CHAPTER 17: TEXTS AND FONTS

Text rendering was our **first real contact with graphics** in WinAPI.

Now we go deeper — not just drawing letters, but **controlling how text behaves, scales, aligns, and interacts with graphics**.

This chapter focuses on:

- How Windows represents fonts internally
- Why **TrueType** changed everything
- How WinAPI lets you manipulate text beyond basic output
- How proper text alignment improves UI quality

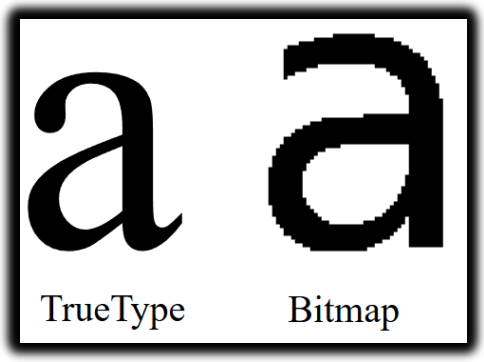
This is where WinAPI text stops being *static* and becomes *programmable*.

TrueType Fonts — Why They Changed Windows Forever

Before TrueType, Windows relied heavily on **bitmap fonts**.

Those fonts were fixed-size, pixel-based, and ugly when scaled.

TrueType, introduced in **Windows 3.1**, replaced that model.



What makes TrueType different?

TrueType fonts describe characters using **mathematical outlines**, not pixels. This single design choice unlocked several critical advantages:

✓ Scalability (WYSIWYG becomes real)

- Fonts can be scaled to any size without distortion
- What you see on screen matches what prints
- No pixelation, no manual font swapping

✓ Platform independence

- The same font files work across systems
- Windows and macOS both support TrueType
- Fonts behave consistently across devices

✓ Programmable geometry

Because characters are outlines, they can be:

- Transformed
- Rotated
- Used as shapes
- Combined with other graphics operations

This is why modern font manipulation is even possible.

Beyond Drawing Text — What WinAPI Actually Allows

With TrueType and WinAPI, text is no longer just “draw and forget”.

You can treat text like **vector graphics**.

I. Scaling Text

You can resize text dynamically without quality loss by:

- Creating fonts with specific logical heights
- Adjusting mapping modes
- Using scalable font resources

Text scaling is clean because TrueType does the math — not pixels.

II. Rotating Text

WinAPI allows rotated text through:

- Logical font orientation
- Glyph outline extraction
- Manual drawing using polygon functions

This enables:

- Vertical text
- Angled labels
- Circular or decorative layouts

Rotation is not a hack — it’s built into how glyphs are defined.

III. Pattern-Filled Text

Instead of solid colors, text interiors can be:

- Filled with patterns
- Filled with bitmaps
- Filled using custom brushes

This works by:

- Converting text outlines into paths
- Filling those paths using GDI drawing functions

At this point, **text behaves like a shape**, not a string.

IV. Fonts as Clipping Regions

One of the most powerful ideas in this chapter:

Text can define where drawing is allowed.

By converting font outlines into regions:

- Text becomes a mask
- Other graphics render *inside* the letters
- Backgrounds, gradients, or animations can appear inside text

This is advanced GDI usage — and very underused.

V. Text Alignment and Justification

Bad alignment makes an app feel amateur.
WinAPI gives you control — you just have to use it.

Alignment options:

- Left-aligned
- Right-aligned
- Centered
- Baseline-controlled

Using functions like:

- SetTextAlign
- GetTextExtentPoint32

You can calculate text dimensions and position text **precisely**, not by guessing.

Proper alignment:

- Improves readability
- Makes layouts predictable
- Gives the UI a finished look

VI. Why This Chapter Matters

This chapter isn't about fonts.

It's about:

- Treating text as **data + geometry**
- Understanding how Windows renders characters
- Moving from "drawing strings" to **designing typography**

Once you understand this:

- Custom controls make sense
- Owner-drawn UI becomes easier
- Graphics + text stop fighting each other

WINDOWS TEXT OUTPUT FUNCTIONS

Windows provides multiple text output functions, each designed for a **different level of control**.
Some are low-level and precise. Others trade precision for convenience.

Understanding **when to use which** is the real lesson here.

TextOut — The Fundamental Text Function

TextOut is the **most basic and direct** way to draw text in WinAPI.

Purpose

- Outputs a string at a specific position in a device context
- No formatting, no wrapping, no background handling by default

Arguments

- hdc
  Handle to the device context where text is drawn.
- xStart, yStart
  Starting position in **logical coordinates**.
- pString
  Pointer to the character buffer.
- iCount
  Length of the string in characters.
  ⚠ **Not NULL-terminated** — you must specify the length explicitly.

Coordinate Behavior

By default, Windows draws text starting at the **upper-left corner of the first character**

Coordinates are affected by:

- Mapping mode
- Current font
- Text alignment flags

TextOut does exactly what you tell it — nothing more.

Text Positioning and Alignment

Text positioning isn't just about x and y values.
It's also controlled by **text alignment state**.

I. SetTextAlign

SetTextAlign changes how Windows interprets the xStart and yStart parameters.

Horizontal alignment flags

- TA_LEFT
- TA_RIGHT
- TA_CENTER

Vertical alignment flags

- TA_TOP
- TA_BOTTOM
- TA_BASELINE

These flags determine **what part of the text** is anchored to the coordinates.

II. TA_UPDATECP — Current Position Mode

When you include TA_UPDATECP:

- xStart and yStart in TextOut are ignored
- Windows uses the **current position**
- The current position is updated *after* drawing text

The current position is set using:

- MoveToEx
- LineTo

This mode is especially useful for:

- Sequential text output
- Multiline text rendering
- Console-like layouts

⚠ Note: TA_CENTER does **not** update the current position.

III. TabbedTextOut — Text with Columns

When text contains **tab characters**, using multiple TextOut calls becomes messy.

TabbedTextOut solves this.

What it does

- Expands \t characters into aligned columns
- Uses an array of tab stop positions
- Ideal for tables, lists, and columnar layouts

Key arguments

- iNumTabs
  Number of tab stops.
- piTabStops
  Array of tab stop positions (in pixels).
- xTabOrigin
  Starting reference point for tab calculations.

Default behavior

If iNumTabs is 0 or piTabStops is NULL:

- Tabs are spaced at **every 8 average character widths**

This gives predictable alignment without manual calculations.

IV. ExtTextOut — Precision Control

ExtTextOut is where **real control begins**.

It extends TextOut by adding:

- Clipping
- Background filling
- Per-character spacing

Important arguments

IOptions - Controls rendering behavior:

- ETO_CLIPPED → Clip text to a rectangle
- ETO_OPAQUE → Fill background rectangle before drawing text

&rect - Used for:

- Clipping when ETO_CLIPPED is set
- Background fill when ETO_OPAQUE is set

You can use one, both, or neither.

V. Intercharacter Spacing

The last parameter, pxDistance, allows **manual spacing control**.

- It's an array of integers
- Each value defines spacing after a character
- Setting it to NULL uses default spacing

This is extremely useful for:

- Text justification
- Tight columns
- Custom typography effects

This is **low-level typography control**, not cosmetic fluff.

VI. DrawText — High-Level Convenience

DrawText is a **layout engine**, not just a drawing function.

What it handles for you

- Word wrapping
- Alignment
- Vertical positioning
- Tab expansion
- Clipping

Instead of coordinates, you provide a **rectangle**, and Windows does the rest.

Arguments

- hdc
  Device context
- pString
  Pointer to the string
- iCount
  Length of string
  Use -1 for NULL-terminated strings
- &rect
  Bounding rectangle
- iFormat
  Formatting flags

VII. Key DrawText Flags

Alignment

- DT_LEFT (default)
- DT_RIGHT
- DT_CENTER

Vertical positioning

- DT_TOP (default)
- DT_BOTTOM
- DT_VCENTER

Line handling

- DT_SINGLELINE
  Treats CR/LF as characters
- DT_WORDBREAK
  Wraps text at word boundaries

Clipping

- DT_NOCLIP
  Allows text to overflow the rectangle

Spacing

- DT_EXTERNALLEADING
  Includes extra line spacing

Tabs

- DT_EXPANDTABS
  Expands \t characters
- DT_TABSTOP
  Custom tab stops (use carefully)

VIII. When to Use What

TextOut

- Fast
- Simple
- No layout help

TabbedTextOut

- Columnar text
- Structured output
- Minimal formatting logic

ExtTextOut

- Precise control
- Custom spacing
- Clipping and backgrounds

DrawText

- UI text
- Labels
- Paragraphs
- Anything rectangular

IX. Key Takeaways

- TextOut draws text — nothing else
- SetTextAlign changes how coordinates behave
- TabbedTextOut is for structured layouts
- ExtTextOut gives **surgical control**
- DrawText trades control for convenience

If you understand **why** each exists, WinAPI text stops being confusing and starts being predictable.

X. Specifying Text within a Rectangle

Instead of giving an (x, y) position, **DrawText** uses a **RECT** structure.
The text is drawn **inside the rectangle**.

- pString → pointer to the text
- iCount → number of characters
- If the string is **NULL-terminated**, set iCount = -1 and Windows finds the length automatically.

XI. Text Formatting Options

The iFormat parameter controls how text appears inside the rectangle.

Horizontal alignment

- DT_LEFT → left aligned (default)
- DT_RIGHT → right aligned
- DT_CENTER → centered

Vertical alignment

- DT_TOP
- DT_BOTTOM
- DT_VCENTER

Other options

- DT_SINGLELINE → draws text on one line only (ignores new lines)

XII. Line Breaks and Word Wrapping

- By default, **carriage return and linefeed** create new lines.
- DT_SINGLELINE disables this behavior.
- DT_WORDBREAK wraps text at **word boundaries** for multi-line text.
- DT_NOCLIP allows text to extend outside the rectangle.

XIII. Tab Handling

- Text may contain **tab characters** (\t or 0x09).
- DT_EXPANDTABS enables proper tab handling.
- By default, **tab stops occur every 8 characters**.
- DT_TABSTOP allows custom tab spacing, but it can conflict with other flags.

Note: A tab character does not print a symbol. It moves the cursor to the next tab stop.

XIV. DrawTextEx – Enhanced Text Handling

DrawTextEx provides better control, especially for tab stops.

Why use DrawTextEx

- More precise tab control than DrawText
- Avoids problems with DT_TABSTOP

Main Arguments

- hdc → device context
- pString → text string
- iCount → text length
- rect → text rectangle
- iFormat → same flags as DrawText
- drawtextparams → extra settings

XV. DRAWTEXTPARAMS Structure

- cbSize → size of the structure
- iTabLength → tab width (in average character units)
- iLeftMargin → left margin
- iRightMargin → right margin
- uiLengthDrawn → number of characters drawn

XVI. Key Points

- Use **DrawText** for simple rectangle-based text.
- Use **DrawTextEx** for **custom tab stops and margins**.
- Set iTabLength to control tab spacing.
- Margins are optional.
- uiLengthDrawn shows how much text was processed.
- DrawTextEx may not exist on very old Windows systems.
- Choose the function based on **complexity and compatibility needs**.

Enhanced Text Rendering & Device Context Attributes

I. Enhanced Text Settings with DrawTextEx

DrawTextEx provides more control than DrawText.

Uses the **DRAWTEXTPARAMS** structure

Allows:

- Precise **tab stop control**
- **Left and right margins**
- Feedback on how many characters were drawn

Why It Matters

- Useful when **exact alignment and spacing** are required
- Helps create **clean and professional text layouts**
- Improves control over text formatting in Windows applications

II. Device Context (DC) Attributes for Text Rendering

A **device context (DC)** stores settings that control how text is drawn.

Key attributes include:

- Text color
- Background color
- Background mode
- Intercharacter spacing

III. Text Color Control

Default text color is **black**

Change text color using: **SetTextColor**

Retrieve current text color using: **GetTextColor**

Windows converts the specified color into a pure color before rendering.

IV. Background Mode and Background Color

Text is drawn over a rectangular background.

Background Mode

- Set using SetBkMode:
- OPAQUE (default) - Background is filled with color
- TRANSPARENT - Background is not drawn

Background Color

- Set using SetBkColor
- Ignored when background mode is TRANSPARENT

Transparent mode improves text visibility over images or patterns.

V. Intercharacter Spacing

Controls the space **between characters**.

- Set using SetTextCharacterExtra
- iExtra = 0 → default spacing
- Negative values are treated as **zero**
- Retrieve spacing using: GetTextCharacterExtra

Used for **fine layout control** and **text justification**.

VI. Using System Colors

To match Windows theme colors, use **system-defined colors** for text and background.

Handling System Color Changes

- Windows sends WM_SYSCOLORCHANGE
- Applications should:
    - ✓ Update text and background colors
    - ✓ Redraw affected areas

This Ensures consistency with user's system settings.

VII. Key Points

- DrawTextEx provides **advanced formatting control**
- Device context attributes affect **text appearance**
- Text color and background settings improve readability
- Background mode controls whether the background is drawn
- Intercharacter spacing fine-tunes text layout
- System colors ensure UI consistency
- Handle WM_SYSCOLORCHANGE to adapt dynamically

VIII. How This Fits the Main Topic

This subtopic is still part of **Text Positioning, Alignment, and Rendering.**

Focus shifts from *where text goes* → *how text looks.*

Leveraging Stock Fonts for Text Rendering in Windows

Text rendering in Windows depends on the **font selected into the device context (DC)**. To simplify font handling, Windows provides **stock fonts**—predefined fonts ready for immediate use.

I. Using Stock Fonts

Stock fonts are built-in fonts supplied by Windows.

They are useful for **quick setup**, **UI consistency**, and **basic text rendering**.

Getting and Selecting a Stock Font:

✓ The font is now active in the device context and used for all text output.

II. Why Fonts Matter

- Fonts affect **readability** and **visual appearance**
- Choosing the right font improves **clarity and layout**
- Stock fonts reduce setup time for common use cases

III. Common Stock Fonts

STOCK FONT	USAGE
`SYSTEM_FONT`	Default proportional system font
`SYSTEM_FIXED_FONT`	Fixed-width font (tables, code)
`OEM_FIXED_FONT`	Terminal / legacy text
`DEFAULT_GUI_FONT`	Standard Windows UI text

DEFAULT_GUI_FONT ensures visual consistency with Windows controls.

IV. Font Metrics and Layout

Font measurements are important for text positioning and layout calculations.

```
HFONT hFont = (HFONT)GetStockObject(DEFAULT_GUI_FONT);
SelectObject(hdc, hFont);
```

Provides:

- Character height
- Average character width

Essential for aligning and spacing text correctly

V. Proportional vs Fixed Fonts

- **Proportional fonts**: character widths vary
- **Fixed fonts**: all characters have the same width

Proportional fonts require extra care when calculating text width.

VI. Beyond Stock Fonts

Limitations

- Limited font styles and sizes
- No control over exact typeface

More Control

- Windows provides **font creation functions**
- Allows precise control over:
    - ✓ Typeface
    - ✓ Size
    - ✓ Weight and style

(Handled in later sections.)

VII. Key Takeaways

- Stock fonts are easy to use and readily available
- Select them using GetStockObject and SelectObject
- Choose fonts based on readability and layout needs
- Use GetTextMetrics for accurate positioning
- For advanced control, create custom fonts

VIII. Where This Fits

Still part of **Text Rendering and Formatting**

Focus: **font selection and usage in a device context**

UNDERSTANDING FONT BASICS IN WINDOWS

Before working with font-related code, it is important to understand how **fonts work in Windows**.

