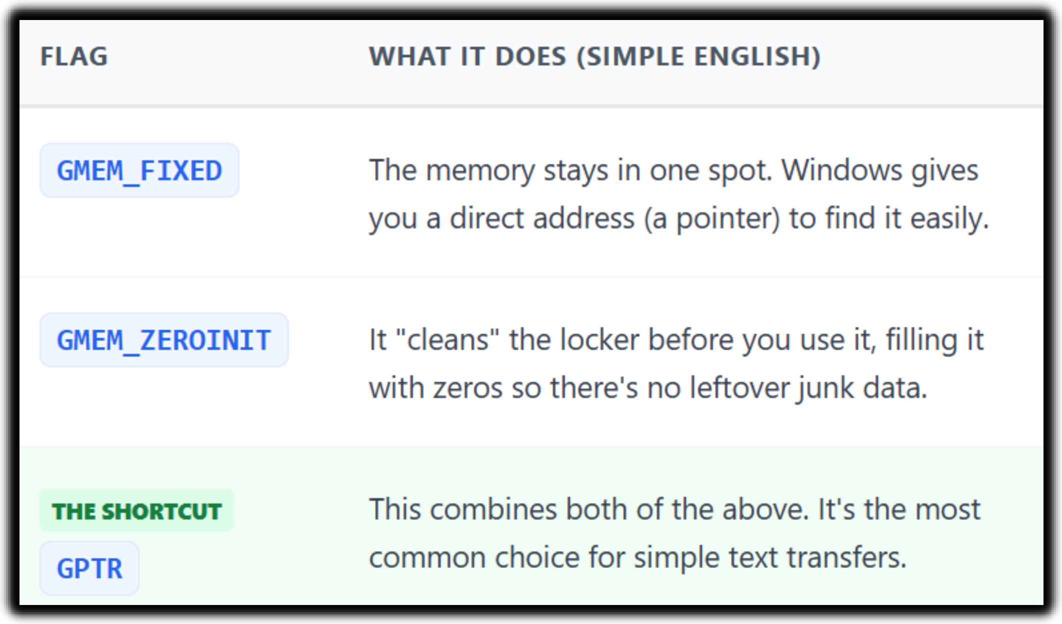
1. **How to act** (Flags)
2. **How much space you need** (Size in bytes)



```
HGLOBAL hGlobal;
hGlobal = GlobalAlloc(GHND, 1024); // Asking for 1KB of "public" memory
```

2. The "Cheat Sheet" for Flags

When you call GlobalAlloc, you use flags to tell Windows exactly what kind of memory "locker" you want.



FLAG	WHAT IT DOES (SIMPLE ENGLISH)
`GMEM_FIXED`	The memory stays in one spot. Windows gives you a direct address (a pointer) to find it easily.
`GMEM_ZEROINIT`	It "cleans" the locker before you use it, filling it with zeros so there's no leftover junk data.
THE SHORTCUT `GPTR`	This combines both of the above. It's the most common choice for simple text transfers.

3. Handles vs. Pointers (The "Key" vs. The "Address")

This is a bit tricky, but important:

- **The Handle (HGLOBAL):** This is the **ID Number** of the memory block. This is what you actually "hand" to the Clipboard.
- **The Pointer:** This is the **exact location** where you type your data.

Analogy: If you want to put a suitcase in a hotel locker, the **Handle** is your claim ticket number. The **Pointer** is the actual physical shelf where you lay the suitcase down.

If GlobalAlloc returns **NULL**, it means the computer is out of memory. Always check for this so your program doesn't crash!

The "Public Locker" Maintenance Kit

Once you have a piece of global memory (your "public locker"), you need to know how to manage it. Think of these four functions as your locker maintenance tools:

1. GlobalAlloc (Rent the locker)

This is where you start. You ask Windows for a specific amount of space. If it works, you get a **Handle** (your claim ticket).

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

2. GlobalReAlloc (Resize the locker)

What if you started writing a long poem to the clipboard, but realized you didn't rent enough space?

- Instead of throwing everything away and starting over, you use this to **expand** the memory block.
- It keeps the data you already wrote and just adds more "shelving" at the end.

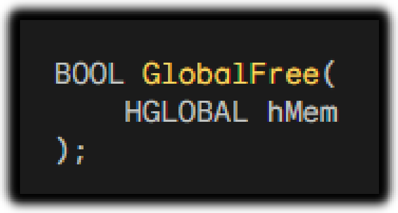
```
DWORD GlobalSize(
    HGLOBAL hMem
);
```

3. GlobalSize (Check the locker size)

If you find a random "claim ticket" (Handle) and don't remember how much space you rented, this function tells you exactly how many bytes are inside. It's a quick way to double-check your work.

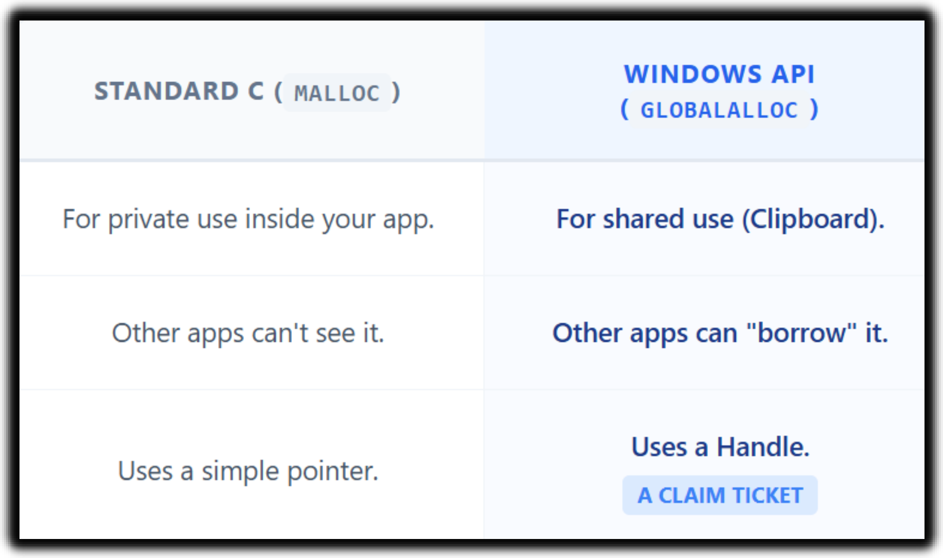
4. GlobalFree (Return the locker)

This is the most important one. When you are done using a piece of memory—and you haven't handed it off to the Clipboard yet—you **must** free it. If you don't, the computer stays "rented out" forever, which is what people mean when they say a program has a **memory leak**.



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

Why bother with these instead of malloc?



STANDARD C (MALLOC)	WINDOWS API (GLOBALALLOC)
For private use inside your app.	For shared use (Clipboard).
Other apps can't see it.	Other apps can "borrow" it.
Uses a simple pointer.	Uses a Handle. A CLAIM TICKET

I. The Big Rule of the Clipboard

Once you give a memory handle to the Clipboard using SetClipboardData, **you stop owning it.** You don't need to call GlobalFree anymore because the Clipboard now owns that memory. It will "clean up" the locker itself once someone else copies something new.

II. Fixed vs. Movable: The "Hotel Room" Analogy

To understand this, imagine your computer's memory is a giant hotel.

- **Fixed Memory (GMEM_FIXED):** This is like a guest who refuses to move. Even if the hotel wants to renovate or clear a whole floor, that guest stays in Room 101. Eventually, if you have too many "fixed" guests scattered around, the hotel can't fit a large tour group anywhere because no one will move to a different room.
- **Movable Memory (GMEM_MOVEABLE):** This is like a guest who doesn't care which room they are in. The hotel manager can move them from Room 101 to Room 505 whenever they want. This keeps the hotel organized and prevents "fragmentation" (having lots of tiny, useless gaps).

III. Why does the Clipboard care?

The Clipboard is the ultimate "Public Space." Because many different apps need to touch that memory, Windows prefers it to be **Movable**.

1. **Old Windows (16-bit):** You *had* to use movable memory. If you used fixed memory, the system would slow down or even crash because it couldn't reorganize itself.
2. **Modern Windows (32-bit/64-bit):** Computers are much smarter now. Even if you say memory is "Fixed," Windows can technically move it behind the scenes using "Virtual Memory."

However, for the Clipboard, the rule is still: **Use Movable.** It's the "polite" way to code. It tells Windows, "Here is some data for the clipboard; feel free to move it around to keep the system fast."

IV. The "Lock and Key" Routine

There is one catch: If Windows can move your data at any time, how do you type into it? If you start typing and Windows moves the "paper" halfway through, you'll leave a mess!

To solve this, we use a two-step process:

1. **GlobalLock:** You tell Windows, "Stop! Hold this memory still for a second." Windows gives you a **Pointer** (the actual address).
2. **GlobalUnlock:** Once you're done typing, you tell Windows, "Okay, I'm done. You can move it again if you need to."

V. The "Share" Flag: GMEM_SHARE

The notes mentioned GMEM_SHARE. In the old days, you had to explicitly say "I want to share this."

In modern Windows, this is mostly handled for you when you use the Clipboard, but it's still good practice to know that clipboard memory is **Shared Property.**

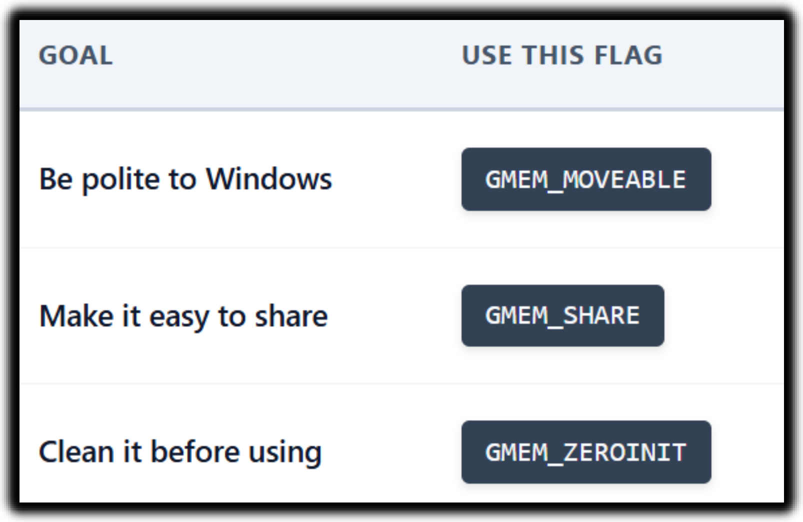
Once you hand it over, it's not yours anymore!

