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

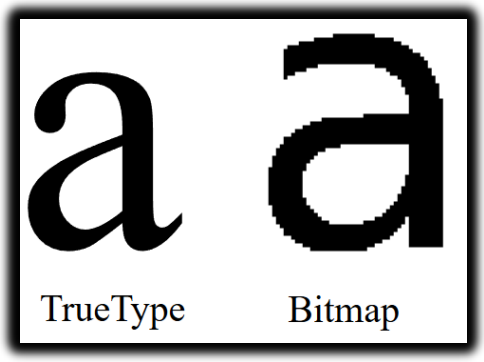
This is where WinAPI text stops being *static* and becomes *programmable*.

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)

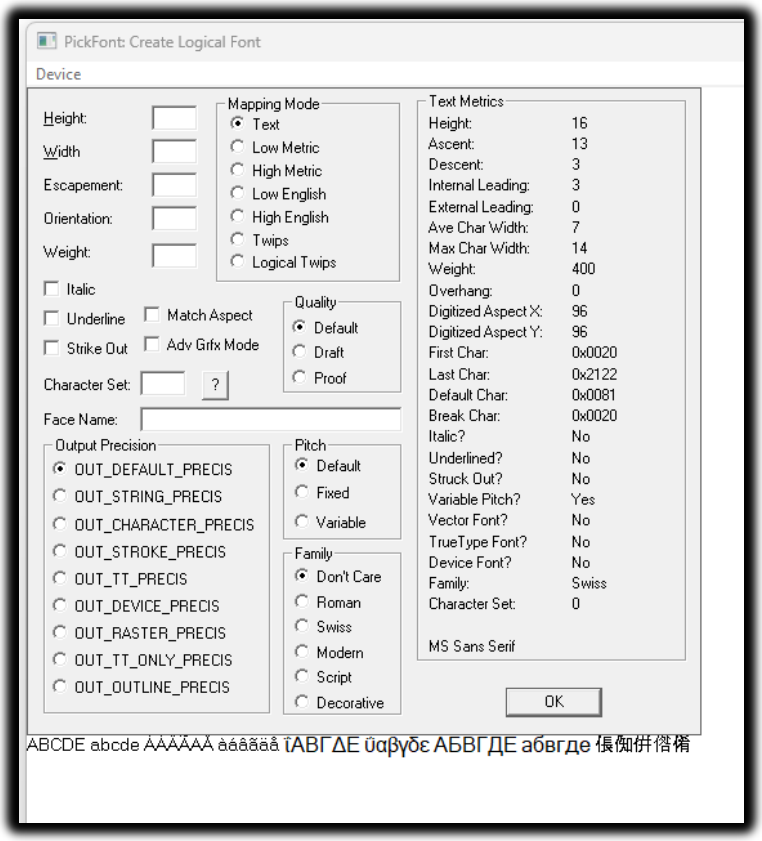
MySetMapMode sets the **ruler type** for the Device Context before drawing, ensuring sizes are interpreted correctly.

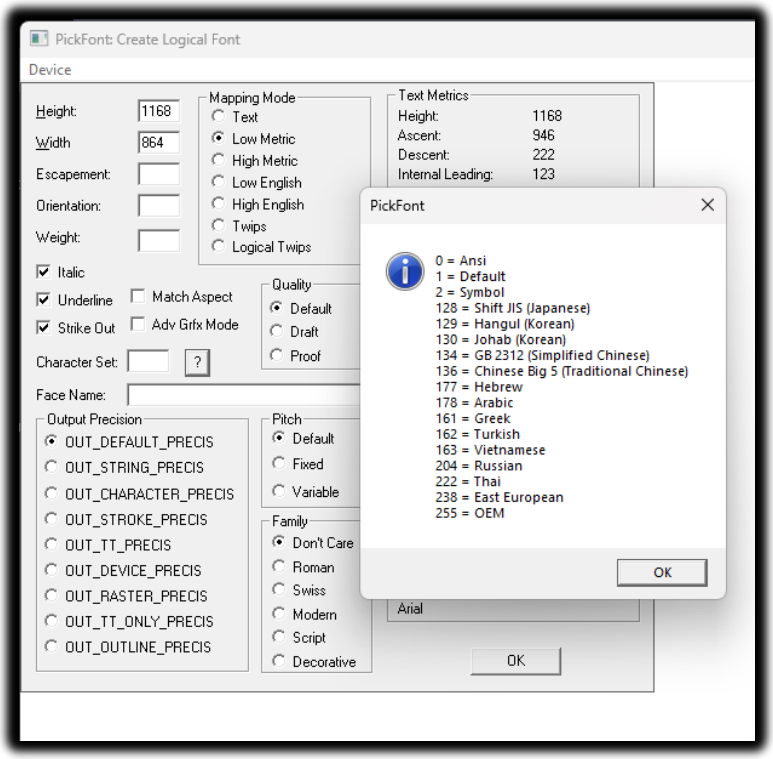
VI. Summary – Flow of Data

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.

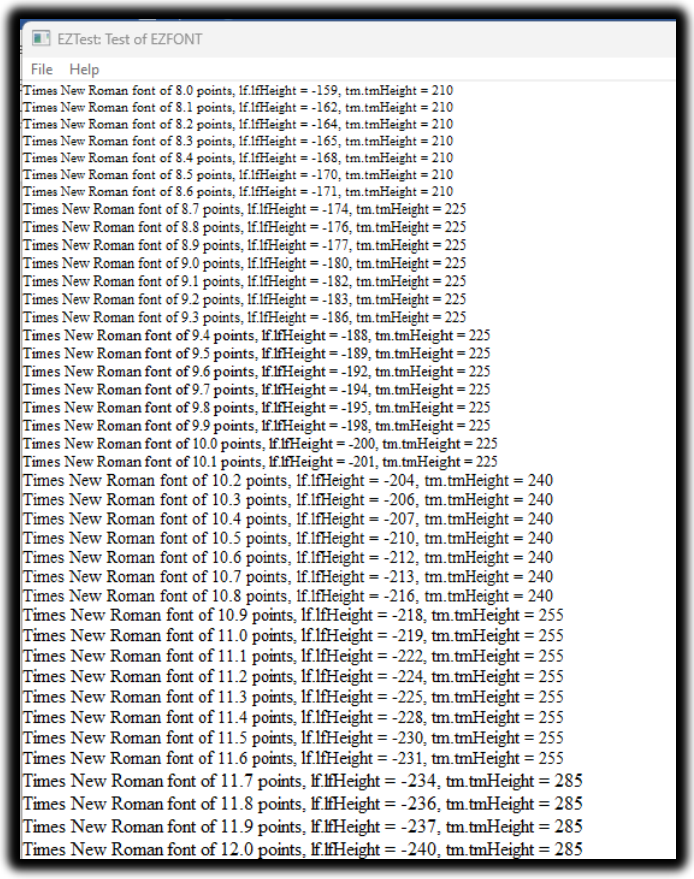
3. The "Symbol" Trap If you set lfCharSet to SYMBOL_CHARSET, Windows treats the font differently. It stops trying to map letters to Unicode standard slots and just gives you the raw glyphs. This is used for "Wingdings" or specialty icon fonts.

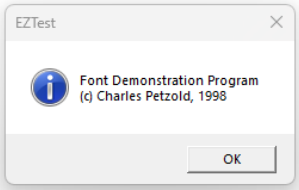
EZFONT PROGRAM

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:

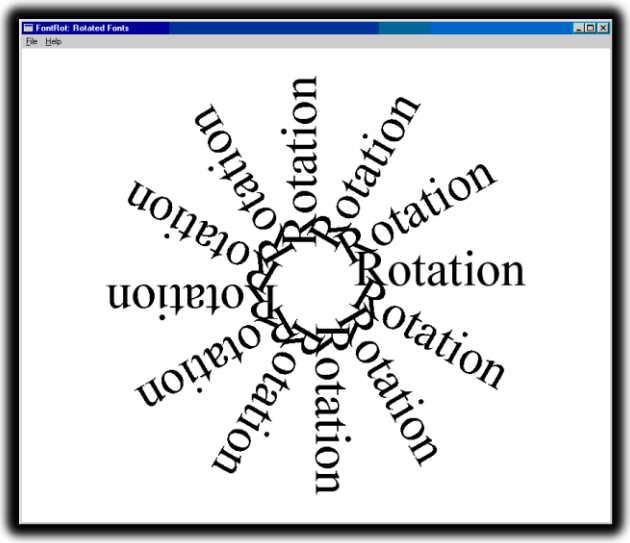
- Help scaling
- Improve consistency
- Do **not** defeat rasterization

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**.

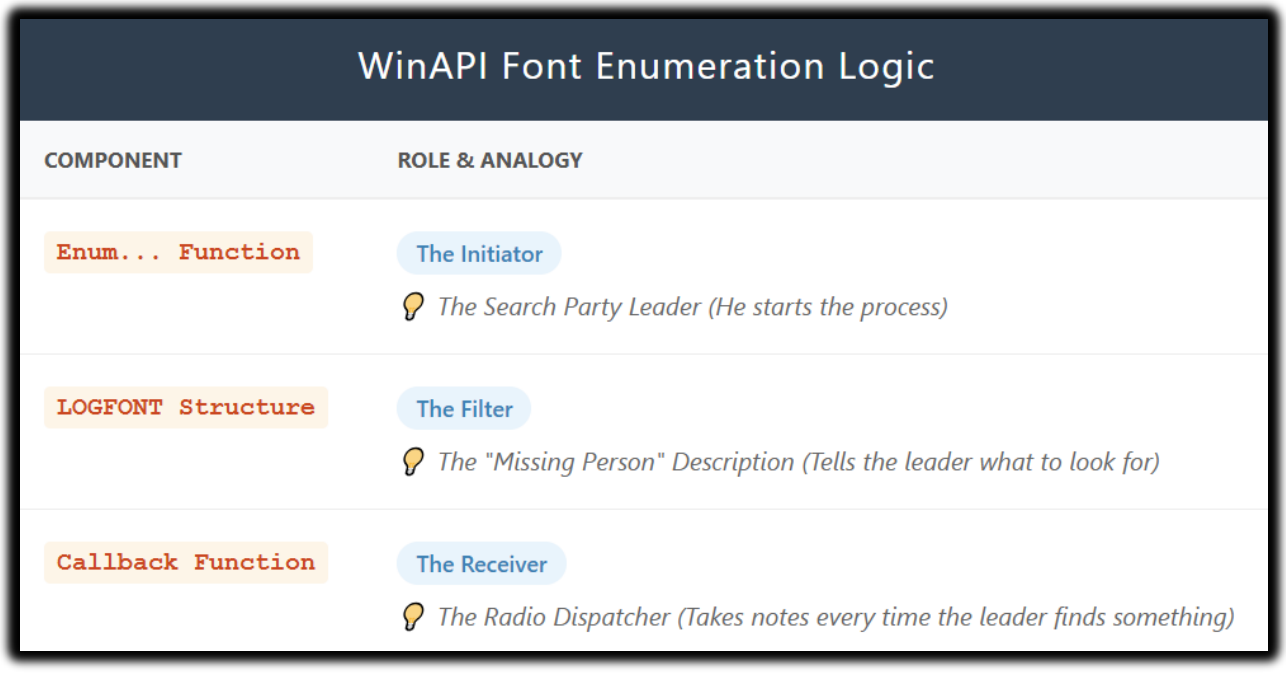
The callback receives:

