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

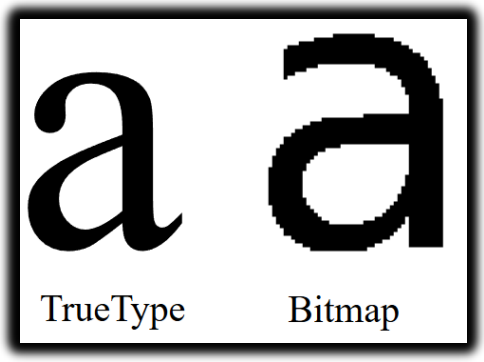
This is where WinAPI text stops being *static* and becomes *programmable*.

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.

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)

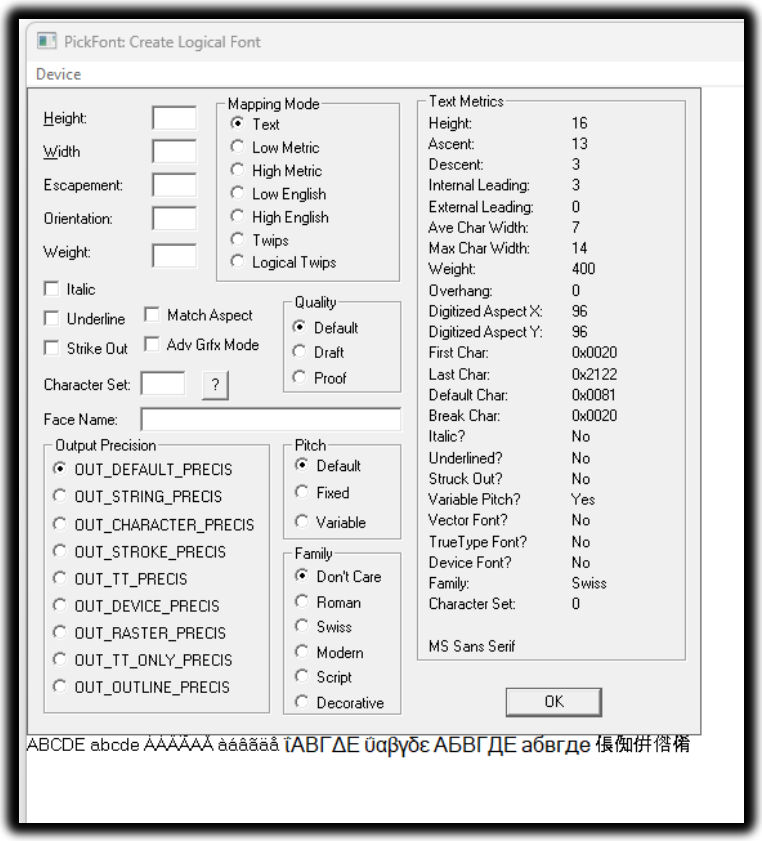
MySetMapMode sets the **ruler type** for the Device Context before drawing, ensuring sizes are interpreted correctly.

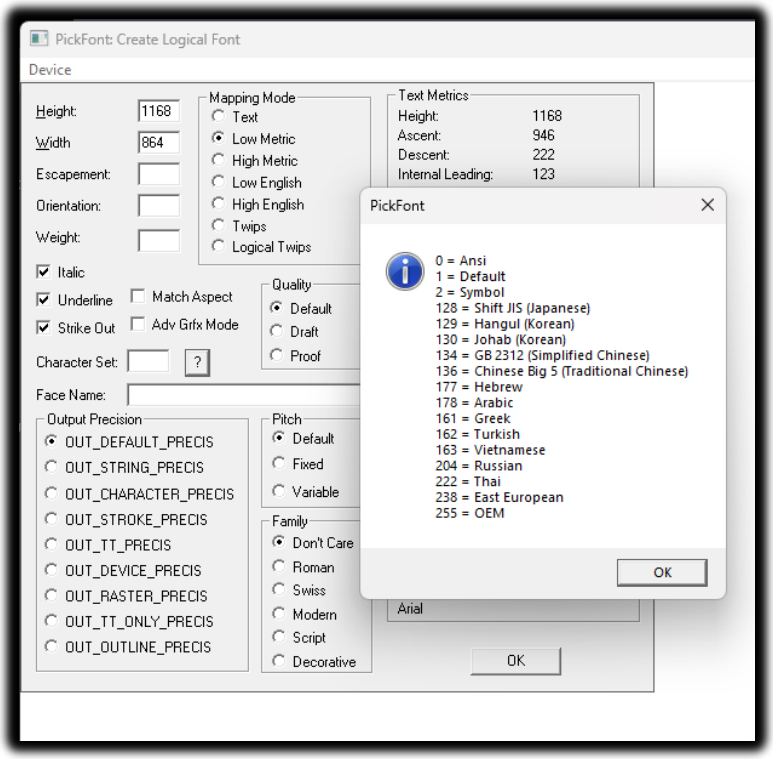
VI. Summary – Flow of Data

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

EZFONT's Role in Windows Font Management:

Addressing Legacy Font Handling Challenges: EZFONT emerged as a response to the complexities associated with font enumeration and approximation in earlier Windows font-handling mechanisms.

Capitalizing on TrueType's Universality: TrueType fonts, being widely available on Windows systems, offered a foundation for simplifying font selection and management, which EZFONT effectively leverages.

```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 ;
}
```

EzCreateFont Function in Depth:

Device Context Preparation:

Preserving Original State: Saving the device context's state ensures that any subsequent modifications within the function remain isolated, preventing unintended side effects in the application.