If you are writing code to put text on the clipboard today, you usually use a combination of these flags:

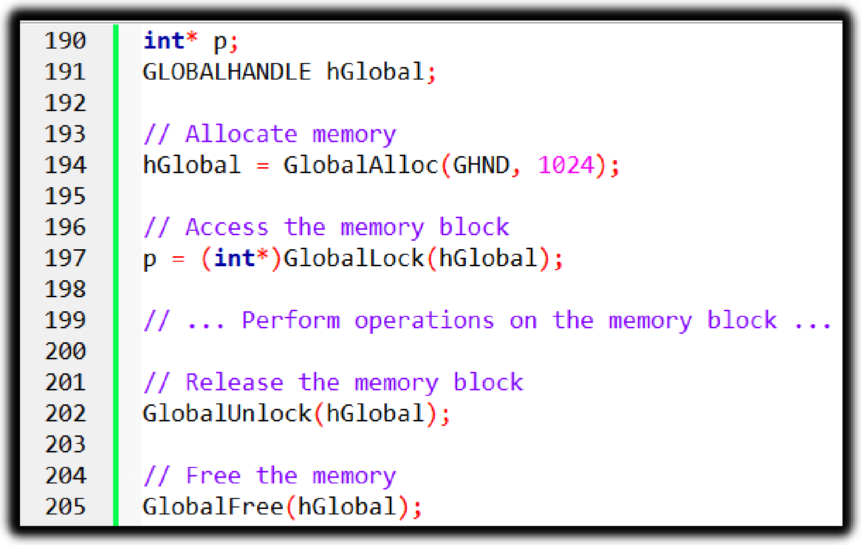
GHND: This is a shortcut for **G**lobal **Hand**le.

It combines GMEM_MOVEABLE and GMEM_ZEROINIT.

It's the "Standard" way to get clipboard memory.



GOAL	USE THIS FLAG
Be polite to Windows	GMEM_MOVEABLE
Make it easy to share	GMEM_SHARE
Clean it before using	GMEM_ZEROINIT



```
int* p;
GLOBALHANDLE hGlobal;

// Allocate memory
hGlobal = GlobalAlloc(GHND, 1024);

// Access the memory block
p = (int*)GlobalLock(hGlobal);

// ... Perform operations on the memory block ...

// Release the memory block
GlobalUnlock(hGlobal);

// Free the memory
GlobalFree(hGlobal);
```

VI. The "Lock and Key" Process

Since a **Handle** is just a "Claim Ticket" and not the actual memory address, you have to follow this workflow:

1. **Locking (GlobalLock):** This is like shouting "Freeze!" to Windows. It converts the **Handle** into a **Pointer** (an actual address you can use). While it's locked, Windows promises not to move that block in virtual memory.
2. **Using:** You type your data into that pointer address.
3. **Unlocking (GlobalUnlock):** This is like saying "At ease." It tells Windows you are finished, and the memory block is now free to be moved around again to keep the system organized.

VII. Pro-Tips for Memory Management

- **The "One Message" Rule:** Don't keep a memory block locked for a long time. Lock it, do your work (like copying text), and unlock it immediately within the same function. This keeps Windows running smoothly.
- **The Lock Count:** Windows keeps a tally. If you call GlobalLock twice, you must call GlobalUnlock twice before the memory is actually movable again.
- **Handle vs. Pointer:** When it comes time to delete the memory (GlobalFree), you **must** use the **Handle** (the claim ticket), not the pointer. If you lose the handle but have the pointer, you can use GlobalHandle(p) to find the ticket number again.

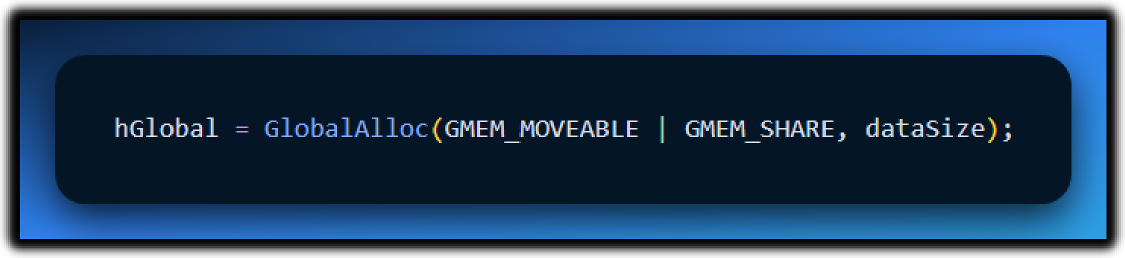
VIII. Why bother in 32-bit Windows?

Even though modern computers have plenty of RAM, we still use **Movable** memory for the Clipboard for two reasons:

1. **Defragmentation:** It allows Windows to slide memory blocks around like Tetris pieces to make room for bigger tasks.
2. **Compatibility:** The Clipboard is a shared space. Using GMEM_MOVEABLE and GMEM_SHARE is the "standard" way apps agree to talk to each other.

IX. The "Standard" Code Line

To get memory specifically for the clipboard, this is the line most programmers use:



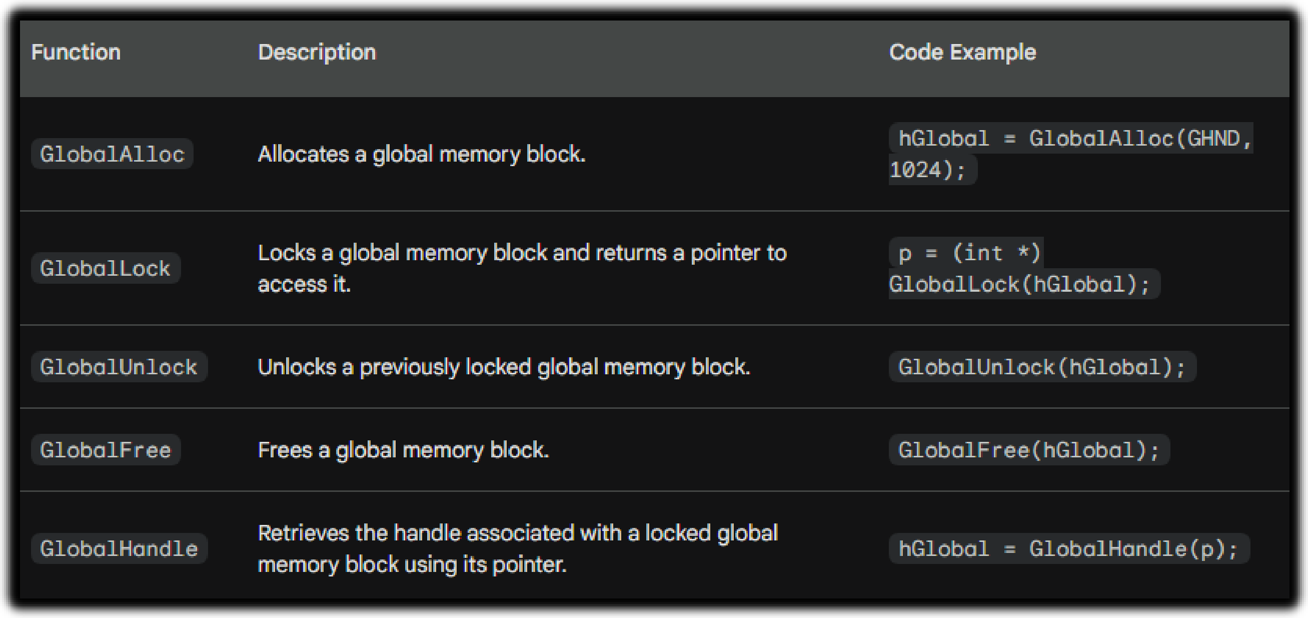
```
hGlobal = GlobalAlloc(GMEM_MOVEABLE | GMEM_SHARE, dataSize);
```

(Note: Many programmers also use the shortcut GHND, which does almost the same thing!)

In Short:

- **Handle:** The ID number for the "public locker."
- **GlobalLock:** Getting the physical key to open the locker.
- **GlobalUnlock:** Putting the key back so the janitor (Windows) can move the locker if needed.

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

The 4-Step Clipboard Handshake

When you want to "Copy" text, your program has to follow this exact sequence. If you skip a step, the program will likely crash or the data won't show up.

Step 1: Prepare the Memory

You can't just hand the clipboard a standard variable. You have to put your text into a "public locker" (Global Memory).

- **Allocate:** Use GlobalAlloc. Make sure to add +1 to the length for the "Null Terminator" (the invisible character that tells Windows "this is the end of the text").
- **Lock & Copy:** Lock the memory to get a pointer, copy your text into it, and **Unlock** it immediately.

Rule: Never give the clipboard a *locked* handle. It's like giving someone a locked suitcase without the key.

Step 2: Open and Clear

Before you can put something in, you have to "claim" the clipboard and clear out whatever was there before (like a previous "Copy" from another app).

- OpenClipboard(hwnd)
- EmptyClipboard()

Step 3: The Hand-Off

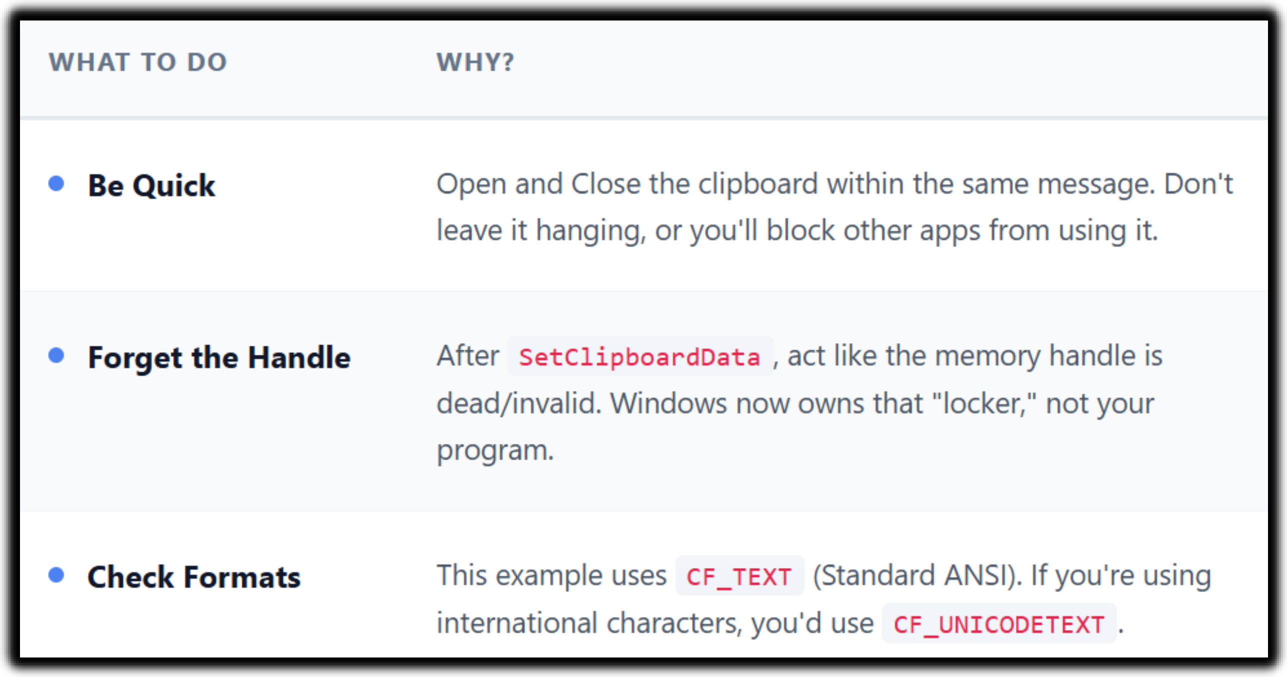
This is the moment where ownership changes.