- 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

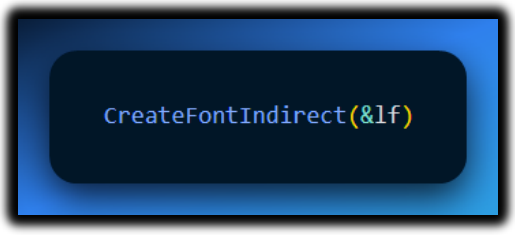
III. Repaint Phase

- 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

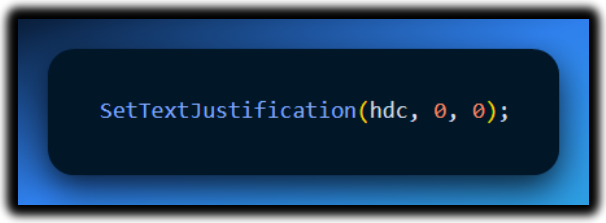
- Draws text at an **exact (x, y)** position.
- No layout logic.
- No wrapping.
- No alignment.

👉 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

Final Mental Model 🧠

GDI never formats text for you.
You measure → wrap → align → draw.

Once you get this, the chaos disappears.

JUSTIFY.C PROGRAM

1. Header Files and Global Context:

windows.h: This header empowers the program with a rich set of Windows API functions for graphics, window management, and user interactions.

resource.h: This file, while not explicitly shown, holds definitions for application resources such as menus, dialogs, and icons, contributing to the user interface and experience.

Global Variables: These variables establish a shared context for the program's elements:

JUSTIFY1: The application's name, used for window titles and identification.

WndProc: The designated window procedure, responsible for handling events and messages within the main window.

2. WinMain: The Program's Entry Point:

Registration: The code commences by registering the window class, defining its attributes and behaviors within the Windows environment.

Window Creation: A main window is subsequently created, bearing the title "Justified Type #1" and prepared to receive user input and display content.

Message Loop: The program enters a continuous loop, patiently listening for and responding to various events such as key presses, mouse clicks, window resizing, and system notifications. This loop maintains the application's responsiveness and interactivity.

3. DrawRuler: Visualizing Text Boundaries:

Ruler Creation: This function meticulously renders horizontal and vertical rulers adorned with tick marks, serving as visual guides for text alignment and formatting. These rulers provide a clear reference for the user, enhancing the visual clarity of the text layout.

4. Justify: Orchestrating Text Formatting:

Paragraph Processing: This function meticulously crafts the appearance of a paragraph of text within a specified rectangular region. It gracefully handles diverse alignment styles, including left, right, center, and justified.

Line Breaking and Spacing: It meticulously calculates optimal line breaks to ensure text fits within the designated area, and meticulously adjusts spacing between words for justified alignment, achieving a visually balanced and pleasing presentation.

5. WndProc: Responding to Window Events:

Message Handling: This function diligently serves as the central hub for handling various events and messages directed towards the main window. It effectively orchestrates actions based on different types of interactions and system signals.

WM_CREATE: Upon window creation, this message prompts initialization of font and printer dialog structures, setting the stage for user customization and printing capabilities.

WM_COMMAND: This message arises in response to menu commands, triggering actions such as:

- Printing: Engaging the printing process, relying on the Justify function to meticulously render text onto the selected printer.
- Font Selection: Presenting a font dialog to empower users with choices for visual aesthetics and readability.
- Alignment Adjustment: Updating the text alignment based on user preferences, ensuring versatility in text presentation.

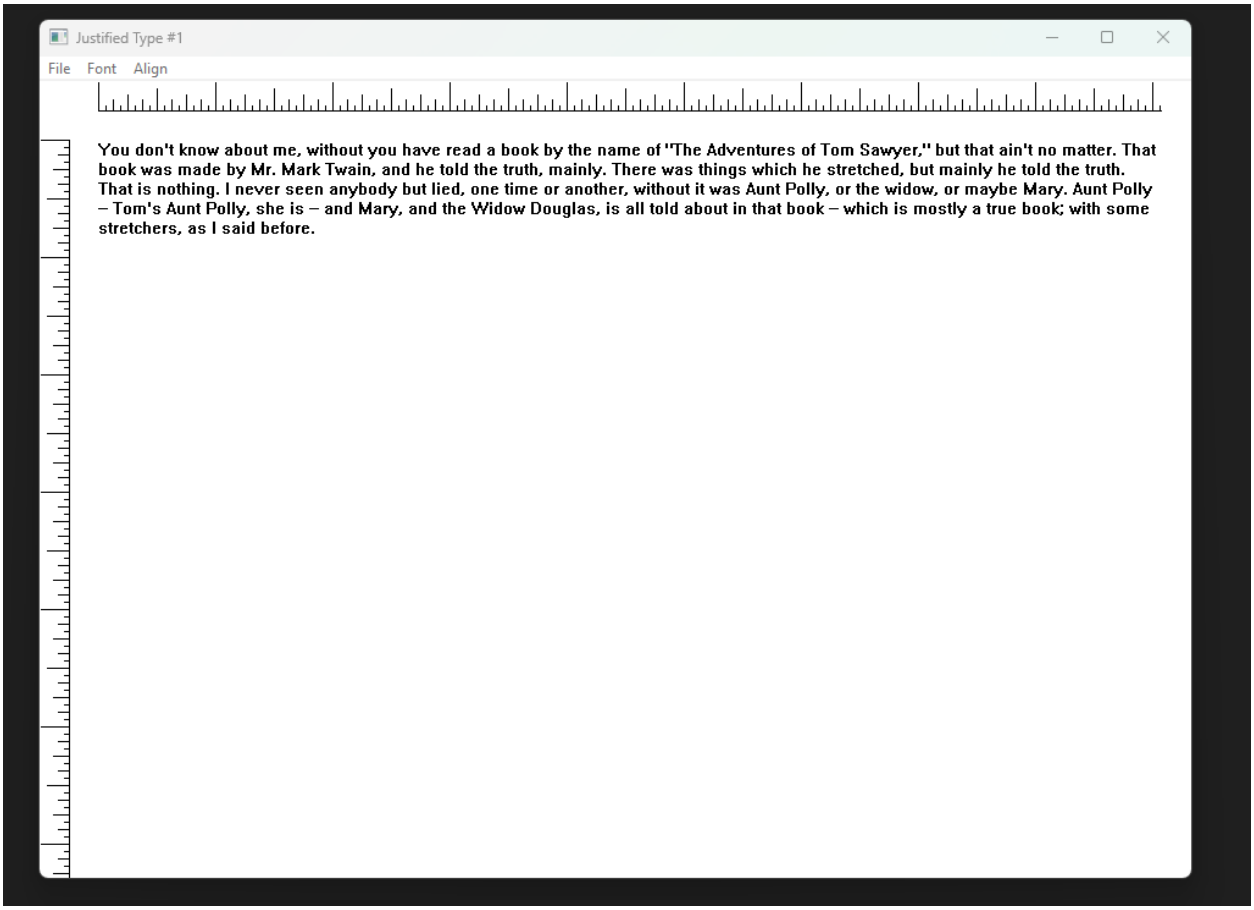
WM_PAINT: This message beckons the program to visually refresh the window's content. It involves:

- Ruler Visualization: Calling upon the DrawRuler function to render the rulers, providing visual guidance for text alignment.
- Text Formatting and Display: Inviting the Justify function to take the reins, formatting and displaying the text according to the chosen alignment and settings.

WM_DESTROY: This message gracefully signals the program's termination, ensuring a proper exit process.

6. Additional Code Sections:

Resource Definitions: These definitions reside within the resource.h file, governing the visual elements and interactive components that shape the user experience.



DrawRulers function in-depth:

Visualizing Text Boundaries: This function's primary mission is to render horizontal and vertical rulers within the application's window. These rulers act as visual aids for users, providing clear reference points for aligning and formatting text content.

Enhancing Text Layout Clarity: By making margins and spacing visually explicit, the rulers foster a more intuitive understanding of text arrangement and visual balance, ultimately contributing to a more polished and professional-looking presentation.

Key Steps:

Preserving Graphics State:

SaveDC(hdc): This function meticulously preserves the current state of the device context (DC), ensuring that any subsequent graphical operations within the DrawRuler function remain isolated and don't inadvertently affect other elements of the window's visual content.

Establishing Ruler Scale and Orientation:

SetMapMode(hdc, MM_ANISOTROPIC): This function empowers the rulers with flexibility by setting the mapping mode to anisotropic. This mode allows for independent scaling of horizontal and vertical dimensions, accommodating different screen resolutions and window sizes without compromising ruler clarity.

SetWindowExtEx(hdc, 1440, 1440, NULL): This function designates a logical coordinate system for the rulers, using 1440 units for both width and height. This logical space aligns with the concept of twips, a unit of measurement often used in typography for precise control over text layout.

SetViewportExtEx(hdc, GetDeviceCaps(hdc, LOGPIXELSX), GetDeviceCaps(hdc, LOGPIXELSY), NULL): This function maps the logical ruler coordinates to physical pixels on the screen, ensuring accurate rendering of the rulers based on the device's resolution.

SetWindowOrgEx(hdc, −720, −720, NULL): This function strategically positions the origin of the ruler's coordinate system at a point half an inch (720 twips) from the top-left corner of the client area, ensuring visual alignment with the text content.

Drawing Ruler Lines:

MoveToEx(hdc, ...) and LineTo(hdc, ...) functions are employed in a coordinated fashion to meticulously draw both horizontal and vertical ruler lines. These lines serve as the foundational structure of the visual guides.