Advanced Graphics Mode: This mode enables precise font control and rendering capabilities.

Resetting Transformations: Resetting transformation matrices to identity ensures accurate font measurements and calculations, unaffected by prior transformations.

Resolution Handling:

Physical vs. Logical Resolution: EZFONT accommodates both physical and logical resolution scenarios, providing flexibility for different device configurations.

DPI Calculation: Determining the horizontal and vertical dots per inch (DPI) is crucial for scaling fonts accurately across devices with varying resolutions.

Font Size Calculation:

Tenths of a Point to Device Units: The desired font height and width, specified in tenths of a point, are converted to device units using the calculated DPI.

Device Units to Logical Units: This subsequent conversion ensures consistency in font size rendering across different devices, independent of their physical resolutions.

LOGFONT Configuration:

Encapsulating Font Properties: The LOGFONT structure serves as a comprehensive container for font attributes, including height, width, typeface, character set, and styling properties (bold, italic, underline, strikeout).

Font Creation and Adjustment:

Initial Font Creation: The CreateFontIndirect function is responsible for generating a font based on the specified LOGFONT settings.

Width Adjustment (Optional): If a specific font width is requested, EZFONT retrieves font metrics using GetTextMetrics and adjusts the LOGFONT's width accordingly, ensuring the desired visual appearance.

Device Context Restoration:

Preserving Application State: Restoring the device context to its original state maintains consistency for the application's subsequent drawing operations.

Font Handle Return:

Handle for Text Rendering: The function returns a font handle (HFONT), which serves as a reference to the newly created font, enabling its use in various text rendering tasks within the application.

Additional Considerations:

- Potential for Further Simplification: EZFONT could potentially streamline font enumeration tasks for developers who still require them, further simplifying font management.
- Integration with ChooseFont Dialog Box: EZFONT might offer a means to simplify font choices presented within the ChooseFont dialog box, enhancing user experience.
- Point Size Specification: Use decipoints (1/10ths of a point) for height.
- Em-Width Adjustment: Control overall character widths independently of height.
- Font Attributes: Set bold, italic, underline, and strikeout styles.
- Logical vs. Physical Resolution: Choose based on device type and font scaling needs.
- Mapping Mode Consistency: Ensure the same mapping mode is used during font creation and text rendering.
- Memory Management: Delete created fonts using DeleteObject before program termination.
- Windows NT-Specific Adjustments: Necessary for accurate font sizing on Windows NT systems.
- Font Enumeration Simplification: EZFONT could potentially simplify font enumeration tasks as well.
- ChooseFont Dialog Integration: Might offer a way to simplify font choices within the dialog.

Windows NT-Specific Adjustments:

In the context of Windows 10 development, the specific adjustments made in EZFONT for Windows NT may no longer be necessary.

Windows 10 has undergone significant evolution since the Windows NT era, and the calls to SetGraphicsMode and ModifyWorldTransform, designed for Windows NT, can be safely omitted.

This suggests that Windows 10's advancements may have obviated the need for these particular adjustments in achieving accurate font sizing.

Font Enumeration Simplification:

Despite the advancements in Windows 10's font handling capabilities, EZFONT retains relevance for developers seeking to simplify font enumeration.

Its potential to streamline font selection processes remains valuable, offering a straightforward approach to filter font choices and simplify the selection of TrueType fonts.

Developers can leverage EZFONT to enhance the efficiency of font-related tasks, even within the context of Windows 10.

ChooseFont Dialog Integration:

Integrating EZFONT with the ChooseFont dialog in Windows 10 applications could yield several benefits for the user experience.

This integration could facilitate the filtering of font choices to those reliably available on the system, simplifying the selection process.