- SetClipboardData(CF_TEXT, hGlobal)
- **Crucial Concept:** Once you run this line, **the memory is no longer yours.** The Clipboard now owns that "locker." You should not try to edit or free that memory ever again.

Step 4: Close the Door

- CloseClipboard()
- Always close it immediately. If you leave it open, no other program on the computer (like Word or Chrome) will be able to use the "Copy/Paste" functions.

Important Rules for "Clean" Code



WHAT TO DO	WHY?
• **Be Quick**	Open and Close the clipboard within the same message. Don't leave it hanging, or you'll block other apps from using it.
• **Forget the Handle**	After `SetClipboardData`, act like the memory handle is dead/invalid. Windows now owns that "locker," not your program.
• **Check Formats**	This example uses `CF_TEXT` (Standard ANSI). If you're using international characters, you'd use `CF_UNICODETEXT`.

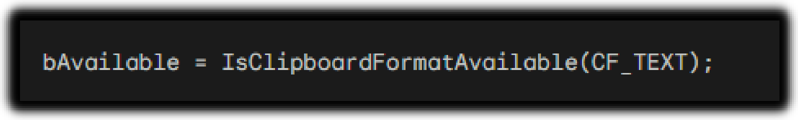
The "Lazy" Summary

1. **Rent** a public memory block and **fill** it with your text.
2. **Unlock** it.
3. **Open** the clipboard and **Wipe** it clean.
4. **Hand over** the memory and **Close** the clipboard.

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.

The "Only One" Rule (Exclusive Access)

The Clipboard is a shared resource, but it has a "one-at-a-time" policy.

- **The Lock:** When you call OpenClipboard, you are putting a lock on it. If another program (like Excel) currently has it open, your call will fail (return FALSE).
- **The Responsibility:** Because you are locking a system-wide feature, you must **close it immediately**.
- **The Danger Zone:** Never leave the clipboard open while showing a **Message Box** or a **Dialog Box**. If the user switches to another app while your pop-up is on the screen, the other app's "Copy/Paste" will be broken because you still hold the lock.

I. Windows: The Invisible Translator

One of the coolest things about the clipboard is that it handles **Unicode conversion** for you automatically.

- **Format Synergy:** If you put CF_UNICODETEXT (Unicode) onto the clipboard, Windows will automatically create CF_TEXT (ANSI) and CF_OEMTEXT (Old DOS style) versions in the background.
- **The Benefit:** This means a modern Unicode program and an old 1990s ANSI program can still copy and paste text to each other without you writing a single line of conversion code.

II. CLIPTEXT Logic: How the Program Thinks

The CLIPTEXT program is a simple window that displays a string. It uses the clipboard to change that string. Here is the "Big Picture" of its message handling:

WM_INITMENUPOPUP (Smart Menus)

Before you even click a menu, the program checks two things:

1. **Is there text in the Clipboard?** If no, "Paste" is grayed out.
2. **Is my window empty?** If yes, "Copy" and "Cut" are grayed out.

IDM_EDIT_PASTE (The Retrieval)

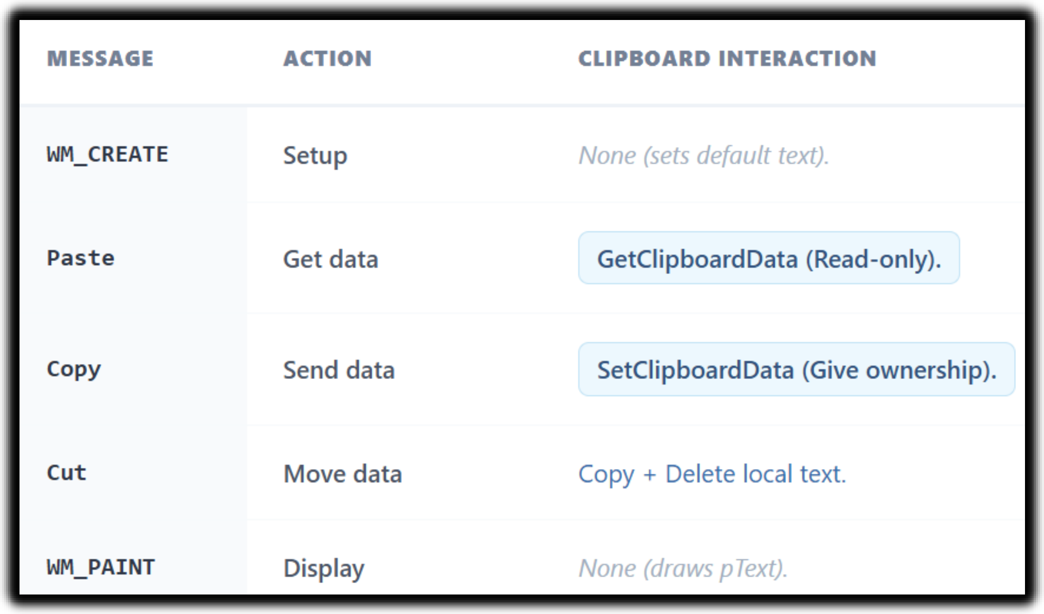
1. Open the clipboard.
2. Call GetClipboardData. **Crucial:** You don't "own" the handle Windows gives you here. You just "borrow" it.
3. GlobalLock the handle, copy the text into your own local pText variable, then GlobalUnlock.
4. Close the clipboard and tell the window to repaint (InvalidateRect).

IDM_EDIT_COPY (The Hand-off)

1. Allocate global memory (GlobalAlloc).
2. Copy your local pText into that global block.
3. OpenClipboard and EmptyClipboard (You must empty it before you can set new data).
4. SetClipboardData and **CloseClipboard**.

III. Key Takeaways for Developers

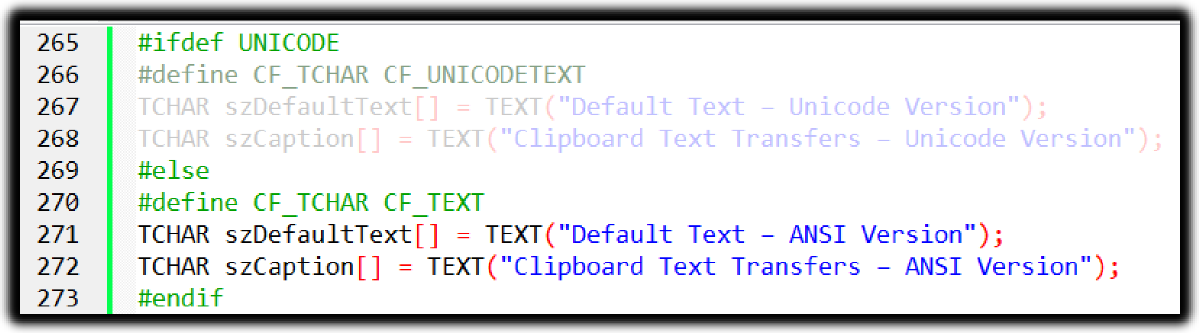
- **Global vs. Local:** Use malloc for the text you want to keep inside your program. Use GlobalAlloc only for the text you are giving away to the clipboard.
- **Format Choice:** Use the TCHAR approach. If the UNICODE flag is on, use CF_UNICODETEXT. If not, use CF_TEXT.
- **Invalidate:** Always call InvalidateRect after a Paste or a Cut. The data has changed, so the screen needs to be "redrawn" to show the new (or missing) text.



MESSAGE	ACTION	CLIPBOARD INTERACTION
WM_CREATE	Setup	*None (sets default text).*
Paste	Get data	GetClipboardData (Read-only).
Copy	Send data	SetClipboardData (Give ownership).
Cut	Move data	Copy + Delete local text.
WM_PAINT	Display	*None (draws pText).*

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:



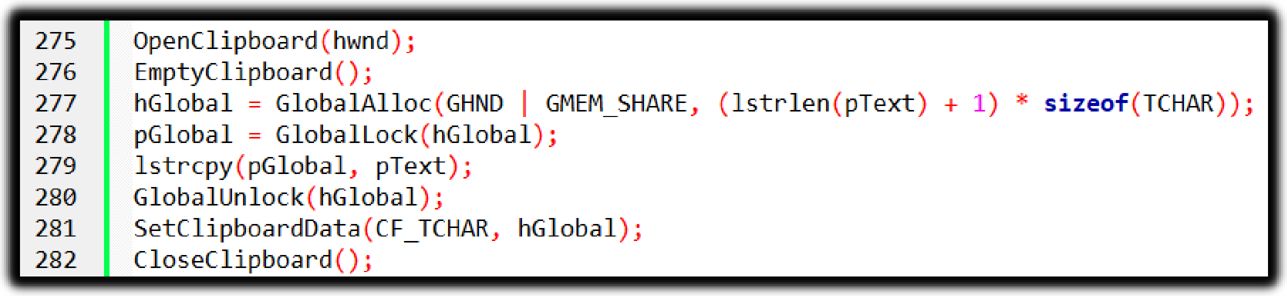
```
#ifdef UNICODE
#define CF_TCHAR CF_UNICODETEXT
TCHAR szDefaultText[] = TEXT("Default Text - Unicode Version");
TCHAR szCaption[] = TEXT("Clipboard Text Transfers - Unicode Version");
#else
#define CF_TCHAR CF_TEXT
TCHAR szDefaultText[] = TEXT("Default Text - ANSI Version");
TCHAR szCaption[] = TEXT("Clipboard Text Transfers - ANSI Version");
#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:

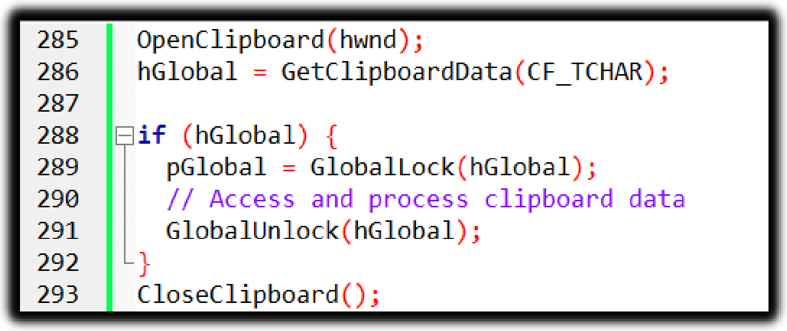
Setting Clipboard Data:



```
OpenClipboard(hwnd);
EmptyClipboard();
hGlobal = GlobalAlloc(GHND | GMEM_SHARE, (lstrlen(pText) + 1) * sizeof(TCHAR));
pGlobal = GlobalLock(hGlobal);
lstrcpy(pGlobal, pText);
GlobalUnlock(hGlobal);
SetClipboardData(CF_TCHAR, hGlobal);
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:



```
OpenClipboard(hwnd);
hGlobal = GetClipboardData(CF_TCHAR);

if (hGlobal) {
    pGlobal = GlobalLock(hGlobal);
    // Access and process clipboard data
    GlobalUnlock(hGlobal);
}
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.

The "Invisible" Conversion

The coolest part of the CLIPTEXT experiment is running two copies of the program: one compiled as **Unicode** and one as **ANSI**.

- **The Experiment:** Copy text from the Unicode version and paste it into the ANSI version.
- **What's Happening:** You didn't write any conversion code, but it still works. When you call SetClipboardData(CF_UNICODETEXT, ...) in the first app, Windows creates a hidden CF_TEXT version. When the second app calls GetClipboardData(CF_TEXT), Windows hands over that translated version automatically.

1. Smart Menu Logic (bEnable)

A professional app doesn't let you click "Paste" if there is nothing to paste. CLIPTEXT uses a simple flag system:

WM_INITMENUPOPUP: This is the trigger. Right before the menu drops down, the program asks:

- "Is there a string in my memory?" (If yes, enable **Copy** and **Cut**).
- "Is there text on the clipboard right now?" (Use IsClipboardFormatAvailable(CF_TEXT)—if yes, enable **Paste**).

2. Beyond the Basics: Pro Features

Once you master "Text," the clipboard can do much more. Here are the three "Next Level" concepts:

A. Clipboard Viewers (Monitoring)

You can turn your program into a **Clipboard Viewer**.

- **How:** You call SetClipboardViewer.
- **The Result:** Windows puts your app on a "notification list." Every time *any* other program in Windows hits "Copy," your app gets a WM_DRAWCLIPBOARD message. This is how "Clipboard Manager" apps work.

B. Custom Data Formats

If you are building a specialized app (like a CAD program or a Music Tracker), CF_TEXT isn't enough.

- **The Fix:** Use RegisterClipboardFormat("MY_AWESOME_FORMAT").
- **The Power:** You can now pass complex C++ structures or objects through the clipboard, and only programs that "know" your custom format will be able to read them.

C. Delayed Rendering (The "Lazy" Copy)

Imagine copying a 500MB high-resolution image. You don't want to shove that into memory unless the user actually hits "Paste."

- **The Hack:** You call SetClipboardData(CF_BITMAP, NULL).
- **The Handshake:** You are telling Windows: "I have the data, but I'm holding it." If another app asks for it, Windows sends your app a WM_RENDERFORMAT message, and *then* you provide the data.

Final Clipboard Golden Rules:

1. **Be a good citizen:** Open, do your work, and **Close** the clipboard instantly.
2. **Handle Ownership:** Once you SetClipboardData, that memory handle belongs to Windows. Forget it exists.
3. **Use the Right Format:** Always check IsClipboardFormatAvailable before trying to Paste.

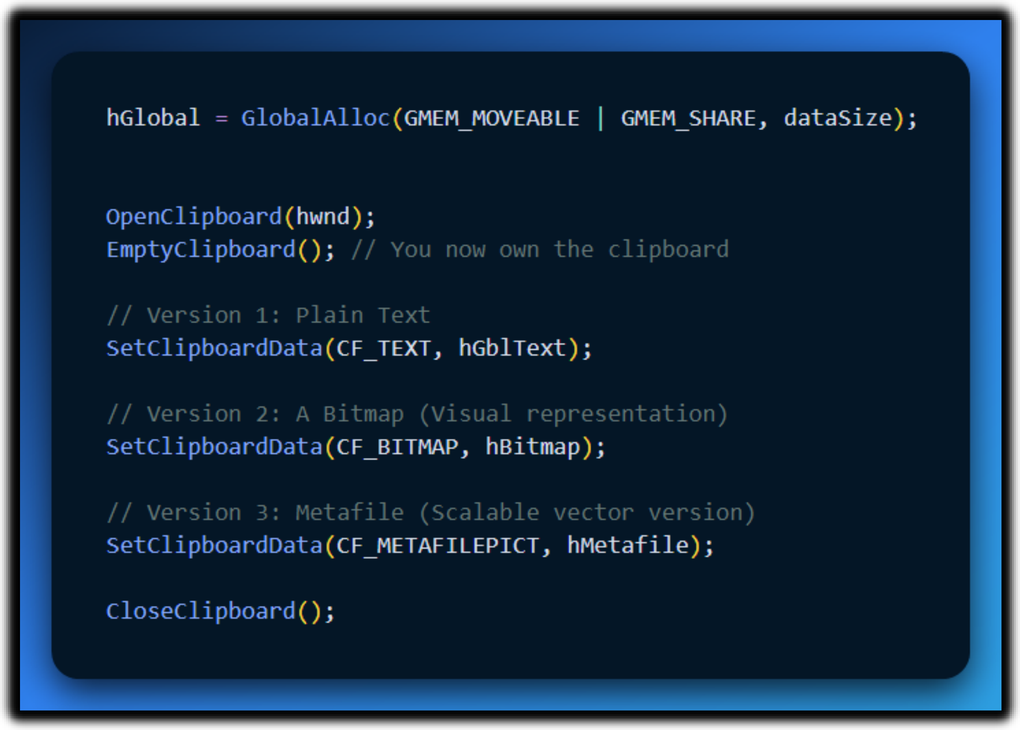
USING MULTIPLE DATA ITEMS WITH THE CLIPBOARD

1. Setting the Stage (Copying Multiple Formats)

When you "Copy" in a sophisticated app (like Excel), the app doesn't just put text on the clipboard. It puts a dozen things there at once.

The Workflow:

1. **Open & Empty:** You must call EmptyClipboard() first. This makes you the "owner" of the current clipboard session.
2. **The "Shotgun" Approach:** You call SetClipboardData() multiple times in a row, once for each format you can provide.

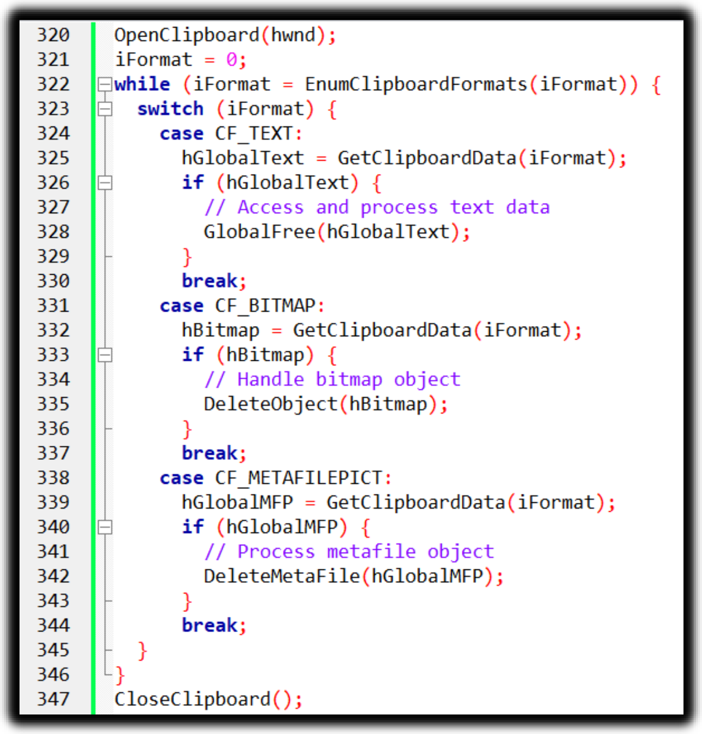


```
hGlobal = GlobalAlloc(GMEM_MOVEABLE | GMEM_SHARE, dataSize);


OpenClipboard(hwnd);
EmptyClipboard(); // You now own the clipboard

// Version 1: Plain Text
SetClipboardData(CF_TEXT, hGblText);

// Version 2: A Bitmap (Visual representation)
SetClipboardData(CF_BITMAP, hBitmap);

// Version 3: Metafile (Scalable vector version)
SetClipboardData(CF_METAFILEPICT, hMetafile);

CloseClipboard();
```

2. Accessing numerous formats / The Retrieval (Pasting the Best Format)

When a program wants to "Paste," it doesn't just grab whatever is on top. It looks for the "richest" format it can handle.

- **Enumeration:** You can use EnumClipboardFormats to loop through everything currently in the "locker."
- **Priority:** Usually, you don't loop. You just ask: *"Is there a Metafile? No? Okay, is there a Bitmap? No? Fine, I'll just take the Text."*