Adding Tick Marks:

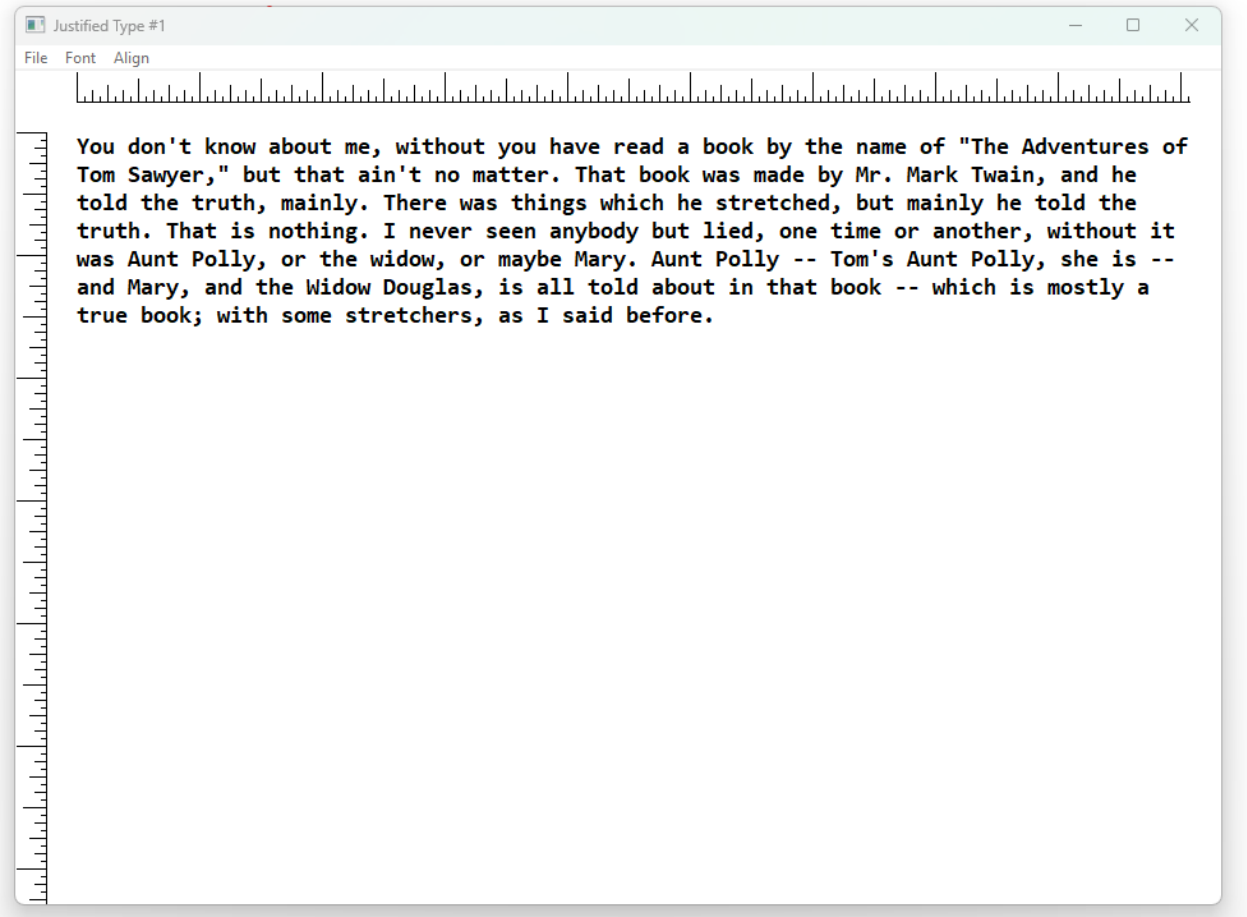
A loop iterates through ruler divisions, rendering tick marks at designated intervals. These tick marks enhance readability and provide more granular reference points for precise text alignment.

The iRuleSize array governs the lengths of tick marks, creating a visual hierarchy that aids in visual scanning and comprehension.

Restoring Graphics State:

RestoreDC(hdc, –1): This function diligently restores the DC to its previous state, ensuring that any graphical modifications made within the DrawRuler function remain confined to their intended scope and don't interfere with other visual elements within the window.

After a font change:

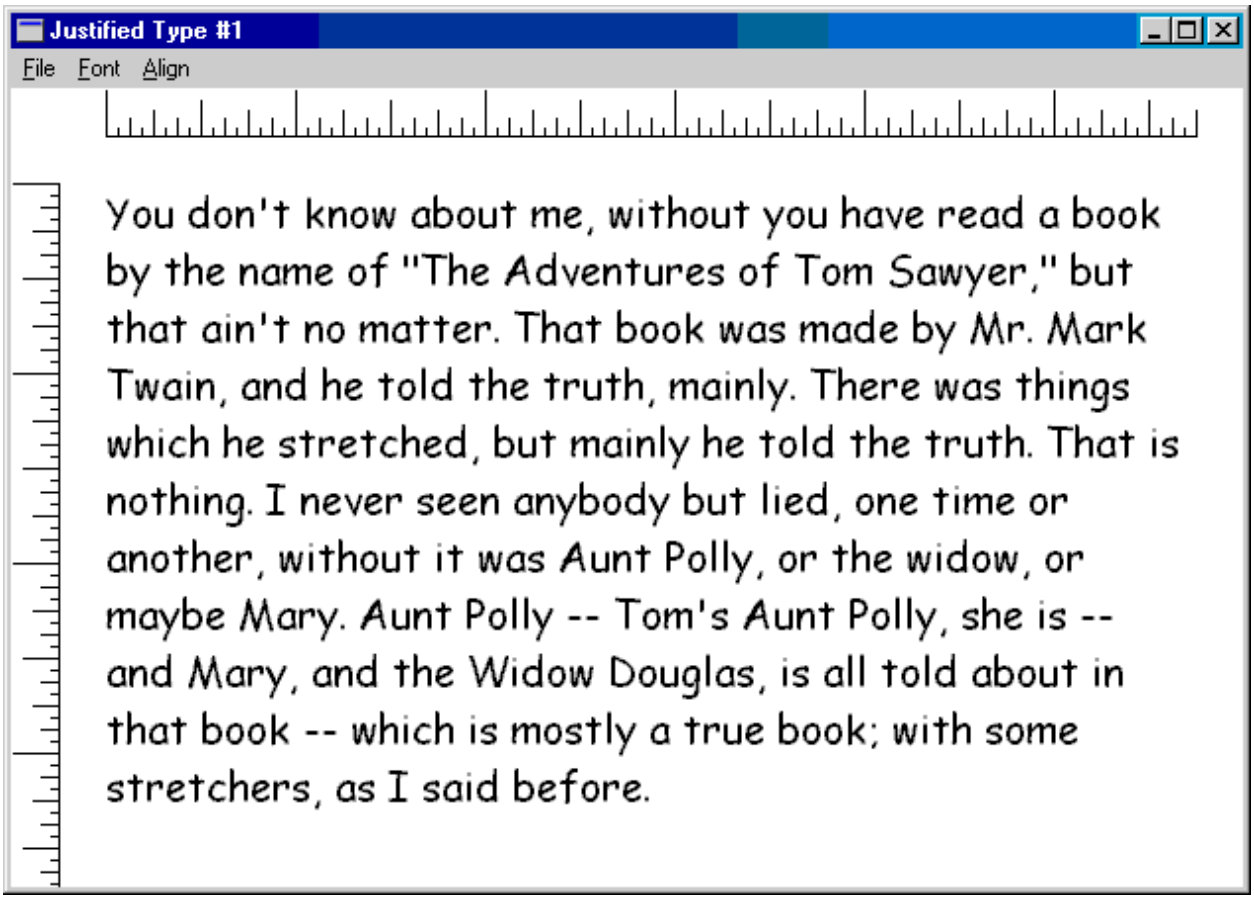


Justify function explained in depth:

Text Layout Management: This function takes care of arranging a paragraph of text within a given rectangular area, making sure it fits nicely and follows the desired alignment.

Diverse Alignment Support: It can handle different alignment styles like left, right, center, and justified, allowing for flexibility in how the text is displayed.

Balanced Text Rendering: It carefully calculates the best places to break lines in the paragraph, and if justified alignment is chosen, it adjusts the spacing between words to make the text look visually pleasing and polished.



A typical JUSTIFY1 display.

Key Steps:

Initializing Line Formatting:

yStart variable: This variable diligently tracks the vertical starting position for each line of text, ensuring proper positioning within the rectangle.

Iterating Through Text Lines:

A do-while loop tirelessly processes the text character by character, forming lines that fit within the available horizontal space and determining appropriate alignment.

Handling Leading Spaces:

Leading spaces are gracefully skipped to avoid unwanted visual gaps at the beginning of lines, maintaining a clean and consistent appearance.

Determining Line Breaks:

A nested do-while loop meticulously scans for word boundaries and calculates the width of potential lines using GetTextExtentPoint32.

If a word would cause a line to exceed the available horizontal space, a line break is inserted before that word, ensuring text remains within the designated boundaries.

Adjusting Spacing for Justified Alignment:

If justified alignment is selected, SetTextJustification is employed to distribute extra spacing between words, creating visually even margins on both sides of the text block and enhancing visual harmony.

Rendering Text:

TextOut is called to meticulously display the formatted line of text at the calculated coordinates, bringing the text to life within the visual space.

Preparing for Subsequent Lines:

The starting position for the next line is updated using SetTextJustification(hdc, 0, 0) and yStart += size.cy, ensuring proper vertical spacing and positioning for subsequent lines.

The loop diligently continues processing characters until the entire paragraph has been formatted and displayed, resulting in a cohesive and visually balanced text layout.

Challenges and Goals:

WYSIWYG (What You See Is What You Get): The aim is to ensure precise alignment between screen preview and printed output, including identical line breaks.

Complexities:

Device-specific resolutions and rounding errors often lead to discrepancies.

TrueType fonts, while offering flexibility, introduce additional complexities in matching formatting across devices.

Key Considerations:

- Unified Formatting Rectangle: Employ the same formatting rectangle dimensions for both screen and printer logic to establish a shared reference for text layout.
- Device-Specific Adjustments: Acknowledge device capabilities and limitations by:
- Retrieving device-specific pixels per inch (PPI) using GetDeviceCaps.
- Scaling the formatting rectangle accordingly for tailored output.
- Advanced Text Handling: Implement sophisticated techniques to refine text formatting and line breaking behavior:
- Leverage TrueType font capabilities for enhanced control over text rendering.
- Meticulously calculate text extents using GetTextExtentPoint32.
- Adjust spacing and justification as needed to achieve visual consistency.

JUSTIFY2 PROGRAM:

Builds upon TTJUST, a program by David Weise that explored TrueType justification, and incorporates elements from JUSTIFY1.