Fonts directly affect **text appearance, readability, performance, and scalability**.

Windows supports **two main font categories**:

- **GDI Fonts**
- **Device Fonts**

Categories of Fonts – (1) GDI Fonts

Fonts managed by the Windows Graphics Device Interface (GDI).

I. Raster (Bitmap) Fonts

- Characters are stored as **bitmaps.**
- Designed for **specific sizes and aspect ratios** and are **not scalable.**
- Very **fast** and **highly readable** at designed sizes.

Characteristics

- Bitmap-based characters
- Fixed sizes only
- Excellent clarity
- High performance

Common Raster Fonts

- System
- FixedSys
- Terminal
- Courier
- MS Serif
- MS Sans Serif
- Small Fonts

II. Stroke Fonts

- Characters defined using **line segments**
- **Continuously scalable**
- Lower performance and clarity
- Best suited for **plotters**

Characteristics

- Line-based definitions
- Scalable
- Poor readability at small sizes
- Weak appearance at large sizes

Common Stroke Fonts

- Modern
- Roman
- Script

III. TrueType Fonts

- **Outline-based**
- Scalable to any size
- High visual quality
- Standard font type in modern Windows

Key Advantages

- Excellent scalability
- Consistent appearance
- High readability

2. Device Fonts

- Built into **output devices** such as printers
- Font rendering handled by the device hardware
- Availability depends on the device

Common in printers and specialized output devices.

3. Typeface Names (Common Examples)

FONT CONSTANT	TYPEFACE NAME	CATEGORY / STYLE
`SYSTEM_FONT`	**System**	Proportional
`SYSTEM_FIXED_FONT`	**FixedSys**	Monospaced
`OEM_FIXED_FONT`	Terminal	Legacy OEM
`Courier`	**Courier**	Slab Serif Mono
`DEFAULT_GUI_FONT`	*MS Serif / MS Sans Serif*	Standard UI
`Small Fonts`	SMALL FONTS	Compact Raster

Serif vs Sans Serif

- **Serif**: small finishing strokes
- **Sans Serif**: no finishing strokes

4. Font Attributes

Fonts are defined by the following characteristics:

- **Typeface** → design of characters
- **Size** → measured in points (e.g., 12 pt)
- **Weight** → normal, bold, light
- **Style** → normal, italic, oblique

Synthesized Attributes

Windows can generate:

- Bold
- Italic
- Underline
- Strikethrough

No separate font files are required for these effects.

5. Key Considerations for Font Selection

- **Readability** → raster fonts are very clear
- **Scalability** → TrueType fonts adapt to any size
- **Performance** → raster fonts are faster
- **Visual appeal** → TrueType fonts look better
- **Device compatibility** → device fonts are hardware-specific

6. TrueType and OpenType Fonts

- TrueType fonts combine **scalability and quality**
- Widely used in modern Windows systems
- **OpenType** extends TrueType with:
    - ✓ Advanced features
    - ✓ Cross-platform support

Covered in more detail in later sections.

7. Font Attribute Synthesis

For GDI raster and stroke fonts, Windows can simulate:

- Bold
- Italic
- Underline
- Strikethrough

Example: Italics are created by **shifting the top of characters**

8. Essential Takeaways

- Windows supports multiple font technologies
- Each font type has **strengths and limitations**
- TrueType fonts are the modern standard
- Understanding font types helps choose the right font for each task

9. Where This Fits

✓ Continues **Text Rendering & Font Management**
✓ Foundation for **font creation and selection APIs**

EXPLORING TRUETYPE FONTS IN WINDOWS

TrueType fonts are a major advancement in font technology.
They provide **scalable, high-quality text** using **mathematical outlines** instead of fixed bitmaps.

Understanding TrueType fonts is essential for effective text rendering in Windows.

How TrueType Fonts Work

I. Character Definition

- Characters are defined using **filled outlines**
- Outlines consist of **straight lines and curves**
- Scaling is done by changing outline coordinates

This allows smooth resizing without loss of quality.

II. Rasterization Process

- When a TrueType font is used at a specific size, Windows performs **rasterization**
- Rasterization converts outlines into screen-ready pixels
- TrueType fonts include **hints**:
    - ✓ Guide the scaling process
    - ✓ Reduce rounding errors
    - ✓ Preserve character shape and clarity

III. Bitmap Creation and Caching

- Scaled outlines are converted into **bitmaps**
- Bitmaps are **cached in memory**
- Reusing cached bitmaps improves performance

IV. Key Characteristics (Summary)

- **Outline-based** → smooth curves and lines
- **Scalable** → any size without distortion
- **Hinting** → improves appearance at small sizes
- **Rasterized** → outlines converted to bitmaps
- **Cached** → faster rendering after first use

V. Common TrueType Fonts in Windows

- **Courier New** → fixed-width, typewriter style
- **Times New Roman** → serif, ideal for documents
- **Arial** → sans-serif, screen-friendly
- **Symbol** → special symbols and characters
- **Lucida Sans Unicode** → wide multilingual support

Modern Windows versions include many additional TrueType fonts.

VI. Advantages of TrueType Fonts

- **Versatile** → works across sizes and devices
- **High visual quality** → smooth and professional
- **Readable** → clear character shapes
- **Aesthetic flexibility** → many styles available

Considerations:

- More processing than bitmap fonts (due to rasterization)
- Font availability depends on system installation

TrueType fonts are the dominant font technology in modern Windows:

- Scalable.
- Visually appealing.
- Widely supported.
- Suitable for both screen and print.

Where This Fits:

- Continues **Font Technology & Text Rendering**.
- Leads naturally into **font creation and selection APIs**

RECONCILING TRADITIONAL AND COMPUTER TYPOGRAPHY

Windows bridges the gap between **traditional typography** and **computer typography**.

Traditional typography focuses on **distinct font designs**, while computer typography often relies on **attribute-based font selection**.

Windows supports **both approaches**, allowing developers to choose fonts by typeface or by attributes such as size, weight, and style.

This flexibility lets developers manage fonts based on **design goals or technical needs**.

Point Size: A Guideline, Not an Exact Measurement

Point size is a **typographic convention**, not a precise ruler.

- Historically, a point is about **1/72 of an inch.**
- Originates from **print typography.**
- In digital systems, point size **does not directly equal actual character height.**

Actual character dimensions vary within a font.

Because of this, **GetTextMetrics** should be used when accurate measurements are required.

Point size guides design, but **metrics control layout**.

Leading: Vertical Spacing Between Lines

Leading controls the vertical rhythm of text.

Internal Leading

- Space inside the font
- Allows room for accents and diacritics
- Part of the font design

External Leading

- Suggested space **between lines**
- Provided by tmExternalLeading in TEXTMETRIC
- Can be used directly or adjusted as needed

Text Metrics

GetTextMetrics provides:

- tmHeight → total line spacing (not pure font size)
- tmInternalLeading
- tmExternalLeading

These values are essential for accurate text layout.

Typography: Design Meets Engineering

Font selection is both **art and science**.

- Font designers balance readability, proportion, and style
- Developers must combine:
    - ✓ Typographic conventions
    - ✓ Technical font metrics

Experimenting with font families, sizes, and spacing helps achieve better visual results.

The Logical Inch in Windows Displays

Windows uses a concept called the **logical inch** to ensure text remains readable on screens.

In older systems like Windows 98:

- System font: **10-point** with **12-point line spacing**
- Display settings:
  - ✓ **Small Fonts** → 96 dpi
  - ✓ **Large Fonts** → 120 dpi
- Logical dpi obtained using:
  - ✓ GetDeviceCaps(LOGPIXELSX / LOGPIXELSY)

I. What Is a Logical Inch?

A logical inch represents **96 or 120 pixels**, not a physical inch.

- Appears larger than a real inch when measured
- Designed to improve text readability on screens
- Accounts for:
  - ✓ Lower pixel density
  - ✓ Greater viewing distance than printed paper

The logical inch is a **scaling mechanism**, not an error.

II. Why It Exists

- Makes small text (e.g., 8-point) readable on screens
- Matches screen width to typical paper margins
- Improves layout consistency for text-heavy displays

Windows 98 vs Windows NT

Windows 98

- Logical dpi matches expected behavior
- Predefined mapping modes work well

Windows NT

- LOGPIXELS values may not match physical dimensions
- Better to rely on **logical dpi**, not physical measurements
- Custom mapping modes recommended for text

Logical Twips Mapping Mode

To maintain consistency, applications can define a custom mapping mode:

- 1 point = **20 logical units**
- Example: 12-point font → 240 units
- Y-axis increases downward, simplifying line layout

Ensures consistent text sizing across Windows versions.

Key Points

- Logical inch intentionally enlarges text for readability
- Difference applies only to **displays**, not printers
- Printers maintain true physical measurements
- Use logical dpi for screen text
- Custom mapping modes improve consistency on Windows NT

Modern Considerations

- User font scaling preferences should be respected
- High-DPI displays reduce reliance on older tricks
- Logical inch concepts still influence Windows text scaling

Final Takeaway

Typography in Windows requires understanding both:

- **Typographic principles**
- **GDI measurement systems**

Mastering this balance leads to **accurate, readable, and professional text rendering**.

UNDERSTANDING LOGICAL FONTS IN WINDOWS

Logical fonts are **abstract descriptions of a font**, defining attributes like **typeface, size, weight, and style**.

They exist as **GDI objects** (HFONT) and become meaningful when **selected into a device context** (SelectObject).

This abstraction ensures **device-independent text rendering**, providing consistent appearance across monitors and printers.

Key Concepts

- **Logical Font** → Abstract blueprint for text appearance.
- **GDI Object** → Handle of type HFONT.
- **Device Independence** → Maps requested font to the closest available on the output device.

Like logical pens or brushes, a logical font doesn't exist in the device until **selected** into a DC.

Creating and Selecting Logical Fonts

1. Creation

Logical fonts are created using:

- CreateFont → 14 separate parameters (less convenient)
- CreateFontIndirect → Pointer to a **LOGFONT** structure (preferred)

LOGFONT Structure (14 fields):

- Typeface name
- Height / Width
- Weight (boldness)
- Style (italic, underline, strikeout)
- Character set, pitch, and more

Methods to set LOGFONT:

1. **Direct specification** → Manually fill LOGFONT fields
2. **Enumeration** → List all fonts on device (rarely used)
3. **ChooseFont** → User-friendly dialog returns LOGFONT

2. Selection

- SelectObject(hdc, hFont) → Activates logical font in DC
- Windows maps it to the **closest real font** available on the device

3. Usage

Query font info:

- GetTextMetrics → Detailed metrics (height, spacing, leading, etc.)
- GetTextFace → Returns the font face name.

Draw text using selected font.

Delete when done:

- DeleteObject(hFont) → Only if not selected in a DC.
- Do not delete stock fonts.

Font Mapping

- Windows may display a font **slightly different** from requested attributes
- Depends on **device font availability** and **system substitutions**

Important Structures

Structure	Purpose
`LOGFONT`	**Defines font attributes** when creating a logical font. Includes height, width, weight, and character set.
`TEXTMETRIC`	**Retrieves metrics** for the selected font. 20+ fields Covers physical characteristics like ascent, descent, and internal leading.

Practical Considerations

- **Font Availability** → Not all devices have the same fonts
- **User Preferences** → Respect size/style choices for readability
- **Font Families** → Prefer widely available families for consistency
- **Modern Systems** → Keep up with TrueType/OpenType enhancements

Summary of Logical Font Lifecycle

1. **Create** → CreateFont / CreateFontIndirect
2. **Select** → SelectObject to activate in DC
3. **Query / Draw** → Use GetTextMetrics, GetTextFace, draw text
4. **Delete** → DeleteObject when finished

Logical fonts allow **flexible, device-independent text rendering**, bridging the gap between **developer intent** and **device capabilities**.

PICKFONT PROGRAM – A FONT LABORATORY

PICKFONT is essentially a **Font Playground**. Instead of guessing what a font looks like in code, this program provides a **Dialog Box control panel** where you can tweak every font attribute—**Height, Width, Weight, Italic, Mapping Mode**—and see the changes instantly.

I. The Data Container (DLGPARAMS)

Passing multiple variables between a main window and a dialog box in C is cumbersome. PICKFONT solves this with a **custom structure called DLGPARAMS**, which acts as a "bucket" for the application state.

Contents of DLGPARAMS:

- **Target:** Are we drawing to the **Screen** or a **Printer**?
- **Font:** The LOGFONT structure (Height, Width, Name, Weight, etc.)
- **Rules:** Mapping modes (pixels, inches, twips) and flags (e.g., Match Aspect Ratio)

The main window owns the bucket, and the dialog box borrows it to change settings.

II. The Main Window (WndProc)

The Main Window is mostly a **canvas**. Its job is simple:

- **Initialization (WM_CREATE):** Launches the dialog box immediately for editing.
- **Painting (WM_PAINT):** Reads the current font settings from DLGPARAMS, creates the font, and draws text on the screen.
- **Update Loop:** Relies on the dialog box to tell it **when to repaint**.

III. The Dialog Box (DlgProc)

This is the **control center** of the program.

- **Inputs:** Typing numbers into "Height" or checking "Italic" doesn't change anything immediately.
- **Update Logic:** Clicking **OK** (or changing a key setting) calls SetLogFontFromFields, which reads the GUI and updates the DLGPARAMS bucket.
- **Feedback:** Tells the Main Window to redraw itself with the new font.

IV. The Bridge Functions (GUI ↔ Code Translators)

Windows understands **LOGFONT structures**, users understand **text boxes and checkboxes**. We need translators:

A. SetLogFontFromFields (GUI → Code)

- Reads **Edit Controls** (GetDlgItemInt) and **Checkbox states** (IsDlgButtonChecked)
- Updates the LOGFONT structure in DLGPARAMS
- Example: Typing "50" in Height → lfHeight = 50

B. SetFieldsFromTextMetric (Code → GUI)

- After Windows selects a font, it may **adjust the requested size**
- Uses GetTextMetrics to retrieve the actual font created
- Updates the Dialog Box so the user sees the **real rendered values**
- Example: Requested Height 50 → Windows creates Height 48 → GUI shows 48

V. Handling Mapping Modes (MySetMapMode)

Fonts are measured differently depending on the **mapping mode**:

MAPPING MODE	EXAMPLE	INTERPRETATION
`MM_TEXT`	**50**	50 pixels (size varies on high-res screens)
`MM_LOENGLISH`	**50**	0.50 inches (consistent physical size)
`Twips`	**50**	1/1440th of an inch (requires math for coordinates)

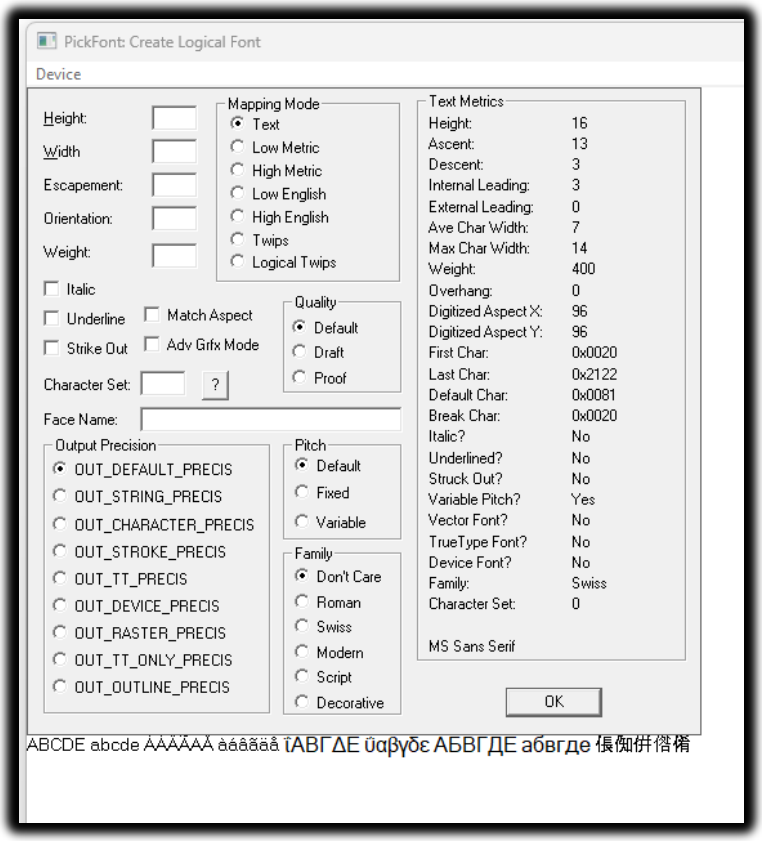
MySetMapMode sets the **ruler type** for the Device Context before drawing, ensuring sizes are interpreted correctly.