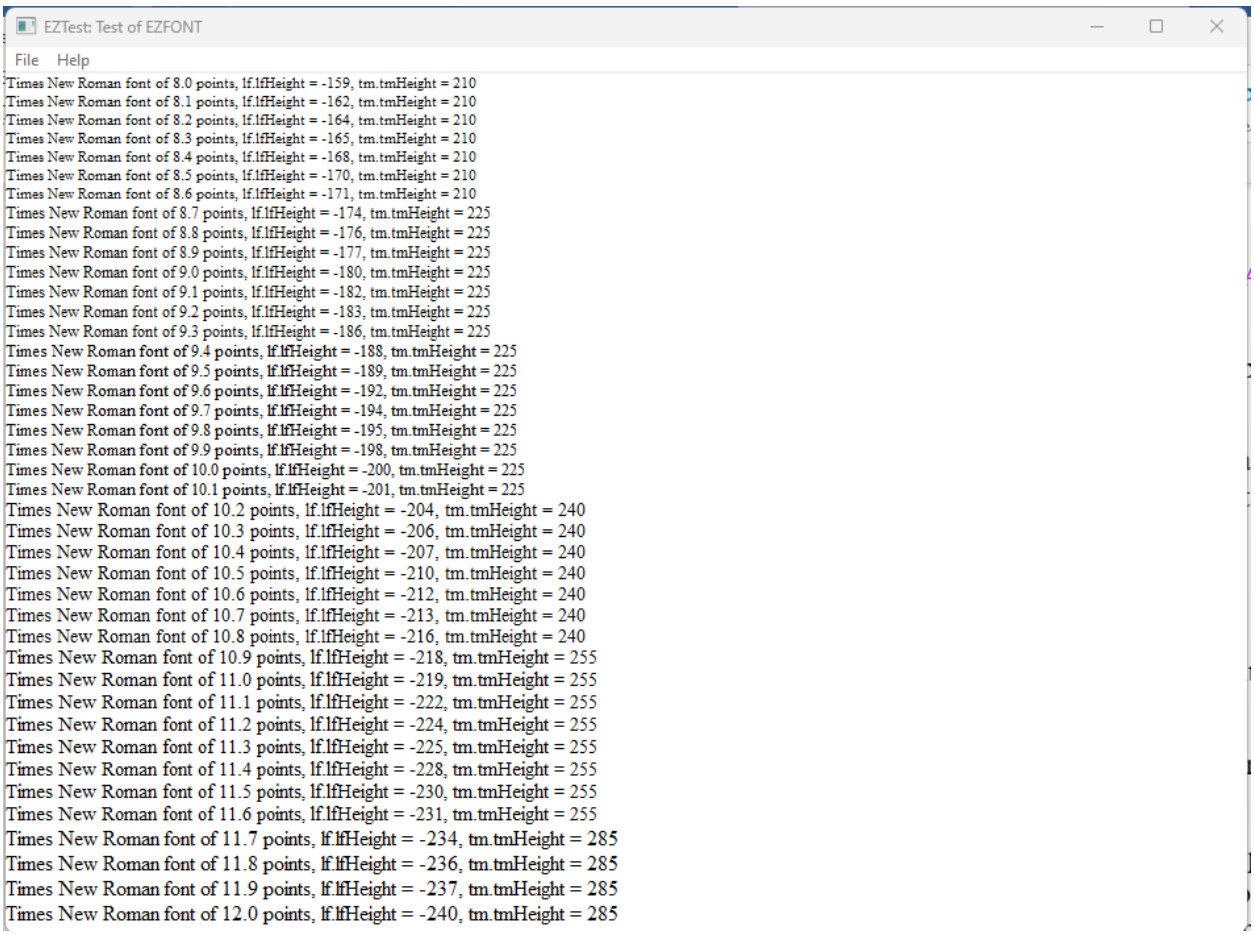
Moreover, it might offer more intuitive customization options, contributing to a user-friendly interface for font-related operations.

Key Considerations for Windows 10:

In the Windows 10 environment, it's crucial to acknowledge the updated font handling APIs that provide a comprehensive set of options for font selection, creation, and management.

When designing font-related features for Windows 10 applications, prioritizing user experience is paramount.

This involves offering intuitive font selection tools, ensuring consistent font rendering across diverse devices and resolutions, and providing clear feedback during font-related operations.

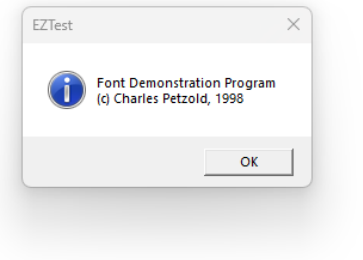


Recommendations:

For developers working with EZFONT in a Windows 10 environment, thorough compatibility testing is recommended.

Identifying and addressing any compatibility issues will help ensure the smooth functioning of EZFONT in the latest operating system.

Additionally, exploring Windows 10's advanced font handling APIs can provide developers with expanded capabilities for font management, aligning with the modern features offered by the operating system. Ultimately, a user-centric design approach should guide the development of font-related features to deliver an optimal experience for Windows 10 users.



EZTEST.C:

Focus on Font Testing: This program's primary purpose is to rigorously evaluate the EZFONT system's ability to accurately create and render TrueType fonts within a Windows environment. It serves as a comprehensive test suite for ensuring font functionality.

Key Functionalities:

Font Size Exploration: The program dynamically creates TrueType fonts spanning a range of point sizes, meticulously examining font generation and rendering across a spectrum of sizes.

Font Metrics Retrieval: It probes font metrics using GetObject and GetTextMetrics, gathering essential information about font dimensions and characteristics, validating their consistency and accuracy.

Font Information Display: Descriptive text is rendered alongside each generated font, showcasing its point size and corresponding metric values, providing a visual and informative assessment of font behavior.

Logical Twips for Scalability: The program employs the logical twips mapping mode to establish a reliable foundation for consistent font scaling across diverse devices and screen resolutions, ensuring accurate font appearance regardless of hardware configuration.

Window Procedure (WndProc):

Initialization:

The program starts by initializing variables, including a static DOCINFO structure and a PRINTDLG structure.

It defines variables to store the client area dimensions and a handle to the printer DC.

Command Handling (WM_COMMAND):

It handles commands, including IDM_PRINT (Print) and IDM_ABOUT (About).

When IDM_PRINT is selected, it opens a Print dialog, obtains the printer DC, and sets up the printing environment.

It calls the PaintRoutine to perform font-related painting on the printer DC.

It handles the printing process, including starting and ending the document and pages.

When IDM_ABOUT is selected, it displays an informational message box about the font demonstration program.

Window Size Handling (WM_SIZE):

It updates the variables storing the client area dimensions when the window size changes.

Painting (WM_PAINT):

It calls BeginPaint and EndPaint to set up and finish the painting process.

It calls the PaintRoutine function to handle the actual painting of font-related information on the window DC.

Cleanup and Quit (WM_DESTROY):

When the window is closed, it posts a quit message to terminate the program.

2. PaintRoutine Function:

Device Context Setup:

It sets the logical mapping mode to logical twips and adjusts the window and viewport extents for consistency.

Font Testing Loop:

It iterates through a range of point sizes from 80 to 120.