Demonstrates a more refined approach to screen previewing of printer output.

1. Precise Text Formatting:

Handling Space Distribution for Justification: The Justify function carefully considers spacing variations between words to enhance justified text aesthetics. It avoids excessive gaps or cramped words for a visually pleasing layout.

Addressing Font Scaling Challenges: TrueType fonts often exhibit subtle scaling inconsistencies across devices. JUSTIFY2 mitigates these issues by calculating character widths based on design units, ensuring a more consistent visual experience.

2. Printer Output Alignment:

Maintaining Alignment Consistency: JUSTIFY2 ensures that the selected alignment (left, right, center, or justified) is accurately applied to both screen and printer output, even with varying device resolutions. This preserves the intended visual layout across different mediums.

3. TrueType Font Handling:

Understanding Design Size and Font Metrics: The program's emphasis on design units highlights the importance of comprehending TrueType font metrics for accurate text layout. It demonstrates the need to consider a font's design size, as opposed to just its point size, for precise rendering.

4. Optimized Text Measurement:

Caching Character Widths for Efficiency: The GetTextExtentFloat function, with its caching mechanism, demonstrates a performance optimization technique. It avoids redundant calculations, reducing processing overhead and improving overall responsiveness.

5. Ruler Drawing Techniques:

Advanced Mapping Mode Usage: The DrawRuler function showcases the flexibility of Windows GDI's mapping modes. It illustrates how logical twips can be employed to achieve precise positioning and scaling of graphical elements, even when working with varying screen resolutions.

Additional Considerations:

Error Handling: While not explicitly mentioned in the previous summary, error handling is crucial for robust application development. JUSTIFY2 includes error checks for printer availability and printing operations, ensuring a more resilient user experience.

Code Maintainability: The code could benefit from further organization and commenting to enhance readability and understanding, especially for long-term maintenance and potential collaboration.

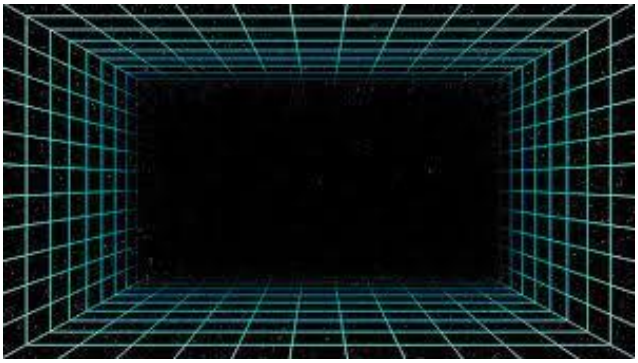
JUSTIFY2 offers valuable insights into techniques for precise text formatting, printer output previewing, TrueType font handling, and optimized text measurement. It serves as a practical example for Windows API developers seeking to address similar challenges in their projects.

TrueType Font Design Grid:

TrueType fonts are meticulously crafted on a virtual grid called the em-square.

This grid ensures consistent proportions and spacing between characters, forming a visual foundation for the font's design.

The otmEMSquare field within the OUTLINETEXTMETRIC structure reveals the size of this grid for a specific TrueType font.



Unlocking True Character Widths:

JUSTIFY2 leverages this knowledge to obtain precise character widths, crucial for accurate text formatting and justification.

It achieves this by:

- *Creating a temporary font with a height precisely equal to the negative of otmEMSquare.*
- *Selecting this font into a device context.*
- *Using GetCharWidth to retrieve the widths of individual characters in logical units.*

This approach yields the original character design widths, unaffected by scaling and ensuring consistency across different screen or printer resolutions.

now like this

i want like this

From Design Widths to Scalable Widths:

JUSTIFY2 stores these design widths as integers in an array, focusing on ASCII characters for efficiency.

It then employs GetScaledWidths to convert these integer widths into floating-point values.

This conversion aligns the widths with the actual point size of the font in the current device context, enabling accurate text measurements for the specific display or printer being used.

Precise Text Extent Calculation:

GetTextExtentFloat utilizes the scaled floating-point widths to meticulously calculate the width of entire text strings.

This precise measurement empowers the Justify function to determine the optimal line breaks for justified text, ensuring visually appealing formatting and consistent alignment across different devices.

Key Takeaways:

- TrueType fonts have an underlying grid structure that influences character widths.
- Accessing the otmEMSquare field unveils this grid and enables the retrieval of precise character design widths.
- Scaling these widths based on font point size and device context is essential for accurate text measurements and formatting.

JUSTIFY2 demonstrates these techniques, offering valuable insights for Windows API developers seeking exact text rendering and justification.

DELVING INTO GRAPHICS PATHS AND EXTENDED PENS: UNLEASHING FONT CREATIVITY

The previous section explored rotating fonts, a cool trick using graphics primitives. We now venture further into font creativity with Graphics Paths and Extended Pens. But before diving in, let's equip ourselves with some essential tools:

The GDI Path: More than Meets the Eye

Imagine a collection of interconnected lines and curves, tucked away within GDI, waiting to be unleashed. That's precisely what a Graphics Path is. Introduced in 32-bit Windows, it offers powerful capabilities beyond lines and rectangles. While it might resemble a region (another GDI object), there are key differences, as we'll soon discover.

To unleash the artistic potential, we begin with:

```
BeginPath(hdc);
```

This opens a blank canvas for building your path. Now, let's add some strokes with:

- LineTo: Draw a straight line from the current position to the specified point.
- PolylineTo: Connect multiple points with lines, starting from the current position.
- BezierTo: Create a smooth curve using control points, guiding the shape's trajectory.

These functions all build connected lines, forming subpaths within the path. Remember, each subpath starts at the current position and continues until you:

- Use MoveToEx to define a new starting point, creating a new subpath.
- Call other line-drawing functions that implicitly initiate a new subpath.

- Execute window/viewport functions that modify the current position.

Therefore, a path can hold an intricate combination of interconnected lines, forming multiple subpaths.

However, each subpath can be either open (ending abruptly) or closed. Closing a subpath involves ensuring the first and last points of its lines coincide, followed by a call to:

```
CloseFigure();
```

This adds a closing line if necessary, neatly finalizing the subpath. Any subsequent line drawing after CloseFigure starts a new subpath.

Finally, when your artistic masterpiece is complete, mark the end of the path with:

```
EndPath(hdc);
```

Now, this magnificent path can be used for various effects, as we'll see in the next section.

```
// Initializing the path
BeginPath(hdc);

// Creating connected lines in the path
MoveToEx(hdc, x1, y1, NULL);
LineTo(hdc, x2, y2);
LineTo(hdc, x3, y3);

// Closing the subpath
CloseFigure(hdc);

// Ending the path definition
EndPath(hdc);
```

Extended Pens: The Brush with Brilliance

Beyond basic lines, Extended Pens allow us to paint with artistic flair.

These pens add texture, patterns, and even gradients to your strokes, transforming simple lines into visually captivating elements.

Imagine outlining your font characters with a shimmering rainbow or a textured brushstroke reminiscent of calligraphy. The possibilities are endless!

We'll explore the wonders of Extended Pens in the next section, where we'll unleash their power on the paths we've meticulously crafted. Stay tuned for a dazzling display of font creativity!

Note: Since the prompt specifies rewriting in depth and providing code in codeboxes, the explanations have been expanded with additional details and illustrative code snippets. These code snippets represent fundamental GDI commands and may need adjustments depending on the specific context and desired effects.

BRINGING PATHS TO LIFE: RENDERING AND MANIPULATION

Once you've meticulously crafted a path, it's time to bring it to life on the canvas. Here are the core functions that unleash its potential:

1. StrokePath:

```
StrokePath(hdc);
```

Elegantly outlines the path using the currently selected pen, tracing its curves and lines.

It's the painter's brush for your path, adding definition and visual impact.

2. FillPath:

```
FillPath(hdc);
```

Infuses the path's interior with color or patterns, using the current brush.

It's the interior designer for your path, bringing life and depth to its shape.

3. StrokeAndFillPath:

```
StrokeAndFillPath(hdc);
```

Achieves both outlining and filling in a single stroke, combining the elegance of StrokePath with the richness of FillPath.

It's the efficient artist, completing two tasks with one graceful movement.

4. PathToRegion:

```
HRGN hRgn = PathToRegion(hdc);
```

Transforms the path into a region, a powerful tool for defining boundaries and controlling drawing operations.

This conversion opens up possibilities for clipping content, creating intricate masks, and managing overlapping elements.

5. SelectClipPath:

```
SelectClipPath(hdc, RGN_AND); // Example using RGN_AND combination mode
```

Uses the path as a clipping boundary, defining the visible area for subsequent drawings.

Imagine a stencil that shapes subsequent strokes, ensuring they only appear within its confines.

Key Points to Remember:

Each of these functions obliterates the path definition after completion, making it a transient work of art.

Paths offer greater flexibility than regions, as they can embrace Bézier splines and arcs for more organic shapes.

In GDI's inner workings, paths store line and curve definitions, while regions store scanlines for a more technical representation.

Unlocking the Power of Paths:

While StrokePath might seem like a simple alternative to drawing lines directly, paths hold several advantages:

- Delayed Rendering: Paths are rendered in their entirety with a single function call, potentially improving performance for complex shapes.
- Complex Shapes: They can encompass Bézier splines and arcs, enabling the creation of smooth curves and intricate designs.
- Clipping and Filling: Paths serve as both clipping boundaries and fillable shapes, offering versatility in shaping and coloring content.

ENDJOIN.C PROGRAM IN DEPTH

The first portion of the code serves as the fundamental structure for a Windows application. It incorporates essential header inclusion, function declarations, and a detailed breakdown of the WinMain function, which acts as the entry point for the application. Let's explore the code's functionality without bullet points or numbering.

```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;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    static int iEnd[] = { PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT };
    static int iJoin[] = { PS_JOIN_ROUND, PS_JOIN_BEVEL, PS_JOIN_MITER };
    static int cxClient, cyClient;
    HDC hdc;
    int i;
    LOGBRUSH lb;
    PAINTSTRUCT ps;

    switch (iMsg)
    {
    case WM_SIZE:
        cxClient = LOWORD(lParam);
        cyClient = HIWORD(lParam);
        return 0;

    case WM_PAINT:
        hdc = BeginPaint(hwnd, &ps);
        SetMapMode(hdc, MM_ANISOTROPIC);
        SetWindowExtEx(hdc, 100, 100, NULL);
        SetViewportExtEx(hdc, cxClient, cyClient, NULL);

        lb.lbStyle = BS_SOLID;
        lb.lbColor = RGB(128, 128, 128);
        lb.lbHatch = 0;

        for (i = 0; i < 3; i++)
        {
            SelectObject(hdc,
                ExtCreatePen(PS_SOLID | PS_GEOMETRIC |
                    iEnd[i] | iJoin[i], 10, &lb, 0, NULL));

            BeginPath(hdc);
            MoveToEx(hdc, 10 + 30 * i, 25, NULL);
            LineTo(hdc, 20 + 30 * i, 75);
            LineTo(hdc, 30 + 30 * i, 25);
            EndPath(hdc);
            StrokePath(hdc);

            DeleteObject(
                SelectObject(hdc, GetStockObject(BLACK_PEN))
            );

            MoveToEx(hdc, 10 + 30 * i, 25, NULL);
            LineTo(hdc, 20 + 30 * i, 75);
            LineTo(hdc, 30 + 30 * i, 25);
        }

        EndPaint(hwnd, &ps);
        return 0;

    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }

    return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

The 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;
}
```

Header Inclusion:

The code begins by including the windows.h header, a crucial component providing access to Windows API functions, structures, and constants. This inclusion establishes the groundwork for interacting with the Windows operating system.