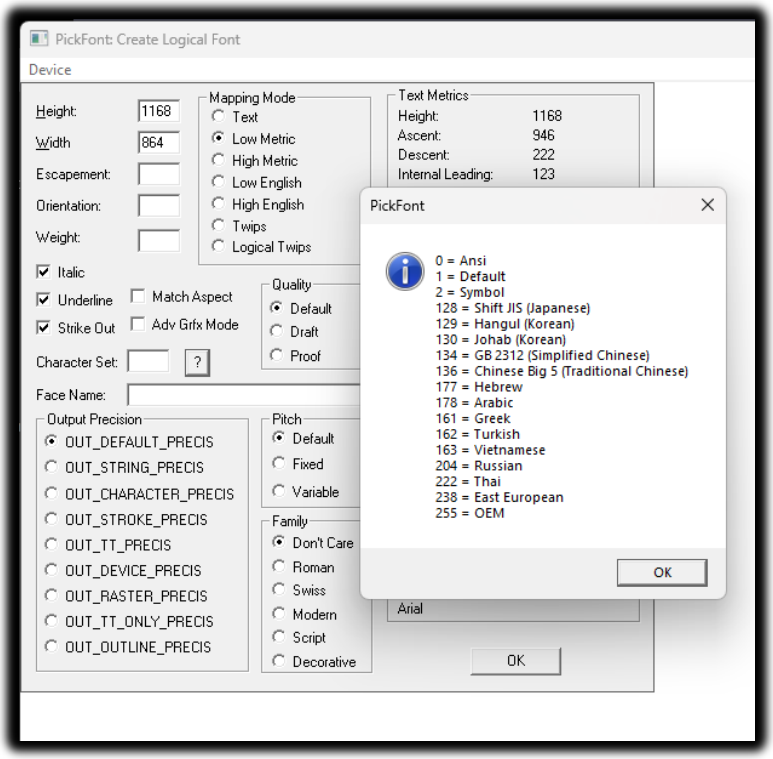
VI. Summary – Flow of Data

1. **Structure:** DLGPARAMS holds the current font settings.
2. **Input:** User changes settings in the dialog box (DlgProc).
3. **Translation:** SetLogFontFromFields converts GUI inputs → LOGFONT.
4. **Output:** Main Window (WndProc) reads DLGPARAMS and draws text.
5. **Feedback:** SetFieldsFromTextMetric updates GUI to reflect **actual rendered values**.

This program is a **perfect example of how Windows handles fonts**, bridging **user-friendly input** with **GDI's logical font system**, mapping modes, and rendering logic.

The program described at once….


PickFont: Create Logical Font
Device
Height:
Width
Escapement:
Orientation:
Weight:
Mapping Mode
Text
Low Metric
High Metric
Low English
High English
Twips
Logical Twips
Italic
Underline
Match Aspect
Strike Out
Adv Grfx Mode
Quality
Default
Draft
Proof
Character Set:
?
Face Name:
Output Precision
OUT_DEFAULT_PRECIS
OUT_STRING_PRECIS
OUT_CHARACTER_PRECIS
OUT_STROKE_PRECIS
OUT_TT_PRECIS
OUT_DEVICE_PRECIS
OUT_RASTER_PRECIS
OUT_TT_ONLY_PRECIS
OUT_OUTLINE_PRECIS
Pitch
Default
Fixed
Variable
Family
Don't Care
Roman
Swiss
Modern
Script
Decorative
Text Metrics
Height: 16
Ascent: 13
Descent: 3
Internal Leading: 3
External Leading: 0
Ave Char Width: 7
Max Char Width: 14
Weight: 400
Overhang: 0
Digitized Aspect X: 96
Digitized Aspect Y: 96
First Char: 0x0020
Last Char: 0x2122
Default Char: 0x0081
Break Char: 0x0020
Italic? No
Underlined? No
Struck Out? No
Variable Pitch? Yes
Vector Font? No
TrueType Font? No
Device Font? No
Family: Swiss
Character Set: 0
MS Sans Serif
OK
ABCDE abcde ÀÁÄÅÅÅ àáâãäå ïΑΒΓΔΕ üαβγδε АБВГДЕ абвгде 俄匈併僭脩


PickFont: Create Logical Font
Device
Height: 1168
Width 864
Escapement:
Orientation:
Weight:
Mapping Mode
Text
Low Metric
High Metric
Low English
High English
Twips
Logical Twips
Italic
Underline
Match Aspect
Strike Out
Adv Grfx Mode
Quality
Default
Draft
Proof
Character Set:
?
Face Name:
Output Precision
OUT_DEFAULT_PRECIS
OUT_STRING_PRECIS
OUT_CHARACTER_PRECIS
OUT_STROKE_PRECIS
OUT_TT_PRECIS
OUT_DEVICE_PRECIS
OUT_RASTER_PRECIS
OUT_TT_ONLY_PRECIS
OUT_OUTLINE_PRECIS
Pitch
Default
Fixed
Variable
Family
Don't Care
Roman
Swiss
Modern
Script
Decorative
Text Metrics
Height: 1168
Ascent: 946
Descent: 222
Internal Leading: 123
Arial
OK
PickFont
0 = Ansi
1 = Default
2 = Symbol
128 = Shift JIS (Japanese)
129 = Hangul (Korean)
130 = Johab (Korean)
134 = GB 2312 (Simplified Chinese)
136 = Chinese Big 5 (Traditional Chinese)
177 = Hebrew
178 = Arabic
161 = Greek
162 = Turkish
163 = Vietnamese
204 = Russian
222 = Thai
238 = East European
255 = OEM
OK

The **PICKFONT** program is a testing lab. Do not overthink the code; it is just a Dialog Box that lets you tweak settings. Read the source code if you want to see how to make a checkbox work.

What actually matters in this chapter is understanding how Windows handles typography. It is a three-step process:

1. **The Request (LOGFONT):** You ask for a specific font.
2. **The Matchmaker (The Mapper):** Windows looks for it.
3. **The Result (TEXTMETRIC):** Windows tells you what you actually got.

VII. The Request: LOGFONT Structure

When you want a font, you fill out a structure called **LOGFONT** (Logical Font). Think of this as filling out a profile for a dating app. You list exactly what you are looking for, but you might not get a perfect match.

Here are the fields that actually matter:

lfHeight (The most important field) This determines size, but the math is tricky.

- **Positive Value:** You are asking for the **Cell Height**. This includes the character *plus* the internal spacing (leading).
- **Negative Value:** You are asking for the **Character Height**. This is just the size of the letter itself, matching the point size (like 12pt).
- **Zero:** Windows picks a default size.

lfWidth Usually, you leave this at **0**. This tells Windows to calculate the width automatically based on the aspect ratio. If you put a number here, you can stretch or squash the font.

lfEscapement and lfOrientation This controls rotation.

- **Escapement:** Rotates the entire line of text (like writing up a hill).
- **Orientation:** Rotates the individual letters.
- *Note:* On modern Windows, these usually need to be the same value to work correctly.

lfWeight How bold is the text?

- **400:** Normal.
- **700:** Bold.
- You can use any number from 0 to 1000, but most fonts only support a few specific weights.

lfItalic, lfUnderline, lfStrikeOut These are simple switches. Set them to **TRUE** to turn them on.

lfCharSet This defines the language symbols (ANSI, Greek, Russian, etc.).

- **Warning:** If you ask for a specific font name (like Arial) but the wrong CharSet (like Symbol), Windows prioritizes the CharSet. You might get a random Symbol font instead of Arial.

lfPitchAndFamily This is your generic backup plan.

- **Pitch:** Fixed (like code) or Variable (like this text).
- **Family:** Serif (Times New Roman), Sans-Serif (Arial), or Script.
- If you ask for *Times New Roman* but the user deleted it, Windows uses this field to find a substitute that looks similar.

VIII. The Matchmaker: Font Mapping Algorithm

You pass your **LOGFONT** to the function **CreateFontIndirect**. Windows now has to find a font on the hard drive that matches your request.

It uses a **Penalty System**. It looks at every font installed and calculates a score. If a font differs from your request, it gets penalty points. The font with the lowest penalty wins.

The Priority List:

1. **CharSet:** Windows will almost never give you the wrong language.
2. **Pitch:** It tries to match Fixed vs. Variable width.
3. **FaceName:** It looks for the specific name you typed.
4. **Height:** It scales the font to fit your size.

The Trap: If you ask for a font that does not exist, Windows will guess. Sometimes its guess is terrible. This is why you should always check what you actually got.

IX. The Result: TEXTMETRIC Structure

After you select the font into your Device Context, you call **GetTextMetrics**. This fills a structure called **TEXTMETRIC**. This is the reality check. It tells you the physical dimensions of the font Windows selected.

The Vertical Geometry

- **tmHeight:** The total vertical size of the font.
- **tmAscent:** How far the letters go up above the baseline (like the top of a *h*).
- **tmDescent:** How far the letters hang down below the baseline (like the bottom of a *g*).
- **tmInternalLeading:** Space inside the tmHeight reserved for accent marks.

The Spacing

- **tmExternalLeading:** The recommended gap between lines of text. Windows does not add this automatically; you have to add it yourself when moving to the next line.
- **tmOverhang:** If Windows had to fake a bold or italic effect (because the real font file was missing), it adds extra pixels to the width. This is the overhang.

The Character Flags

- **tmFirstChar / tmLastChar:** The range of valid letters in this font.
- **tmDefaultChar:** What gets drawn if you type a letter that doesn't exist (usually a box or a question mark).
- **tmBreakChar:** The character used to break words (usually the Spacebar).

X. Summary

1. **PICKFONT** is just a UI to test these concepts.
2. Use **LOGFONT** to describe what you want.
3. Use **lfHeight** (negative value) to set the size.
4. Windows uses the **Mapping Algorithm** to find the closest match.
5. Use **GetTextMetrics** to measure exactly what Windows gave you so you can space your lines correctly.

CHARACTER SETS & FONT MAPPING

I. The Core Concept: Text = Numbers

Computers do not understand letters; they only understand numbers.

- **A Character Set** is a map. It says "Number 65 equals 'A', Number 66 equals 'B'."
- **The Problem:** In the early days, one byte (8 bits) could only store 256 characters. This wasn't enough for English, Russian, Greek, and Japanese all at once.
- **The Windows Solution (Legacy):** Windows created different "Sets" ID numbers.
    - ✓ ANSI_CHARSET (0): Standard English/Western Europe.
    - ✓ GREEK_CHARSET (161): Replaces the upper 128 slots with Greek letters.
    - ✓ RUSSIAN_CHARSET (204): Replaces the upper 128 slots with Cyrillic.

II. The Modern Solution: Unicode

Instead of swapping 256-slot tables, Unicode uses a single massive table (tens of thousands of slots).

- It contains English, Greek, Russian, and Emoji all in one place.
- **In Windows:** Modern apps (compiled as "Unicode") use 16 bits (2 bytes) per character, allowing 65,536 distinct characters without swapping tables.

III. TrueType: The "Big Font" Concept

Old "Raster Fonts" were tied to a specific character set. You had one file for "Arial Greek" and another for "Arial Cyrillic."

TrueType Fonts (and OpenType) are different. They are **"Big Fonts."**

- A single TrueType file (like arial.ttf) contains glyphs for Latin, Greek, Hebrew, Arabic, etc.
- **How Windows uses it:** Even if you ask for GREEK_CHARSET, Windows can use the standard arial.ttf file. It just knows to look inside the "Greek section" of that file.

IV. Windows Font Selection (The Matchmaker)

When you ask Windows for a font, you don't just ask for a name; you ask for a **Character Set**.

The Logic:

1. **You Request:** LOGFONT.lfCharSet = GREEK_CHARSET.
2. **Windows Searches:** It looks for a font that claims to support Greek.
3. **The Match:**
   - ✓ If you requested "Arial," it checks if Arial has Greek data.
   - ✓ If Arial *doesn't* support Greek, Windows ignores your request for "Arial" and substitutes a font that *does* (like "Symbol" or "Microsoft Sans Serif").
   - ✓ **Priority:** Windows prioritizes the **Character Set** over the **Face Name**. It is better to show the correct language in the wrong font than the wrong language in the correct font.

V. The Structures: Input vs. Output

This is how you communicate with the system in C code.

a) The Input: LOGFONT

This is your "Wish List." You fill this out to tell Windows what you want.

```
// LOGFONT structure (partial)
typedef struct tagLOGFONTA {
    LONG lfHeight;
    LONG lfWidth;
    LONG lfEscapement;
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;  // Character set identifier
    // ... (other fields)
} LOGFONTA, *PLOGFONTA, *NPLOGFONTA, *LPLOGFONTA;
```

lfCharSet Role: This is the filter. If you set this to HEBREW_CHARSET, Windows filters out any font that doesn't know Hebrew.

b) The Output: TEXTMETRIC

This is the "Reality Check." After Windows picks a font, you call GetTextMetrics to see what you actually got.

```
// TEXTMETRIC structure (partial)
typedef struct tagTEXTMETRICA {
    LONG tmHeight;
    LONG tmAscent;
    LONG tmDescent;
    LONG tmInternalLeading;
    LONG tmExternalLeading;
    LONG tmAveCharWidth;
    LONG tmMaxCharWidth;
    LONG tmWeight;
    LONG tmOverhang;
    LONG tmDigitizedAspectX;
    LONG tmDigitizedAspectY;
    BYTE tmFirstChar;
    BYTE tmLastChar;
    BYTE tmDefaultChar;
    BYTE tmBreakChar;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStruckOut;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;  // Character set identifier
} TEXTMETRICA, *PTEXTMETRICA, *NPTEXTMETRICA, *LPTEXTMETRICA;
```

- **tmCharSet Role:** This tells you the specific character set of the font Windows selected. Even if you asked for a specific font, you verify here if it actually supports the language you need.

VI. Important Missing Concepts

The original text skipped these practical realities of font programming:

1. Font Fallback (Font Linking) What happens if you try to type a Japanese character in an English-only font like "Tahoma"?

- Windows does **not** show a square box immediately.
- It activates **Font Linking**. It silently switches to a Japanese font (like "MS UI Gothic") just for that one character, then switches back.

2. Code Pages vs. Character Sets

- **Character Set:** A Windows GUI concept (used in LOGFONT). It tells the font renderer which glyphs to draw.
- **Code Page:** A generic text processing concept (used in the command line or file saving). It maps byte values to characters.
- *Rule of Thumb:* In GUI programming (DrawText), worry about Character Sets. In File I/O (ReadFile), worry about Code Pages.