```
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();
```

The Golden Rule of Pasting: When you call GetClipboardData, you are getting a handle to memory that **belongs to the system.**

You can look at it, and you can copy data *out* of it, but you must **never** delete it. If you delete that handle, the clipboard becomes corrupted, and other apps will crash if they try to paste.

---

3. Automatic "Cheat" Formats

Windows is actually working behind the scenes to help you. If you only provide one format, Windows often synthesizes others:

- If you provide **CF_TEXT**, Windows automatically creates **CF_OEMTEXT** and **CF_UNICODETEXT**.
- If you provide **CF_BITMAP**, Windows can often provide a **CF_DIB** (Device Independent Bitmap).

4. Summary for Developers

- **On Copy:** Provide as many formats as you can (Text, RTF, HTML, Bitmap). It makes your app play better with others.
- **On Paste:** Look for the most complex format you support first, and fall back to plain text as a last resort.
- **On Cleanup:** Only delete handles you created that **failed** to get into the clipboard. Once SetClipboardData is called, it's not your problem anymore.

DELAYED RENDERING FOR EFFICIENT CLIPBOARD MANAGEMENT

Delayed Rendering lets you say to Windows: "I have this data ready if anyone wants it, but I'm not going to build it yet."

The Handshake (How it works)

1. The "IOU" (SetClipboardData with NULL)

When the user clicks "Copy," you open the clipboard and call SetClipboardData(CF_TEXT, NULL).

- You aren't giving Windows the text.
- You are giving Windows an **IOU**. You are now the "Clipboard Owner."

2. The Request (WM_RENDERFORMAT)

Nothing happens until someone else hits "Paste." At that moment, Windows realizes it has an IOU but no data. It sends your program a WM_RENDERFORMAT message.

- **Your Job:** Now you actually do the work. Allocate the memory, fill it with the data, and call SetClipboardData again—but this time with the real handle.

Cleaning Up and Closing Down

3. The Eviction (WM_DESTROYCLIPBOARD)

If the user goes to another program and copies something else, your IOU is no longer valid. Windows sends you WM_DESTROYCLIPBOARD.

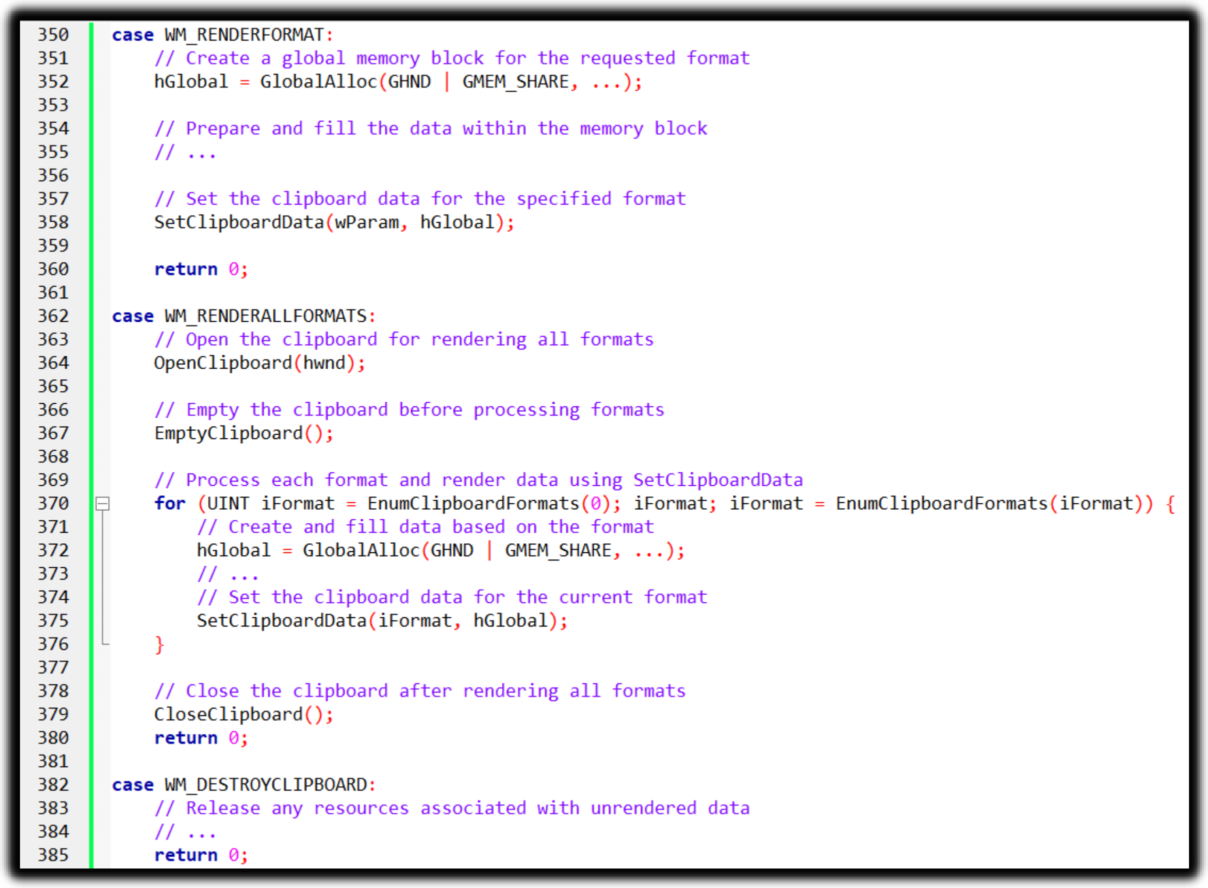
- This is your signal to stop worrying about that data. You don't need to provide it anymore.

4. The Final Call (WM_RENDERALLFORMATS)

What if your program is closing, but you still "owe" Windows data on the clipboard?

- Windows sends you WM_RENDERALLFORMATS.
- You must now quickly "render" all those IOUs into real data before your program disappears from memory. This ensures that even after your app is closed, the user can still paste what they copied.

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;
```

The "Pro" Shortcuts for Delayed Rendering

The notes mention a few "cheats" that real-world developers use to keep their code simple:

I. Combining Messages

If your program only handles one type of data (like just Text), you don't need two separate logic blocks for WM_RENDERFORMAT and WM_RENDERALLFORMATS. You can point them both to the same function. Whether someone clicked "Paste" or your program is "Closing," the action is the same: **Give Windows the data.**

II. When to Ignore WM_DESTROYCLIPBOARD

The notes say this message is "optional." Here is why:

If your data is just a small string of text, who cares? Let it sit in your program's memory.

However, if you have a 1GB video file ready to be rendered, you *want* to listen for WM_DESTROYCLIPBOARD. Once you get that message, you can delete that massive temporary file and free up the user's computer.

Why Delayed Rendering is the Gold Standard

For small apps, this is overkill. But for **any** app that handles images, spreadsheets, or long documents, it is mandatory because:

1. **The Speedy Copy:** The user hits Ctrl+C and the app responds instantly because it hasn't actually "done" anything yet.
2. **System Health:** You aren't clogging up the "Global Memory" with data that might never be used.

PRIVATE DATA FORMATS: SHARING BEYOND STANDARD FORMATS

The "Secret Language" Formats

Standard formats like CF_TEXT are like speaking English—everyone understands them. **Private Formats** are like a secret code only your specific app understands.

I. The "I'll Do It Myself" Approach (CF_OWNERDISPLAY)

This is the most advanced way to use the clipboard.

- **The Deal:** You tell the clipboard, "I'm not going to give you a file or a string. I'm just going to draw the preview myself."
- **The Result:** When the user opens the Windows "Clipboard Viewer," Windows sends a message to **your** program saying, "Hey, draw something in this box."
- **Who uses this?** Excel uses this to draw those little grid lines and cell colors in the preview window that a standard text box couldn't show.

II. The "Half-and-Half" Approach (DSP Formats)

DSP stands for "Display." These are clever because they have a "Public face" and a "Private heart."

- You store data in a way that the **Clipboard Viewer** can still show a preview (like a basic picture or text).
- However, other programs will ignore it because it's tagged as "Private."
- It's perfect for when you want the user to see what they copied, but you don't want them pasting it into an app that would break it.

How to Tell "Is it mine?"

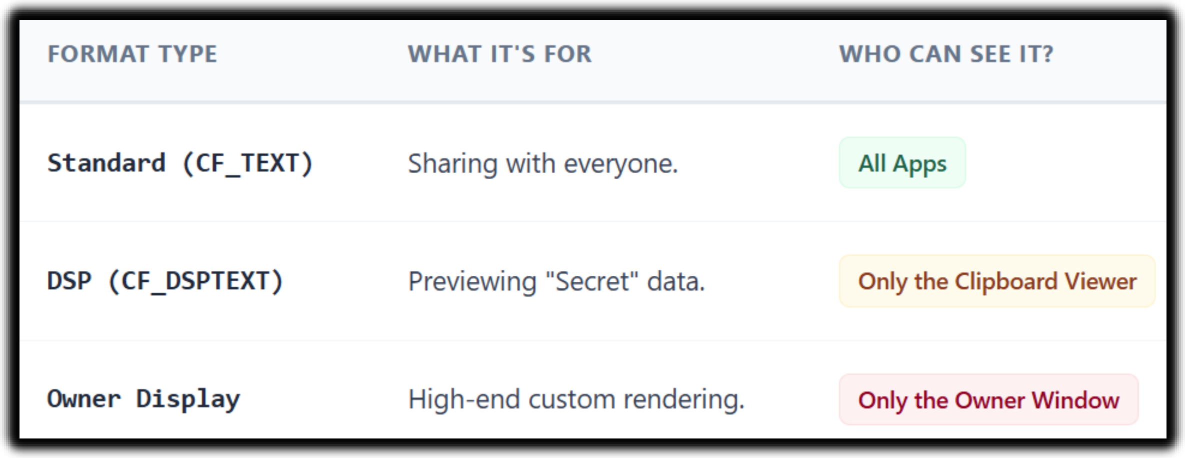
Sometimes you want your app to behave differently if the data on the clipboard came from **you** versus from **Notepad**.

1. **GetClipboardOwner**: This gives you the ID (Handle) of the window that put the data there.
2. **GetClassName**: You check that ID to see if the window class name matches your app.

Example: In a professional Graphics App, if you copy a "Circle" and paste it back into the same app, it stays a "Circle" (you can change its radius). If you paste it into Paint, it just becomes a flat "Picture" (pixels).

Why this matters for the Developer:

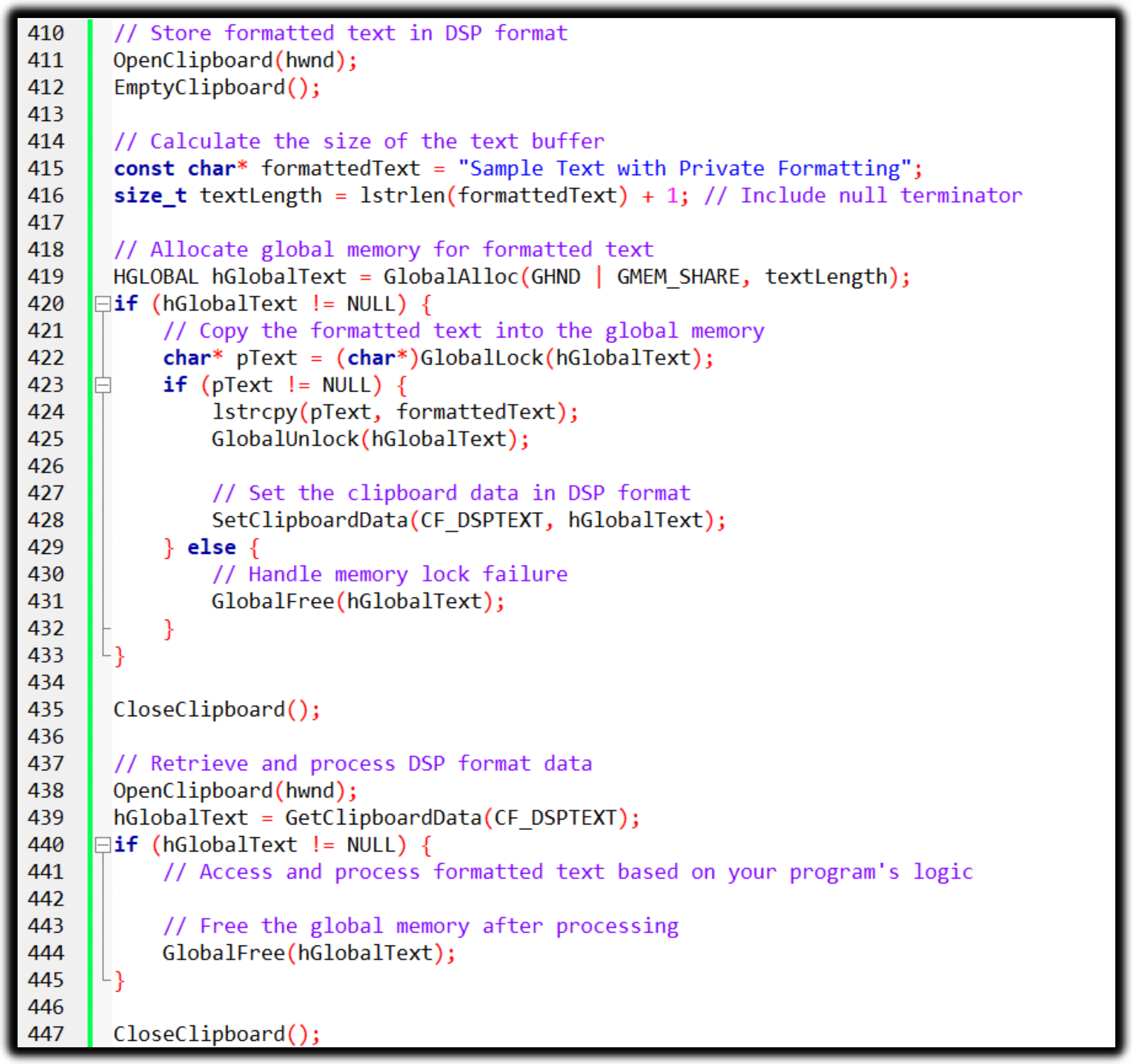
- **Modularity:** It allows you to pass whole "Objects" (like a 3D model or a spreadsheet formula) through the clipboard.
- **Professionalism:** Using CF_OWNERDISPLAY makes your app look integrated into Windows because its clipboard previews will look exactly like the actual app.



FORMAT TYPE	WHAT IT'S FOR	WHO CAN SEE IT?
`Standard (CF_TEXT)`	Sharing with everyone.	All Apps
`DSP (CF_DSPTEXT)`	Previewing "Secret" data.	Only the Clipboard Viewer
`Owner Display`	High-end custom rendering.	Only the Owner Window

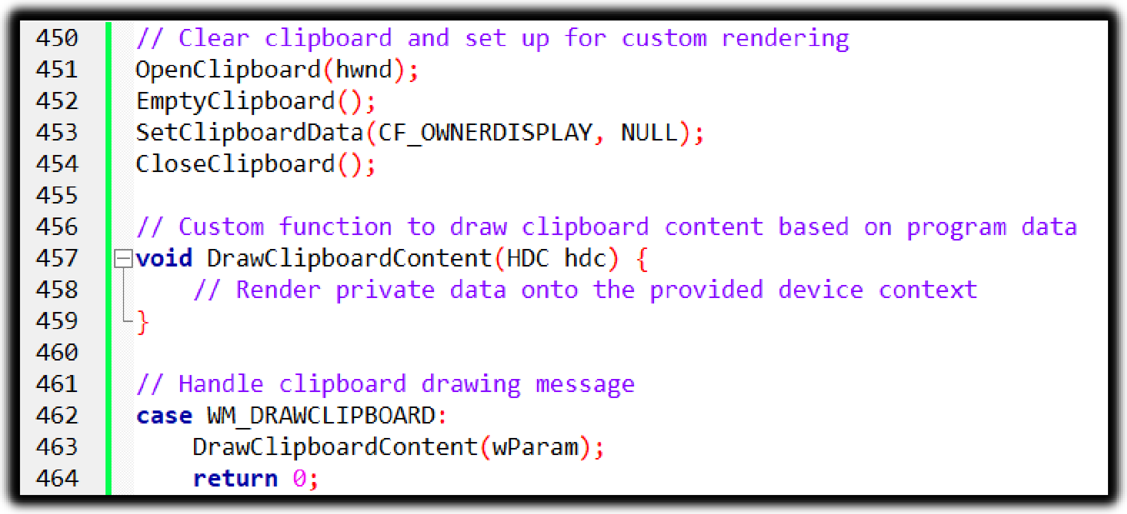
Code Examples:

I. Using DSP Format:



```
// Store formatted text in DSP format
OpenClipboard(hwnd);
EmptyClipboard();

// Calculate the size of the text buffer
const char* formattedText = "Sample Text with Private Formatting";
size_t textLength = lstrlen(formattedText) + 1; // Include null terminator

// Allocate global memory for formatted text
HGLOBAL hGlobalText = GlobalAlloc(GHND | GMEM_SHARE, textLength);
if (hGlobalText != NULL) {
    // Copy the formatted text into the global memory
    char* pText = (char*)GlobalLock(hGlobalText);
    if (pText != NULL) {
        lstrcpy(pText, formattedText);
        GlobalUnlock(hGlobalText);

        // Set the clipboard data in DSP format
        SetClipboardData(CF_DSPTEXT, hGlobalText);
    } else {
        // Handle memory lock failure
        GlobalFree(hGlobalText);
    }
}

CloseClipboard();

// Retrieve and process DSP format data
OpenClipboard(hwnd);
hGlobalText = GetClipboardData(CF_DSPTEXT);
if (hGlobalText != NULL) {
    // Access and process formatted text based on your program's logic

    // Free the global memory after processing
    GlobalFree(hGlobalText);
}

CloseClipboard();
```

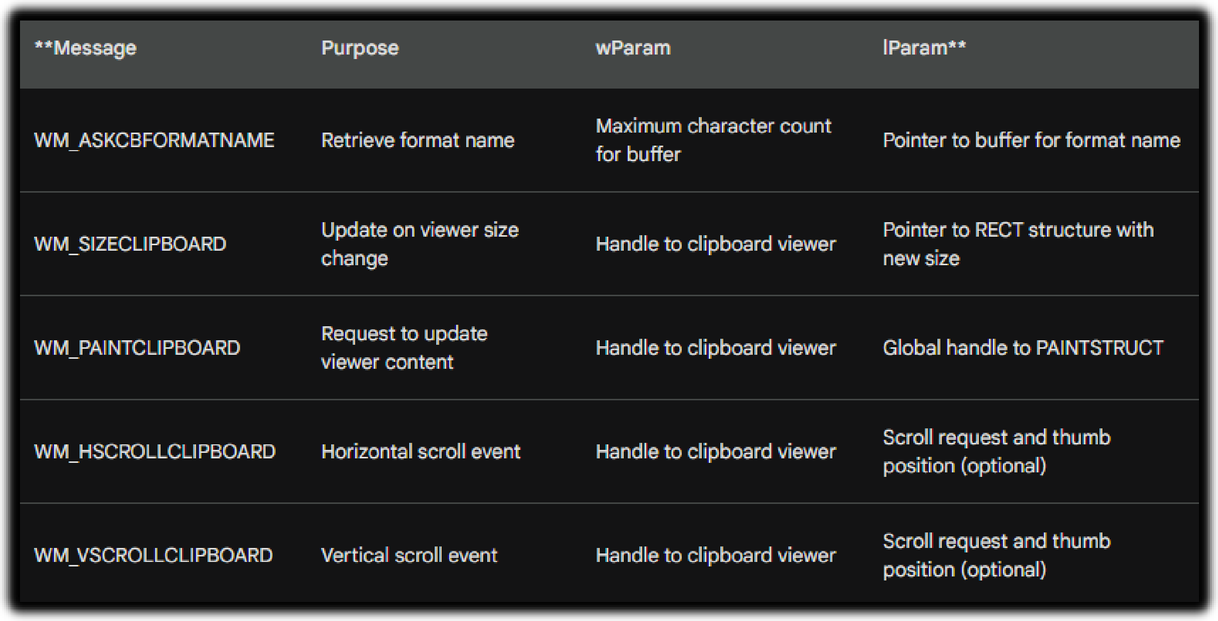
II. Using CF_OWNERDISPLAY:


```
// Clear clipboard and set up for custom rendering
OpenClipboard(hwnd);
EmptyClipboard();
SetClipboardData(CF_OWNERDISPLAY, NULL);
CloseClipboard();

// Custom function to draw clipboard content based on program data
void DrawClipboardContent(HDC hdc) {
    // Render private data onto the provided device context
}

// Handle clipboard drawing message
case WM_DRAWCLIPBOARD:
    DrawClipboardContent(wParam);
    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:

I. WM_ASKCBFORMATNAME (What's the name?)

The viewer asks: "What should I call this data in my menu?"

Your job: Copy a simple name (like "My Custom Graphic") into the buffer Windows provides.

II. WM_SIZECLIPBOARD (How big is the box?)

The viewer tells you: "The user just resized my window."

Your job: Take note of the new width and height so you know how much room you have to draw your preview.

III. WM_PAINTCLIPBOARD (Draw it now!)

This is the most important one. The viewer says: "Okay, the screen needs an update. Draw your data right here."

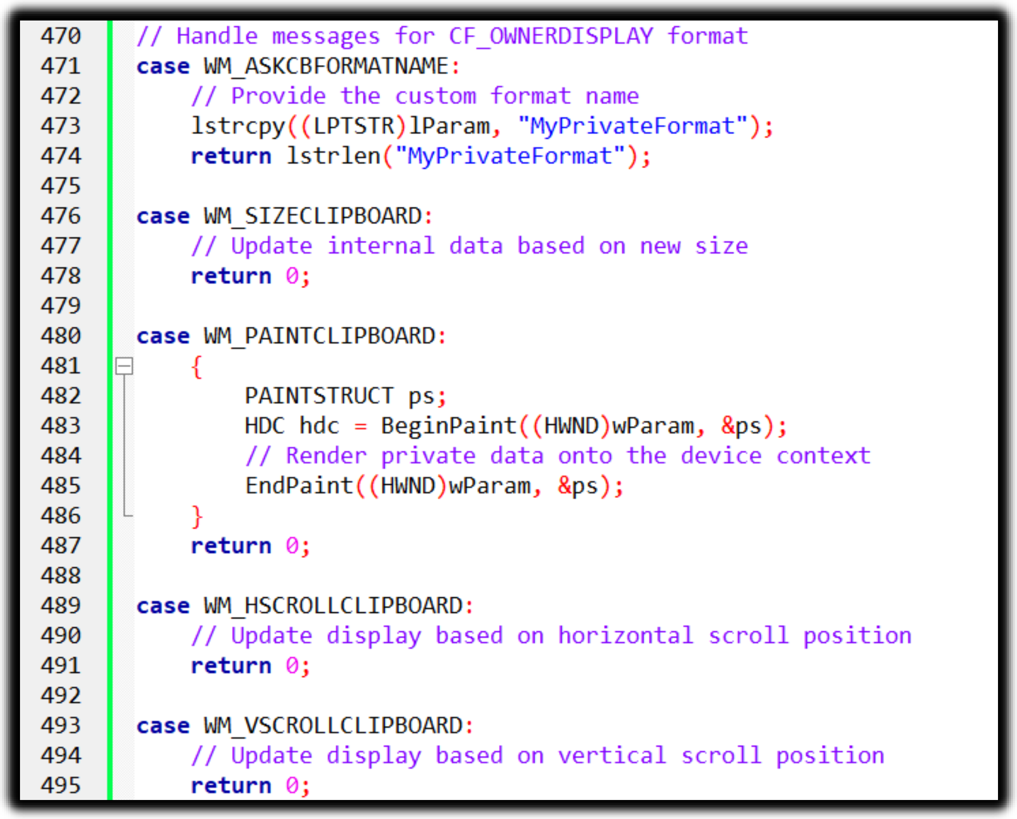
Your job: Grab the **Device Context** (the "canvas") from the message and use your standard drawing commands to show a preview of what was copied.

IV. WM_HSCROLL / WM_VSCROLLCLIPBOARD (Moving around)

The viewer says: "The user is clicking the scrollbars."

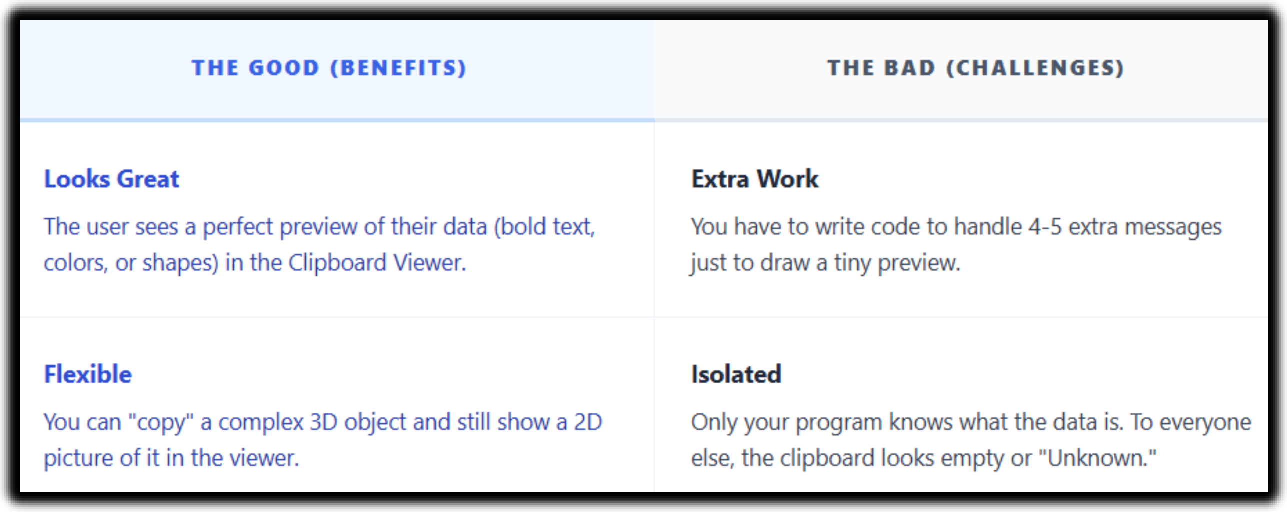
Your job: Shift the preview left, right, up, or down so the user can see the rest of the data.

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;
```

