CHARACTER MESSAGES

We are now crossing the bridge from "Hardware" to "Text."

- **Previous Section:** "The user pressed Button #30." (WM_KEYDOWN)
- **This Section:** "The user typed the letter 'a'." (WM_CHAR)

1. The Big Concept: Translation

Windows takes the raw keystroke messages you learned earlier, processes them (checking Shift, Caps Lock, and Language settings), and translates them into **Character Messages**.

You usually ignore WM_KEYDOWN for text entry and listen for **WM_CHAR** instead.

2. The Four Character Messages

a. WM_CHAR (The Main Event)

This is the most important message for text input.

- **When it happens:** When a user presses a key that corresponds to a printable character (A-Z, 0-9, symbols).
- **wParam:** Contains the **Character Code** (ASCII or Unicode value). *Example:* If the user types "A", wParam is 65.
- **lParam:** Contains the physical details (Repeat count, Scan code) copied directly from the WM_KEYDOWN message that caused this.

b. WM_DEADCHAR (The Accent Helper)

This is for international characters (diacritics).

The Concept: A "Dead Key" is a key that doesn't make a character appear immediately. It waits for the *next* keystroke to combine them.

Example: On many keyboards, to type **é**, you first press the ´ (Acute Accent) key.

- User presses ´: Nothing appears on screen. The cursor does not move (it is "dead"). Windows sends WM_DEADCHAR.
- User presses **e**: Windows combines them and sends WM_CHAR for **é**.

c. WM_SYSCHAR (System Characters)

- **The Concept:** These are characters generated while holding the **Alt** key.
- **Usage:** Windows uses these for menu shortcuts (e.g., Alt+F to open the "File" menu).
- **Your Job:** Usually, you ignore these. You pass them to DefWindowProc so Windows can handle the menu navigation for you.

d. WM_SYSDEADCHAR (System Dead Characters)

- **The Concept:** A "Dead Key" pressed while holding **Alt**.
- **Usage:** Extremely rare. Unless you are writing very specific system tools for international keyboards, you will never touch this.

MESSAGE	TYPE	MEANING	ACTION
`WM_CHAR`	Standard	"User typed a valid letter."	Process this (Add letter to screen).
`WM_DEADCHAR`	Standard	"User pressed an accent key."	Usually ignore (Windows handles the logic).
`WM_SYSCHAR`	System	"User typed an Alt-Command."	Ignore (Pass to Windows).
`WM_SYSDEADCHAR`	System	"User pressed Alt+Accent."	Ignore.

DEAD CHARACTERS, ENCODING, AND PROCESSING LOGIC

1. Dead Characters: The "Two-Step" Typing

The Concept: Keys that wait.

Standard keys print immediately. "Dead Keys" do not print anything when you press them; they change the meaning of the *next* key you press. This is essential for international languages (like French or German).

How it works (The é example):

- **Step 1:** You press the **Accent Key** (e.g., ^ or ´).
    1. Result: Nothing appears on screen. The cursor waits.
    2. Message: Windows sends WM_DEADCHAR.
- **Step 2:** You press a **Letter Key** (e.g., e).
    1. Result: Windows combines them and prints **é**.
    2. Message: Windows sends WM_CHAR containing the code for é.

Correction Note: Your text mentions typing the accent *after* the 'e', but standard "Dead Key" behavior usually requires typing the accent *first*, then the letter.

2. Nonsystem vs. System Characters

This distinction is simple: **Is the ALT key held down?**

Nonsystem Characters (WM_CHAR):

- ✓ These are normal letters, numbers, and symbols.
- ✓ **Goal:** Putting text into the document.

System Characters (WM_SYSCHAR):

- ✓ These are generated when you hold **Alt** and press a key.
- ✓ **Goal:** Controlling the window (opening menus, activating shortcuts).
- ✓ **Action:** You usually ignore these and let Windows handle the menu logic.

3. ANSI vs. Unicode: The Alphabet Size

This is the difference between "Old Windows" and "Modern Windows."

FEATURE	ANSI (THE OLD WAY)	UNICODE (THE MODERN WAY)
Bits per Character	8 Bits (1 Byte)	16 Bits (2 Bytes)
Total Characters	256	65,536+ (Millions)
Language Support	Limited (English + 1 Region)	Universal (All languages simultaneously)
C Data Type	`char`	`wchar_t`
Recommendation	**Avoid if possible.**	**Always use this in Windows.**

Key Takeaway: In WM_CHAR, the wParam will hold the code. If your app is Unicode (which it should be), that code is a 16-bit number capable of representing Emoji, Chinese characters, or Mathematical symbols.

4. How to Process Character Messages

The Golden Rule: Only look at WM_CHAR.

When writing your Window Procedure, the logic usually looks like this:

1. **Ignore the Hardware Info (lParam):** You rarely care about repeat counts or scan codes when processing text.
2. **Read the Character Code (wParam):** This is the **actual ASCII or Unicode value**.

Important Distinction:

- ✓ In WM_KEYDOWN, wParam is a **Virtual Key** (e.g., VK_A = 0x41). It represents the physical button.
- ✓ In WM_CHAR, wParam is the **Character** (e.g., 'a' = 0x61 or 'A' = 0x41). It represents the text result.

```
case WM_CHAR:
    TCHAR ch = (TCHAR) wParam; // Get the character code

    // Check for "Backspace" (Character 8)
    if (ch == 8) {
        // Delete last character
    }
    // Check for "Enter" (Character 13)
    else if (ch == 13) {
        // Move to next line
    }
    // Normal Text
    else {
        // Add 'ch' to your string buffer and repaint
    }
    return 0;
```

ANSI vs. Unicode and the Message Sequence.

1. The "Magic" Switch: ANSI vs. Unicode

How does Windows know whether to send you an 8-bit character (char) or a 16-bit character (wchar_t)?

It depends on how you registered your window class at the very beginning of your program (Chapter 3).

- **RegisterClassA**: You get **ANSI** messages (old school).
- **RegisterClassW**: You get **Unicode** messages (modern standard).

Note: You rarely type these names directly. You usually type RegisterClass, and your compiler macros automatically pick the "W" version if you are building a modern Unicode app.

---

2. The Message Sandwich

Character messages do not come from the hardware. They are created artificially by your message loop.

The Workflow:

1. **Hardware:** User presses the 'A' key.
2. **Queue:** Windows puts WM_KEYDOWN in your queue.
3. **Loop Step 1 (GetMessage):** Your loop grabs WM_KEYDOWN.

4. **Loop Step 2 (TranslateMessage): This is the critical moment.**
    - ✓ Windows looks at the WM_KEYDOWN.
    - ✓ It checks the Shift state (is Shift up or down?).
    - ✓ It determines that "Key A" + "No Shift" = "Character 'a'".
    - ✓ **It posts a NEW message (WM_CHAR) back into your queue.**
5. **Loop Step 3 (DispatchMessage):** The program processes the original WM_KEYDOWN.
6. **Next Loop:** The program finds the new WM_CHAR waiting in the queue and processes it.

---

3. The Sequence: Order of Operations

Because TranslateMessage inserts the character message immediately, they always arrive in a "sandwich" order:

1. **WM_KEYDOWN** (The Button Press)
    - ✓ *Input:* "The physical button #65 was pushed."
2. **WM_CHAR** (The Text Result)
    - ✓ *Input:* "The letter 'a' was generated."
3. **WM_KEYUP** (The Release)
    - ✓ *Input:* "The physical button #65 was let go."

---

4. The Example: Typing "a"

This example highlights the difference between the **Key ID** and the **Text Result**.

Scenario: User presses the 'A' key **without** holding Shift.

MESSAGE	WPARAM (THE DATA)	MEANING
`WM_KEYDOWN`	`0x41`	`0x41` is the Virtual Key Code for the **Physical 'A' Button**.
`WM_CHAR`	`0x61`	`0x61` is the ASCII/Unicode value for the **lowercase letter 'a'**.
`WM_KEYUP`	`0x41`	`0x41` is the Virtual Key Code for the **Physical 'A' Button**.

Why is this important?

Notice that WM_KEYDOWN and WM_KEYUP have the **same** wParam (0x41), regardless of whether you are typing "a" or "A".

Only WM_CHAR tells you the actual case (0x61 for 'a', 0x41 for 'A').

Shift-Typing, Repeat Logic, and Control Codes.

1. The Anatomy of a Capital Letter ("Shift + A")

When you type a capital letter, you are actually performing a complex 5-step dance with the hardware. Because "Shift" is a separate physical key, it generates its own messages.

The Sequence:

STEP	ACTION	MESSAGE	WPARAM	MEANING
1	Press Shift	`WM_KEYDOWN`	0x10	"Shift key went down."
2	Press 'A'	`WM_KEYDOWN`	0x41	"Physical 'A' button went down."
3	Translation	`WM_CHAR`	0x41	"The text 'A' (Uppercase) was generated."
4	Release 'A'	`WM_KEYUP`	0x41	"Physical 'A' button went up."
5	Release Shift	`WM_KEYUP`	0x10	"Shift key went up."

Note: Notice that Step 3 (WM_CHAR) is the only one you care about for text input. Steps 1, 2, 4, and 5 are "hardware noise" that you usually ignore unless you are writing a game or a custom hotkey handler.

2. Handling Repeat Count (The "Machine Gun" Effect)

If you hold down the 'A' key, the keyboard sends a stream of WM_KEYDOWN messages.

Because TranslateMessage runs on every message in the loop, it also generates a stream of WM_CHAR messages.

The Loop:

1. WM_KEYDOWN (Repeat Count = 1) $\rightarrow$ WM_CHAR (Repeat Count = 1)
2. WM_KEYDOWN (Repeat Count = 2) $\rightarrow$ WM_CHAR (Repeat Count = 2)
3. WM_KEYDOWN (Repeat Count = 3) $\rightarrow$ WM_CHAR (Repeat Count = 3)

This ensures that if you hold a key in Notepad, the letters appear on screen at the exact same rate the hardware is firing.

3. Checking for Unicode (IsWindowUnicode)

If you are writing a reusable library or a plugin, you might not know if the window you are working with is old (ANSI) or new (Unicode).

- **The Function:** IsWindowUnicode(hwnd)
- **Returns TRUE:** The window expects 16-bit characters (wchar_t).
- **Returns FALSE:** The window expects 8-bit characters (char).

4. Control Characters (Ctrl + Key)

This is a relic from the teletype days that still exists in Windows today.

When you hold Ctrl and press a letter, Windows generates a WM_CHAR message, but the wParam is a special Control Code (numbers 1 to 26), not the letter itself.