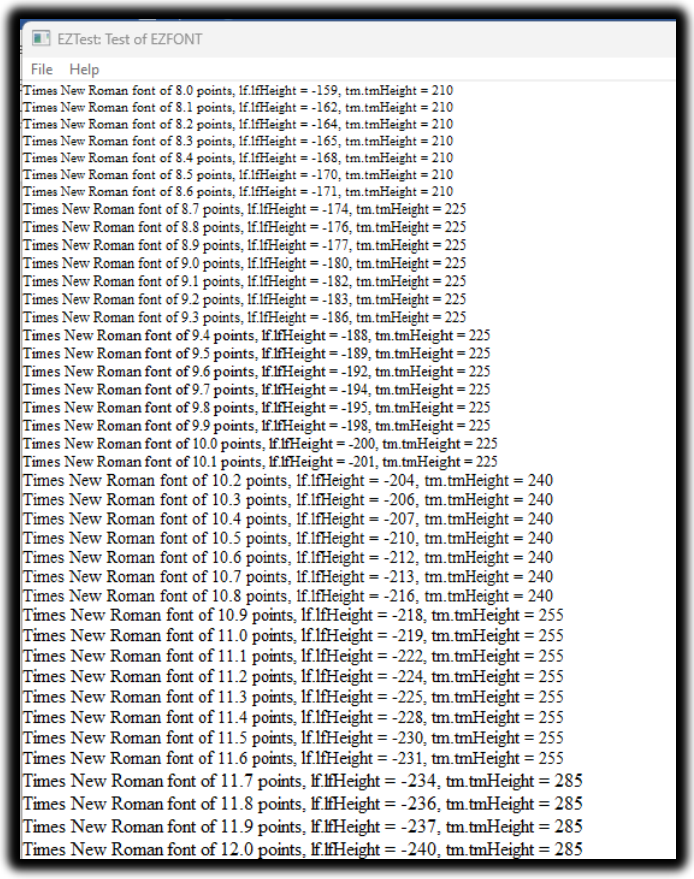
3. The "Symbol" Trap If you set lfCharSet to SYMBOL_CHARSET, Windows treats the font differently. It stops trying to map letters to Unicode standard slots and just gives you the raw glyphs. This is used for "Wingdings" or specialty icon fonts.

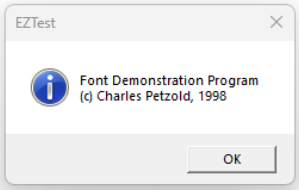
EZFONT PROGRAM

In the past, Windows struggled with font enumeration —basically, the system had a hard time finding and displaying the right fonts correctly.

We created EZFONT to fix that. By leaning into the flexibility of TrueType fonts, we've made font management straightforward and much more predictable for every user.

```c
/*--------------------------------------
   EZFONT.C -- Easy Font Creation
               (c) Charles Petzold, 1998
  --------------------------------------*/

#include <windows.h>
#include <math.h>
#include "ezfont.h"

HFONT EzCreateFont (HDC hdc, TCHAR * szFaceName, int iDeciPtHeight,
                    int iDeciPtWidth, int iAttributes, BOOL fLogRes)
{
     FLOAT      cxDpi, cyDpi ;
     HFONT      hFont ;
     LOGFONT    lf ;
     POINT      pt ;
     TEXTMETRIC tm ;

     SaveDC (hdc) ;

     SetGraphicsMode (hdc, GM_ADVANCED) ;
     ModifyWorldTransform (hdc, NULL, MWT_IDENTITY) ;
     SetViewportOrgEx (hdc, 0, 0, NULL) ;
     SetWindowOrgEx   (hdc, 0, 0, NULL) ;

     if (fLogRes)
     {
          cxDpi = (FLOAT) GetDeviceCaps (hdc, LOGPIXELSX) ;
          cyDpi = (FLOAT) GetDeviceCaps (hdc, LOGPIXELSY) ;
     }
     else
     {
          cxDpi = (FLOAT) (25.4 * GetDeviceCaps (hdc, HORZRES) /
                                  GetDeviceCaps (hdc, HORZSIZE)) ;

          cyDpi = (FLOAT) (25.4 * GetDeviceCaps (hdc, VERTRES) /
                                  GetDeviceCaps (hdc, VERTSIZE)) ;
     }

     pt.x = (int) (iDeciPtWidth  * cxDpi / 72) ;
     pt.y = (int) (iDeciPtHeight * cyDpi / 72) ;

     DPtoLP (hdc, &pt, 1) ;

     lf.lfHeight         = - (int) (fabs (pt.y) / 10.0 + 0.5) ;
     lf.lfWidth          = 0 ;
     lf.lfEscapement     = 0 ;
     lf.lfOrientation    = 0 ;
     lf.lfWeight         = iAttributes & EZ_ATTR_BOLD      ? 700 : 0 ;
     lf.lfItalic         = iAttributes & EZ_ATTR_ITALIC    ?   1 : 0 ;
     lf.lfUnderline      = iAttributes & EZ_ATTR_UNDERLINE ?   1 : 0 ;
     lf.lfStrikeOut      = iAttributes & EZ_ATTR_STRIKEOUT ?   1 : 0 ;
     lf.lfCharSet        = DEFAULT_CHARSET ;
     lf.lfOutPrecision   = 0 ;
     lf.lfClipPrecision  = 0 ;
     lf.lfQuality        = 0 ;
     lf.lfPitchAndFamily = 0 ;

     lstrcpy (lf.lfFaceName, szFaceName) ;

     hFont = CreateFontIndirect (&lf) ;

     if (iDeciPtWidth != 0)
     {
          hFont = (HFONT) SelectObject (hdc, hFont) ;
          GetTextMetrics (hdc, &tm) ;
          DeleteObject (SelectObject (hdc, hFont)) ;
          lf.lfWidth = (int) (tm.tmAveCharWidth * fabs (pt.x) / fabs (pt.y) + 0.5) ;

          hFont = CreateFontIndirect (&lf) ;
     }

     RestoreDC (hdc, -1) ;
     return hFont ;
}
```





EZFONT, EZTEST, and FONTDEMO — What Actually Matters

This chapter is about **font correctness**, not magic.
EZFONT exists to solve one problem:

Create a TrueType font at a predictable physical size across devices.

Everything else is implementation detail.

I. EzCreateFont — The Real Purpose

What EzCreateFont actually does

- Converts **decipoint font sizes** into device-correct logical units
- Creates a TrueType font that:
  - ✓ Scales correctly
  - ✓ Prints correctly
  - ✓ Survives different DPIs and devices

That's it.

You don't need a paragraph for every API call — the code already documents that.

II. What's worth remembering

✅ Device Context Isolation

- The function **saves and restores the DC**
- This prevents font creation from breaking later drawing code
- This is defensive programming, not fancy graphics work

✅ DPI Matters

- Font sizes are meaningless without DPI
- EZFONT explicitly calculates DPI to avoid:
  - ✓ Fonts appearing larger on printers.
  - ✓ Fonts shrinking on high-resolution displays.

✅ Decipoints Are Intentional

- Fonts are specified in **1/10th point units**
- This matches typography standards
- It allows precise font intent even if rasterization rounds later

III. What no longer matters (Windows 10+)

❌ NT-era Graphics Hacks

- SetGraphicsMode.
- ModifyWorldTransform.
- Identity matrices.

These were **Windows NT survival hacks**.
On modern Windows, they are unnecessary and can be safely ignored.

IV. LOGFONT — We don't need to overexplain It

LOGFONT is just a struct:

- Height
- Width
- Weight
- Charset
- Style flags

It's not mystical.
It's a container passed to CreateFontIndirect.

The only thing worth emphasizing:

- Height is king.
- Width is optional and adjusted only if explicitly requested.

V. Font Width Adjustment — Why It Exists

Sometimes:

- Height alone produces ugly proportions
- Different fonts scale differently

EZFONT optionally:

- Measures the font
- Tweaks width to match visual intent

This is about **appearance**, not correctness.

VI. EZTEST.C — Why This Program Exists

EZTEST is **not an app**.
It is a **measurement instrument**.

Its job

- Generate fonts at incremental sizes
- Measure them
- Display their metrics
- Expose rounding and rasterization limits

That's all.

VII. Important Insight (Keep This)

Rasterization Is the Bottleneck

- Fonts are drawn with pixels
- Pixels are discrete
- Very small size changes often:
    - ✓ Render identically on screen
    - ✓ Differ only on printers

This is not a bug.
This is physics.

VIII. FONTDEMO.C — Same Engine, Different Goal

FONTDEMO is about **human perception**, not metrics.

- Uses the same PaintRoutine
- Uses the same EZFONT engine
- Focuses on:
    - ✓ Visual comparison
    - ✓ Printing
    - ✓ User interaction

Think of it as:

- **EZTEST = lab instrument**
- **FONTDEMO = showroom**

IX. Printing vs Screen — The One Thing Users Miss

This deserves exactly **one clean explanation**, not ten.

- Screens:
  - ✓ Low DPI
  - ✓ Rasterized early
  - ✓ Limited size fidelity
- Printers:
  - ✓ High DPI
  - ✓ Vector-aware
  - ✓ Much better size differentiation

If fonts matter:
👉 **Always test print output**

X. Mapping Modes — Set Expectations Properly

Logical mapping modes:

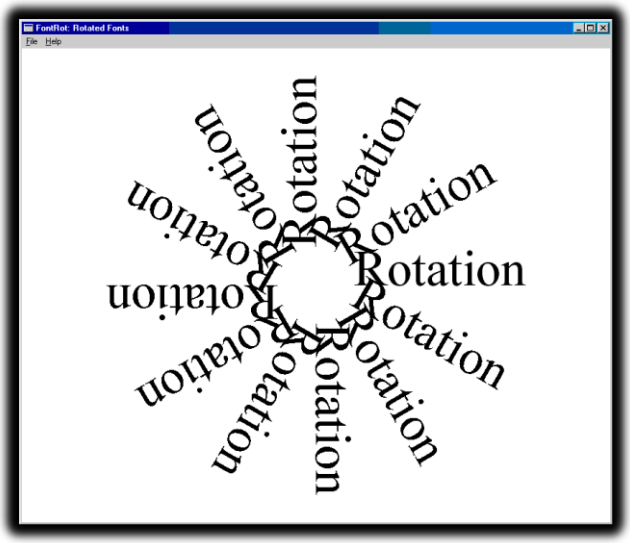
- Help scaling
- Improve consistency
- Do **not** defeat rasterization

They influence math, not pixels.

XI. Final Takeaways (This Is the Close)

- EZFONT solves **font size correctness**
- EZTEST proves **measurement accuracy**
- FONTDEMO proves **visual behavior**
- Rasterization limits are unavoidable
- Printers reveal truth screens hide
- NT-era hacks are historical, not required
- The code explains *how*
- The notes should explain *why*

FONT ROTATION IN WINDOWS GDI: A DYNAMIC EXPLORATION



This section should teach *how rotation works*, *why it works*, and *when to use which method*. Nothing more.

The **FONTROT** program demonstrates that TrueType text in Windows GDI doesn't have to sit flat.

Using font rotation, text can be drawn at any angle, opening the door to dynamic layouts, labels, gauges, and visual effects.

At its core, this program shows **two valid ways to rotate text** in WinAPI:

1. Rotating the **font itself**
2. Rotating the **graphics world**

Method 1: Rotating the Font (LOGFONT-Based Rotation)

FONTROT uses the simplest and most reliable approach: **rotating the font via LOGFONT**.

How it works

- A base TrueType font is created using EzCreateFont
- Its LOGFONT structure is retrieved
- Rotation is applied using:
  - ✓ lfEscapement
  - ✓ lfOrientation
- Values are specified in **tenths of a degree**
  - ✓ Example: 300 = 30°

Each rotated font is:

- Created with CreateFontIndirect
- Selected into the device context
- Used to draw text
- Deleted immediately after use

This is efficient, clean, and safe.

I. Why the Text Rotates Around the Center

Before drawing:

- The viewport origin is moved to the **center of the window**
- Text alignment is set to **baseline-based positioning**

This makes the center of the window act as the **rotation pivot**, producing the circular "starburst" effect.

Without this step, rotation would occur around the top-left corner — which looks wrong.

II. What the Demo Shows Visually

- 12 rotations
- 30° increments
- Full 360° sweep

Each draw uses a **new font**, not a transform.
This highlights a key idea:

In GDI, rotation can live in the font, not the math.

III. Key Takeaways from FONTROT

- TrueType fonts rotate cleanly in GDI
- Rotation is controlled entirely through LOGFONT
- The viewport origin defines the rotation axis
- Fonts must be deleted — every time — no exceptions

This method works on **all modern Windows versions** and doesn't rely on NT-only features.

Method 2: World Transforms (XFORM Matrix)

For more complex transformations, Windows also supports **world transforms** using the XFORM structure.

What XFORM controls

- Scaling
- Rotation
- Shearing
- Translation

Core functions

- SetWorldTransform
- ModifyWorldTransform

These apply transformations to **everything drawn afterward**, not just text.

I. When to Use XFORM (And When Not To)

Use LOGFONT rotation when:

- You're rotating text only
- You want simplicity
- You want predictable output
- You care about font hinting and clarity

Use XFORM when:

- You need to rotate *multiple objects together*
- You're combining rotation + scaling + translation
- You're building diagram-style or CAD-like visuals

II. Important Reality Check

- World transforms originated in **Windows NT**
- They still exist in Windows 10+
- But they are **legacy-level tools**

For modern applications:

- GDI → fine for learning and classic apps
- Direct2D / Direct3D → better for serious graphics work

III. One Rule You Must Follow

If you use world transforms:

Always restore the default transform

Failing to do this will silently break future drawing code.

IV. Bottom Line

- FONTROT proves that **text rotation in GDI is real and reliable**
- LOGFONT rotation is the cleanest solution
- XFORM is powerful but heavier
- Modern Windows keeps both for compatibility
- Understanding both makes you dangerous (in a good way 😎)

FONT ENUMERATION

Font enumeration is how a program **asks Windows: "What fonts do you have?"**
GDI walks through installed fonts and reports them back to your program.

Used for:

- Font pickers
- Preview lists
- Filtering fonts by type (TrueType, raster, fixed-pitch, etc.)
- Measuring font capabilities before use

Core Enumeration Functions

I. EnumFonts (Legacy, Still Works)

- Old-school enumeration API
- Enumerates **all fonts** or fonts matching a specific name
- Limited control, limited detail
- Still functional, but **not recommended for modern apps**

Use case:

- Simple font listing
- Legacy codebases

```
EnumFonts(hdc, szTypeFace, EnumProc, pData);
```

II. EnumFontFamilies (TrueType-Oriented)

- Improved handling of **TrueType fonts**
- Works in **two stages**:
  - ✓ Enumerates font families
  - ✓ Enumerates individual fonts within each family
- Better than EnumFonts, but still limited

Use case:

- Programs focused on TrueType fonts only

```
EnumFontFamilies(hdc, szFaceName, EnumProc, pData);
```

III. EnumFontFamiliesEx ✅ (Recommended)

- **Modern, flexible, and precise**
- Accepts a LOGFONT structure to control:
  - ✓ Charset
  - ✓ Pitch
  - ✓ Family
  - ✓ Typeface filtering
- Best choice for **32-bit and later Windows**

Use case:

- Professional applications
- Font filtering, categorization, analysis

Rule:
If you care about control → use EnumFontFamiliesEx.

```
EnumFontFamiliesEx(hdc, &logfont, EnumProc, pData, dwFlags);
```

Callback Function (Critical Concept)

Enumeration doesn't return an array.
Instead, **GDI calls your callback function once per font**.