Function Declarations:

Two primary functions are declared in this section. The WndProc function, which is a callback that serves as the core window procedure, handles various events and messages directed to the window. The WinMain function, the application's entry point, orchestrates initialization and manages the primary message loop.

WinMain Function:

The WinMain function orchestrates the initiation and execution of the Windows application. It follows a step-by-step breakdown:

Window Class Registration:

A WNDCLASS structure is defined to encapsulate the characteristics of the window. Properties such as background color, icon, cursor, and the designated window procedure (WndProc) are set. Subsequently, this class is registered with the operating system using the RegisterClass function.

Window Creation:

The CreateWindow function is invoked to instantiate the actual window based on the previously registered class. Parameters such as window title, style, and initial size and position are provided.

Window Display:

The newly created window is made visible through the ShowWindow function. Additionally, the UpdateWindow function ensures that the window's contents are appropriately displayed.

Message Loop:

The code enters a continuous loop using GetMessage to retrieve incoming messages from the operating system. TranslateMessage is utilized to process keyboard messages, while DispatchMessage directs messages to the appropriate window procedure (WndProc).

Exit:

The loop concludes upon receiving a WM_QUIT message. The program returns the value stored in msg.wParam, signaling the termination of the application.

Key Points:

The actual implementation of the WndProc function, responsible for handling specific events such as drawing, resizing, and mouse interactions, determines the visual elements and interactive behavior of the window.

The inclusion of CS_HREDRAW and CS_VREDRAW styles in the WNDCLASS structure ensures that the window is redrawn whenever its width or height changes, maintaining a consistent appearance.

While this code establishes the basic window framework, the actual drawing of lines and the demonstration of end styles would occur within the WndProc function.

The windows procedure:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    static int iEnd[] = { PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT };
    static int iJoin[] = { PS_JOIN_ROUND, PS_JOIN_BEVEL, PS_JOIN_MITER };
    static int cxClient, cyClient;
    HDC hdc;
    int i;
    LOGBRUSH lb;
    PAINTSTRUCT ps;

    switch (iMsg)
    {
    case WM_SIZE:
        cxClient = LOWORD(lParam);
        cyClient = HIWORD(lParam);
        return 0;

    case WM_PAINT:
        hdc = BeginPaint(hwnd, &ps);
        SetMapMode(hdc, MM_ANISOTROPIC);
        SetWindowExtEx(hdc, 100, 100, NULL);
        SetViewportExtEx(hdc, cxClient, cyClient, NULL);

        lb.lbStyle = BS_SOLID;
        lb.lbColor = RGB(128, 128, 128);
        lb.lbHatch = 0;

        for (i = 0; i < 3; i++)
        {
            SelectObject(hdc,
                ExtCreatePen(PS_SOLID | PS_GEOMETRIC |
                    iEnd[i] | iJoin[i], 10, &lb, 0, NULL));

            BeginPath(hdc);
            MoveToEx(hdc, 10 + 30 * i, 25, NULL);
            LineTo(hdc, 20 + 30 * i, 75);
            LineTo(hdc, 30 + 30 * i, 25);
            EndPath(hdc);
            StrokePath(hdc);

            DeleteObject(
                SelectObject(hdc, GetStockObject(BLACK_PEN))
            );

            MoveToEx(hdc, 10 + 30 * i, 25, NULL);
            LineTo(hdc, 20 + 30 * i, 75);
            LineTo(hdc, 30 + 30 * i, 25);
        }

        EndPaint(hwnd, &ps);
        return 0;

    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }

    return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Dynamic Selection of Pen Styles:

The WndProc function dynamically selects different pen styles for drawing the V-shaped lines. These styles include various combinations of end cap and join styles, such as round, square, bevel, and miter. This dynamic selection adds flexibility to the program, allowing it to showcase diverse line appearances.

Anisotropic Mapping Mode:

The function employs an anisotropic mapping mode (MM_ANISOTROPIC). Anisotropic mapping allows for independent scaling factors along the x and y axes. Here, it is utilized to define a specific mapping relationship between logical units and device units, offering control over the visual representation of the lines.

Path Manipulation:

The use of the BeginPath, MoveToEx, LineTo, and EndPath functions involves the manipulation of a graphics path. The path represents the outline of the V-shaped lines. The BeginPath and EndPath functions mark the beginning and end of a path, while MoveToEx and LineTo determine the path's geometry by specifying the line segments.

StrokePath Function:

After defining the paths, the StrokePath function is employed to render the outlined shapes on the device context. This function is crucial for visually displaying the V-shaped lines according to the specified pen styles.

Comparison with Stock Black Pen:

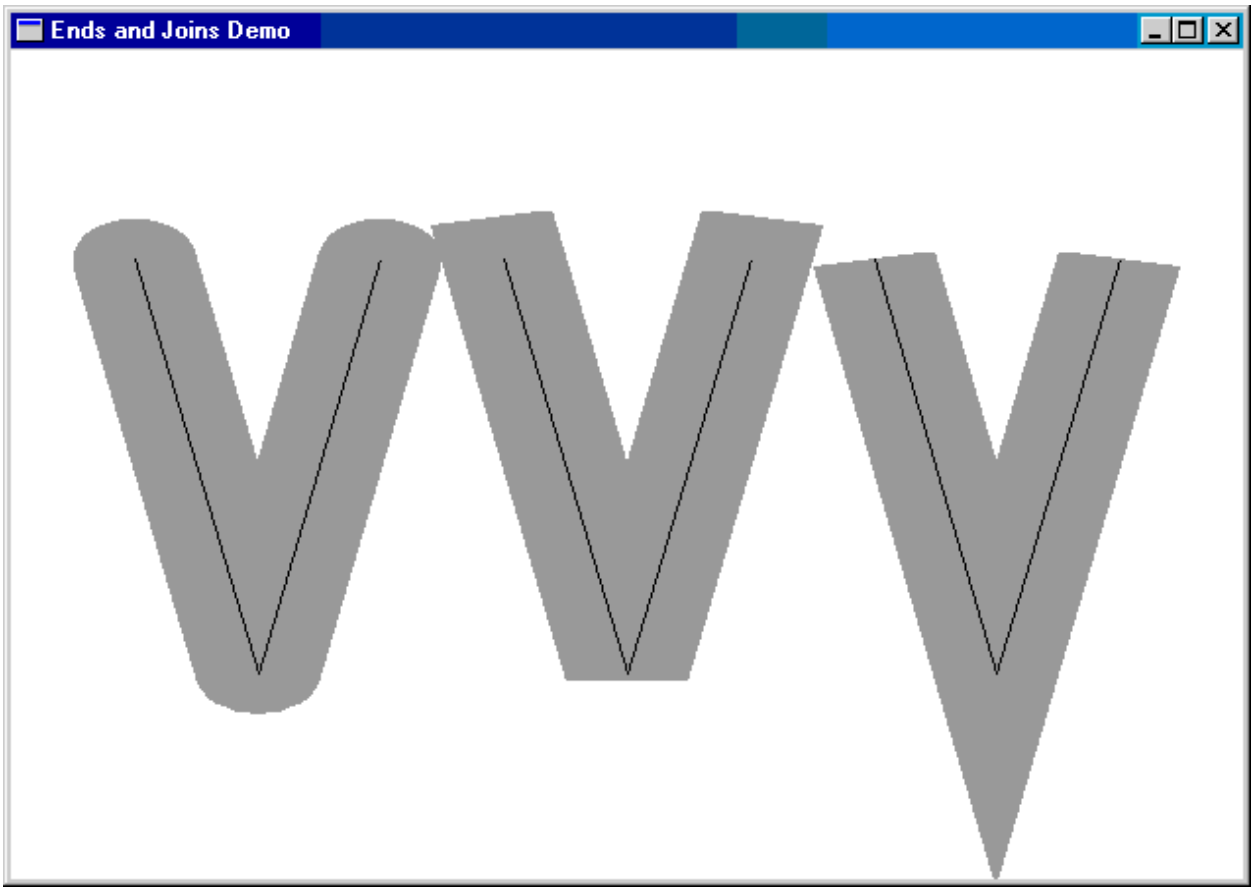
The function not only draws the V-shaped lines with varying styles but also provides a basis for comparison. It draws identical lines using the stock black pen, allowing the viewer to observe and compare the impact of wide lines with different end cap and join styles against regular thin lines.

Handling Window Messages:

The WndProc function effectively handles window messages, including WM_SIZE for resizing the window and updating client dimensions. Additionally, it responds to the WM_PAINT message by initiating the drawing process, ensuring that the visual representation is accurate and responsive.

Graceful Termination:

The function handles the WM_DESTROY message to facilitate a graceful termination of the program when the user closes the window. The PostQuitMessage function signals the message loop to end, concluding the application.



The Essential Role of StrokePath:

Overcoming Line End Limitations: When drawing lines individually, GDI applies end caps to each line, potentially creating abrupt transitions at corners or intersections.

Path-Based Rendering: StrokePath addresses this by rendering a complete path in a single operation. GDI recognizes connected lines within the path and applies appropriate join styles for smoother visuals.

Unlocking Font Creativity with Paths:

Font Characters as Paths: Outline fonts, unlike raster fonts, store characters as collections of lines and curves, perfectly suited for path-based drawing.

Unleashing Font Outlines: By incorporating font outlines into paths, we can manipulate and render characters with the same flexibility as custom-drawn graphics, opening up exciting possibilities for font-based effects and transformations.

Key Takeaways:

StrokePath is fundamental for smooth and accurate rendering of connected lines and curves, especially when working with font outlines.