The Math:

- **Ctrl + A** = 1
- **Ctrl + B** = 2
- ...
- **Ctrl + Z** = 26

Why does this matter?

Some of these numbers overlap with standard keyboard functions. This is why "Ctrl+H" acts as Backspace in many terminal applications.

5. Common Control Codes:

KEY COMBO	ASCII VALUE	STANDARD FUNCTION
`Ctrl + G`	`7`	**Bell** (Makes a system beep sound).
`Ctrl + H`	`8`	**Backspace** *(Same as VK_BACK).*
`Ctrl + I`	`9`	**Tab** *(Same as VK_TAB).*
`Ctrl + J`	`10`	**Line Feed** (Starts a new line).
`Ctrl + M`	`13`	**Carriage Return** *(Same as Enter).*

Implication for your Code: If you are processing WM_CHAR, and the user presses Enter, you will receive the number 13. You don't need to check VK_RETURN; you just check for character 13.

Control Characters, Escape Codes, and Menu Accelerators

This table summarizes the four-character messages:

Message	Description
WM_CHAR	Sent when a character is typed
WM_DEADCHAR	Sent before a character is displayed
WM_SYSCHAR	Sent when a system character is typed
WM_SYSDEADCHAR	Sent before a system character is displayed

This section explains why pressing Backspace is mathematically identical to pressing Ctrl+H, and how Windows decides whether Ctrl+O writes a character or opens a file.

1. The "Duplicate" Phenomenon

The Concept: Two ways to dial the same number.

In the ASCII standard, specific keys on your keyboard map to the exact same numerical values as certain Ctrl + Letter combinations. Your program cannot tell the difference between them.

The Equivalency Table:

KEY NAME	WINUSER.H IDENTIFIER	HEX CODE	CTRL EQUIVALENT	MEANING / ACTION
Backspace	`VK_BACK`	`0x08`	**Ctrl + H**	Move cursor back one space.
Tab	`VK_TAB`	`0x09`	**Ctrl + I**	Move cursor to next tab stop.
Line Feed	N/A	`0x0A`	**Ctrl + J**	Move cursor down one line.
Enter	`VK_RETURN`	`0x0D`	**Ctrl + M**	Carriage Return; return to start of line.
Escape	`VK_ESCAPE`	`0x1B`	**Ctrl + [**	Cancel / Exit current operation.
Spacebar	`VK_SPACE`	`0x20`	N/A	Insert a blank space character.
Shift	`VK_SHIFT`	`0x10`	N/A	Modifier key (either Left or Right).

Why this matters: If you write a program to detect Backspace, you are automatically also detecting Ctrl+H. You don't need to write separate code for the shortcut; they are physically the same signal to the computer.

2. ANSI C Escape Codes

The Concept: How to type the untypable.

In C programming, you cannot just type "Enter" inside a string of code.

- **Wrong:** char c = ' '; (This breaks the compiler).
- **Right:** char c = '\n'; (This is the "Escape Code").

These codes act as **aliases** for the Control Characters mentioned above.

ESCAPE SEQUENCE	NAME	MATCHES KEY...	HEX VALUE
`\b`	Backspace	Backspace	`0x08`
`\t`	Horizontal Tab	Tab	`0x09`
`\n`	Newline	Ctrl+Enter (Line Feed)	`0x0A`
`\r`	Return	Enter (Carriage Return)	`0x0D`
`\\`	Backslash	N/A	`0x5C`

Coding Tip: When processing WM_CHAR, you can check against these aliases to make your code readable: if (wParam == '\b') is much clearer than if (wParam == 0x08).

---

3. Menu Accelerators: The "Short-Circuit"

The Concept: When a Keystroke becomes a Command.

Normally, Ctrl + Letter generates a WM_CHAR message (a control character). **However**, if you define that combination as a **Menu Accelerator** (a shortcut), Windows intercepts it before it ever becomes a character.

The Scenario: Ctrl + O

- **Case A (No Accelerator):**
    1. User presses Ctrl + O.
    2. Windows generates WM_CHAR.
    3. wParam is **15** (The ASCII code for Control-O).
    4. Your window receives a character message.
- **Case B (With Accelerator "Open"):**
    1. User presses Ctrl + O.
    2. Windows checks the Accelerator Table.
    3. It sees "Open File" is mapped to this key.
    4. **It destroys the WM_CHAR message.**
    5. It sends a WM_COMMAND message (ID = "Open File") instead.

The Takeaway: If you add standard shortcuts (Ctrl+C, Ctrl+V, Ctrl+O) to your application menus, your WM_CHAR handler will **never see those keys**. They are "stolen" by the menu system to trigger commands.

Duplicate Control Characters and the "Which Message?" Debate.

This section answers a very common question: *"If the Enter key sends both a KeyDown code AND a Character code, which one should I listen to?"*

1. The "Duplicate" Table

First, you must understand that some keys on your keyboard are just "shortcuts" for older Ctrl combinations. To the computer, they are mathematically identical.

THE KEY YOU PRESS	THE CODE IT SENDS (HEX)	THE C ESCAPE CODE	IT IS IDENTICAL TO...
Backspace	`0x08`	`\b`	Ctrl + H
Tab	`0x09`	`\t`	Ctrl + I
Ctrl + Enter	`0x0A`	`\n` (Line Feed)	Ctrl + J
Enter	`0x0D`	`\r` (Carriage Return)	Ctrl + M
Escape	`0x1B`	`\e`	Ctrl + [

Key Takeaway: If your program checks for Ctrl+H, it will trigger every time the user hits **Backspace**. They are the same signal.

---

2. The Dilemma: WM_KEYDOWN vs. WM_CHAR

These special keys (Backspace, Tab, Enter, Esc) have a "Dual Nature." They generate **two** messages every time you press them:

1. **WM_KEYDOWN** (Virtual Key Code, e.g., VK_RETURN)
2. **WM_CHAR** (ASCII Code, e.g., 13)

The Big Question: Which message should you use to handle the action?

Option A: The Traditional Way (WM_KEYDOWN)

In the old days (and in some specific cases today), programmers used WM_KEYDOWN.

- **Why?** Because these keys are often used for "Control" (moving cursors, changing focus) rather than "Typing" (putting ink on paper).
- **Logic:** "Enter is a command, not a letter, so I will handle it alongside the Arrow Keys in WM_KEYDOWN."

Option B: The Modern/Better Way (WM_CHAR)

For most standard applications, it is much smarter to handle them in WM_CHAR.

- **Reason 1: Consistency.** You are already checking WM_CHAR for letters like 'A', 'B', and 'C'. It is cleaner to check for 'Backspace' in the exact same switch statement rather than splitting your logic into two different places.
- **Reason 2: Simplicity.** In WM_CHAR, you don't have to worry about Shift states.
  - ✓ *In WM_KEYDOWN:* You have to manually check if Shift is held to know if VK_TAB means "Next Field" or "Previous Field."
  - ✓ *In WM_CHAR:* Windows has already done that translation for you.

---

3. The Decision Guide

Here is the simple rule of thumb for your code:

IF YOU WANT TO...	USE THIS MESSAGE	THE LOGIC WHY?
Move the Cursor (Left, Right, Up, Down)	`WM_KEYDOWN`	These are **Navigation** keys. They don't generate characters; they move the "view".
Edit Text (Type letters, Backspace, Enter, Tab)	`WM_CHAR`	These are **Data** keys. `Backspace` and `Enter` are treated as "Editing Text" because they change the string buffer.
Issue Commands (Escape to cancel, F1 for Help)	`WM_KEYDOWN`	These are **Function** keys. They trigger logic paths, not text input.

Why? Because Backspace and Enter are effectively "Editing Text." They change the string you are typing. Therefore, they belong with the other characters in WM_CHAR.

Recommended Approach for Control Keys and the fascinating mechanism of Dead Characters (Accents)

```
case WM_CHAR:
    switch (wParam)
    {
        case '\b':  // Backspace (0x08)
            // Code to delete the character to the left
            break;

        case '\t':  // Tab (0x09)
            // Code to move cursor to the next tab stop
            break;

        case '\r':  // Carriage Return / Enter (0x0D)
            // Code to move cursor to the start of the next line
            break;

        case '\n':  // Linefeed / Ctrl+Enter (0x0A)
            // Code to move down one line (if distinct from Enter)
            break;

        case 0x1B:  // Escape (No standard C letter, usually 0x1B)
            // Code to cancel operation
            break;

        default:
            // This is a normal letter (A, b, $, etc.)
            // Insert it into the document
            break;
    }
    return 0;
```

As discussed, handling special keys inside WM_CHAR is cleaner than WM_KEYDOWN.

Here is the corrected version of the code snippet from your notes (fixing the typos in the escape sequences).

The Logic: Instead of memorizing Hex codes like 0x08, we use standard C "Escape Sequences" (\b, \t, etc.).

1. Dead Characters: The "Ghost" Keys

The Concept: A key that waits for a partner.

On US keyboards, every key prints immediately. On International keyboards (like German, French, or Spanish), some keys are **"Dead Keys."** They do not generate a character immediately; they modify the *next* key you press to add an accent mark.

The Workflow:

1. **Step 1:** User presses the Dead Key (e.g., the Acute Accent ´).
    - ✓ **Result:** Nothing appears on screen.
    - ✓ **Message:** WM_DEADCHAR.
2. **Step 2:** User presses a letter (e.g., a).
    - ✓ **Result:** Windows combines them.
    - ✓ **Message:** WM_CHAR (with the code for á).

2. The "German Keyboard" Example

Your notes use the German layout to illustrate this.

- **The Key:** The key near "Enter" (where + is on US keyboards).
- **The Action:**
    - ✓ **Unshifted:** Acts as an Acute accent (´).
    - ✓ **Shifted:** Acts as a Grave accent (`).
- **The Sequence:**
    1. User presses the Dead Key (´). Windows sends WM_DEADCHAR. (Program usually ignores this).
    2. User presses A.
    3. Windows checks: "Does ´ + A exist?"
    4. Yes! It is á.
    5. Windows sends WM_CHAR with the code for á.

3. Error Handling: When Accents Fail

What happens if the user tries to put an accent on a letter that doesn't support it? (e.g., Putting an accent on the letter 's').

The "Resurrection" Logic: Since ´s is not a valid character, Windows assumes the user actually wanted to type both characters separately.

User Action: Press Dead Key (´) ⏩ Press s.