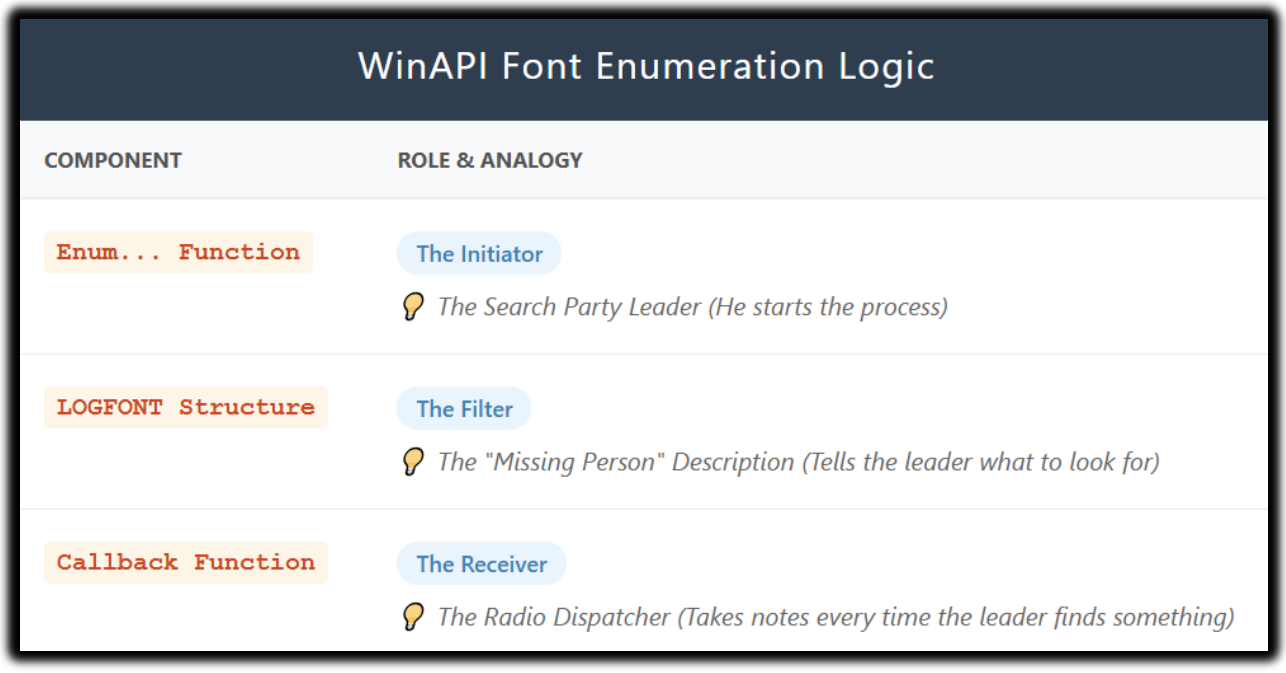
The callback receives:

- Font description
- Metrics
- Font type info

What you can do inside the callback:

- Build font lists
- Filter fonts (TrueType only, fixed-width, etc.)
- Store metrics for layout
- Reject unwanted fonts

Key idea:
👉 *You don't pull fonts — Windows pushes them to you.*



Structures You Must Know

I. LOGFONT – The Blueprint

Defines how a font looks:

- Typeface name
- Height, width
- Weight (boldness)
- Italic, underline
- Pitch & family

Used for:

- Creating fonts
- Filtering enumeration
- Font selection

II. TEXTMETRIC – Font Measurements

Describes font geometry:

- Height
- Ascent / descent
- Average character width
- Line spacing

Used for:

- Text layout
- Line calculations
- Precise positioning

III. ENUMLOGFONT

- Extends LOGFONT
- Includes:
    - ✓ Full font name
    - ✓ Style name
- Useful for **TrueType fonts**

IV. NEWTEXTMETRIC

- Extension of TEXTMETRIC
- TrueType-specific metrics
- Reveals advanced font characteristics

Used for:

- High-precision text rendering
- Professional layout engines

V. ChooseFont – The Easy Button

Instead of enumerating manually:

- Use ChooseFont
- Windows shows the **standard font dialog**
- User selects font
- You receive a filled LOGFONT

Advantages:

- No callbacks
- No filtering logic
- User-friendly
- Faster to implement

Best for:

- GUI apps
- Editors
- Tools where the user chooses fonts

```
CHOOSEFONT cf;
LOGFONT lf;
```

Summary

- Enumeration = Windows tells you what fonts exist
- EnumFontFamiliesEx = best API
- Callback = where the real work happens
- LOGFONT defines fonts
- TEXTMETRIC measures fonts
- ChooseFont = skip enumeration, let the user pick

This chapter is **not about memorizing APIs** —
it's about understanding **how GDI exposes font data and how you intercept it**.

CHOOSEFONT — A USER-FRIENDLY SHORTCUT

ChooseFont is the **fastest, safest way** to let users select fonts without manually enumerating them.

Instead of walking through GDI font lists yourself, Windows:

- Displays the **standard font dialog**
- Lets the user pick
- Returns a fully populated LOGFONT

👉 Practical, stable, user-approved UI.

I. Why ChooseFont Exists

Manual enumeration is:

- Verbose
- Error-prone
- Overkill for most GUI apps

ChooseFont:

- Eliminates callbacks
- Eliminates filtering logic
- Returns valid, system-approved font data

Use it when **users choose**, not when **apps analyze** fonts.

II. CHOOSEFONT Structure (Control Center)

This structure configures the dialog **and** receives results.

Core Fields

lpLogFont → LOGFONT

This is where the chosen font is described:

- Typeface
- Height / width
- Weight (boldness)
- Italic / underline / strikeout
- Escapement & orientation
- Charset
- Output precision
- Clipping precision
- Quality
- Pitch & family

This LOGFONT is later passed to: **CreateFontIndirect**

III. Flags (Behavior Control)

Important flags you'll actually use:

- CF_SCREENFONTS
  → Show only screen-usable fonts
- CF_EFFECTS
  → Enable underline + strikeout UI
- CF_INITTOLOGFONTSTRUCT
  → Dialog opens using the LOGFONT you supply
- CF_NOVECTORFONTS
  → Excludes vector fonts (rarely needed today)

Flags decide **what the user sees and can do**.

IV. ChooseFont Function Behavior

- Displays modal font dialog
- User selects font
- On success:
  - ✓ Returns TRUE
  - ✓ Fills CHOOSEFONT + LOGFONT

If it fails:

- Returns FALSE
- CommDlgExtendedError() gives the reason

Always check return value.

Typical Program Flow

I. Initialization Phase

- GetDialogBaseUnits
  → Establishes default character dimensions
- GetObject(SYSTEM_FONT, LOGFONT)
  → Uses system font as a sane starting baseline

This avoids weird defaults.

II. User Action

- Menu item like **“Font!”**
- Calls ChooseFont
- User interacts with dialog

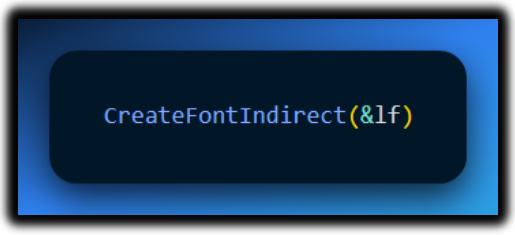
III. Repaint Phase

- InvalidateRect
- Window repaints
- New font is now active

This keeps rendering clean and predictable.

Rendering with the Selected Font

I. Font Creation



Creates an HFONT from the LOGFONT returned by ChooseFont.

II. Text Output

Draws text using the selected font and color.

Displaying Font Attributes

Many demos show:

- Typeface name
- Height
- Weight
- Italic / underline flags

Purpose:

- Transparency
- Debugging
- Educational value

Important Technical Realities (This Matters)

lfHeight Is Context-Sensitive

- **Screen DC** → pixels
- **Printer DC** → resolution-dependent

Never assume point size == height.

Logical Inch Concept

Logical inch bridges:

- Point size
- Device resolution
- Printing vs screen

Understanding this is **mandatory** for accurate layout.

Mapping Mode Pitfalls

- Metric modes (MM_LOMETRIC, MM_HIMETRIC)
  → Can distort font sizing

Best practice:

- Use MM_TEXT during font selection and rendering
- Switch mapping modes only for geometry

hDC in CHOOSEFONT

- Used only to **list printer fonts**
- Does **not** auto-scale lfHeight

You still compute height yourself.

iPointSize — The Hidden Weapon

- Font size in **1/10 point units**
- Device-independent
- Ideal for:
  - ✓ Screen
  - ✓ Printer
  - ✓ Mixed contexts

See EZFONT.C for conversion logic.

Unicode & Advanced Fonts

I. UNICHARS Program

- Enumerates every character in a font
- Uses TextOutW
- Works on NT and Win98+

Best for:

- Unicode fonts
- Multilingual testing
- Symbol exploration

Examples:

- Lucida Sans Unicode
- Bitstream CyberBit

LOGFONT Deep Cuts (Often Ignored)

- lfEscapement, lfOrientation
  → Text rotation
- lfCharSet
  → Language & script compatibility
- Script field
  → Critical for localization & global apps

Ignore these and your app breaks internationally.

Let's explain a bit more.

When you are building software that uses text, most people just focus on the font name and size. But if you ignore these three settings in the LOGFONT structure, your app will fail for users in other countries.

I. Rotating the Text

The settings lfEscapement and lfOrientation control how text is tilted or rotated.

- **lfEscapement:** This tilts the entire line of text at an angle.
- **lfOrientation:** This rotates each individual character.

If you don’t set these correctly, you can’t create vertical labels or slanted designs, which are common in charts and sidebars.

II. Language Compatibility (lfCharSet)

This setting tells the computer which set of symbols to use (like Greek, Cyrillic, or Kanji).

Even if you have the right font, the letters might look like gibberish or empty boxes if the lfCharSet is wrong.

It bridges the gap between the font file and the specific language the user speaks.

III. The Script Field

This is the most important part for making a global app. It tells the system which regional version of a font to use.

- **Why it matters:** Many languages share the same font names but use different "scripts" (like Arabic vs. Hebrew).
- **The Risk:** If you ignore this, your app might look fine in English but will break completely when someone tries to use it in Asia, the Middle East, or Eastern Europe.

IV. Bottom Line

To make an app that works everywhere in the world, you must define the rotation and the language script correctly.

Best Practices (Straight Truth)

- Use ChooseFont for UI apps
- Use EnumFontFamiliesEx for analysis tools
- Always use MM_TEXT for font operations
- Calculate font heights explicitly
- Test with Unicode fonts
- Restore original fonts after use
- Delete font objects — **always**

Final Words on ChooseFont

ChooseFont is not a shortcut for lazy developers. It is the **correct abstraction** for letting users choose fonts.

At its core, ChooseFont is a pre-built Windows dialog that allows users to select fonts and text settings. Some new developers assume that using it is "cheating" because they aren't manually listing fonts or building the UI themselves.

In reality, using ChooseFont is the **professional** way to build Windows software.

I. It Handles the Hard Parts You Can't See

Choosing a font is far more complicated than it looks. Under the surface, you're dealing with rotation, character sets, language scripts, DPI scaling, and accessibility rules.

- **The manual way:** You write hundreds of lines of code and still risk letting users pick a font that breaks text rendering.
- **The ChooseFont way:** Windows handles all the deep details—lfCharSet, scripts, scaling, and compatibility—and gives you a fully valid LOGFONT structure that just works.

This is like ordering a fully pre-cooked meal instead of trying to grow the wheat, milk the cow, and build the oven yourself. More effort doesn't mean better results.

II. It Prevents International Bugs Before They Exist

Custom font pickers often fail outside the developer's own language.

A beginner might list only font names like *Arial* or *Times New Roman and call it done. But fonts have different scripts—Arabic, Greek, Hebrew, Japanese—and ignoring them leads to invisible or broken text.

ChooseFont includes script selection by design. A user in Japan can explicitly choose the Japanese script of a font, and the text renders correctly. No extra logic. No special cases. No surprises.

If you skip this, your "pizza" may look fine—but it's missing the salt. For users in Korea or Israel, it's completely unusable.

III. It Feels Like a Real Windows App

Users have muscle memory. They know where the font list is. They know how to filter, preview, and select fonts.

When you use ChooseFont, your app behaves like Word, Notepad, and every other serious Windows application. When you roll your own picker, you risk missing features, breaking accessibility, or forcing users to learn a new "kitchen layout" just to grab a snack.

IV. It's Already a Calibrated Oven

ChooseFont is a professional oven that's already tuned for:

- High-DPI displays.
- Accessibility settings.
- System themes and scaling.

Build your own, and you're responsible for calibrating all of that. If you don't, your UI ends up tiny, blurry, or inconsistent on modern monitors.

V. Smart vs. Hardworking

- **Hardworking dev:** Spends three days building a custom font picker. It looks cool—and breaks when the system language is set to Chinese.
- **Smart dev:** Spends five minutes using ChooseFont. It works everywhere, and they spend the saved time making the app faster or adding real features.

Using ChooseFont isn't lazy. It's smart engineering: relying on a proven, system-level tool to solve a hard problem correctly—so you can focus on what actually makes your app better.

UNICHARS.C — UNICODE CHARACTER EXPLORER

UNICHARS.C is a graphical utility that **visualizes 16-bit Unicode characters** using a scrollable grid. It's not a toy program — it's a **reference-grade Unicode and font inspection tool**.

Core Idea

- Displays Unicode characters in a **16 × 16 grid** (256 characters per page)
- Each character corresponds to a **16-bit Unicode code**
- A **vertical scrollbar** lets the user move through Unicode blocks

One page = one Unicode block.

Window & Layout Design

Main window includes:

- Client area for drawing characters
- Vertical scrollbar for navigation

Grid layout:

- 16 rows × 16 columns
- Character placement calculated using:
    - ✓ Character height
    - ✓ Average width
    - ✓ External leading

This guarantees **clean alignment and consistent spacing**.

Font Handling

Default Font

- Lucida Sans Unicode
- Size: **12 points**
- Chosen because:
    - ✓ Wide Unicode coverage
    - ✓ Stable rendering on older systems

Font Customization (ChooseFont)

- Menu item: **"Font!"**
- Invokes ChooseFont
- User can change:
    - ✓ Typeface
    - ✓ Size
    - ✓ Style

After selection:

- Font recreated
- Window invalidated
- Display repainted immediately

👉 Dynamic, user-driven rendering.

Scrolling Logic

- Vertical scrollbar controls **Unicode block index**
- Supports:
  - ✓ Line scrolling
  - ✓ Page scrolling
- Scroll position directly maps to:
  - ✓ Unicode block offset
  - ✓ Characters drawn in the grid

Scrolling updates are reflected visually in real time.

Painting & Rendering

- Uses **TextOutW**
- Wide-character rendering ensures:
    - ✓ Correct Unicode output
    - ✓ Compatibility with multilingual scripts

During WM_PAINT:

- Program computes which Unicode range to display
- Characters are drawn cell-by-cell
- Grid structure remains fixed

Unicode Organization

- Unicode space divided into **256-character blocks**
- Each block contains:
    - ✓ 256 consecutive Unicode code points
- Scrollbar lets users explore:
    - ✓ Scripts
    - ✓ Symbols
    - ✓ Language ranges

This makes Unicode **visible and tangible**, not abstract.