For each point size, it creates a Times New Roman font using EzCreateFont, retrieves font information, and prints details about the font.

Cleanup:

It deletes the created font object to release resources.

3. WinMain Function:

Window Class Registration:

It registers the window class with the window procedure (WndProc) and other attributes.

Window Creation:

It creates the main window with specified attributes.

Message Loop:

It enters the message loop, handling messages until the program is terminated.

Return:

It returns the wParam of the last message as the program's exit code.

Conclusion:

The EZTEST.C program is designed to create a window, handle user commands for printing and displaying information about the font demonstration program, and paint font-related details in the client area. It works in conjunction with the EzCreateFont function and is part of a font demonstration system.

FONTDEMO.C:

Visual Font Showcase: This program excels in providing an engaging and interactive demonstration of font capabilities, seamlessly integrating the EZFONT system to offer a user-friendly visual experience.

Notable Features:

Window Creation and Message Handling: It establishes a standard Windows window to serve as a canvas for font rendering, skillfully handling various system messages to ensure smooth window interactions and responsiveness.

Font Rendering Orchestration: The program invokes the PaintRoutine function (shared with EZTEST.C) to execute the actual font rendering process, leveraging EZFONT's font creation utilities to produce visually appealing text elements.

User-Friendly Menu: It offers a convenient menu system, empowering users to print the displayed fonts for physical reference or to access program information, fostering accessibility and exploration.

Printing Functionality: The program incorporates robust printing capabilities, allowing users to effortlessly capture the visual font demonstration on paper, extending its utility beyond onscreen experiences.

1. Include Statements and External Declarations:

Include Statements:

It includes necessary header files, such as windows.h, EzFont.h, and resource.h.

External Declarations:

It declares external functions, including PaintRoutine and WndProc, as well as external variables like szAppName and szTitle.

2. WndProc (Window Procedure):

Window Class Registration:

It registers the window class with the window procedure (WndProc) and other attributes.

Window Creation:

It creates the main window with specified attributes.

Message Loop:

It enters the message loop, handling messages until the program is terminated.

Return:

It returns the wParam of the last message as the program's exit code.

3. PaintRoutine Function:

Print Command Handling (WM_COMMAND):

It handles the IDM_PRINT command.

It opens a Print dialog, obtains the printer DC, and sets up the printing environment.

It calls the PaintRoutine to perform font-related painting on the printer DC.

It handles the printing process, including starting and ending the document and pages.

About Command Handling (WM_COMMAND):

It handles the IDM_ABOUT command.

It displays an informational message box about the font demonstration program.

Window Size Handling (WM_SIZE):

It updates the variables storing the client area dimensions when the window size changes.

Painting (WM_PAINT):

It calls BeginPaint and EndPaint to set up and finish the painting process.

It calls the PaintRoutine function to handle the actual painting of font-related information on the window DC.

Printing Cleanup (WM_DESTROY):

It handles cleanup related to printing when the window is destroyed.

4. WinMain Function:

Window Class Registration:

It registers the window class with the window procedure (WndProc) and other attributes.

Window Creation:

It creates the main window with specified attributes.

Message Loop:

It enters the message loop, handling messages until the program is terminated.

Return:

It returns the wParam of the last message as the program's exit code.

Conclusion:

The FONTDEMO.C program is designed to create a window, handle user commands for printing and displaying information about the font demonstration program, and paint font-related details in the client area. It works in conjunction with the EzCreateFont function and is part of a font demonstration system.

Intertwined Yet Distinct:

Shared Font Rendering Logic: Both programs strategically utilize the PaintRoutine function to streamline font rendering tasks, promoting code reusability and maintainability.

EZFONT as the Font Engine: FONTDEMO.C effectively harnesses the font creation prowess of the EZFONT system to fuel its visual demonstrations, showcasing the practical applications of EZFONT's capabilities.

Diverse Purposes, Harmonious Collaboration: While EZTEST.C focuses on meticulous font testing, FONTDEMO.C excels in providing an engaging user experience, underscoring the versatility of the EZFONT system in both development and demonstration scenarios.

Here's a breakdown of the key points to mention:

Rasterization Limitations:

The PaintRoutine function in EZTEST.C encounters challenges in precisely rendering font sizes due to the inherent constraints of rasterization on displays.

Fonts are ultimately composed of pixels, which are discrete units of screen space.

Rendering font sizes with very fine increments (0.1 points in this case) can lead to situations where multiple sizes appear visually identical on screen, as the pixel grid cannot accommodate every subtle size variation.

Printing vs. On-Screen Rendering:

FONTDEMO.C offers the ability to print the font output, often resulting in more accurate size differentiation compared to on-screen rendering.

Printers typically have higher resolutions and employ different rendering techniques, allowing them to represent a wider range of font sizes with greater precision.

Mapping Modes and Font Scaling:

The use of logical twips mapping mode in EZTEST.C aims to promote consistent font scaling across devices.

However, it's crucial to acknowledge that mapping modes primarily affect font scaling behavior, not the underlying rasterization process itself.