CF_OWNERDISPLAY



THE GOOD (BENEFITS)	THE BAD (CHALLENGES)
Looks Great The user sees a perfect preview of their data (bold text, colors, or shapes) in the Clipboard Viewer.	**Extra Work** You have to write code to handle 4-5 extra messages just to draw a tiny preview.
Flexible You can "copy" a complex 3D object and still show a 2D picture of it in the viewer.	**Isolated** Only your program knows what the data is. To everyone else, the clipboard looks empty or "Unknown."

BETTER ALTERNATIVES?

If CF_OWNERDISPLAY feels like too much work, you have two other choices:

1. DSP Formats (The "Halfway" House)

Use this if you want a preview without the heavy coding. You provide a standard format (like a Bitmap) but tag it as "Private."

Result: The viewer shows the picture, but other apps won't try to paste it.

2. Registered Formats (The "Public" Secret)

Use RegisterClipboardFormat if you want other developers to be able to use your data.

Result: If another company wants to support your app, they can "look" for your specific format name and read 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.

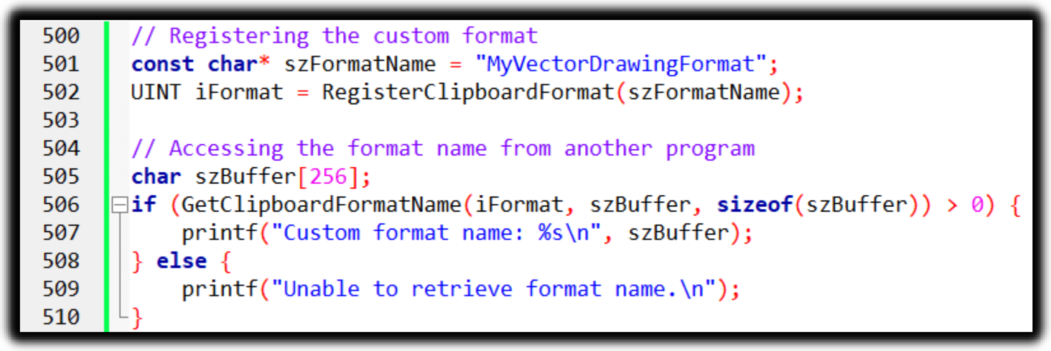
Sharing Format Information:

- 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:



```c
// Registering the custom format
const char* szFormatName = "MyVectorDrawingFormat";
UINT iFormat = RegisterClipboardFormat(szFormatName);

// Accessing the format name from another program
char szBuffer[256];
if (GetClipboardFormatName(iFormat, szBuffer, sizeof(szBuffer)) > 0) {
    printf("Custom format name: %s\n", szBuffer);
} else {
    printf("Unable to retrieve format name.\n");
}
```

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

In Windows, the clipboard doesn't broadcast to everyone at once. Instead, it uses a **linked list (the Chain)**. If you want to know when the clipboard changes, you have to "plug yourself in" at the front of the line.

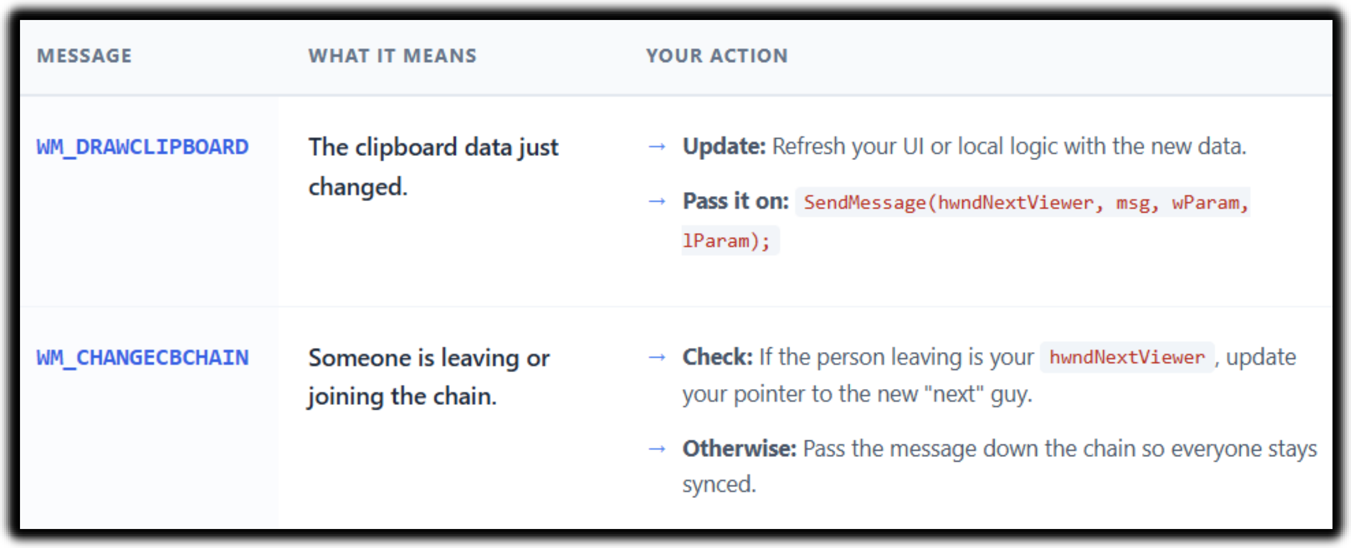
I. Plugging Into the Chain

To become the observer, you call SetClipboardViewer(hwnd).

- **The Swap:** Windows makes *you* the head of the chain.
- **The Link:** The function returns the HWND of the guy who *used* to be first.
- **Your Job:** You must save that HWND (let's call it hwndNextViewer). You are now responsible for him.

II. The Responsibility (Message Flow)

You aren't just a listener; you are a **relay station**. If you don't pass the message along, every program behind you in the chain "goes deaf."



MESSAGE	WHAT IT MEANS	YOUR ACTION
`WM_DRAWCLIPBOARD`	The clipboard data just changed.	→ **Update:** Refresh your UI or local logic with the new data. → **Pass it on:** `SendMessage(hwndNextViewer, msg, wParam, lParam);`
`WM_CHANGECBCHAIN`	Someone is leaving or joining the chain.	→ **Check:** If the person leaving is your `hwndNextViewer`, update your pointer to the new "next" guy. → **Otherwise:** Pass the message down the chain so everyone stays synced.

III. Leaving the Chain

When your program closes, you can't just vanish, or you'll leave a "hole" in the linked list. You must call ChangeClipboardChain(yourHwnd, hwndNextViewer). This tells Windows to stitch the person before you directly to the person after you.