Windows Reaction:

1. It realizes ´ + s = Invalid.
2. It sends **two** messages back-to-back:
    - ✓ **Message 1 (WM_CHAR):** The accent itself (´).
    - ✓ **Message 2 (WM_CHAR):** The letter (s).

3. Result on Screen: ´s

Why is this smart?

It means your program does not need to know how accents work. You just listen for WM_CHAR.

- If the user typed it correctly, you get one character (á).
- If the user messed up, you get two characters (´ then s).
- Either way, you just print what you receive!

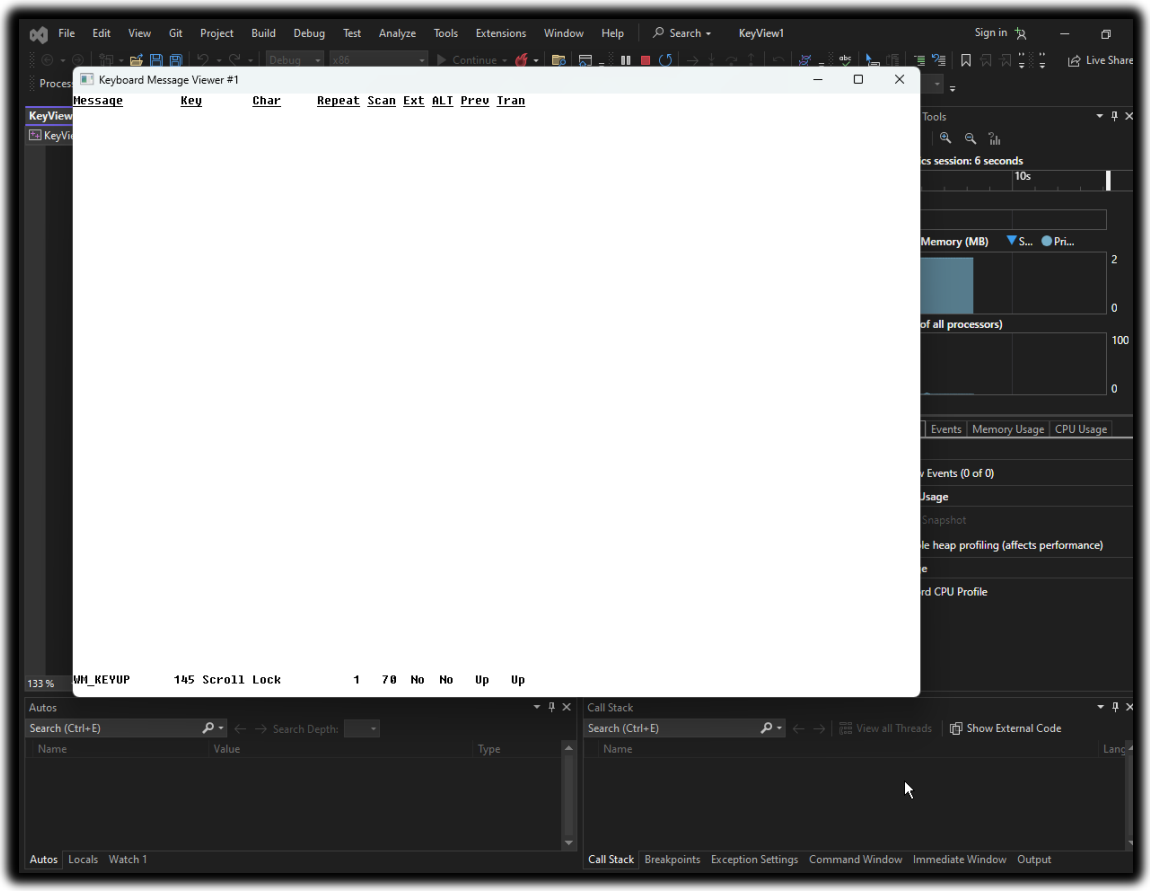
4. Testing Tip

To see this in action without flying to Germany:

1. Go to Windows **Settings** > **Time & Language** > **Language**.
2. Add the **"German (Germany)"** keyboard layout.
3. Switch to it using the language bar (bottom right of taskbar).
4. Run a diagnostic tool (like the KEYVIEW program mentioned in the text) to watch the messages fly by as you type.

KEYVIEW1, the "Keyboard X-Ray" program

This program is the most useful tool in this chapter. It is a diagnostic utility that lets you **see** the invisible messages we have been discussing (like WM_KEYDOWN, WM_CHAR, and WM_DEADCHAR) in real-time.



1. Why is the app empty at first?

When you first run KEYVIEW1, the window is blank.

- **Reason:** This program is a **History Log**. It displays a list of the keys you *have* pressed.
- **Action:** As soon as you press a key, the window acts like a typewriter or a Command Prompt: it pushes the old lines up and writes the new message at the bottom.

---

2. How the Program Works (The Logic Flow)

The notes outline a smart way to handle a "scrolling log" in C. Here is the simplified breakdown of the machinery:

A. The Storage (The "History Book")

Instead of just printing to the screen, the program keeps a saved copy of every message in memory.

- **The Data Structure:** It uses an **Array of MSG structures**.
- **Dynamic Sizing (WM_SIZE):** This is the clever part. The program calculates exactly how many lines of text fit on your screen (Window Height ÷ Text Height). It allocates *exactly* enough memory to store that many messages. If you make the window bigger, it re-allocates more memory to store more history.

B. The Input Loop (WM_KEYDOWN / WM_CHAR)

Every time you press a key, the WndProc wakes up.

1. **Capture:** It grabs the message details (wParam, lParam).
2. **Store:** It saves this message into the array.
3. **Scroll:** It physically shifts the pixels of the window up by one line (using ScrollWindow).
4. **Invalidate:** It tells Windows, "The very bottom line is now empty and dirty; please let me paint it."

C. The Paint Job (WM_PAINT)

When Windows asks to repaint that bottom line, the program reads the MSG array and formats the data into columns.

The Columns Displayed:

1. **Message Name:** (e.g., "WM_KEYDOWN")
2. **Virtual Key:** (e.g., "65" or "VK_A")
3. **Character:** (The actual letter, e.g., "A")
4. **Repeat Count:** (How many times it fired)
5. **Extended Flag:** (Is it a keypad key?)
6. **Alt Status:** (Is Alt held down?)
7. **Previous State:** (Was the key already down?)

3. Why this program is important

KEYVIEW1 is your "Reality Check."

- **Theory:** You read that "Dead Keys send WM_DEADCHAR."
- **Practice:** You run KEYVIEW1, switch your keyboard to "German," press the accent key, and **watch** the WM_DEADCHAR appear on the screen.

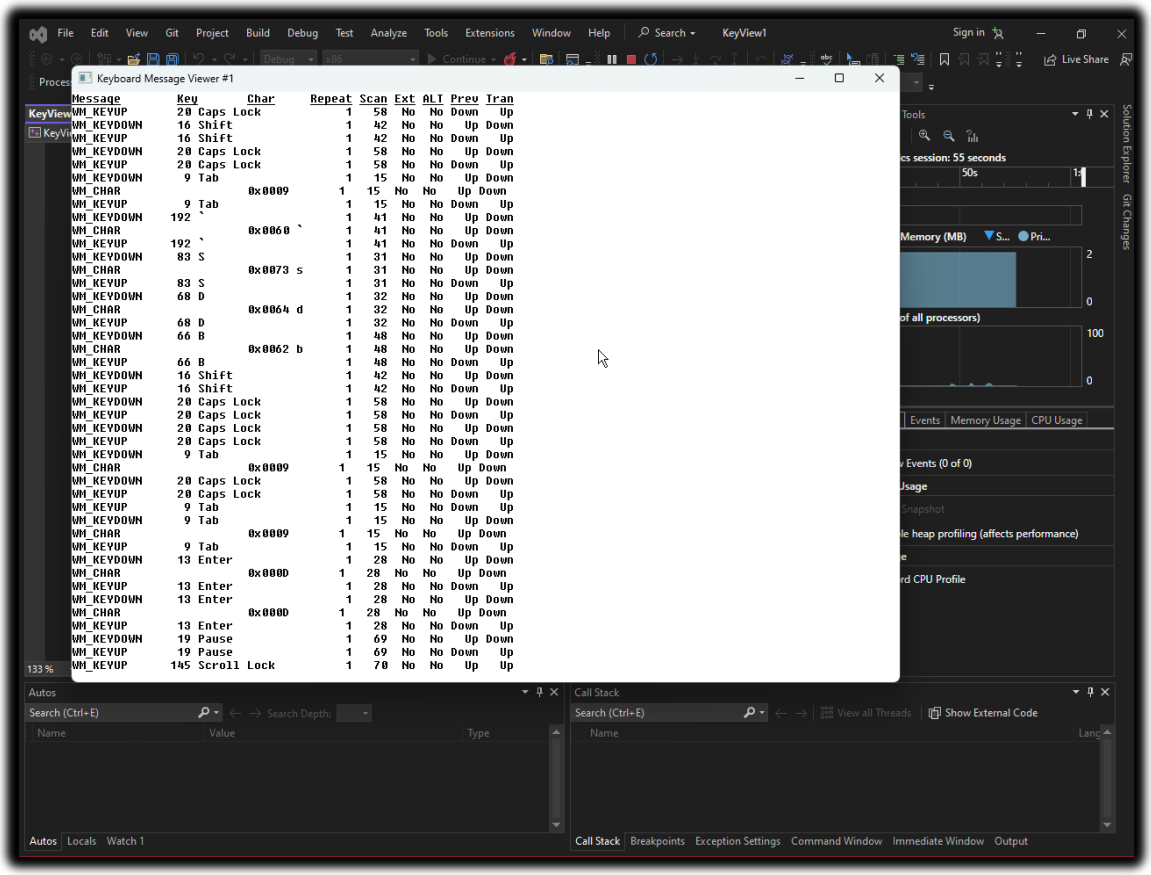
The XOR identity $(A \oplus B) \oplus B = A$ elegantly proves that the invisible "Message Sandwich" (KeyDown → Char → KeyUp) is actually happening — because the paired KeyDown and KeyUp cancel perfectly, leaving a detectable trace only if the hidden Char message in the middle disturbs the state.

4. Summary of the Code Logic

- **WinMain**: Sets up the window.
- **WndProc**: Sorts the incoming mail.
- **WM_SIZE**: Decides how big the history book needs to be.
- **WM_PAINT**: Draws the text on the screen.
- **ScrollWindow**: The visual trick to make text move up smoothly.

This concludes Chapter 6 on the Keyboard! You have moved from understanding physical buttons (WM_KEYDOWN) to text input (WM_CHAR) and finally to visualizing it all with KEYVIEW1.