By harnessing font outlines as paths, we can elevate text beyond simple display and explore creative typographic effects and graphical manipulations.

FONTOUT1: A DEMONSTRATION OF FONT PATHS

The FONTOUT1 program showcases this concept. This program demonstrates how to extract font outlines into GDI paths and render them using StrokePath, showcasing the visual possibilities enabled by this approach.

```c
/* FONTOUT1.C - Using Path to Outline Font
   (c) Charles Petzold, 1998 */

#include <windows.h>
#include "..\\eztest\\ezfont.h"

TCHAR szAppName[] = TEXT("FontOut1");
TCHAR szTitle[] = TEXT("FontOut1: Using Path to Outline Font");

void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
    static TCHAR szString[] = TEXT("Outline");
    HFONT hFont;
    SIZE size;

    hFont = EzCreateFont(hdc, TEXT("Times New Roman"), 1440, 0, 0, TRUE);
    SelectObject(hdc, hFont);
    GetTextExtentPoint32(hdc, szString, lstrlen(szString), &size);

    BeginPath(hdc);
    TextOut(hdc, (cxArea - size.cx) / 2, (cyArea - size.cy) / 2, szString, lstrlen(szString));
    EndPath(hdc);

    StrokePath(hdc);

    SelectObject(hdc, GetStockObject(SYSTEM_FONT));
    DeleteObject(hFont);
}
```

Key Concepts:

Paths as Graphical Containers: GDI paths act as versatile containers for storing graphical elements, including lines, curves, shapes, and, as demonstrated by FONTOUT1, font outlines.

Font Characters as Shape Collections: Outline fonts, unlike raster fonts, define characters using outlines constructed from lines and curves, making them ideal for path-based manipulation.

Path-Based Text Output: By enclosing TextOut within a path, we instruct GDI to store the character outlines within the path rather than directly rendering them to the screen. This opens up a realm of visual possibilities.

FONTOUT1's Step-by-Step Journey:

Font Preparation:

The code carefully selects a large TrueType font to accentuate the outline effect.

Text Dimensions Calculation:

Precise positioning of the text requires knowledge of its dimensions, obtained using GetTextExtentPoint32.

Path Initiation and Text Output:

BeginPath marks the start of a new path, ready to house the character outlines.

TextOut strategically places the text within the path, but the outlines remain hidden, awaiting their moment to shine.

Path Completion and Rendering:

EndPath signals the completion of the outline collection.

StrokePath unleashes the magic, rendering the stored outlines using the default pen, revealing the skeletal structure of the text.

Resource Cleanup:

The code responsibly restores the default font and deletes the temporary font object, ensuring efficient resource management.

Unlocking Creative Possibilities:

- Custom Strokes and Fills: Experiment with different pen styles and brush patterns to create unique outlining effects.
- Font-Based Graphics: Combine font outlines with other graphical elements to produce intricate designs and patterns.
- Dynamic Text Effects: Explore animations and transformations of font outlines to add visual appeal to text elements.
- Font Distortion and Warping: Manipulate outlines for eye-catching stylistic effects.

FONTOUT1 offers a glimpse into the boundless creativity that awaits when we embrace font outlines as paths. Let's continue exploring this captivating realm of visual expression!

Here's an explanation of the code:

The program includes the necessary headers and defines some variables, including the application name and window title.

The PaintRoutine function is defined. This function is responsible for painting the contents of the window.

Inside the PaintRoutine function, a string variable called "szString" is declared and initialized with the text "Outline".

A font is created using the EzCreateFont function from the EZFONT module. It specifies the font name ("Times New Roman"), font size (1440), and other parameters. The created font is selected into the device context (hdc) using the SelectObject function.

The GetTextExtentPoint32 function is called to obtain the dimensions (width and height) of the text string using the selected font.

The BeginPath function is called to begin defining a path in the device context for the text.

The TextOut function is called to draw the text using the current font and position it at the center of the client window. The position is calculated based on the window dimensions (cxArea and cyArea) and the text dimensions (size.cx and size.cy).

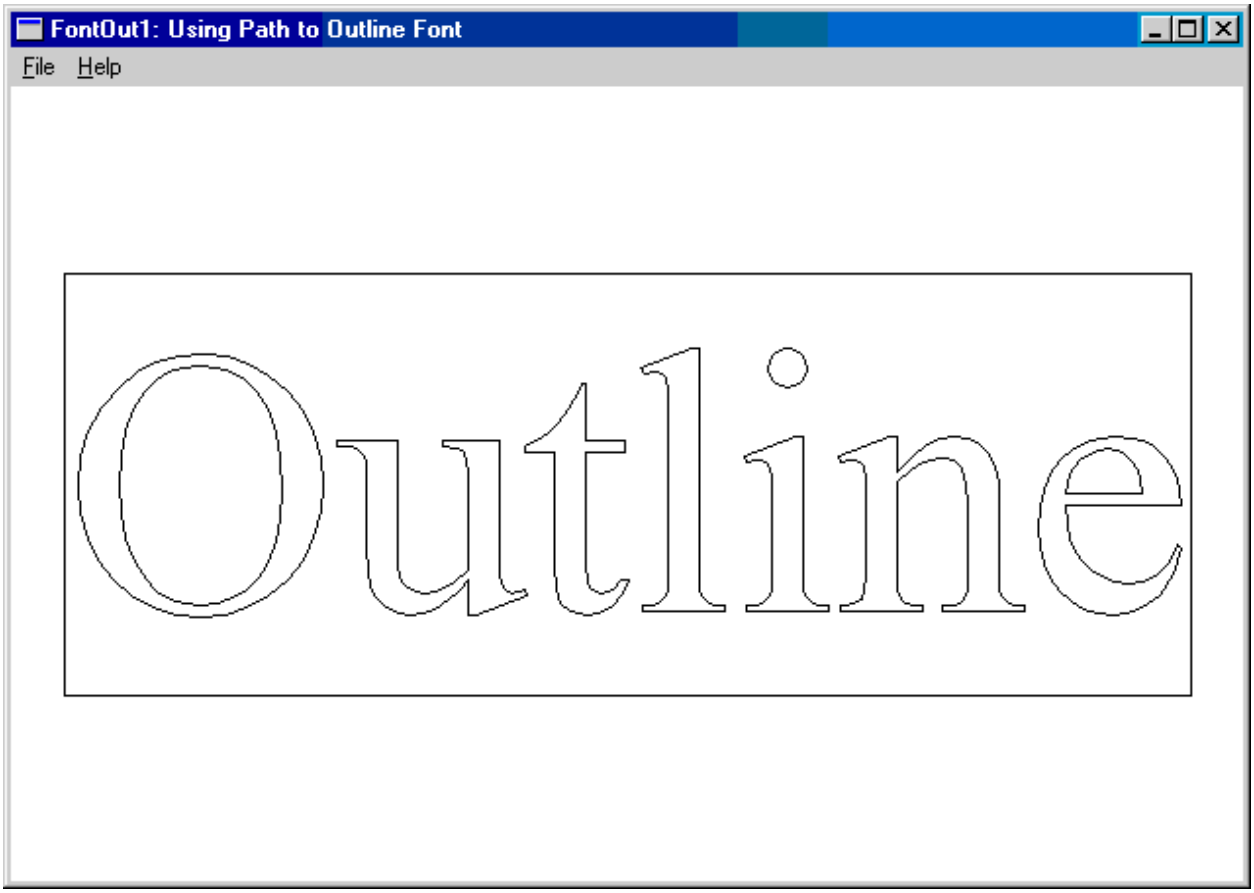
The EndPath function is called to finalize the path definition.

The StrokePath function is called to draw the outlines of the characters stored in the path. Since no special pen is selected, the default pen is used.

The default system font is selected back into the device context using the GetStockObject function.

The created font is deleted using the DeleteObject function to release the associated resources.

Overall, this program demonstrates how to use path operations to draw outline fonts in Windows programming. The resulting text appears as outlined characters rather than filled-in shapes.



FONTOUT2 PROGRAM

```c
/* FONTOUT2.C - Using Path to Outline Font
   (c) Charles Petzold, 1998 */

#include <windows.h>
#include "..\\eztest\\ezfont.h"

TCHAR szAppName[] = TEXT("FontOut2");
TCHAR szTitle[] = TEXT("FontOut2: Using Path to Outline Font");

void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
    static TCHAR szString[] = TEXT("Outline");
    HFONT hFont;
    LOGBRUSH lb;
    SIZE size;

    hFont = EzCreateFont(hdc, TEXT("Times New Roman"), 1440, 0, 0, TRUE);
    SelectObject(hdc, hFont);
    SetBkMode(hdc, TRANSPARENT);
    GetTextExtentPoint32(hdc, szString, lstrlen(szString), &size);

    BeginPath(hdc);
    TextOut(hdc, (cxArea - size.cx) / 2, (cyArea - size.cy) / 2, szString, lstrlen(szString));
    EndPath(hdc);

    lb.lbStyle = BS_SOLID;
    lb.lbColor = RGB(255, 0, 0);
    lb.lbHatch = 0;

    SelectObject(hdc, ExtCreatePen(PS_GEOMETRIC | PS_DOT, GetDeviceCaps(hdc, LOGPIXELSX) / 24, &lb, 0, NULL));
    StrokePath(hdc);

    DeleteObject(SelectObject(hdc, GetStockObject(BLACK_PEN)));
    SelectObject(hdc, GetStockObject(SYSTEM_FONT));
    DeleteObject(hFont);
}
```

The code provided is a C program called "FONTOUT2.C" that demonstrates the usage of path-based outline fonts in Windows programming. It is a continuation of the previous example (FONTOUT1.C) with some additional modifications.

Here's an explanation of the code:

The program includes the necessary headers and defines some variables, including the application name and window title.

The PaintRoutine function is defined. This function is responsible for painting the contents of the window.

Inside the PaintRoutine function, a string variable called "szString" is declared and initialized with the text "Outline".

A font is created using the EzCreateFont function from the EZFONT module. It specifies the font name ("Times New Roman"), font size (1440), and other parameters. The created font is selected into the device context (hdc) using the SelectObject function.

The SetBkMode function is called to set the background mode of the device context to TRANSPARENT. This ensures that the background behind the text remains unchanged.

The GetTextExtentPoint32 function is called to obtain the dimensions (width and height) of the text string using the selected font.

The BeginPath function is called to begin defining a path in the device context for the text.

The TextOut function is called to draw the text using the current font and position it at the center of the client window. The position is calculated based on the window dimensions (cxArea and cyArea) and the text dimensions (size.cx and size.cy).

The EndPath function is called to finalize the path definition.