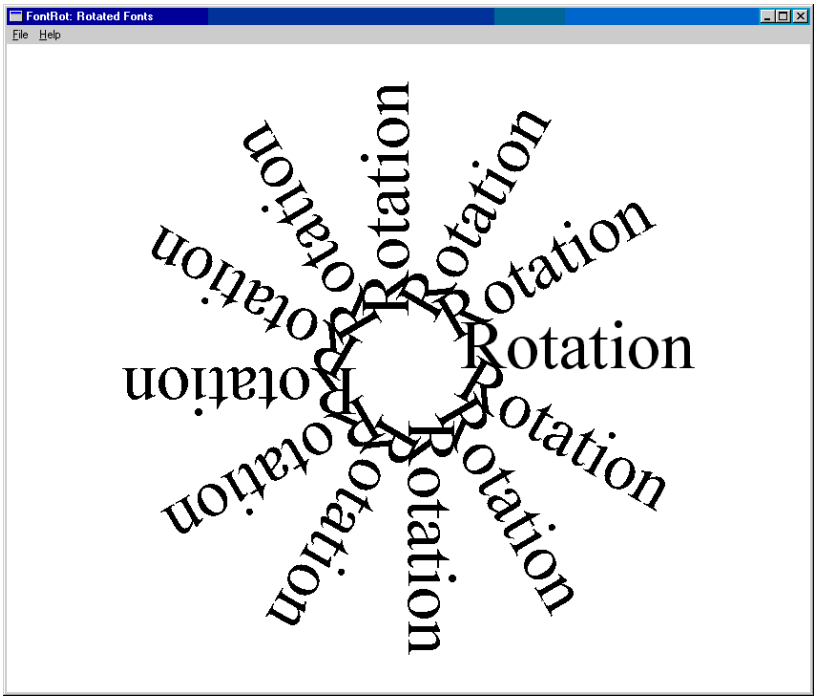
Real-World Implications:

This phenomenon highlights the importance of understanding the nuances of font rendering and rasterization when designing applications that heavily rely on precise font size control.

It's essential to anticipate potential discrepancies between on-screen and printed output, especially when working with fonts at very small sizes or with very fine size adjustments.

FONT ROTATION IN WINDOWS GDI: A DYNAMIC EXPLORATION

The FONTROT program unveils the captivating ability to rotate TrueType fonts within the realm of Windows GDI. It gracefully demonstrates how text can be liberated from rigid horizontal alignment and brought to life at a variety of angles, introducing a new dimension of expressiveness to text-based graphics.



Behind the Scenes: A Technical Tour

Laying the Foundation:

The program starts by using EzCreateFont to create a temporary font called Times New Roman with a size of 54 points. It then carefully retrieves the font's LOGFONT structure, which contains important information for manipulating the font. Finally,

the program responsibly releases the temporary font, showing good resource management practices.

Preparing the Canvas:

To achieve a clean presentation, the program carefully makes the background transparent, ensuring that the rotated text stands out. It establishes a consistent text alignment by aligning the text to its baseline in a visually pleasing manner. Importantly, it moves the starting point of the drawing area to the center, creating a central point around which the text rotates gracefully.

Rotation Unleashed: 12 Steps of Elegance

The program performs a 12-step dance, each step representing a 30-degree rotation increment, covering a full circle of possibilities. In each step:

It adjusts the LOGFONT structure's lfEscapement and lfOrientation fields with the desired rotation angle in tenths of a degree.

It uses CreateFontIndirect to create a new font embodying the specified rotation.

The newly created font is carefully selected into the device context, ready to leave its unique mark on the canvas.

The word "Rotation" slowly appears on the screen, with each letter elegantly angled according to the font's rotation.

To ensure efficient memory usage, the program gracefully says goodbye to the rotated font, maintaining a clean and optimal memory landscape.

Visual Symphony: A Showcase of TrueType Flexibility

The careful execution of this dance produces a captivating visual showcase. The screen is adorned with twelve instances of the word "Rotation," each elegantly angled at a unique degree, forming a stunning starburst of text.

This compelling demonstration reaffirms the impressive flexibility of TrueType fonts within Windows GDI, empowering developers to craft visually dynamic and expressive text-based experiences with professionalism and finesse.

Key Takeaways:

The LOGFONT structure acts as a versatile tool for controlling font attributes, such as rotation angles, with precision.

GDI adeptly handles text rotation, even with intricate TrueType fonts, showcasing its versatility in rendering text.

The viewport origin plays a crucial role in text rotation, serving as the axis around which text gracefully revolves.

FONTROT serves as a prime example of responsible font resource management, emphasizing the significance of proper font handling for maintaining performance and avoiding memory leaks.

If you are seeking a more versatile solution for graphics rotation and other linear transformations, and your programs are limited to running on Windows NT, you can utilize the XFORM matrix and the world transform functions.

XFORM MATRIX: THE MASTER CHOREOGRAPHER:

This structure, defined as XFORM, serves as a mathematical blueprint for choreographing diverse transformations within Windows NT's graphics realm.

It encompasses six crucial elements:

- eM11, eM12, eM21, eM22: These elements collectively define the core transformation matrix, governing scaling, rotation, shearing, and other linear transformations.
- eDx, eDy: These elements specify horizontal and vertical translations, allowing movement of objects within the coordinate space.

World Transform Functions: Stage Directors:

These functions act as stage directors, meticulously applying the transformations encoded within XFORM matrices to the graphics world:

- SetWorldTransform(hdc, &xform): Instructs the device context hdc to embrace the specified xform matrix as its prevailing world transformation, shaping subsequent drawing operations.
- ModifyWorldTransform(hdc, &xform, MWT_LEFTMULTIPLY): Gracefully blends the specified xform matrix with the existing world transform, typically through left-multiplication, expanding the range of achievable transformations.