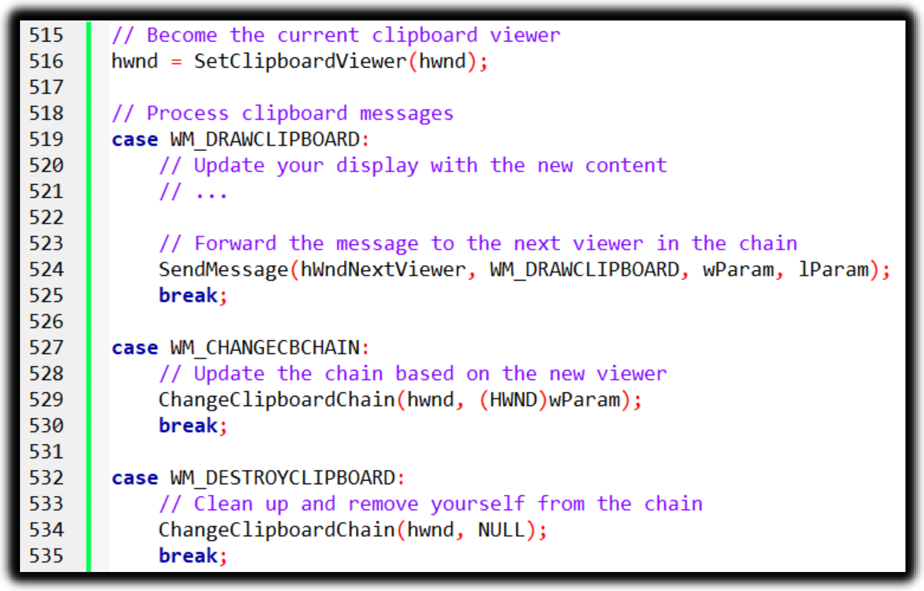
Refined Logic Flow

Think of it like a bucket brigade: >
1. Windows hands a bucket (WM_DRAWCLIPBOARD) to the first person.
2. That person looks inside, does their work, and hands it to the next person.
3. If someone in the middle drops the bucket (doesn't forward the message), the rest of the line gets nothing.

Key API Checklist

SetClipboardViewer: "Put me at the front."
ChangeClipboardChain: "I'm leaving; stitch the chain back together."
SendMessage: The tool used to keep the chain alive.

Example Code:



```
// Become the current clipboard viewer
hwnd = SetClipboardViewer(hwnd);

// Process clipboard messages
case WM_DRAWCLIPBOARD:
    // Update your display with the new content
    // ...

    // Forward the message to the next viewer in the chain
    SendMessage(hWndNextViewer, WM_DRAWCLIPBOARD, wParam, lParam);
    break;

case WM_CHANGECBCHAIN:
    // Update the chain based on the new viewer
    ChangeClipboardChain(hwnd, (HWND)wParam);
    break;

case WM_DESTROYCLIPBOARD:
    // Clean up and remove yourself from the chain
    ChangeClipboardChain(hwnd, NULL);
    break;
```

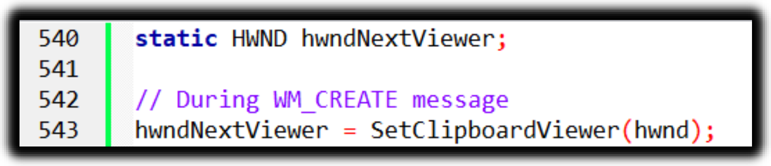
Clipboard Viewer Chain: In-Depth Analysis

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.

I. Joining the Chain: The "Handshake"

Think of becoming a clipboard viewer as inserting yourself at the front of a line. You don't just "listen"—you become the new gatekeeper.

- **The Registration (SetClipboardViewer):** You tell Windows, "I want to be the first to know about clipboard changes."
- **The Handoff:** Windows gives you the ID (HWND) of the program that *used* to be first.
- **The Secret Sauce:** You **must** save that ID (usually in a static variable like hwndNextViewer). If you lose it, you break the chain, and every program that joined before you stops working.

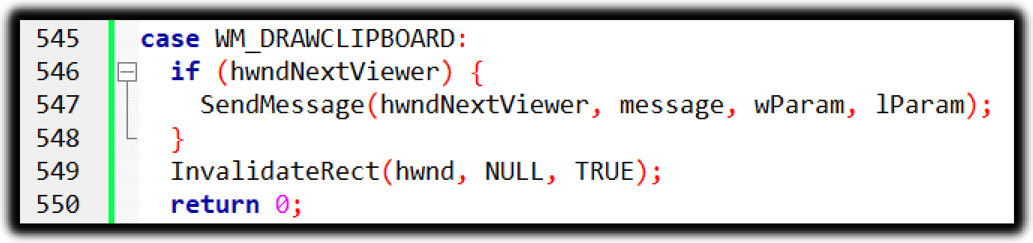


```
static HWND hwndNextViewer;

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

II. Handling WM_DRAWCLIPBOARD: The Relay

When the clipboard content changes, Windows shouts at **you** first. Your job is to act as a relay station: look at the data, then pass the "shout" down the line.

- **The Relay (Forwarding):** Unless you are the only one in line (hwndNextViewer == NULL), you **must** use SendMessage to pass the WM_DRAWCLIPBOARD message to the next guy. If you don't, the chain breaks at your feet.
- **The Refresh (Invalidating):** Trigger a redraw of your own window (Invalidate) so your UI can display the fresh clipboard data.



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

III. Updating the UI (WM_PAINT)

Once WM_DRAWCLIPBOARD tells you something has changed, you actually have to go get the data and show it.

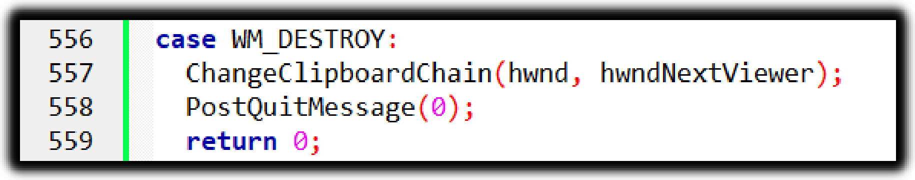
- **The Workflow:** To peek at the data, you must **Open** the clipboard, **Get** the specific format you want (text, bitmap, etc.), and **Close** it immediately so other apps can use it.
- **Update:** Once you have the data in hand, redraw your window to reflect the new contents.

Pro Tip: Don't hold the clipboard open while you're doing heavy processing or waiting for user input. If you don't call CloseClipboard, no other program (including yours) can copy/paste anything.

IV. Leaving the Chain: The "Clean Exit"

You can't just close your app and hope for the best. If you disappear without a word, the "link" to the programs behind you is severed.

- **The "Unplug" (ChangeClipboardChain):** Call this to tell Windows, "I'm out." You provide your handle and the handle of the guy behind you (hwndNextViewer).
- **The Repair:** Windows then sends a WM_CHANGECBCHAIN message to the head of the chain. This message travels down the line until it reaches the person who was pointing to *you*. They then update their pointer to point to the person *after* you, effectively stitching the hole closed.

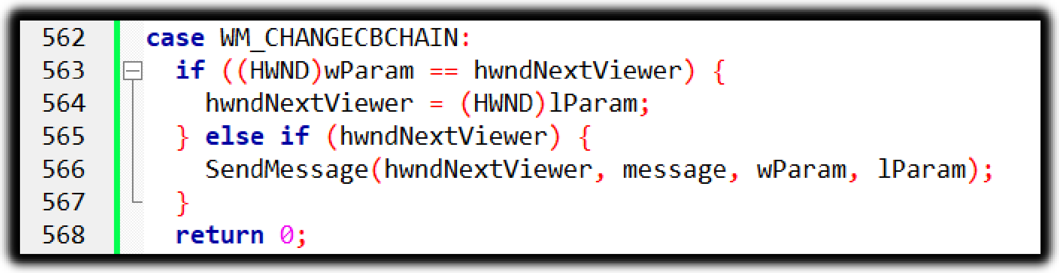


```
case WM_DESTROY:
  ChangeClipboardChain(hwnd, hwndNextViewer);
  PostQuitMessage(0);
  return 0;
```

V. Handling WM_CHANGECBCHAIN: The Self-Healing Chain

This message is how the "linked list" repairs itself. When any program leaves the chain, every viewer must check if their "neighbor" is the one who just vanished.

- **The Check:** Look at wParam. This is the handle of the program that is leaving.
- **Case A: Your neighbor is leaving.** If wParam == hwndNextViewer, the person you were supposed to talk to is gone. You must now update your hwndNextViewer to lParam (the new person Windows just put in that spot).
- **Case B: Someone else is leaving.** If wParam isn't your neighbor, you don't care about the change, but the people behind you might. **Forward the message** to your hwndNextViewer so the repair search continues down the line.



```
case WM_CHANGECBCHAIN:
  if ((HWND)wParam == hwndNextViewer) {
    hwndNextViewer = (HWND)lParam;
  } else if (hwndNextViewer) {
    SendMessage(hwndNextViewer, message, wParam, lParam);
  }
  return 0;
```

VI. Additional Functions:

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

VII. 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.

VIII. Conclusion: Why This Matters

The clipboard viewer chain functions as a linked list, and your application represents a single node within that list. If you fail to properly manage handoffs (SetClipboardViewer) or neglect to repair the chain when exiting (ChangeClipboardChain), you can disrupt clipboard notifications for every other application on the system.

The correct logic is a simple but critical loop:

Join → Relay messages → Repair the chain on exit

Each step is essential. Joining ensures your app receives notifications, relaying messages preserves the chain for downstream viewers, and repairing the chain prevents breakage when your app closes.

Refer to *ClipView* in Chapter 12.

CLIPVIEW: A SIMPLE CLIPBOARD VIEWER PROGRAM

CLIPVIEW is a minimal clipboard viewer that demonstrates the mechanics of joining and maintaining the clipboard viewer chain and displaying clipboard data. It focuses on handling the CF_TEXT format.

I. Core Messages and Responsibilities

- WM_CREATE: Joins the clipboard viewer chain and saves the next viewer.
- WM_CHANGECBCHAIN: Maintains the chain as viewers are added or removed.
- WM_DRAWCLIPBOARD: Relays the notification and triggers a repaint.
- WM_PAINT: Reads CF_TEXT from the clipboard and renders it.
- WM_DESTROY: Removes itself cleanly from the chain.

II. Clipboard Access

- Uses OpenClipboard / GetClipboardData to retrieve text data.
- Locks the global memory handle and renders it with DrawText.
- Releases resources and closes the clipboard when finished.

III. Scope and Limitations

- Handles only CF_TEXT.
- More complete viewers must enumerate formats (EnumClipboardFormats) and resolve custom formats (GetClipboardFormatName).
- Owner-display formats (CF_OWNERDISPLAY) require forwarding paint and scroll messages to the clipboard owner.

IV. Clipboard Owner Interaction

- The clipboard owner's window handle is obtained via GetClipboardOwner.
- Required messages are sent directly to the owner for rendering owner-managed formats.

CLIPVIEW is intentionally small. It illustrates the essential structure of a clipboard viewer—chain management, notification relaying, and data access—without addressing advanced formats or owner-driven rendering.

End of Chapter 12