Clicking any key, it registers:



The Font Choice: Fixed-Pitch

The Problem: Normal Windows fonts (Variable Pitch) are like handwriting. An 'i' is very thin, and a 'W' is very wide. If you try to make columns using a normal font, the lines will wiggle and never align perfectly.

The Solution: SYSTEM_FIXED_FONT The program forces Windows to use a "Fixed-Pitch" (Monospace) font, similar to a typewriter or code editor.

- **Feature:** Every character is exactly the same width.
- **Result:** You can align columns perfectly just by counting spaces.

```
// Select the old-school fixed system font
SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
```

The Header Trick: "Manual" Underlining

The Problem: The programmer wanted headers (like "Message", "Char", "Repeat") to be underlined. **The Standard Way:** Create a special Font object with the "Underline" flag set to TRUE. (This is tedious and requires more code). **The KEYVIEW1 Way:** The "Overstrike" method. The program defines two separate strings:

1. **szTop:** The actual text (e.g., "Message Key Char")
2. **szUnd:** A string of underscores (e.g., "_____ ___ ____")

During WM_PAINT, it draws szTop first, and then draws szUnd **at the exact same position**.

The "Transparency" Fix

The Issue: By default, Windows prints text in **Opaque Mode**.

- This means it draws a white rectangle (the background) *behind* every letter before drawing the letter itself.
- **The Conflict:** If you print the text, then try to print the underscores on top, the "white box" of the underscores will **erase** the text you just wrote!

The Solution: SetBkMode(TRANSPARENT) This command tells Windows: *"When you print these underscores, do NOT draw the white box background. Just draw the black pixels of the line."*

The Visual Difference:

MODE	RESULT OF DRAWING ON TOP OF TEXT/GRAPHICS
`OPAQUE` (Default)	***Blocks Out***: *The background color (usually white) fills the character cell, erasing whatever was underneath.*
`TRANSPARENT`	***Overlays***: *The background is ignored; only the character pixels are drawn, leaving the underlying graphics visible.*

```
// Tell Windows not to erase the background when printing text
SetBkMode(hdc, TRANSPARENT);
TextOut(hdc, x, y, szTop, ...); // Draw the words
TextOut(hdc, x, y, szUnd, ...); // Draw the lines on top
```

Summary of KEYVIEW1

To wrap up this chapter, KEYVIEW1 is the ultimate "lab equipment" for understanding the keyboard.

- It **Validates** the theory: You can see the messages arriving in the order we discussed.
- It **Demonstrates** advanced C techniques: Dynamic memory allocation, scrolling logic, and text transparency tricks.
- It **Diagnoses**: If you ever write a program and the keyboard feels "weird," you can run this tool to see exactly what signals your specific keyboard hardware is sending.

Foreign-Language Keyboard Problem

Foreign-Language Keyboard Problem, also known as the "Extended ASCII" or "Code Page" problem.

When using a foreign-language keyboard layout, Windows programs may display incorrect characters.

This is because the programs are not aware of the new keyboard layout and are still interpreting the character codes according to the default English keyboard layout.

This section deals with the confusion that happens when a single number (e.g., 0xE1) can mean two completely different letters depending on which language "Dictionary" (Code Page) the computer is using.

1. The Core Problem: Ambiguous Codes

In the older ANSI (8-bit) system, there are only 256 slots for characters.

- **Slots 0-127:** Standard English (A-Z, 0-9). These are safe and universal.
- **Slots 128-255:** The "Wild West." These slots are reused by different languages.

The Conflict:

- In **Western Europe**, Slot 0xE1 is defined as **á** (a-acute).
- In **Greece**, Slot 0xE1 is defined as **α** (alpha).

If your program receives the number 0xE1, it doesn't know which letter to draw unless it knows which "Map" (Charset) to use.

2. Example: German Keyboard

For example, if you switch to the German keyboard layout and type the letters "abcde," you will get the following WM_CHAR messages:

Character Code	Character
0×61	a
0×62	b
0×63	c
0×64	d
0×65	e

Typing "abcde" gives standard codes (0x61 - 0x65).

- **Observation:** Since German uses the Latin alphabet, the basic letters match English perfectly.
- **The Issue:** The problem usually arises with the *other* keys (like ö, ä, ß), which live in the upper 128-255 range. If the program expects English, these might display as math symbols or lines instead of German letters.

3. Example: Greek Keyboard

If you switch to the Greek keyboard layout and type "abcde," you will get the following WM_CHAR messages:

Character Code	Character
0xE1	á
0xE2	â
0xF8	ø
0xE4	ā
0xE5	å

These are not the characters that you would expect to see. This is because the Greek keyboard layout maps the a, b, c, d, and e keys to different character codes than the English keyboard layout.

Input: You type on a Greek keyboard.

Message: WM_CHAR sends 0xE1.

The Mismatch:

- **Greek Windows** knows this is **Alpha (α)**.
- **English Program** thinks this is **a-acute (á)**.

Result: You type Greek, but the screen shows Western European accents.

4. Example: Russian Keyboard

If you switch to the Russian keyboard layout and type "abcde," you will get the following WM_CHAR messages:

Character Code	Character
0xF4	ô
0xE8	è
0xF1	ñ
0xE2	â
0xF3	ó

These are not the characters that you would expect to see. This is because the Russian keyboard layout maps the a, b, c, d, and e keys to different character codes than the English keyboard layout, and the Russian version of Windows is able to interpret these character codes correctly.

5. The Solution: Informing GDI

The "Graphics Device Interface" (GDI) is the part of Windows responsible for drawing text on the screen. It needs to know which "Charset" to use to paint the pixels correctly.

The Strategy described in your notes: The program must detect that the keyboard layout has changed and tell GDI to switch its mapping.

1. **The Tool: SetWindowsHookEx** (specifically WH_KEYBOARD_LL).
    - ✓ This function installs a "spy" (a Hook) that intercepts messages before they reach the normal application.
2. **The Action:**
    - ✓ The hook detects which keyboard layout is active.
    - ✓ It interprets the incoming byte codes using the correct layout.
    - ✓ It updates the Font Charset (e.g., switching from ANSI_CHARSET to GREEK_CHARSET).

Modern Context Note: In modern Windows programming (Unicode), this problem largely disappears because every character (Greek, Russian, English) has a unique ID number that never overlaps. However, for the ANSI applications described in these notes, manually managing these Charsets via Hooks or Message handling (WM_INPUTLANGCHANGE) is crucial.

CHARACTER SETS AND FONTS

1. Character Sets (The ID Card)

A character set is just a big table where every letter, number, and symbol gets a unique ID number (an index).

Analogy: Think of a **Prisoner Registry**. Inmate #65 is always "A". It doesn't matter if Inmate #65 is tall, short, or wearing a hat—they are still #65.

RE Perspective: When you see 0x41 in a hex editor, that's the ID for 'A' in the ASCII character set.

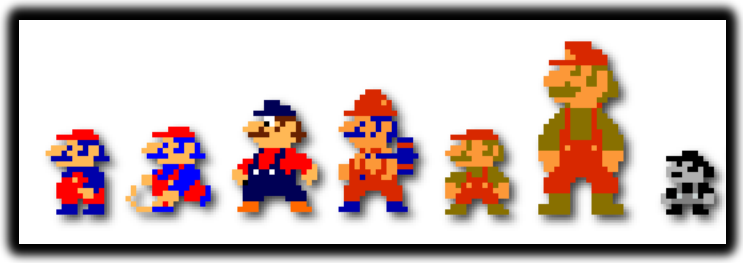
2. Glyphs & Fonts (The Avatar)

A **Glyph** is the actual drawing. A **Font** is a collection of these drawings.

Analogy: Think of **Video Game Skins**. You have one character (the ID), but you can swap skins (the Font) to make them look 8-bit, 4K, or cartoonish.

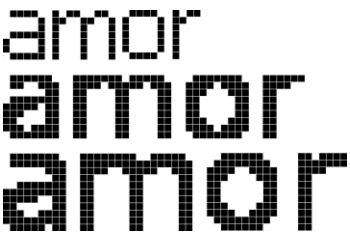
Type 1: Bitmap Fonts (The OG) 👾

These are old-school. They are made of a fixed grid of pixels, like a Mario sprite from 1985.



How they work: The font file is literally a "map of bits" (1s and 0s). 1 means "color this pixel," 0 means "leave it blank."

The Scaling Problem: Since they are fixed grids, they don't like being stretched.



Analogy: It's like a **Lego set**. If you try to make a small Lego house look "huge" by just standing closer to it, it just looks like big blocky plastic. It doesn't get more detailed; it just gets jagged (aliased).

RE Perspective: In WinAPI, if you see functions like CreateFont requesting specific pixel heights that don't match the bitmap's native size, Windows has to "guess" how to stretch it, usually resulting in an ugly, blurry mess.

💡 Practice Question?

"Since Bitmap fonts are just arrays of pixels, why do you think malware authors sometimes use custom bitmap fonts for their 'CrackMe' serial entry boxes instead of standard Windows TrueType fonts?"

Type 2: Vector Fonts (The Math Fonts) 📐

Instead of a grid of pixels, these fonts are **Math Formulas**.

How they work: Instead of saying "Put a pixel at X,Y," a vector font says "Draw a curve from Point A to Point B with this much bend."

The "Infinite Zoom" Hack: Because it's math, you can scale it to the size of a skyscraper or the size of a grain of rice, and it stays perfectly smooth.

Analogy: It's like a **Recipe**. If you have a recipe for 1 cake, you can multiply the ingredients by 100 to make a giant cake. The taste (quality) stays the same; only the scale changes.

RE Perspective: You won't see many "raw" vector fonts anymore; they were the ancestors of TrueType.

Type 3: TrueType Fonts (The Gold Standard)

This is what 99% of Windows uses. It's a Vector font with "Special Sauce" called **Hinting**.

TrueType (TTF) fonts *are* a type of vector font, using mathematical curves (quadratic splines) to define letter shapes, making them scalable without pixelation, just like other vector graphics (e.g., SVG files); the key difference is TTF is a font format for text input, while an SVG is a general graphic file.

The Problem: Even though math (vectors) is perfect, computer screens have pixels. Sometimes, a tiny math-defined curve lands right in the middle of a pixel, making it look blurry or "fuzzy."