Code Illustration: Rotating Text:

```c
#include <windows.h>

// Function to rotate text by a specified angle in degrees
void RotateText(HDC hdc, const TCHAR* text, int x, int y, int angle) {
    XFORM xform = { 0 };  // Initialize XFORM structure
    xform.eM11 = cos(angle * M_PI / 180);
    xform.eM12 = sin(angle * M_PI / 180);
    xform.eM21 = -sin(angle * M_PI / 180);
    xform.eM22 = cos(angle * M_PI / 180);

    SetWorldTransform(hdc, &xform);  // Apply rotation to the world transform
    TextOut(hdc, x, y, text, lstrlen(text));  // Render rotated text
    SetWorldTransform(hdc, NULL);  // Restore default world transform
}
```

NT-Specific Capabilities for Transformation:

The advanced transformation techniques discussed here are exclusive to Windows NT systems. These capabilities provide developers with powerful tools for manipulating graphics using linear transformations.

Versatile Transformations with XFORM and World Transforms:

The XFORM matrix and world transform functions offer a wide range of possibilities for linear transformations. These transformations include scaling, rotation, shearing, and translation. By leveraging these capabilities, developers can precisely control the positioning, orientation, and size of graphical elements.

Cumulative Effects and ModifyWorldTransform:

One of the advantages of using XFORM and world transforms is the ability to combine multiple transformations. By using the ModifyWorldTransform function, developers can apply multiple transformations successively, creating intricate and complex visual effects.

Restoring the Default World Transform:

To ensure proper behavior and avoid unintended impacts on subsequent drawing operations, it is crucial to restore the default world transform after applying custom transformations. This can be achieved by using the SetWorldTransform function with a NULL parameter.

Additional Notes:

Compatibility: While the discussed techniques are specific to Windows NT, it's worth noting that modern versions of Windows offer alternative approaches using graphics APIs like Direct2D. These APIs provide similar transformation capabilities and are compatible with a broader range of Windows systems.

Matrix Mastery: Understanding matrix operations is essential for effectively utilizing XFORM and world transforms. Mastery of matrices allows developers to manipulate transformations precisely and create visually appealing graphics.

Although the XFORM and world transform functions, introduced in Windows NT, are still available in Windows 10 for compatibility with legacy applications, newer graphics APIs like Direct2D and Direct3D offer more advanced and efficient options for graphics rendering and transformation.

While you can use XFORM and world transform functions in Windows 10, considering the specific requirements of your application, exploring the newer graphics APIs may provide enhanced transformation capabilities.

FONT ENUMERATION

Font enumeration is the process of retrieving a list of all available fonts on a device using the Graphics Device Interface (GDI). This list can then be utilized by a program for various purposes, such as font selection or presenting options to the user. Let's explore the enumeration functions and how to use the ChooseFont function as an alternative for font selection.

A Trio of Enumeration Functions:

EnumFonts: A Legacy Yet Functional Tool: While rooted in earlier Windows versions, EnumFonts remains a viable option for basic font enumeration tasks. It dutifully provides information about either all installed fonts or those matching a specified typeface.

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

EnumFontFamilies: Enhanced TrueType Support: Designed specifically for TrueType fonts, this function excels in environments where TrueType font handling is paramount. It operates in a two-stage process, first identifying font families and then delving into individual fonts within those families.

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

EnumFontFamiliesEx: Fine-Grained Control for Modern Windows: Representing the recommended approach for 32-bit Windows systems, EnumFontFamiliesEx offers the highest degree of customization and control over the enumeration process. It gracefully

accepts a LOGFONT structure, enabling developers to meticulously specify enumeration criteria.

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

ChooseFont Function:

The ChooseFont function simplifies font enumeration and selection in applications:

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

Initialize the CHOOSEFONT and LOGFONT structures:

```
ZeroMemory(&cf, sizeof(CHOOSEFONT));
cf.lStructSize = sizeof(CHOOSEFONT);
cf.lpLogFont = &lf;
cf.Flags = CF_SCREENFONTS | CF_EFFECTS;
```

Invoke the ChooseFont dialog:

```
if (ChooseFont(&cf)) {
    // The selected font information is now in the lf member of the LOGFONT structure.
    // You can use it for further processing in your application.
}
```

This approach avoids the direct use of enumeration functions, and the user can interactively choose a font through the standard Windows font selection dialog.

In summary, font enumeration functions provide a way to gather information about available fonts, but for user-friendly font selection, the ChooseFont function is a more convenient option.

The Callback Function: An Important Partner:

Within GDI's realm, the callback function holds a crucial role. GDI faithfully calls upon this function for every enumerated font, granting it the task of processing the font information it provides.

Flexibility at Your Fingertips: Developers have the ability to customize the behavior of the callback function. This allows them to perform a variety of tasks, such as building font lists

for user selection, gathering font metrics for precise text layout, and identifying fonts that meet specific criteria.

Structures: Blueprints for Font Information:

LOGFONT: The Font Architect: This essential structure meticulously defines the core attributes of a font, such as its typeface, size, style, and pitch.

TEXTMETRIC: Revealing Font Metrics: This structure elegantly unveils crucial metrics like font height, width, and spacing, providing essential insights for text layout calculations.

ENUMLOGFONT: Enhanced Information for TrueType Fonts: Building upon the LOGFONT structure, this expanded structure reveals the full name and style of a font, offering a more comprehensive understanding of its characteristics.

NEWTEXTMETRIC: Additional Insights for TrueType Fonts: Designed specifically for TrueType fonts, this structure complements TEXTMETRIC by providing additional fields tailored to uncover the unique attributes of TrueType fonts.