Platform Compatibility

- Works on:
    - ✓ Windows NT
    - ✓ Windows 98
- Uses APIs supported by both
- TextOutW ensures Unicode consistency across platforms

Why UNICHARS Matters

Practical Uses

- **Font exploration**
  - ✓ Inspect Unicode coverage
  - ✓ Find missing glyphs
- **Unicode research**
  - ✓ Study scripts and symbols
  - ✓ Understand encoding ranges
- **Developer reference**
  - ✓ Font selection via ChooseFont
  - ✓ Unicode-safe text rendering
- **Localization prep**
  - ✓ Verify font compatibility for multilingual apps

Key Takeaways

UNICHARS is a **Unicode visualization tool**. Uses:

- ✓ ChooseFont
- ✓ TextOutW
- ✓ Scrollbar-based navigation

- Demonstrates:
  - ✓ Unicode-safe rendering
  - ✓ Font switching
  - ✓ Precise text layout
- Ideal reference for:
  - ✓ Internationalization
  - ✓ Font inspection
  - ✓ Unicode-aware WinAPI apps

UNDERSTANDING PARAGRAPH FORMATTING

The goal of paragraph formatting is simple: place text neatly between the margins and control how it lines up—left, right, centered, or justified.

DrawText is fine for quick jobs, but it doesn't give you enough control when layouts get more complex.

For better formatting, Windows gives you a few core tools:

- **GetTextExtentPoint32** tells you how wide and tall a piece of text will be with the current font.
- **TextOut** draws text exactly where you want it.
- **SetTextJustification** adjusts spacing so text lines up with both margins.

Each alignment has a clear purpose:

- **Left-aligned** text is best for body content because it's easy to read.
- **Right-aligned** text works well for dates or page numbers.
- **Centered** text is good for titles, headings, or short quotes.
- **Justified** text looks clean and professional, but overuse can stretch words too much and hurt readability.

Formatting a Single Line

```
// Get text extents
GetTextExtentPoint32(hdc, szText, lstrlen(szText), &size);

// Left-aligned text
xStart = xLeft;
TextOut(hdc, xStart, yStart, szText, lstrlen(szText));

// Right-aligned text
xStart = xRight - size.cx;
TextOut(hdc, xStart, yStart, szText, lstrlen(szText));

// Centered text
xStart = (xLeft + xRight - size.cx) / 2;
TextOut(hdc, xStart, yStart, szText, lstrlen(szText));

// Justified text
SetTextJustification(hdc, xRight - xLeft - size.cx, 3);  // Distribute extra space among 3 spaces
xStart = xLeft;
TextOut(hdc, xStart, yStart, szText, lstrlen(szText));

// Clear justification error for next line
SetTextJustification(hdc, 0, 0);
```

This section focuses on **measuring text**, **breaking it into lines**, and **formatting it correctly** when drawing with GDI.

Core GDI Text Functions (Know These First)

I. GetTextExtentPoint32

- Measures the **pixel width and height** of a string for the selected font.
- Used to:
  - ✓ Fit text inside margins
  - ✓ Decide when to wrap a line
  - ✓ Compute line spacing

Think of it as:
👉 *"How much screen space will this text consume?"*

II. TextOut

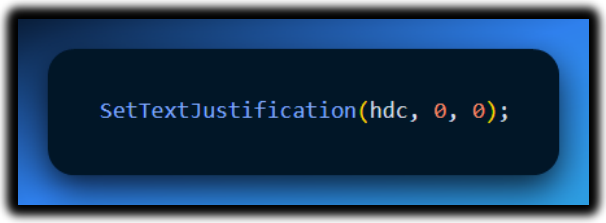
- Draws text at an **exact (x, y)** position.
- No layout logic.
- No wrapping.
- No alignment.

👉 You compute everything first, then TextOut just paints.

III. SetTextJustification

- Enables **justified text**
- Distributes extra horizontal space **between words**
- Only affects the **current line**

⚠️ Important:



Always call this **before drawing the next line**, or spacing will stack incorrectly.

Formatting Multi-Line Text (Correct Mental Model)

Formatting text is a **4-step pipeline**, not magic.

I. Calculate Available Width (Step 1)

Available width = right_margin - left_margin

This value defines the **maximum width** a line may occupy.

II. Break Text into Lines (Word Wrapping)

Goal:
👉 Build lines that **do not exceed available width**

Core idea:

- Split text into words
- Add words one by one to a line
- Measure line width after each addition
- If it overflows → start a new line

This logic exists **before any drawing happens**.

Word Wrapping

Internally, a word-wrapping algorithm:

- Uses strlen to track text length
- Uses GetTextExtentPoint32 to measure pixel width
- Stores completed lines in an array or buffer

You do **not** draw while wrapping.

III. Align Each Line (Step 3)

Once lines are built, alignment is applied **per line**.

Supported alignments:

- Left
- Right
- Center
- Justified

Key idea:

- Alignment = **padding math**
- Justification = **space redistribution**

Justification is special:

- Requires counting spaces
- Uses SetTextJustification
- Must be reset after each line

IV. Advance Vertical Position (Step 4)

After drawing a line: **y += line_height**

Line height comes from:

- Font height
- External leading
- Optional paragraph spacing

Alignment Concepts

Left Alignment

- Default
- No adjustment needed

Right Alignment

- Shift text right so it ends at the right margin

Center Alignment

- Equal padding on both sides

Justified Alignment

- Extra space distributed **between words**
- Last line is usually **not justified**
- Reset justification after every line

Text Metrics & Spacing Considerations

These affect **how your text feels**, not just how it fits.

I. Font Size & Style

- Larger fonts → larger line height
- Bold fonts → wider text
- Italics → different metrics

Always measure after selecting the font.

II. Line Spacing

- Tight spacing = dense, hard to read
- Loose spacing = airy, easier on eyes

Use metrics, don’t guess.

III. Paragraph Spacing

- Add extra vertical gap **between paragraphs**
- Do NOT rely on blank lines

IV. Indentation

- Horizontal offset at line start
- Common for:
    - ✓ Paragraphs
    - ✓ Quotes
    - ✓ Lists

V. Tab Stops

- Used for column-like layouts
- Avoid hardcoding spaces

Best Practices (Hard Rules)

Measure text — never assume
Wrap text before drawing
Reset justification after each line
Use readable fonts
Keep spacing consistent
Don’t justify everything
Avoid excessive font styles

Font Selection Guidelines

Good fonts:

- Clear letter shapes
- Good spacing
- Designed for screens

Safe defaults:

- Arial
- Verdana
- Calibri
- Segoe UI

Avoid:

- Decorative fonts for body text
- Too many font families

What This Section Is REALLY About

This section is **not about fancy UI**.

It teaches:

- How GDI thinks about text
- Why layout is manual
- How professional text rendering is built step-by-step

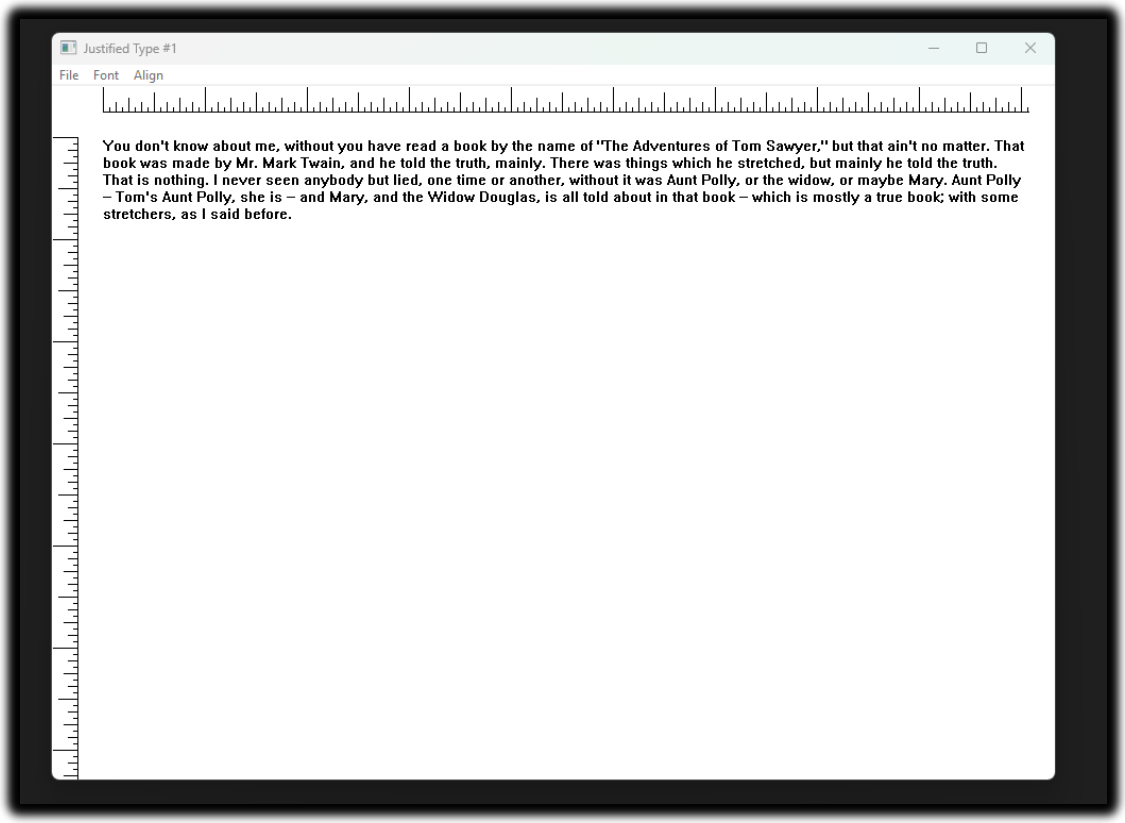
Everything here transfers directly to:

- Printers
- Dialog controls
- Custom editors
- Report rendering

GDI never formats text for you.
You measure → wrap → align → draw.
Once you get this, the chaos disappears.

JUSTIFY.C PROGRAM

Formatting and Justifying Text in Windows GDI



Purpose of JUSTIFY.C

This program demonstrates how to:

- Format a paragraph inside a rectangle
- Align text as left, right, center, or justified
- Make on-screen text match printed output as closely as possible
- Visually show margins and layout using rulers

It focuses on **manual text layout**, not controls.

Program Structure Overview

JUSTIFY.C is built around four main parts:

- Program setup and message loop
- Ruler drawing for layout guidance
- Text formatting and justification logic
- Window message handling

Each part has a clear responsibility.

Header Files and Global Context

windows.h

Provides access to:

- Window creation
- GDI drawing
- Fonts and text rendering
- Message handling

resource.h

Defines:

- Menu commands
- Dialog identifiers
- Icons and UI resources

Global Context

- Application name constant used for window titles
- WndProc declared as the main message handler

This shared context allows all parts of the program to work together.

WinMain: Application Entry Point

WinMain controls the lifetime of the program.

Window Class Registration

- Defines window behavior
- Associates WndProc with the window
- Sets default cursor, icon, and background

Window Creation

- Creates the main window
- Uses a title indicating justified text demonstration
- Prepares the window for drawing and interaction

Message Loop

- Listens for system and user events
- Dispatches messages to WndProc
- Keeps the program responsive

DrawRuler: Visual Layout Guides

I. Purpose

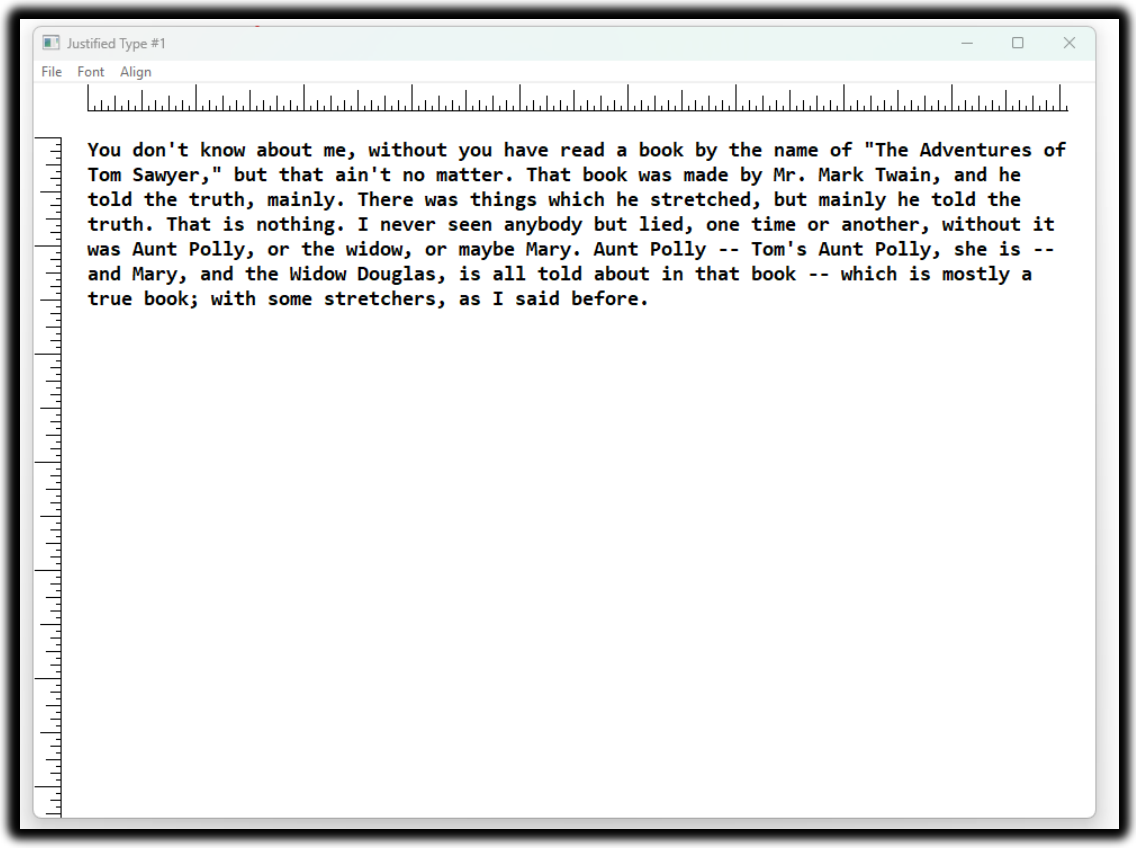
Draws horizontal and vertical rulers around the text area.

Rulers help:

- Visualize margins
- Understand alignment
- Debug spacing and justification

II. Key Steps in DrawRuler

Preserving Graphics State - SaveDC stores the current device context state so changes do not leak outside this function.



III. Mapping Mode Setup

MM_ANISOTROPIC is used to allow independent scaling in both directions.

This makes rulers adapt correctly to:

- Window size
- Screen resolution
- Printer resolution

IV. Logical Coordinate System

- A logical space of 1440 units is used
- This corresponds to twips, common in typography
- Makes ruler measurements predictable

V. Mapping to Device Pixels

Logical units are mapped to physical pixels using the device DPI.
This ensures rulers remain accurate on different displays.

VI. Origin Adjustment

The origin is shifted by half an inch.
This aligns rulers naturally with the text area.

VII. Drawing Ruler Lines

MoveToEx and LineTo draw the main ruler axes.

VIII. Tick Marks

- A loop draws tick marks at regular intervals
- Different tick sizes create visual hierarchy
- Helps users read spacing more easily

IX. Restoring Graphics State

RestoreDC returns the device context to its original state.