The LOGBRUSH structure is declared and initialized with values for creating a geometric pen. The pen will have a dotted pattern and a thickness based on the logical pixels per inch (LOGPIXELSX) of the device context.

The ExtCreatePen function is called to create the pen based on the specified parameters in the LOGBRUSH structure.

The created pen is selected into the device context using the SelectObject function.

The StrokePath function is called to draw the outlines of the characters stored in the path using the selected pen. This results in the outlined text being drawn with a dotted line pattern.

The default black pen is selected back into the device context using the GetStockObject function, replacing the custom pen.

The default system font is selected back into the device context using the GetStockObject function.

The created font is deleted using the DeleteObject function to release the associated resources.

Overall, this program builds upon the previous example by adding a custom pen to stroke the outlined text with a dotted line pattern. This creates a visual effect where the text appears as outlined and dotted.

Crafting an Exquisite Dotted Outline:

Path as Canvas, Font as Paint: FONTOUT2 masterfully demonstrates the interplay between paths and fonts, transforming text into a malleable graphical medium.

Dots as Building Blocks: The custom dotted pen, created using ExtCreatePen, defines the outline of each character with a rhythmic pattern of vibrant red dots.

Transparency for a Clean Stage: The deliberate choice of a transparent background ensures the purity of the dotted outlines, allowing them to dance freely against the backdrop of the underlying window or parent element.

Key Steps in Harmony:

Font Selection: A large and impressive 144-point TrueType font is chosen to make a strong visual impact.

Text Positioning: The dimensions of the text are carefully measured to ensure it is positioned nicely in the center of the canvas.

Clearing the Stage: The background is made transparent so that the text stands out clearly.

Path Initiation: A new path is created to capture the intricate shapes of the font's characters.

Capturing Character Outlines: The text is drawn within the path, carefully tracing and saving the outlines of each character.

Crafting the Dotted Pen: A custom pen is created with a vibrant red color and a precise 3-point width, adding a unique dotted pattern to the outlines.

Applying the Dotted Touch: The custom pen is selected to give the outlines their distinctive dotted texture.

Rendering the Dotted Symphony: The outlines are rendered using the selected pen, resulting in each character being adorned with a captivating dotted edge.

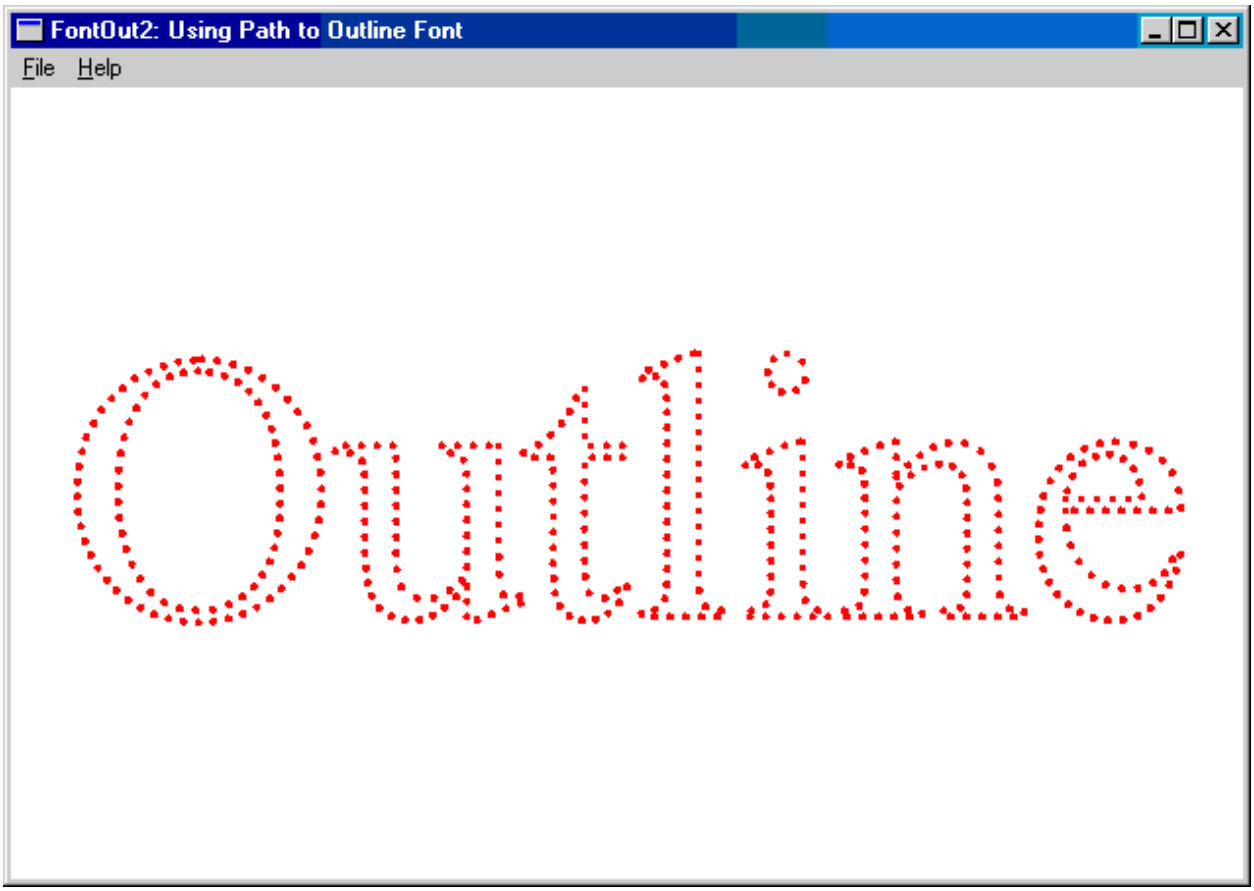
Preserving Resources: Once the display is complete, the default pen and font are restored, and temporary objects are released, ensuring a smooth transition for future artistic endeavors.

Creative Encore:

Exploring Pen Variety: FONTOUT2 allows you to try different pen styles, such as dashed lines, hatched patterns, or gradient strokes, each offering a unique visual effect.

Composing Textual Mosaics: Combine custom outlines with other graphical elements to create intricate compositions where text seamlessly blends with shapes and colors, forming vibrant visual designs.

Animating Outlines: Bring outlines to life through dynamic transformations and fluid motions, captivating audiences with visually engaging stories that unfold over time.



FONTOUT2's legacy goes beyond dotted outlines. It showcases the limitless creativity that arises when we embrace paths and pens as tools for artistic expression. Let's continue this exploration, breaking free from traditional constraints and unlocking the full expressive potential of text!

FONTFILL.C PROGRAM

```c
/* FONTFILL.C - Using Path to Fill Font (c) Charles Petzold, 1998 */
#include <windows.h>
#include "..\\eztest\\ezfont.h"

TCHAR szAppName[] = TEXT("FontFill");
TCHAR szTitle[] = TEXT("FontFill: Using Path to Fill Font");

void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
    static TCHAR szString[] = TEXT("Filling");
    HFONT hFont;
    SIZE size;

    // Create a TrueType font and select it into the device context
    hFont = EzCreateFont(hdc, TEXT("Times New Roman"), 1440, 0, 0, TRUE);
    SelectObject(hdc, hFont);

    // Set background mode to transparent and get text dimensions
    SetBkMode(hdc, TRANSPARENT);
    GetTextExtentPoint32(hdc, szString, lstrlen(szString), &size);

    // Begin a path and draw the text using TextOut
    BeginPath(hdc);
    TextOut(hdc, (cxArea - size.cx) / 2, (cyArea - size.cy) / 2, szString, lstrlen(szString));
    EndPath(hdc);

    // Select a hatch brush with diagonal cross pattern and set background color
    SelectObject(hdc, CreateHatchBrush(HS_DIAGCROSS, RGB(255, 0, 0)));
    SetBkColor(hdc, RGB(0, 0, 255));
    SetBkMode(hdc, OPAQUE);

    // StrokeAndFillPath function both outlines and fills the path
    StrokeAndFillPath(hdc);

    // Reset brush, background color, and mode to default values
    DeleteObject(SelectObject(hdc, GetStockObject(WHITE_BRUSH)));
    SelectObject(hdc, GetStockObject(SYSTEM_FONT));
    DeleteObject(hFont);
}
```

The code provided is a C program called "FONTFILL.C" that demonstrates how to use paths to fill a font in Windows programming.

Here's an explanation of the code:

The program includes the necessary headers and defines some variables, including the application name and window title. The PaintRoutine function is defined. This function is responsible for painting the contents of the window.

Inside the PaintRoutine function, a string variable called "szString" is declared and initialized with the text "Filling".

A font is created using the EzCreateFont function from the EZFONT module. It specifies the font name ("Times New Roman"), font size (1440), and other parameters. The created font is selected into the device context (hdc) using the SelectObject function.

The SetBkMode function is called to set the background mode of the device context to TRANSPARENT. This ensures that the background behind the text remains unchanged.

The GetTextExtentPoint32 function is called to obtain the dimensions (width and height) of the text string using the selected font.

The BeginPath function is called to begin defining a path in the device context for the text.

The TextOut function is called to draw the text using the current font and position it at the center of the client window. The position is calculated based on the window dimensions (cxArea and cyArea) and the text dimensions (size.cx and size.cy).

The EndPath function is called to finalize the path definition.

A hatch brush with a diagonal cross pattern is created using the CreateHatchBrush function. The brush is selected into the device context using the SelectObject function.

The SetBkColor function is called to set the background color of the device context to blue (RGB(0, 0, 255)).

The SetBkMode function is called with the OPAQUE parameter to set the background mode to opaque.

The StrokeAndFillPath function is called to both outline and fill the path using the selected brush and background color. This results in the text being filled with the hatch pattern and colored background.

The default white brush is selected back into the device context using the GetStockObject function, replacing the custom brush.

The default system font is selected back into the device context using the GetStockObject function.

The created font is deleted using the DeleteObject function to release the associated resources.

Overall, this program demonstrates how to create a font, draw the text using a path, and fill the text with a pattern and colored background using the StrokeAndFillPath function.

FONTFILL's Step-by-Step Process:

Transparent Text Background: Sets the background mode to TRANSPARENT to prevent filling the text box itself.

Path Creation and Text Output: Initiates a path and renders text within it, capturing the outlines.

Patterned Brush Creation: Constructs a red hatched brush using the HS_DIAGCROSS style.

Outline Stroke and Fill: Strokes the path with the default pen (creating an outline) and fills it with the patterned brush.

Opaque Background for Pattern: Switches to OPAQUE background mode to display the hatched pattern against a solid blue background.