CHOOSEFONT: A USER-FRIENDLY SHORTCUT:

Simplifying Font Selection: When you need to allow users to choose fonts, the ChooseFont function gracefully presents a familiar dialog box. It empowers users to effortlessly explore and select fonts within a visually intuitive interface.

CHOOSEFONT Structure:

Delving into lf: The Font Architect: This LOGFONT structure meticulously defines the font's core attributes, including typeface, height, width, escapement, orientation, weight, italic, underline, strikeout, character set, output precision, clipping precision, quality, pitch, and family.

Fine-Tuning Dialog Behavior: Flags: Developers can leverage flags within the CHOOSEFONT structure to tailor the dialog box's appearance and functionality, such as:

- CF_SCREENFONTS: Display only screen-compatible fonts.
- CF_EFFECTS: Enable strikeout and underline effects.

- CF_INITTOLOGFONTSTRUCT: Initialize the dialog with a font specified in the lf field.
- CF_NOVECTORFONTS: Exclude vector fonts.

ChooseFont Function Extended:

Guiding Font Selection: ChooseFont presents a visual interface for font exploration and selection, eliminating the need for manual font enumeration.

Return Value: Upon successful font selection, ChooseFont returns true and populates the CHOOSEFONT structure with the user's choices.

Program Flow Enhancements:

Initialization Insights:

GetDialogBaseUnits provides default character height, ensuring consistent text display across devices.

GetObject retrieves the system font's LOGFONT structure, establishing a baseline for font settings.

User-Driven Font Exploration: The "Font!" menu item activates ChooseFont, inviting users to discover their preferred fonts.

Repainting with Selected Font: InvalidateRect triggers a visual update, showcasing the newly chosen font in all its glory.

Unveiling Font Information:

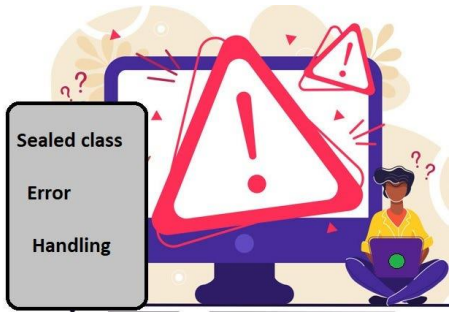
CreateFontIndirect: This function crafts a font object based on the LOGFONT structure, ready for text rendering.

TextOut: This function paints text onto the window's canvas, utilizing the selected font and color.

Displaying Font Attributes: The program visually presents key LOGFONT structure fields, providing transparency into the font's characteristics and empowering users to make informed font choices.

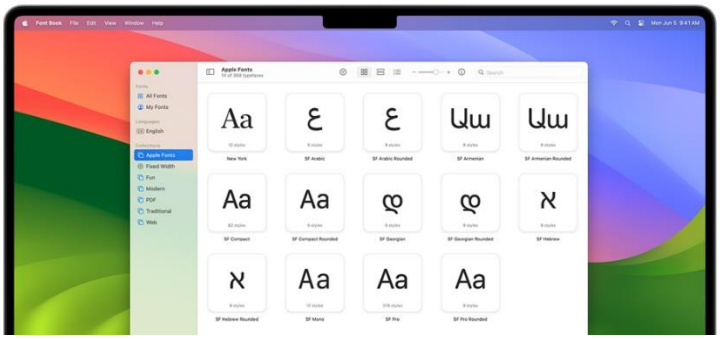
Additional Considerations:

Error Handling: Robust applications should gracefully handle scenarios where ChooseFont fails to return a valid font selection.



Font Compatibility: Developers should consider font availability across different platforms and devices to ensure consistent rendering.

Accessibility: ChooseFont can be further enhanced with accessibility features, such as font size adjustment and high-contrast text options, to accommodate users with diverse needs.



Integration with Text Rendering: Seamlessly integrate ChooseFont and font customization options into text rendering functions like TextOut for a cohesive user experience.

Advanced Font Handling: Explore techniques like font fallback and font linking for comprehensive font management and improved text display.

I'll expand upon the previous notes, providing additional insights and examples:

Enriching Font Dialog Customization:

Beyond Basic Flags: While flags like CF_SCREENFONTS and CF_EFFECTS offer fundamental control, delve deeper into more nuanced customizations:

Exerting Precision: Fine-tune output and clipping precision with lfOutPrecision and lfClipPrecision.

Prioritizing Quality: Set desired output quality with lfQuality.

Pitch Perfect: Define pitch (fixed or variable) and font family with lfPitchAndFamily.

LOGFONT Structure: Unveiling Font Essence:

Beyond Common Attributes: Explore lesser-known LOGFONT fields to expand font control:

Angles of Expression: lfEscapement and lfOrientation govern text angles for creative effects.

Character Set Harmony: lfCharSet ensures compatibility with diverse languages and symbols.

Script Field: Embracing Global Communication:

Bridging Language Barriers: The "Script" field plays a crucial role in supporting multilingual text rendering and localization, ensuring accurate character display for a global audience.

Logical Inch: Bridging Point Size and Font Height:

Context-Driven Consistency: The logical inch concept ensures font heights remain proportional to screen resolutions and printing devices, maintaining visual consistency across different display environments.

Key for Text Metrics: Understanding logical inch calculations is essential for precise text layout and positioning within applications.

Mapping Mode Matters:

Metric mapping modes (e.g., MM_LOMETRIC, MM_HIMETRIC) under Windows NT can lead to inconsistencies between logical coordinates and physical font sizes.