This prevents side effects on text rendering.

WndProc: Message Handling Core

WndProc controls all window behavior.

WM_CREATE

- Initializes font settings
- Prepares print and font dialogs
- Sets initial alignment mode

WM_COMMAND

Triggered by menu actions.

Handles:

- Printing the justified text
- Opening font selection dialog
- Changing alignment mode dynamically

WM_PAINT

Responsible for drawing content.

Steps:

- Call DrawRuler to show layout guides
- Call Justify to format and draw text

WM_DESTROY

- Cleans up resources
- Posts quit message
- Ends the program cleanly

Justify Function: Core Text Logic

This function performs **all paragraph layout work**.

I. Responsibilities

- Fit text inside a rectangle
- Break text into lines
- Apply alignment
- Draw the final result

II. Supported Alignments

- Left
- Right
- Center
- Justified

Justify Function Workflow

I. Line Position Tracking

yStart tracks the vertical position of each line.

II. Text Processing Loop

A loop processes characters one by one:

- Builds lines that fit horizontally
- Stops before exceeding available width

III. Skipping Leading Spaces

Leading spaces are ignored at the start of a line.
This prevents uneven left margins.

IV. Line Break Detection

- Words are tested using GetTextExtentPoint32
- If a word would overflow, a line break is inserted
- This guarantees text stays inside the rectangle

V. Justified Alignment Handling

When justification is active:

- Extra space is calculated
- SetTextJustification spreads space between words
- Both margins become visually even

VI. Rendering the Line

TextOut draws the formatted line at the correct position.

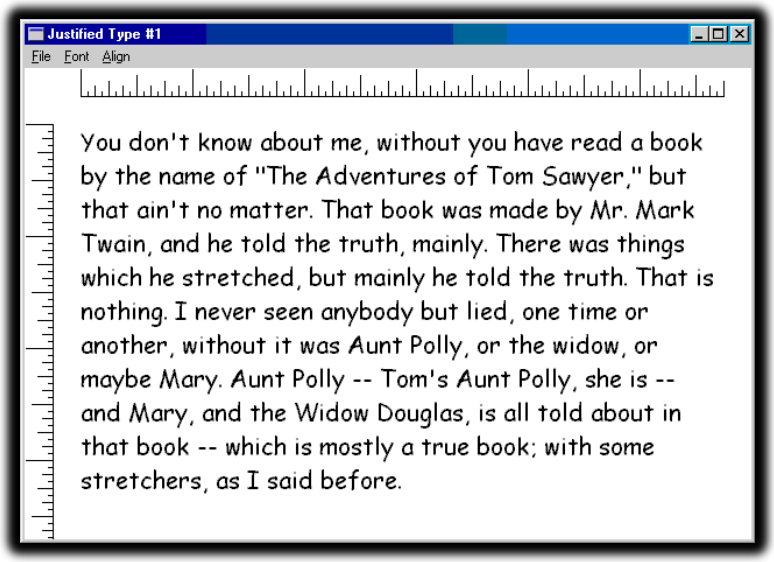
VII. Preparing for the Next Line

- SetTextJustification is reset to zero
- yStart is advanced by line height
- The next line begins cleanly

VIII. Loop Completion

The loop continues until the entire paragraph is rendered.

IX. After a Font Change



WYSIWYG Goals and Challenges

I. Goal

Screen output should match printed output:

- Same line breaks
- Same spacing
- Same alignment

II. Challenges

- Screens and printers use different resolutions
- Rounding errors affect text width
- TrueType fonts behave differently across devices

Key Techniques for Better Consistency

I. Unified Formatting Rectangle

Use the same rectangle dimensions for:

- Screen drawing
- Printer drawing

This creates a shared layout reference.

II. Device-Aware Scaling

- Query device DPI using GetDeviceCaps
- Scale layout logic based on device resolution

III. Accurate Text Measurement

- Always measure text using GetTextExtentPoint32
- Never assume character sizes

IV. Careful Justification Control

- Apply justification per line
- Always reset spacing afterward

What This Program Teaches

JUSTIFY.C is not about UI polish.

It teaches:

- Manual text layout
- Real justification logic
- Device-independent formatting
- Why text rendering is hard

This knowledge transfers directly to:

- Printing systems
- Text editors
- Report generators
- Layout engines

JUSTIFY2 PROGRAM

JUSTIFY2 is simply an improved version of JUSTIFY1.

Its main goal is better accuracy when showing printer output on the screen, especially when using TrueType fonts.

It is based on:

- TTJUST by David Weise (TrueType justification demo)
- Concepts already used in JUSTIFY1

Nothing new conceptually — just more precision.

I. What JUSTIFY2 Improves

Better Justified Text

The Justify function is more careful when spreading spaces between words.
This avoids:

- Huge gaps between words
- Text looking squeezed or uneven

Result: cleaner justified text.

II. Screen and Printer Match Better

JUSTIFY2 focuses on WYSIWYG behavior.

What you see on screen should match what comes out of the printer:

- Same alignment
- Same line breaks
- Same spacing

This is harder than it sounds because:

- Screens and printers use different resolutions
- Fonts scale differently on each device

JUSTIFY2 handles this more carefully than JUSTIFY1.

III. TrueType Font Handling (Why JUSTIFY2 Exists)

TrueType fonts are designed on a hidden grid called the em-square.

This grid:

- Defines the real shape and width of characters
- Is independent of screen resolution or printer DPI

The size of this grid is stored in:

- otmEMSquare inside OUTLINETEXTMETRIC

JUSTIFY2 uses this information to get real character widths instead of guessed ones.

IV. Getting Accurate Character Widths

JUSTIFY2 does the following:

- Creates a temporary font using the em-square size
- Selects it into a device context
- Uses GetCharWidth to read exact character widths

These widths:

- Represent the original font design
- Are not affected by scaling
- Work consistently across devices

The widths are stored as integers for ASCII characters only, to keep things fast.

V. Scaling Widths for Real Use

Design widths alone are not enough. JUSTIFY2 converts them into usable sizes by:

- Scaling them to match the current font size
- Producing floating-point widths

This is done using:

- GetScaledWidths
- Cached width values for performance

This allows accurate measurement on:

- Screen
- Printer

VI. Precise Text Measurement

GetTextExtentFloat uses the scaled widths to:

- Measure entire strings very accurately
- Help Justify decide where lines should break

This avoids:

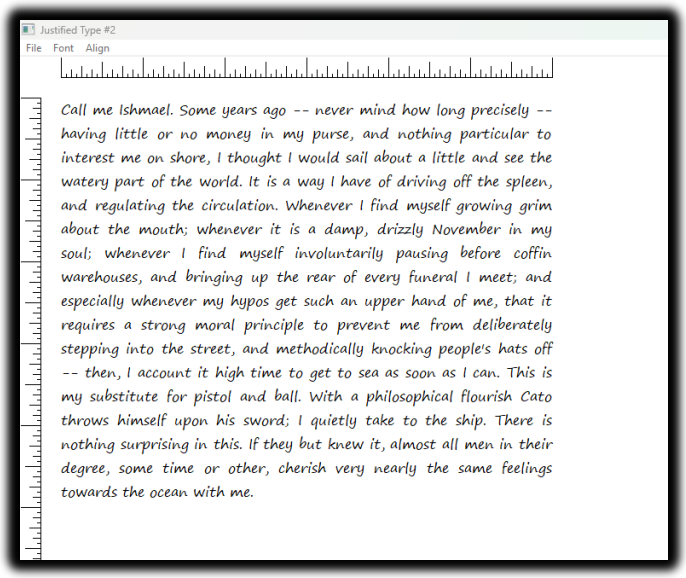
- Random line wrapping
- Printer vs screen mismatch

VII. Ruler Drawing (Still the Same Idea)

The DrawRuler function:

- Uses mapping modes
- Works in logical units (twips)
- Scales cleanly on different devices

This is mostly unchanged from JUSTIFY1, just reused cleanly.



VIII. Error Handling and Code Quality

JUSTIFY2 adds basic checks for:

- Printer availability
- Print failures

The code could still be cleaner, but it is more robust than JUSTIFY1.

GRAPHICS PATHS AND EXTENDED PENS

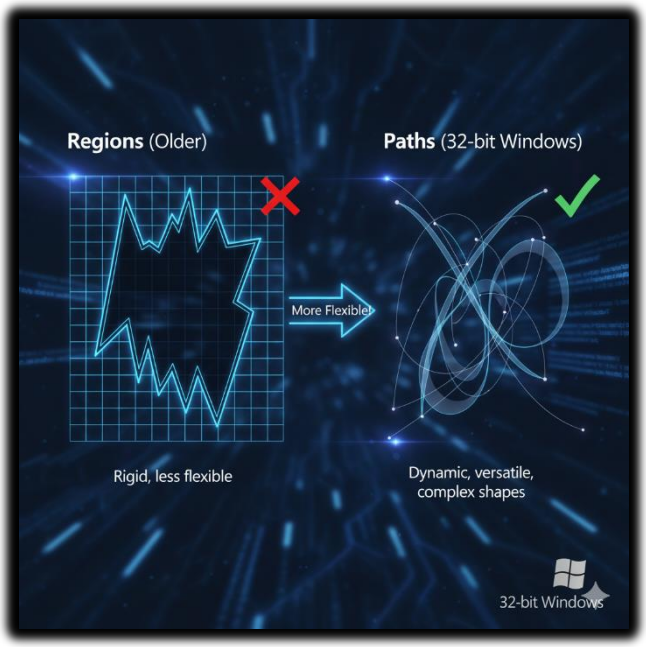
This section goes deeper into font and shape drawing using **Graphics Paths** and **Extended Pens**. Before this, we used basic graphics primitives. Now we move to more flexible and powerful drawing tools.

I. Graphics Paths (The Core Idea)

A Graphics Path is a hidden container inside GDI that stores:

- Lines
- Curves
- Shapes

Nothing is drawn immediately. You build the shape first, then draw it later.
Paths were introduced in 32-bit Windows and are more flexible than regions.



II. Starting a Path

You begin a path by telling GDI to start recording drawing commands.

At this point:

- Nothing appears on screen
- GDI just remembers what you draw

III. Adding Lines and Curves to a Path

You build the path using normal drawing functions:

- LineTo
  Draws a straight line from the current position
- PolylineTo
  Draws multiple connected straight lines
- BezierTo
  Draws smooth curves using control points

All these commands:

- Add geometry to the path
- Do not draw anything yet

IV. Subpaths (Important Concept)

A single path can contain many subpaths.

A subpath:

- Starts at a point
- Continues until it is closed or broken

A new subpath starts when:

- You call MoveToEx
- You close the previous subpath
- Certain mapping or viewport changes happen

V. Closing a Subpath

A subpath can be:

- Open (just ends)
- Closed (forms a loop)

To close it:

- The last point connects back to the first point
- GDI adds a closing line if needed

After closing:

- Any new drawing starts a new subpath

VI. Ending the Path

When you finish building the shape:

- You tell GDI the path is complete

Now the path is ready to be rendered or reused.

VII. Rendering a Path (Making It Visible)

Once the path exists, you can draw it in different ways.

StrokePath

- Draws only the outline
- Uses the currently selected pen
- Similar to tracing the shape

FillPath

- Fills the inside of the path
- Uses the current brush
- No outline is drawn

StrokeAndFillPath

- Draws the outline
- Fills the inside
- Done in one operation

PathToRegion

- Converts the path into a region
- Regions are used for:
    - ✓ Clipping
    - ✓ Hit-testing
    - ✓ Defining drawing boundaries

Paths are more flexible than regions because:

- Paths store curves and splines
- Regions store scanlines only

SelectClipPath

- Uses the path as a clipping area
- Future drawing is restricted to the path shape
- Useful for masking effects

VIII. Important Behavior (Very Important)

After you render a path:

- The path is destroyed
- You must rebuild it if you need it again

Paths are temporary by design.

IX. Why Use Paths Instead of Direct Drawing?

Paths offer major advantages:

- Delayed rendering
  Build complex shapes first, draw once
- Complex shapes
  Supports Bézier curves and smooth arcs
- Clipping and filling
  One shape can outline, fill, and clip

This makes paths ideal for:

- Font outlines
- Special text effects
- Advanced graphics work

X. Extended Pens

Extended Pens allow:

- Thick strokes
- Custom styles
- Patterns and advanced effects

They work perfectly with paths and are covered next.

This section is not magic.

Paths are simply:

- Record first
- Draw later

Once you get that mental model, everything here clicks fast.

ENDJOIN.C PROGRAM

ENDJOIN.C demonstrates how to draw lines with different end caps and joins using Windows GDI, and introduces path-based rendering for smoother visuals and font creativity.

I. Basic Windows Program Structure

The program starts with standard Windows setup:

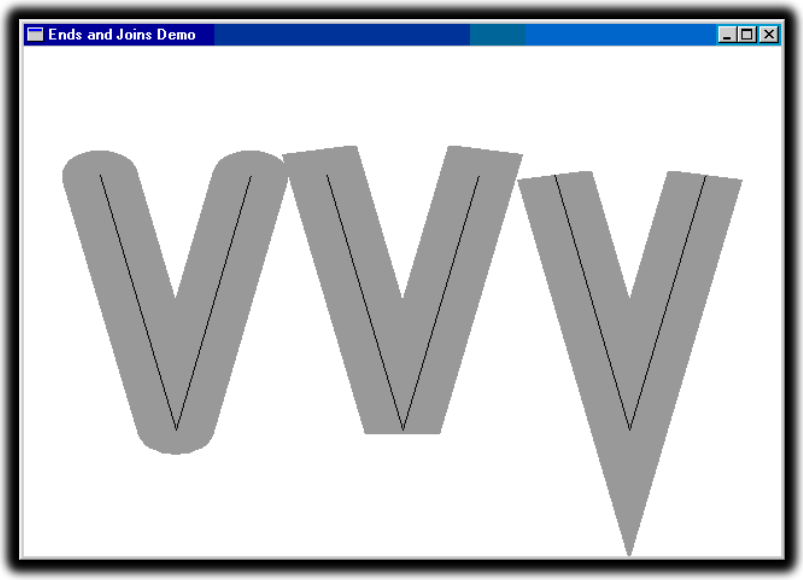
- **Header inclusion:** windows.h is included to access Windows API functions, structures, and constants.
- **Function declarations:**
  - ✓ WinMain is the entry point of the application.
  - ✓ WndProc handles window messages like painting, resizing, and closing.

II. WinMain Function

- **Window Class Registration:** A WNDCLASS structure is defined, setting the window's icon, cursor, background color, and linking WndProc as the message handler. The class is registered with RegisterClass.
- **Window Creation:** CreateWindow instantiates the window, sets title, style, position, and size.
- **Show and Update Window:** ShowWindow makes it visible, and UpdateWindow forces immediate redraw.
- **Message Loop:** GetMessage retrieves messages, TranslateMessage handles keyboard input, and DispatchMessage sends messages to WndProc.
- **Exit:** When WM_QUIT is received, the loop ends, returning msg.wParam.