The Solution (Hinting): TrueType fonts contain extra code (Hints) that tells Windows: *"Hey, if you're rendering me at a tiny size, nudge this line slightly to the left so it aligns perfectly with the pixel grid."*

Analogy: Think of a **Tailor**. The vector is the suit pattern, but the "Hinting" is the tailor pinning the fabric to fit a specific person's body so it doesn't look baggy.

Why this matters for Reversing 🛠️

When you are looking at **WinAPI** calls like CreateFontA or ChooseFont, you'll see parameters for nHeight and nWidth.

- If it's a **TrueType** font, Windows will use those math formulas to perfectly render the size you asked for.
- If you see an app loading a custom .ttf or .otf file from its resources, it might be doing its own text rendering to avoid standard Windows string-searching tools (like Strings.exe).

💡 Practice Question?

"If you are reverse-engineering a piece of malware and you see it loading a font file from memory rather than the C:\Windows\Fonts folder, why might the author be doing that? Is it just for aesthetics, or is there a 'stealth' reason?"

Compatibility: The "Translator" Problem 🧩

If your **App** speaks French (Latin-1) but your **Font** only speaks English (ASCII), the accented letters like é or ç will look like hot garbage (usually weird boxes or ?).

The Reality: A font is just a lookup table. If the App asks for ID #233 (é), and the Font table ends at #127, the Font has no "drawing" (glyph) to show.

It panics and shows a default "I don't know" symbol (the **Tofu** block).

Case Study: The KEYVIEW1 Glitch 🕵️

The Problem: KEYVIEW1 was trying to show foreign languages but used the **System Font**.

Why it failed: The System Font is an old-school **Bitmap Font**. It's lightweight and fast for Windows to draw, but it's "lazy"—it only carries a basic set of characters.

It doesn't have the drawings for Cyrillic, Kanji, or complex accents.

The Fix: The app needs to switch to a **TrueType Font** (like Arial or Tahoma). These are "Chonky" fonts—they contain thousands of glyphs for almost every language.

WinAPI Tool: GetStockObject 🛠️

Windows has a "closet" full of pre-made tools (Fonts, Brushes, Pens) called **Stock Objects**. Instead of creating a font from scratch, you just ask Windows to "hand you one."

- **Analogy:** It's like a **Rental Shop**. You don't want to build a ladder; you just call the shop and say, "Give me the standard ladder" (SYSTEM_FONT).
- **The Catch:** If you ask for SYSTEM_FONT, you get that old bitmap font. To fix KEYVIEW1, you'd ask for ANSI_VAR_FONT or a specific TrueType object.

Technical Snippet:

```
// Grabbing a pre-made font handle
HFONT hFont = (HFONT)GetStockObject(ANSI_VAR_FONT);

// Telling a window to use that font
SendMessage(hWnd, WM_SETFONT, (WPARAM)hFont, TRUE);
```

RE Insight: Finding Hidden Text

In malware analysis, if you see a program calling CreateFont or EnumFontFamilies, it might be checking if a specific language font exists. If it finds a "Chinese" font, it might execute a different payload.

The KEYVIEW1 Lesson: In your reversing journey, if you see weird ???? in a strings utility but the app displays them fine, it's because the app is using a specific **Character Set/Font** combo that your tool isn't mimicking.

💡 Practice Question?

"If I'm reversing a program and I see it calling GetStockObject with the value 17 (DEFAULT_GUI_FONT), is it likely using a jagged Bitmap font or a smooth TrueType font?"

The "Legacy" Bitmap Trio: System, FixedSys, & Terminal

This section is a trip down memory lane. These files (.FON) are the "ancestors" of modern typography.

1. The System Font (The "Default" Ghost)

This is the bare-bones font Windows uses when it doesn't know what else to do. It's a proportional font (meaning an 'i' is thinner than a 'W').

The Files: * VGAFIX.FON (Standard 96 DPI screens)

8514FIX.FON (High-res 120 DPI screens—"High-res" by 1990s standards!)

Where to find it: Open **Registry Editor** (regedit) and look at HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts. You'll see "System" pointing to one of these.

2. FixedSys (The Programmer's Antique)

Every character is the exact same width. It was the default for Notepad for a decade. It's incredibly easy for a CPU to render because it's just a grid.

The Files: Same as the System font (VGAFIX.FON), but accessed differently by the OS.

Where to find it: Open **Notepad**, go to *Edit* > *Font*, and look for "Fixedsys". If you type iiiiii and WWWWWW, they will line up perfectly.

3. The Terminal Font (The OEM/Console Vibe)

This font is used for the "Command Prompt." It uses the **OEM Character Set** (the old DOS-style box-drawing characters like ┌ and ■).

The Files: * VGAOEM.FON (96 DPI) and 8514OEM.FON (120 DPI)

Where to find it: Open a **Command Prompt (cmd.exe)**, right-click the title bar, go to **Properties**, then the **Font** tab. You will see "Terminal" as an option.

RE Perspective: Why do we care about .FON files? 🕵️

Malware authors (especially those making **Keyloggers** or **Console-based tools**) love these fonts because they are "embedded" in the system. They don't need to install anything.

If you see a binary loading a file with a .FON extension instead of .TTF, you know it's trying to be:

1. **Fast:** No math involved, just blitting pixels to the screen.
2. **Retro/Stealthy:** It might be using those old DOS box-drawing characters to build a fake "BIOS" screen or a "Technical Support" scam window.

📂 Real-World Path

You can find most of these legacy files sitting quietly in: C:\Windows\Fonts\ *(Note: Windows often hides them from the standard Font Viewer because they are System files. You might need to use the Command Prompt to dir them!)*

💡 Practice Question?

"If you're reversing a program and it's using the **Terminal** font, and you see it printing characters with hex values like 0xDB or 0xB1, what kind of visual UI is the program likely building?"

MS Sans Serif: The "Classic Windows" Suit

This is the default bitmap font for old-school buttons, menus, and labels. It's clean, but because it's a **Bitmap**, it only looks good at specific "Sweet Spot" sizes.

- **The File:** SSERIFE.FON
- **The "Sweet Spots":** 8, 10, 12, 14, 18, and 24 points.
- **Analogy:** Think of it like **Digital Zoom** on an old camera. At 1x or 2x zoom, the photo is crisp. If you try to zoom into 1.5x, the camera has to "fake" the pixels, and everything looks blurry and jagged. This font works the same way—it's pixel-perfect only at those specific sizes.

The Magic Button: DEFAULT_GUI_FONT 🔘

In the old days, programmers didn't want to manually pick a font for every button. They just told Windows: *"Give me the default look for this screen."*

- **How it works:** You pass this ID to GetStockObject(DEFAULT_GUI_FONT).
- **The Intelligence:** Windows looks at your screen resolution (DPI). If you're on a tiny screen, it gives you a small version; if you're on a big screen, it gives you a larger one.
- **Analogy:** It's like ordering the **"Chef's Special"** at a restaurant. You don't know exactly what ingredients they'll use today, but you trust it'll be the standard, reliable meal that fits the occasion.

The "Standard Issue" Stock Fonts 📦

Windows keeps a few other fonts in its "inventory" (Stock Objects) that you can grab instantly.

CONSTANT	THE VIBE	REAL WORLD USE
`ANSI_FIXED_FONT`	Monospaced (Fixed-pitch). Every letter is the same width.	*Think Notepad or a simple Hex Editor view. Great for alignment.*
`ANSI_VAR_FONT`	Proportional. "W" is wider than "i".	*Standard reading text in most older Windows apps.*
`DEVICE_DEFAULT_FONT`	The "Hardware Choice."	*Whatever font the specific device (like an old dot-matrix printer) prefers.*

Reverse Engineering Perspective: Hunting for UI Clues

When you're reversing a stripped binary (no symbols, no names) and you see GetStockObject called with the value 17 (which is DEFAULT_GUI_FONT):

1. **UI Identification:** You know the program is about to draw a standard Windows Dialog or Window.
2. **Age Check:** If the program relies heavily on SSERIFE.FON, it's likely a legacy tool or intentionally designed to be "Old School" to stay small and fast.
3. **Hooking Opportunity:** If you want to mess with how an app looks, you can "hook" GetStockObject. Every time the app asks for DEFAULT_GUI_FONT, you can force it to return a handle to *Comic Sans* just to be chaotic.

Practice Question?

"If I have a modern 4K monitor and an old program asks for MS Sans Serif at size 10, but the SSERIFE.FON file doesn't have a 4K-ready version of that size, what do you think Windows does to the font to make it visible? (Hint: Think back to our 'Lego' analogy)."

Program code in StockFont folder chapter 6...

STOKFONT: The Font Explorer

The TL;DR

It's a 16 * 16 grid that shows you all 256 possible characters in a specific font. If a font is a "book of drawings," this app is the **Table of Contents**.

1. The Entry Point (WinMain)

- **The Vibe:** This is the "Ignition Switch." It starts the engine, creates the window, and begins the **Message Loop**.
- **The Loop:** A while(GetMessage(...)) loop is just a hungry monster waiting for you to do something (click, scroll, or press a key). Once it gets a "snack" (a message), it throws it to the WndProc.

2. The Brain (WndProc)

- **The Vibe:** This is the **Traffic Controller**. It sits there and decides what to do based on the message.
    - ✓ **User scrolls?** "Update the font ID."
    - ✓ **Window needs redrawing?** "Go to WM_PAINT."
    - ✓ **Screen resolution changes?** (WM_DISPLAYCHANGE) "Recalculate the layout."

3. The Canvas (WM_PAINT)

This is the most important part of the code for us.

The Logic: 1. Grab the "Hand" (HDC): Get the Device Context.

2. Pick the Font: Use GetStockObject(iFontID).

3. Hold the Font: Use SelectObject(hdc, hFont). This is like putting a specific stamp in your hand before you start stamping paper.

4. The Nested Loop: It runs two loops (one for rows, one for columns) to draw the 16 * 16 grid.

5. Drawing: It uses TextOut() or DrawText() to put the character on the screen.

4. The "Character Set" (The Secret Decoder Ring) 🌎

The notes mention that the Character Set identifier is **crucial**. Why?

The Problem: Computers only understand numbers (0 to 255).

The Decoder: The "Character Set" is the **Mapping**.

- ✓ If the ID is ANSI_CHARSET, number 128 might be a Euro symbol.
- ✓ If the ID is SYMBOL_CHARSET, number 128 might be a Greek letter or a tiny skull icon.

RE Perspective: If you are reversing an app and see it using SYMBOL_CHARSET, it's likely not showing "text"—it's probably drawing UI icons or custom symbols to make the app look unique (or to hide its true strings).