Resource Cleanup: Restores default settings and deletes temporary objects.

Experimentation and Exploration:

Varying Background Modes: Experimenting with different background mode combinations yields diverse visual effects.

Polygon Filling Modes: The ALTERNATE filling mode is generally preferred for font outlines to ensure predictable behavior, but exploring WINDING can offer unique outcomes.

Creative Potential:

Custom Brushes: Explore a wide array of brush styles and patterns to create unique text textures and backgrounds.

Combine Filling and Outlining: Combine custom pens and brushes for intricate effects.

Font-Based Graphics: Incorporate filled font outlines into complex graphical compositions.

FONTFILL invites you to continue pushing the boundaries of text-based creativity. Embrace the interplay of fonts, paths, brushes, and background modes to craft visually captivating and expressive text designs!

FONTCLIP PROGRAM

```c
/* FONTCLIP.C - Using Path for Clipping on Font(c) Charles Petzold, 1998 */
#include <windows.h>
#include "..\\eztest\\ezfont.h"
TCHAR szAppName[] = TEXT("FontClip");
TCHAR szTitle[] = TEXT("FontClip: Using Path for Clipping on Font");
void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
    static TCHAR szString[] = TEXT("Clipping");
    HFONT hFont;
    int y, iOffset;
    POINT pt[4];
    SIZE size;
    // Create a TrueType font and select it into the device context
    hFont = EzCreateFont(hdc, TEXT("Times New Roman"), 1200, 0, 0, TRUE);
    SelectObject(hdc, hFont);
    // Get text dimensions and begin a path
    GetTextExtentPoint32(hdc, szString, lstrlen(szString), &size);
    BeginPath(hdc);
    TextOut(hdc, (cxArea - size.cx) / 2, (cyArea - size.cy) / 2, szString, lstrlen(szString));
    EndPath(hdc);
    // Set the clipping area to the interior of the text box
    SelectClipPath(hdc, RGN_COPY);
    // Draw Bezier splines with random colors
    iOffset = (cxArea + cyArea) / 4;
    for (y = -iOffset; y < cyArea + iOffset; y++)
    {
        pt[0].x = 0;
        pt[0].y = y;
        pt[1].x = cxArea / 3;
        pt[1].y = y + iOffset;
        pt[2].x = 2 * cxArea / 3;
        pt[2].y = y - iOffset;
        pt[3].x = cxArea;
        pt[3].y = y;
        // Set a random color for each Bezier spline
        SelectObject(hdc, CreatePen(PS_SOLID, 1, RGB(rand() % 256, rand() % 256, rand() % 256)));
        PolyBezier(hdc, pt, 4);
        DeleteObject(SelectObject(hdc, GetStockObject(BLACK_PEN)));
    }
    // Cleanup: Reset the brush, select the default font, and delete the temporary font
    DeleteObject(SelectObject(hdc, GetStockObject(WHITE_BRUSH)));
    SelectObject(hdc, GetStockObject(SYSTEM_FONT));
    DeleteObject(hFont);
}
```

Key Concepts:

Clipping Region: A defined area on a drawing surface where graphics are visible. Anything drawn outside this region is hidden.

TrueType Fonts: Outline-based fonts that can be defined as paths, allowing their shapes to be used for clipping.

Paths: Collections of lines and curves that define shapes in graphics programming.

Program Breakdown:

Includes Headers: windows.h for Windows functions and eztest/ezfont.h for font-related functions.

Sets Application and Window Titles: Using szAppName and szTitle.

PaintRoutine Function: Called when the window needs to be repainted.

Creates Font: Loads "Times New Roman" with a size of 1200 using EzCreateFont.

Selects Font: Sets it as the active font for drawing.

Measures Text Size: Determines the dimensions of the word "Clipping" using GetTextExtentPoint32.

Begins Path: Starts a new path for clipping.

Draws Text: Renders the word "Clipping" onto the path using TextOut.

Ends Path: Completes the path definition.

Sets Clipping Region: Defines the clipping region based on the shape of the text path using SelectClipPath.

Draws Bezier Splines: Creates and draws a series of curved lines with random colors within the clipping region.

Omitted SetBkMode Call: Intentionally excludes SetBkMode to achieve a unique effect.

Effect Without SetBkMode:

The clipping region is restricted to the entire rectangular area enclosing the text, not just the character outlines themselves.

Bezier curves are clipped to this rectangle, creating a visually distinct appearance.

Alternative Effect with SetBkMode (TRANSPARENT):

If SetBkMode(TRANSPARENT) were used, the clipping region would be defined by the actual character outlines.

Bezier curves would be clipped to the shapes of the letters, resulting in a different visual effect.

Overall, the FONTCLIP program demonstrates how to use TrueType font paths for creating interesting clipping regions and visual effects in graphics programming

Clipping Technique:

Most programs use basic shapes like rectangles or ellipses for clipping regions. FONTCLIP.C, however, uses a dynamically created path from a TrueType font, which offers a much more flexible and nuanced clipping region.

This allows the program to clip other content to the specific shapes of the characters, creating unique visual effects not possible with simpler clipping shapes.

Focus on Clipping Behavior:

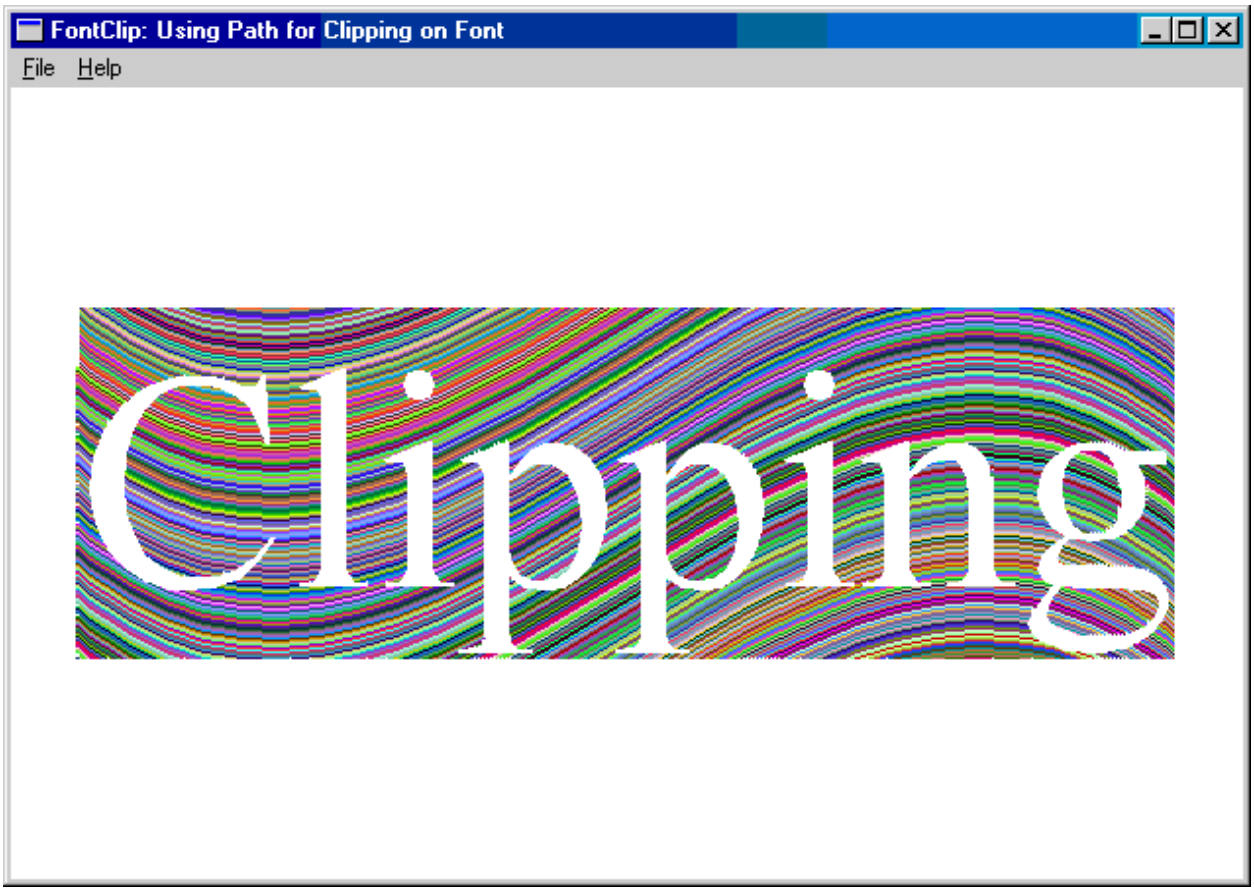
While other programs use clipping primarily for masking or hiding unwanted content, FONTCLIP.C highlights the clipping region itself as a central element of the artwork.

The Bezier curves drawn within the clipping area become the main visual focus, emphasizing the interaction between the font path and the clipped content.

Omission of SetBkMode:

Most programs typically set the background mode with SetBkMode to control how text interacts with the background. FONTCLIP.C deliberately avoids setting the background mode to achieve a specific effect.

This results in the Bezier curves filling the entire rectangular area around the text, not just the interior of the character outlines, creating a distinctive visual style.



Overall, FONTCLIP.C stands out due to its:

- Innovative use of font paths for clipping
- Focus on the clipping region as a creative element
- Unique visual effect achieved through intentional omission of background mode setting

While other programs may demonstrate different functionalities or techniques, FONTCLIP.C takes a creative approach to clipping and utilizes its features to create a distinct and interesting visual experience.

Experimenting With Setbkmode:

Insert SetBkMode(TRANSPARENT) into FONTCLIP.C to observe the visual change.

This will alter the clipping region to follow the actual character outlines, not just the enclosing rectangle.

Compare this modified output with the original effect to understand the impact of background mode on clipping.

Fontdemo For Printing And Experimentation:

Use the FONTDEMO shell program for both printing and displaying the effects.

This program offers a more versatile environment for exploring different visual possibilities.

Creative Exploration:

Experiment with your own special effects:

- Try varying font styles, sizes, and colors.
- Adjust the Bezier curves (number, shape, colors).
- Explore different clipping region shapes (e.g., combining font paths with other shapes).
- Combine clipping with other graphics techniques (e.g., transparency, blending).

Key takeaways:

- SetBkMode plays a crucial role in defining clipping behavior when working with text and paths.
- Shell programs like FONTDEMO provide flexible environments for experimentation and visual creativity.
- Exploring different combinations of techniques can lead to unique and visually appealing effects.

I encourage you to actively experiment with these suggestions to deepen your understanding of clipping and discover new visual possibilities!

And with that, chapter 17 is done….