The window redraws automatically with CS_HREDRAW and CS_VREDRAW to maintain appearance when resized. Main function:

```c
/* ENDJOIN.C - Ends and Joins Demo (c) Charles Petzold, 1998 */
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT("EndJoin");
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;

    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = szAppName;

    if (!RegisterClass(&wndclass))
    {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 0;
    }

    hwnd = CreateWindow(szAppName, TEXT("Ends and Joins Demo"), WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);

    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);

    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```



III. WndProc – Handling Window Events

The real work happens in WndProc. Its main responsibilities:

1. **Dynamic Pen Styles:**
   Different end caps and joins (round, square, bevel, miter) are selected dynamically to draw V-shaped lines, showcasing visual differences.
2. **Anisotropic Mapping Mode:**
   MM_ANISOTROPIC allows independent scaling of X and Y axes, giving precise control over how logical units map to pixels.
3. **Path-Based Drawing:**
   - ✓ BeginPath starts a path.
   - ✓ MoveToEx sets the starting point.
   - ✓ LineTo draws line segments.
   - ✓ EndPath ends the path.
   - ✓ StrokePath renders the entire path at once with the selected pen, ensuring smooth joins and proper end caps.
4. **Comparison with Stock Pen:**
   The program draws the same lines with the default black pen, letting you see the effect of custom widths and styles versus simple lines.
5. **Window Message Handling:**
   - ✓ WM_SIZE updates client area dimensions.
   - ✓ WM_PAINT triggers the drawing process.
   - ✓ WM_DESTROY calls PostQuitMessage to exit gracefully.

IV. Why StrokePath Matters

- **Line End Limitation:** Drawing lines individually applies end caps to each segment, which can make corners look jagged.
- **Path-Based Rendering:** StrokePath treats connected lines as a single object, applying join styles for smooth transitions.

V. Unlocking Font Creativity

Paths aren't just for lines—they can be used with font outlines:

- **Font Characters as Paths:** Outline fonts store characters as lines and curves.
- **Manipulating Fonts:** By treating font outlines as paths, you can apply stroke, fill, and transformation effects just like graphics.

VI. Key Takeaways

- **StrokePath is essential** for smooth rendering of connected lines and curves.
- **Paths allow advanced graphics**: precise control of joins, end caps, and visual effects.
- **Font outlines + paths = creative text effects**, giving flexibility beyond simple text display.

FONTOUT1 & FONTOUT2

These programs showcase **font outlines as GDI paths** and how to render them creatively using pens. They take text from a normal string to a fully manipulable graphical object.

I. Core Concepts

- **Paths as Containers:** A GDI path stores lines, curves, shapes, or font outlines for later rendering.
- **Font Outlines as Shapes:** TrueType fonts store characters as lines and curves, perfect for path-based manipulation.
- **Path-Based Text Output:** By enclosing text in a path (BeginPath → TextOut → EndPath), characters aren't drawn immediately—they're stored as vector outlines, ready for stroking or filling.

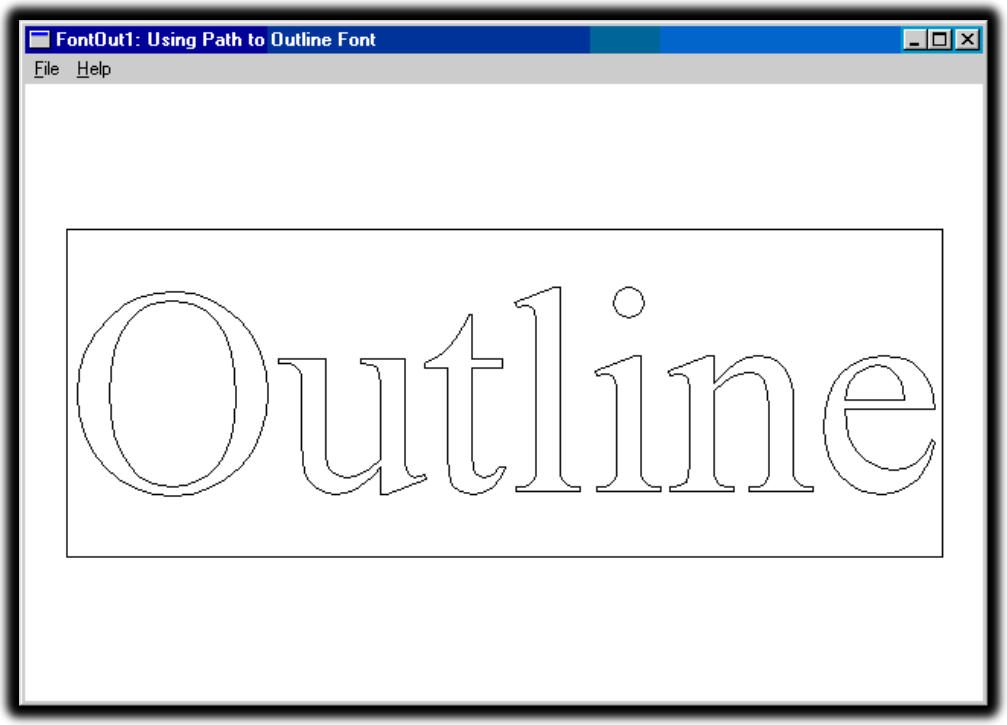
II. FONTOUT1 – Basic Outline Rendering

- **Goal:** Draw text as outlines instead of filled shapes.
- **Process:**
  - ✓ Select a large TrueType font.
  - ✓ Measure text dimensions using GetTextExtentPoint32 for positioning.
  - ✓ Begin a path with BeginPath.
  - ✓ Draw text into the path using TextOut.
  - ✓ End the path with EndPath.
  - ✓ Stroke the path with StrokePath to render outlines using the default pen.
- **Cleanup:** Restore default font and delete the temporary font.

Result: Text appears as outlined characters, exposing the skeleton of each letter.

Creative Potential:

- Change pen styles and colors.
- Combine outlines with other graphics.
- Animate or distort characters.



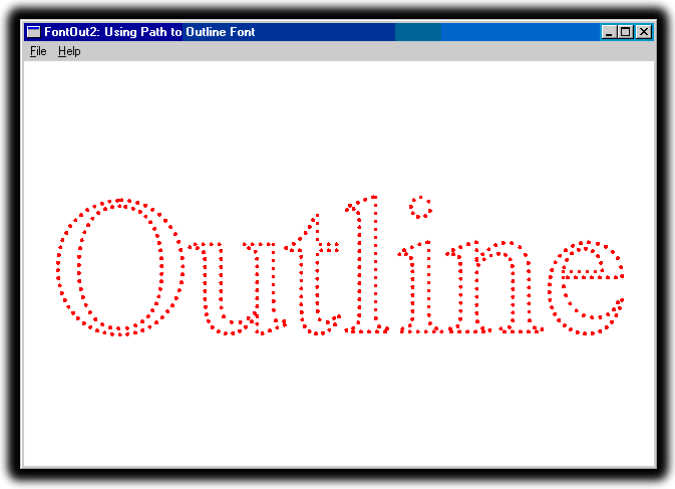
III. FONTOUT2 – Advanced Outlined Text with Custom Pens

- **Enhancement:** Adds **custom pens** to stroke text outlines with visual effects.
- **Key Steps:**
  - ✓ Create a TrueType font and set a **transparent background** for clarity.
  - ✓ Begin a path and draw the text inside it.
  - ✓ Create a custom pen using ExtCreatePen (e.g., dotted, dashed, colored).
  - ✓ Select the pen and StrokePath to render the text outlines with style.
  - ✓ Restore default pen and font, and release temporary objects.

Result: Outlined text appears with **dotted or styled edges**, giving a decorative effect.

Creative Possibilities:

- Experiment with pen styles (dashed, hatched, gradient).
- Combine with shapes or other text for intricate designs.
- Animate outlines or apply transformations for dynamic visual effects.



IV. Key Takeaways

- Paths let you **manipulate text like graphics**, not just print letters.
- Fonts as paths unlock **stylistic and artistic control**, including stroke styles, fills, and animations.
- FONTOUT2 shows how **pens transform outlines** into visually expressive designs.
- Transparent backgrounds and careful text positioning enhance presentation and flexibility.

FONTFILL & FONTCLIP – REVAMPED NOTES

These programs demonstrate **advanced text rendering using paths, pens, brushes, and clipping**. They show how fonts become fully manipulable graphics objects in Windows GDI.

I. FONTFILL – Filling Text with Patterns

- **Goal:** Fill font outlines with patterns and background colors while optionally stroking edges.
- **Core Steps:**
  - ✓ Create a TrueType font and select it into the device context.
  - ✓ Set **transparent background** to avoid affecting the window behind text.
  - ✓ Begin a path and draw the text with TextOut.
  - ✓ End the path.
  - ✓ Create a **patterned brush** (e.g., diagonal cross hatch).
  - ✓ Set **opaque background** for the pattern.
  - ✓ Use StrokeAndFillPath to outline and fill the text in one step.
  - ✓ Restore default pen, brush, and font; delete temporary objects.

Creative Possibilities:

- Use different brushes and patterns (hatching, gradients, textures).
- Combine filling and outlining for richer visual effects.
- Integrate filled fonts into complex graphics or compositions.

Takeaway: Paths let you treat text as a canvas—strokes and fills can be applied like any graphic shape.

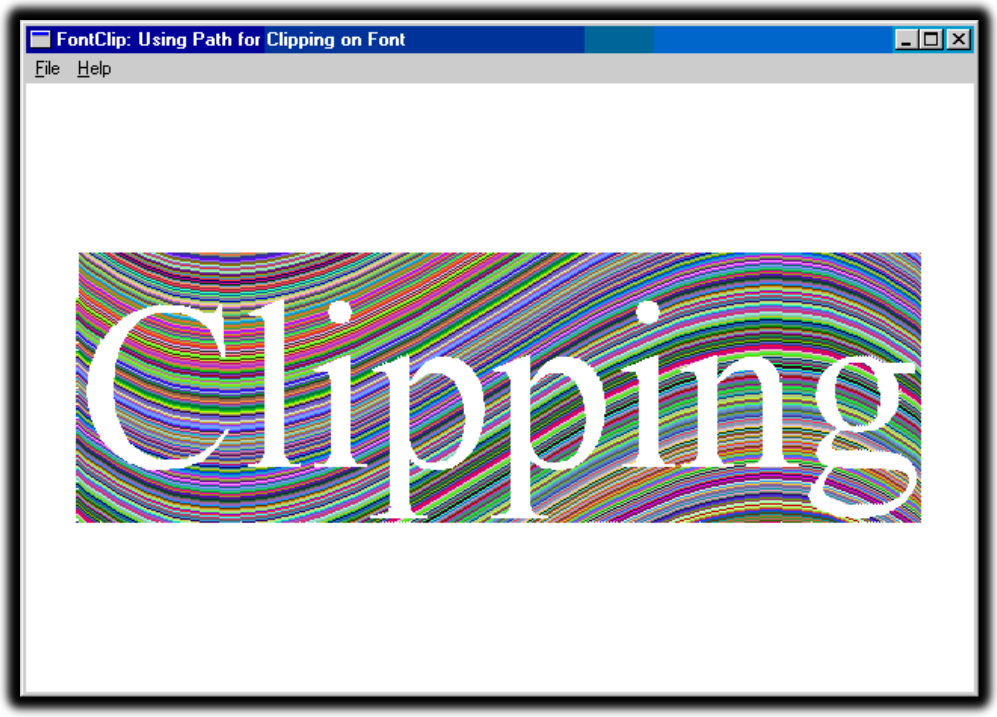
II. FONTCLIP – Clipping with Font Paths

- **Goal:** Use font outlines as a clipping region to control where graphics appear.
- **Core Steps:**
  - ✓ Create a TrueType font and measure text dimensions.
  - ✓ Begin a path and draw text into it.
  - ✓ End the path and use SelectClipPath to set the clipping region.
  - ✓ Draw other graphics (e.g., Bezier curves) restricted to the font shape.
- **SetBkMode Impact:**
  - ✓ Transparent: clipping region matches actual character outlines.
  - ✓ Opaque or omitted: clipping region is the rectangular bounding box of the text.

Creative Potential:

- Combine font clipping with other shapes for intricate designs.
- Experiment with colors, Bezier curves, and font styles.
- Layer multiple clipping regions for complex masking effects.

Takeaway: FONTCLIP turns text into a **masking tool**, making font shapes the active design element rather than just a medium for letters.



General Font-Path Insights

- **Paths**: Store outlines of fonts, lines, curves, and shapes; can be stroked, filled, or converted into clipping regions.
- **Pens and Brushes**: Custom pens define outline style; brushes define fill style/pattern.
- **Stroke vs StrokeAndFill**:
  - ✓ StrokePath draws only the edges.
  - ✓ StrokeAndFillPath draws edges and fills interiors in one step.
- **Clipping Regions**: Let text outlines control where other graphics appear.
- **Transparency & Background Modes**:
  - ✓ Transparent: text doesn't overwrite existing content.
  - ✓ Opaque: background is filled; affects clipping behavior.

Creative Notes:

- Fonts + Paths = Full graphic manipulation of text.
- Combine with animations, transformations, and patterns for visual effects.
- You can layer multiple paths, pens, brushes, and clipping masks to craft unique compositions.

FINAL STATEMENT – KEY TAKEAWAYS & WINAPI MASTERING TIPS 🎯

Things to Remember from Font & Graphics Paths

1. **Treat Text as Graphics**
    - ✓ TrueType fonts can be converted into paths.
    - ✓ Once in a path, text can be **stroked, filled, or used as a clipping region**.
    - ✓ Paths give you creative control—think of letters as shapes, not just characters.
2. **Paths vs Regions**
    - ✓ Paths are flexible: support Bézier curves, complex shapes, and multi-subpaths.
    - ✓ Regions are simpler: store filled pixel data, not curves.
    - ✓ Use paths when you want **precision, effects, and dynamic transformations**.
3. **Pens and Brushes Matter**
    - ✓ Pens define outlines; brushes define fills.
    - ✓ Extended pens allow patterns, textures, and gradients—your tool for artistic flair.
    - ✓ Experiment: dotted, dashed, hatched, or gradient strokes make fonts visually expressive.
4. **Background Mode & Transparency**
    - ✓ Transparent mode keeps the window behind visible; opaque mode fills it.
    - ✓ Background mode affects clipping behavior and visual output—know which one to use.
5. **Experimentation is Key**
    - ✓ Mix fonts, paths, clipping, and graphical effects.
    - ✓ Try different alignments, brushes, and mapping modes.
    - ✓ Observe how each function changes the visual outcome—learning comes from doing.

How to Be Good at WinAPI Coding

1. **Master the Core Concepts**
    - ✓ Messages, windows, device contexts, and GDI objects are the foundation.
    - ✓ Understand **how DCs, pens, brushes, paths, and fonts interact**.
2. **Think in Steps, Not Lines**
    - ✓ Focus on what the program *does*, not just the code.
    - ✓ Separate logical steps: initialization, drawing, resource cleanup, and termination.
3. **Always Manage Resources**
    - ✓ Select/restore objects in DCs.
    - ✓ Delete temporary pens, brushes, and fonts.
    - ✓ Prevent leaks; WinAPI is unforgiving.
4. **Draw & Observe**
    - ✓ WinAPI is very visual—run code, tweak parameters, and watch changes.
    - ✓ Seeing line joins, strokes, fills, and clipping regions in action teaches more than reading alone.
5. **Experiment with Advanced Features**
    - ✓ Paths, extended pens, and clipping regions unlock creativity.
    - ✓ Combine features gradually—first get a single path working, then layer brushes, fills, and clipping.
6. **Be Patient and Iterative**
    - ✓ WinAPI coding can be verbose and low-level.
    - ✓ Start simple, test often, and build complexity in layers.

Bottom line: WinAPI mastery isn't about memorizing every function—it's about understanding **how windows, messages, and GDI objects interact**, and then **using creativity to manipulate them**. Think of it as **painting with code**: the more you experiment, the stronger your intuition becomes.

WinAPI Mastery

Painting with Code: Experiment & Intuition

Windows API