5. Reversing the STOCKFONT Logic 🕵️

If you were looking at this program in a debugger (like x64dbg), here is what you would see:

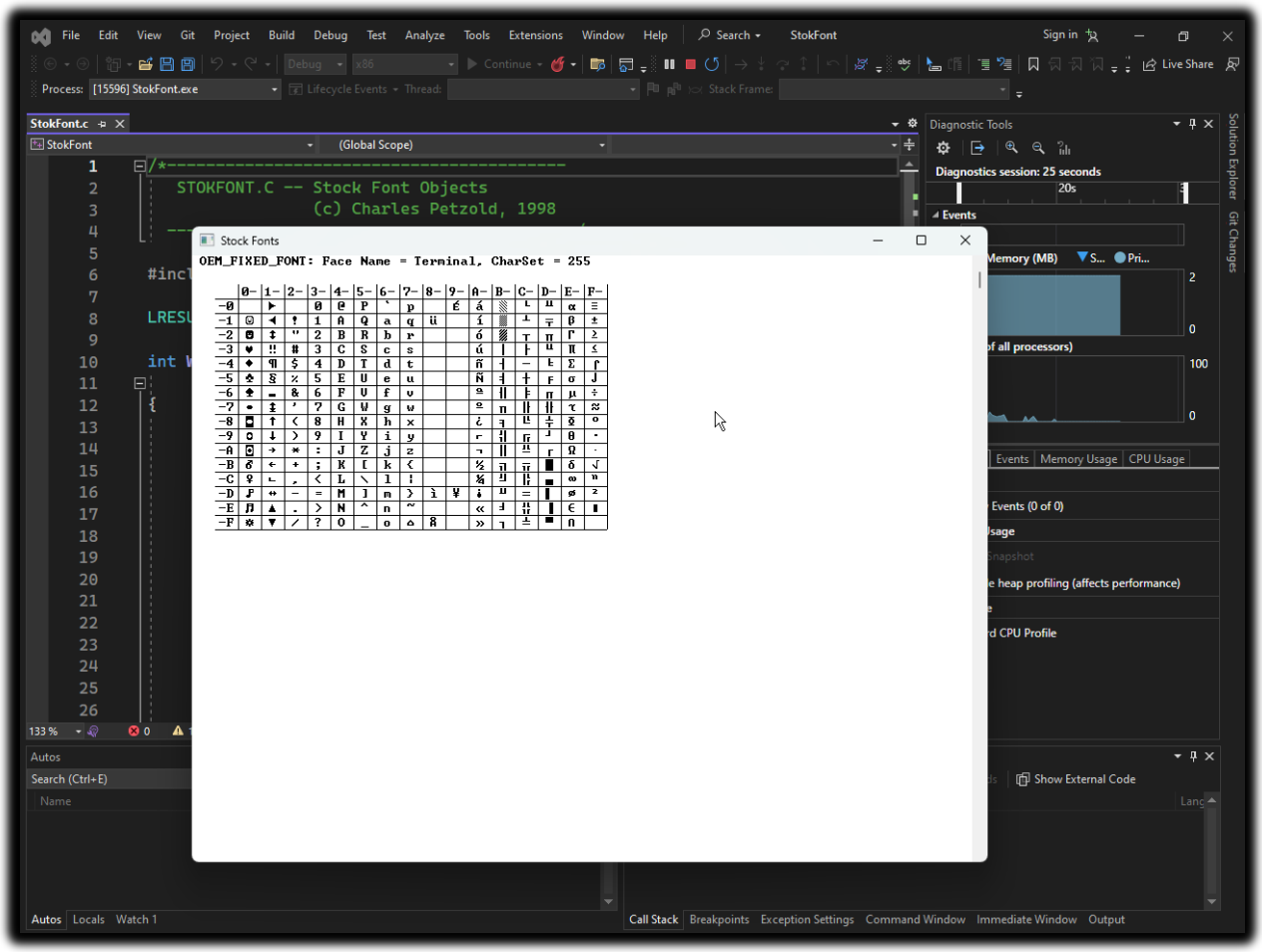
1. **The "SelectObject" Pattern:** You'll see a call to GetStockObject followed immediately by SelectObject. This is the classic WinAPI "Set-up" pattern.
2. **The Loop:** You'll see a CMP (compare) and a JNE/JLE (jump) instruction that repeats 16 times for the columns, inside another loop that repeats 16 times for the rows.
3. **The Interaction:** When you scroll, WndProc receives WM_VSCROLL. In the assembly, you'll see the code updating a global variable (the iFontID) and then calling InvalidateRect.

InvalidateRect basically tells Windows, "This window is dirty, please call WM_PAINT to clean (redraw) it."

💡 Practice Question?

"If I change the SelectObject call in STOKFONT to use a random HPEN (a pen for drawing lines) instead of an HFONT, what do you think TextOut will do? Will it crash, or will it just look weird?"

Output:


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
StokFont
Sign in
Live Share
Process: [15596] StokFont.exe
StokFont.c
StokFont
(Global Scope)
/*--
STOKFONT.C -- Stock Font Objects
(c) Charles Petzold, 1998
#incl
LRESU
int W
{
Stock Fonts
OEM_FIXED_FONT: Face Name = Terminal, CharSet = 255
Diagnostic Tools
Diagnostics session: 25 seconds
20s
Events
Memory (MB)
S...
Pri...
2
0
of all processors)
100
0
Events
Memory Usage
CPU Usage
Events (0 of 0)
Snapshot
e heap profiling (affects performance)
d CPU Profile
133 %
0
Autos
Search (Ctrl+E)
Name
Show External Code
Autos
Locals
Watch 1
Call Stack
Breakpoints
Exception Settings
Command Window
Immediate Window
Output
Solution Explorer
Git Changes

The Great Character Set Battle: OEM vs. ANSI ⚔️

This is the "final boss" of font basics—understanding how Windows handles different languages and legacy systems. Let's break down these two competing worlds: **OEM** (The Past) and **ANSI** (The Present).

1. The OEM Character Set (The "Retro" Set)

- This is the character set created by IBM for the original PC. It's what **MS-DOS** used.
- It's famous for **"Box-Drawing"** characters. Before we had fancy buttons, we used characters like ┌, ─, and ■ to draw fake windows in a black terminal.
- Think of it like **ASCII Art**. It wasn't just for letters; it was for building "graphics" out of text.
- **RE Perspective:** If you see a binary using OEM_CHARSET, it might be a console tool or a legacy piece of malware trying to draw a "cool" retro interface in a command prompt.

2. The ANSI Character Set (The "Modern" Standard)

- This is the default for Windows. It's based on **Latin-1**.
- It dumped the "box-drawing" stuff to make room for 64 extra characters used in Western Europe (like ©, ®, and accented letters like ñ).
- This is the **"Global Passport"** for Western languages. It's clean, standard, and what you'll see in 99% of modern .txt files or Windows apps.

3. The Two "System" Fonts 🖥️

Windows keeps two "Old Reliable" fonts in its back pocket:

1. **System Font (SYSTEM_FONT):** Usually uses the **OEM** set. This is the OS's "Panic" font—it's what it uses for basic dialogs and menus if nothing else is specified.
2. **System Fixed Font (SYSTEM_FIXED_FONT):** Usually uses the **ANSI** set. It's "Fixed-Pitch," meaning every letter is the same width. **Analogy:** Think of this as the **Typewriter Font**. Perfect for code because columns always line up.

4. Internationalization: The "Local Rules" 🗺️

In your 2022 notes, you mentioned that these change in Russian or Greek versions of Windows.

- **The Logic:** Windows is a shapeshifter. If you install the Russian version, SYSTEM_FONT doesn't just change its *look*; it swaps its **Decoder Ring** to the Cyrillic character set.
- **Why it breaks things:** If you write a program that *only* understands ANSI and you run it on a Greek machine, your text will turn into **"Mojibake"** (random nonsense characters).

5. The Developer's "Font Toolbelt" 🔧

When you're reversing or coding, these three functions are your "Inspectors":

- **GetStockObject:** The "Vending Machine." You put in an ID (like SYSTEM_FONT), and it gives you a handle (HFONT).
- **GetTextFace:** The "ID Check." It tells you the name of the font (e.g., "Arial" or "System").
- **GetTextMetrics:** The **"Biometrics."** This gives you a TEXTMETRIC struct, which is like a medical report for the font. It tells you the height, the width, and—most importantly—the **Character Set** it uses.

6. RE Insight: The "Environmental Check" 🕵️

Malware often checks the "Locale" or "Character Set" of a machine before doing anything.

- **The Logic:** If a Russian hacker writes malware, they might code it to **self-destruct** if it detects the Cyrillic character set (RUSSIAN_CHARSET). Why? To avoid getting arrested by their local police!
- **What to look for:** In your debugger, keep an eye on GetTextMetrics. If you see the code checking the tmCharSet field, it's trying to figure out what country it's currently "visiting."

Practice Question? 💡

"If I use GetTextFace on a handle I got from GetStockObject(SYSTEM_FONT), and it returns 'System', but all my text is coming out as Greek letters, where is the 'problem'—in my code, or in the OS settings?"

Unicode Upgrade for KEYVIEW1.

This section marks the transition from "Old Windows" (ANSI) to "Modern Windows" (Unicode). We are taking the exact same program (KEYVIEW1) and tweaking it to handle 16-bit characters instead of 8-bit ones.

1. The Core Change: RegisterClassW

The most important line in the upgrade is how you register the window.

Original (ANSI): RegisterClassA

Result: Windows sends WM_CHAR messages where wParam is an **8-bit** number (0-255).

New (Unicode): RegisterClassW

Result: Windows sends WM_CHAR messages where wParam is a **16-bit** number (0-65,535).

Note: Just by changing this one function call, you fundamentally change the data your Window Procedure receives.

2. The "Solid Block" Mystery

The notes mention a specific bug: when you type Greek or Russian in the Unicode version, you see **Solid Blocks** instead of letters.

Why does this happen? It is a conflict between **The Code** (Logic) and **The Paint** (Visuals).

1. **Logic (Success):** Your program successfully received the correct Unicode ID for the Greek letter (e.g., 0x03A9 for Omega).
2. **Visuals (Fail):** You are still using SYSTEM_FIXED_FONT. This is an old, simple font that only contains pictures (glyphs) for the first 256 characters (English/Western).
3. **The Result:** When you ask that font to "Draw character #937 (Omega)," it looks in its library, realizes it stops at #255, and draws its "Error/Missing" symbol: a solid block or rectangle.

The Fix: To see the actual Greek letters, you would need to select a modern font (like "Courier New" or "Lucida Console") that includes those foreign glyphs.

3. Differences: ANSI vs. Unicode Versions

FEATURE	ANSI VERSION	UNICODE VERSION
Registration	`RegisterClassA`	`RegisterClassW`
Data Size	8-bit (`char`)	16-bit (`wchar_t` / `TCHAR`)
Drawing Function	`TextOutA`	`TextOutW`
Capacity	256 Characters (Confusing overlaps)	65,536+ Characters (Unique IDs)

4. Why Unicode Wins (The Benefits)

The notes highlight three key advantages:

1. **Unambiguous Codes:**
   - ✓ **In ANSI:** The number 0xE1 might be "á" (Western) or "α" (Greek). You have to guess based on the active keyboard.
   - ✓ **In Unicode:** "á" is always 0x00E1. "α" is always 0x03B1. There is zero confusion.
2. **Global Display:**
   - ✓ You can mix languages. You can have English, Russian, and Arabic text on the same line, because they all have different ID numbers.
3. **GDI Power:**
   - ✓ Modern Windows graphics engines are built for Unicode. Using it makes your app faster and more compatible with newer features.

🚀 TrueType: Breaking the "256" Barrier

Bitmap fonts are like a small box that only fits 256 LEGO bricks. **TrueType fonts** are like a digital blueprint that can generate an infinite number of bricks.

- **The 256 Limit:** Old bitmap fonts used 1 byte (8 bits) to identify a character. $2^8 = 256$. That's it. You ran out of room fast.
- **The TrueType Flex:** Because TrueType is based on **Math (Vectors)**, it doesn't just store "pixels"; it stores instructions. This allows it to hold thousands of characters (glyphs) in a single file, covering everything from English to Ancient Greek.

🌎 "Big Fonts": The Multilingual Mega-Packs

In the Windows 98/NT era, Microsoft introduced "Big Fonts" (like Arial, Tahoma, and Times New Roman).

Think of a font file (like Arial.ttf) as a **multilingual toolbox**. Instead of having ten different files for ten different languages, Windows stores them all inside one single font file.

🛠️ The Character Set "Menu"

Inside that one font file, you have different **Character Sets** (sub-sections) that the program can choose from:

- **Western (ANSI):** The standard set for English, French, and German.
- **Cyrillic:** The set used for Russian and Bulgarian.
- **Greek:** The set used for Greek letters (often used in math and science).
- **Specialty Sets:** Other specific sections for Turkish, Baltic, or Central European languages.

In Windows programming, CreateFont is your way of building a font from scratch. While GetStockObject gives you a pre-made "Stock" font, CreateFont lets you customize every detail.

🛠️ The DIY Font Builder

If GetStockObject is like ordering a "Pre-set Meal," CreateFont is like a **Custom-Built Burger**. It has 14 different settings (parameters), but you usually only need to worry about these three:

- **lfFaceName (The Brand):** The name of the font you want to use (e.g., "Arial" or "Times New Roman").
- **lfCharSet (The Language):** This is the "Decoder Ring." It tells Windows which language section of the font to use (e.g., CYRILLIC_CHARSET for Russian).
- **lfPitch (The Style):** This decides if the font is **Fixed-width** (like a typewriter where every letter is the same size) or **Proportional** (where an "i" is thinner than a "W").

💡 One Goal, Two Different IDs

Windows has a confusing quirk: it uses two different numbers to identify the same language sets. Think of it like a person having both a **Social Security Number** (for the government) and a **Driver's License Number** (for the road).

🔢 The Two ID Types

- **Character Set ID (1 Byte):** This is the "Old School" ID. It was created for Windows 1.0 when memory was tiny. It is used specifically for **Fonts** (the LOGFONT structure).
- **Code Page ID (2 Bytes):** This is the "Modern" ID. It was added later to handle the complex mapping of thousands of different characters.

❓ Why does Windows use both?

1. **History:** Windows keeps the old 1-byte ID so that very old programs still work (Backward Compatibility).
2. **MS-DOS Support:** The Character Set ID helps Windows talk to the old MS-DOS "OEM" characters.
3. **Global Use:** The Code Page ID allows Windows to support almost every language on Earth at the same time.

How to use them in your code

If you want to...	Use this ID	Use this Function
Build a Font	Character Set ID	`CreateFont()`
Convert Text	Code Page ID	`MultiByteToWideChar()`

Handling Keyboard Changes (WM_INPUTLANGCHANGE).

When a user changes their keyboard language (like switching from English to Russian in the taskbar), Windows sends a WM_INPUTLANGCHANGE message to your program.

- **The Shortcut:** Windows puts the **Character Set ID** of the new language directly into the wParam.
- **The KEYVIEW2 Logic:** This sample program watches for that message. As soon as you switch keyboards, it takes that ID from wParam, calls CreateFont, and instantly updates the window so the new language displays correctly.

RE Perspective: The 14-Arg Monster

When reversing a program that supports multiple languages, look for MultiByteToWideChar. If you see a hardcoded **Code Page ID** like 1251 (Cyrillic) or 932 (Japanese), you know exactly what region that malware or tool was built to target.

When you're looking at assembly, CreateFont is a beast because it pushes **14 values** onto the stack (or uses many registers in x64).

The Trap: Malware often uses CreateFont to make text look "invisible" or "weirdly sized" so it doesn't show up correctly if you're just looking at a screen capture.

The Tell: If you see CreateFontA (the ANSI version) being called with a weird lfCharSet value (like 0xA1 for Greek), you know the app is targeting a specific region.

When you start analyzing those malware samples, keep a close eye on MultiByteToWideChar. In malware analysis and reverse engineering, this function is a major pivot point.

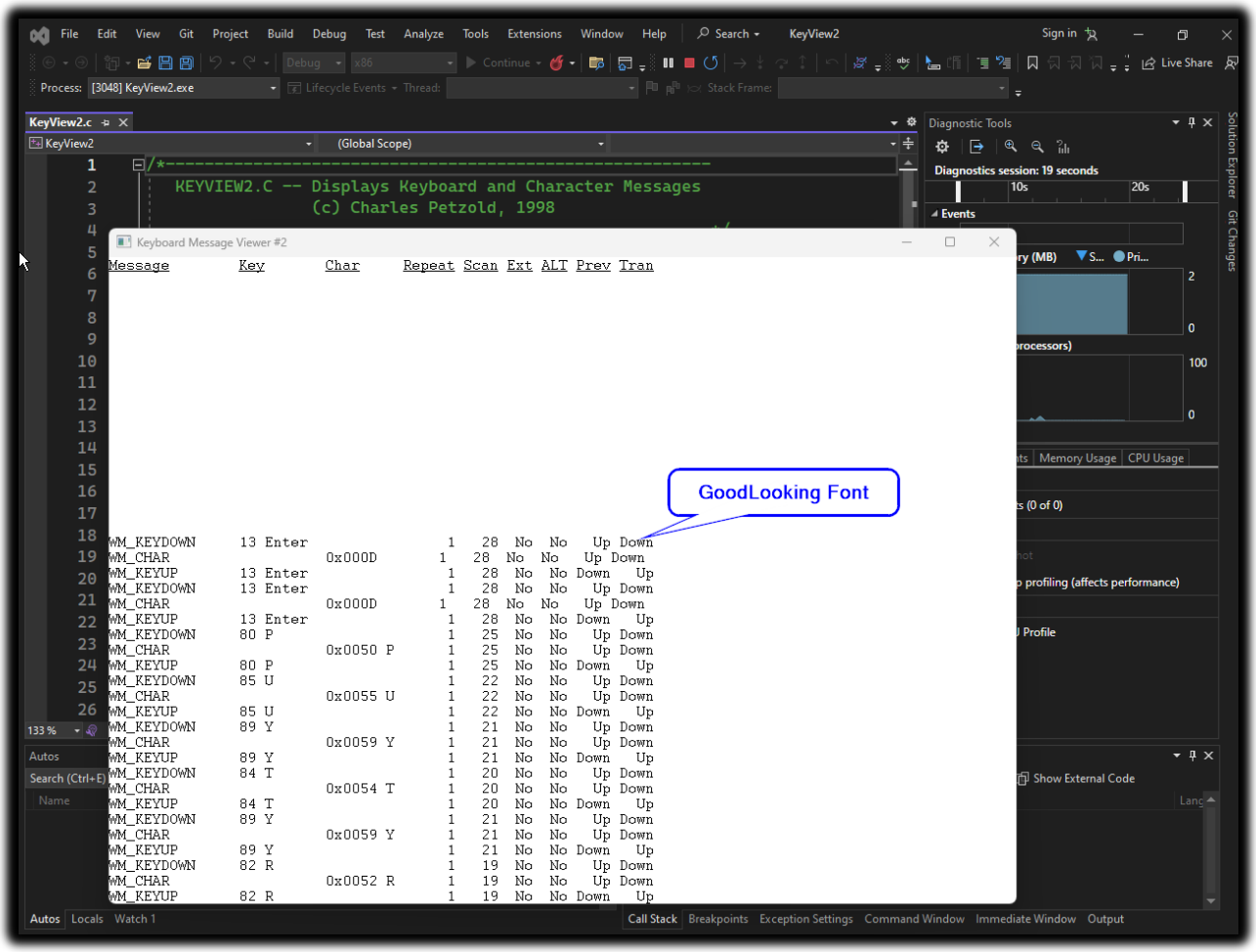
The Story: Imagine a piece of malware trying to hide its strings. It might store them as obfuscated ASCII (MultiByte) and only convert them to UTF-16 (WideChar) right before calling a sensitive API like CreateFileW.

The Analogy: MultiByteToWideChar is the translator at the border. If you set a breakpoint on this function, you can often see the "true" string being born just before the program uses it to do something malicious.

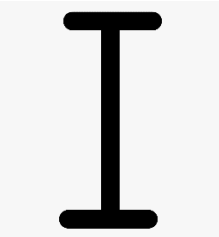
💡 Practice Question?

"I'm reversing a program and I see it calling CreateFont with the lfFaceName set to 'Wingdings'. If I try to read the strings it prints using TextOut, will they look like English words or something else?"

KeyView2 program folder in Chapter 6...



What is the Caret?



The Caret: The "Insert Here" Beacon

The Caret is that blinking line or box that tells the user: "If you type right now, the text lands *here*."

The "One Golden Pen" Analogy

Imagine a massive office building (Windows) where everyone shares **one single golden pen** (The Caret).

- When you walk into a room (Window), you pick up the pen and start using it.
- When you leave the room, you **must** destroy that pen so the person in the next room can forge their own.
- If you leave the room and keep the pen in your pocket, the next person is stuck clicking around with nothing to show for it. **This is why we create on Focus and destroy on Kill Focus.**

The Caret Toolbelt (The "Big 5")

Function	Technical Description
`CreateCaret()`	**Initialization:** Defines the caret shape (width, height, or bitmap). Created in a hidden state.
`SetCaretPos()`	**Positioning:** Moves the caret to specific X and Y client coordinates.
`ShowCaret()`	**Visibility:** Displays the caret and begins the blinking animation.
`HideCaret()`	**Concealment:** Removes the caret from the screen. Hide calls are cumulative.
`DestroyCaret()`	**Termination:** Deletes the current caret and frees the resource.

👮 The Rules of Engagement

1. The Focus Handshake (Crucial!)

You don't just own a caret; you "rent" it when you are the **Main Character** (have Focus).

- **WM_SETFOCUS:** "I'm in charge now!" → **Call CreateCaret + ShowCaret.**
- **WM_KILLFOCUS:** "I'm heading out." → **Call DestroyCaret.**

2. The "Don't Paint Over Me" Rule

The caret is basically a tiny ghost that flips pixels on and off to blink. If you try to paint your window while the caret is visible, you might get "pixel poop" (graphical glitches).

The Workflow: HideCaret → *Do your drawing* → ShowCaret.

3. The "Invisibility Stack" (The Bug Trap) 🪤

HideCaret is **additive**. If you call it 3 times, you have to call ShowCaret 3 times to see it again.

Analogy: It's like putting three blankets over the caret. Removing one blanket still leaves it hidden.

🕵️ Reverse Engineering Perspective: Caret Shenanigans

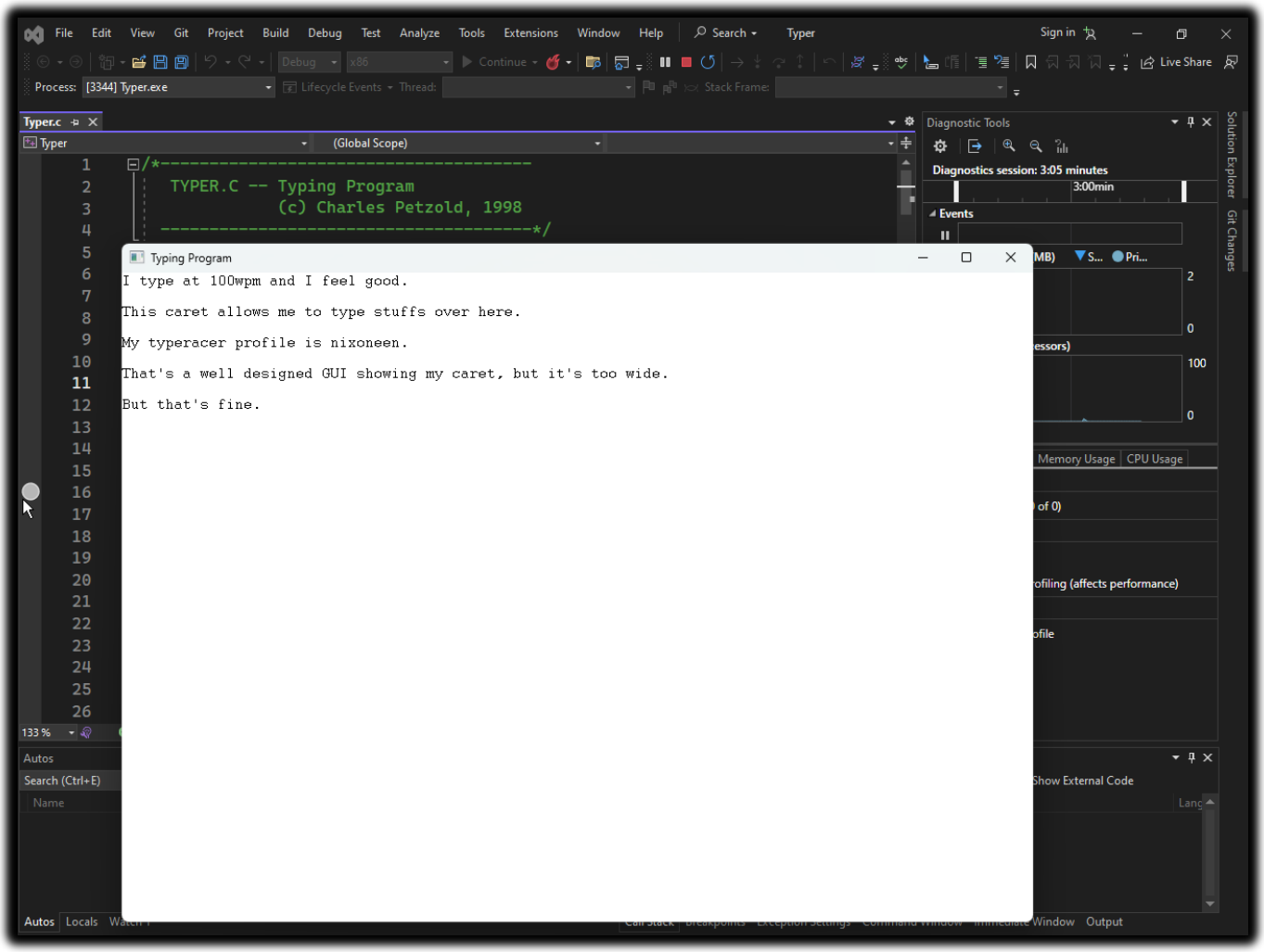
In the world of **Malware and Reverse Engineering**, the caret is a signal.

1. **Fake Login Screens:** If you're looking at a piece of malware that spawns a fake "Windows Update" or "Bank Login" window, checking the WndProc for CreateCaret helps you identify where the "Username/Password" fields are located in the code.
2. **Anti-Analysis:** Some weird programs might call SetCaretBlinkTime to something insane (like 0 or 10,000ms) to see if you're a human or an automated sandbox. If the "human" doesn't notice the blink is broken, the malware might stay dormant.
3. **Keyloggers:** A keylogger doesn't need a caret, but it *watches* for WM_SETFOCUS. When a window gets focus and creates a caret, the logger knows: "Okay, the user is about to start typing somewhere important."

💡 Practice Question?

"If I call CreateCaret in WM_CREATE instead of WM_SETFOCUS, and the user switches to a different app (like Notepad), what happens to the blinking line in my app? Does it keep blinking, or does it vanish?"

The typer program in Chapter 6 Folder Typer folder...



TYPER: The Basic Text Editor Logic

1. Caret Management (The Focus Rule)

The TYPER program follows the strict WinAPI rule for carets:

- **When the window is active (WM_SETFOCUS):** The program calls CreateCaret and ShowCaret.
- **When the window loses focus (WM_KILLFOCUS):** The program calls HideCaret and DestroyCaret.
- **Why?** Windows only allows one caret to blink across the entire operating system at any given time.

2. Character Storage and Fixed-Pitch Fonts

To keep the logic simple, TYPER uses a **Fixed-Pitch font** (like Courier or FixedSys).

- **The Advantage:** Every character is the same width. This makes calculating the caret position easy. If a character is 8 pixels wide, the 5th character is always at $8 \times 5 = 40$ pixels.
- **The Buffer:** The program allocates a block of memory (a buffer) to store every character the user types.

3. Handling Input: WM_KEYDOWN vs. WM_CHAR

TYPER splits keyboard input into two categories:

- **WM_KEYDOWN:** Handles "Action" keys that don't produce a letter (Arrow keys, Home, End, Delete).
- **WM_CHAR:** Handles "Content" keys (Letters, Numbers, Backspace, Enter, Tab).

4. The Drawing Safety Rule

Whenever TYPER needs to update the screen (like typing a new letter) outside of a WM_PAINT message, it follows this sequence:

1. **HideCaret**: Stop the blinking so the drawing doesn't look messy.
2. **Draw the text**: Put the new character on the screen.
3. **ShowCaret**: Turn the blinking back on.

🕵️ Reverse Engineering Perspective: Malware & Input

When you are analyzing malware, understanding the TYPER logic helps you find where "data entry" happens.

- **Keylogging:** Many keyloggers do not use GetMessage. They use **Hooks** (SetWindowsHookEx) to steal messages like WM_CHAR before they ever reach the TYPER program.
- **Screen Scraping:** If malware wants to steal what you typed in a custom editor, it might call GetCaretPos. By knowing where the caret is, it can determine exactly which part of the window the user is interacting with.
- **Buffer Overflows:** In simple programs like TYPER, the text is stored in a buffer. If the program does not check the size of the buffer when you type, a researcher could perform a "Buffer Overflow" by typing too many characters, potentially overwriting the program's return address in memory.
- **Invisible Inputs:** Malware often creates hidden windows with carets to trick the OS into thinking a user is active, or to capture input meant for another application (UI Redressing/Clickjacking).

🏁 Final Review: 20 Questions to Master Fonts, Character Sets, and Carets

These questions cover everything we have revamped. Use these to test your intuition for technical work and RE.

1. What is the main difference between a **Character** and a **Glyph**?
2. Why does a **Bitmap Font** look jagged when you scale it up too large?
3. In WinAPI, which function do you use to grab a pre-defined font like SYSTEM_FONT?
4. Why did the **KEYVIEW1** program show incorrect characters for foreign languages?

5. What is the "hinting" process in **TrueType** fonts used for?
6. If you see VGAFIX.FON on a system, what specific font type is it storing?
7. What is the "additive" property of the HideCaret function?
8. Why should you never call CreateCaret during WM_CREATE?
9. Which message tells a window it is time to destroy its caret?
10. What is the **OEM Character Set** famous for (historically used in DOS)?
11. Which character set is the default for Western European Windows applications?
12. If a malware author uses CreateFont with lfCharSet set to SYMBOL_CHARSET, what is their likely goal?
13. How does a **Fixed-Pitch** font make it easier to calculate the xCaret position?
14. What happens if you try to draw on a window without calling HideCaret first?
15. Which WinAPI function provides "biometric" details about a font, like its height and average width?
16. In the TYPER program, which message handles the **Arrow Keys**?
17. Why are **TrueType** fonts better for international applications than Bitmap fonts?
18. What is the danger of a program not checking its input buffer size during WM_CHAR?
19. What does the GetStockObject identifier DEFAULT_GUI_FONT actually provide to the user?
20. From an RE perspective, why would you monitor an application's calls to SetCaretBlinkTime?