Use Logical Twips for graphics that align seamlessly with font size.

Temporarily revert to MM_TEXT for font selection and display to ensure proper font height interpretation.

Device Context Sensitivity:

lfHeight is pixel-based, suitable for video displays.

Adjust lfHeight for printer device contexts based on printer resolution.

The hDC field in CHOOSEFONT merely lists printer fonts, not influencing lfHeight.

Leveraging iPointSize for Flexibility:

This field provides font size in 1/10 point units, adaptable to various device contexts and mapping modes.

Reference EZFONT.C for conversion code, tailorable to specific needs.

Unicode Character Exploration:

The UNICHARS program showcases all characters within a font, ideal for studying Unicode-rich fonts like Lucida Sans Unicode and Bitstream CyberBit.

Its TextOutW function ensures compatibility across Windows NT and Windows 98.

Additional Insights for Font Mastery:

Prioritize MM_TEXT for Font Operations: Stick to MM_TEXT for font selection and display to avoid interpretation conflicts.

Calculate Font Heights Accurately: Ensure precise font height calculation for both screen and printer device contexts.

Experiment with Unicode Fonts: Embrace applications like UNICHARS to explore diverse character sets and enhance text rendering capabilities.

Stay Updated with Font Technologies: Keep abreast of advancements in font rendering and management techniques to deliver optimal user experiences.

UNICHARS.C PROGRAM

The "UNICHARS.C" program is designed to display 16-bit character codes in a graphical user interface. Let's break down its functionality in paragraphs:

The program creates a window that allows users to explore and visualize Unicode characters. The window includes a vertical scrollbar that enables navigation through different pages of Unicode characters. Each page consists of a grid displaying 16 rows and 16 columns of Unicode characters.

Upon initialization, the program sets up the default font and appearance for the Unicode character display. The default font is "Lucida Sans Unicode," and its size is determined to be 12 points, ensuring legibility and a visually pleasing presentation. The ChooseFont common dialog box is also integrated, allowing users to customize the font used for character display.

Users can access the font customization feature by selecting the "Font!" option from the program's menu. This invokes the ChooseFont dialog, providing options to modify the font type, size, and other attributes. Once a new font is chosen, the program refreshes the display to reflect the selected font, ensuring a dynamic and user-friendly experience.

The main window of the program incorporates a vertical scrollbar, facilitating the navigation of Unicode character pages. Users can scroll up or down, either one line at a time or an entire page, to explore different sets of Unicode characters. The scrollbar's position is visually represented, offering a clear indication of the current page being viewed.

The core of the program lies in its ability to paint the Unicode characters onto the window's canvas. It calculates the layout for each character, considering factors such as character width, height, and external leading. The grid structure organizes the characters into rows and columns, making it easy for users to navigate and locate specific characters of interest.

As the program processes paint messages, it dynamically updates the display based on user interactions. For instance, when scrolling through pages, the program adjusts the content to show the corresponding set of Unicode characters for the selected page. The grid format allows for an organized and systematic presentation of characters, aiding users in their exploration.

In summary, the "UNICHARS.C" program provides an interactive platform for users to explore and visualize 16-bit Unicode characters. It integrates font customization through the ChooseFont dialog, incorporates a user-friendly navigation system with a scrollbar, and dynamically updates the display based on user interactions. The program's design aims to enhance the user's experience in discovering and understanding Unicode characters.

Delving into the World of Unicode Characters:

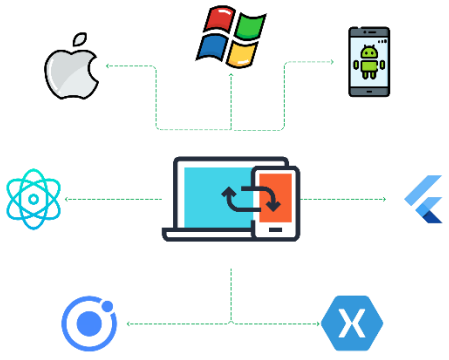
UNICHARS Program: The UNICHARS program provides a visual representation of the vast array of Unicode characters. It organizes these characters in a carefully arranged grid, with each character having a unique hexadecimal code that serves as its digital identifier.

Exploring Character Blocks: Users can embark on an engaging exploration of the 256 character blocks within Unicode. Each block contains 256 individual characters. A vertical scrollbar facilitates seamless navigation through this multilingual landscape, allowing users to discover and examine various characters.

Customizing Fonts: The program offers a personal touch by allowing users to select their preferred font. By accessing the "Font!" menu item and invoking the ChooseFont dialog, users can choose a font that aligns with their aesthetic preferences. By default, the program uses Lucida Sans Unicode, a font widely recognized for its comprehensive support of Unicode characters.

Key Features Amplified:

Cross-Platform Compatibility: The program seamlessly operates on both Windows NT and Windows 98, ensuring accessibility to a wider audience across different operating systems.



Unicode Focus: The program prioritizes Unicode support by utilizing TextOutW, a function specifically designed for accurate rendering of Unicode characters. This ensures faithful representation of diverse scripts and symbols, promoting effective communication across languages.

Font Customization: Users can personalize their experience by selecting their preferred font. This feature allows individuals to cater to their aesthetic and readability preferences, enhancing their exploration of Unicode characters.

Character Exploration: UNICHARS serves as a valuable tool for visually examining a wide range of Unicode characters. It fosters a deeper understanding of multilingual communication, character encoding, and the intricacies of human language in the digital domain.

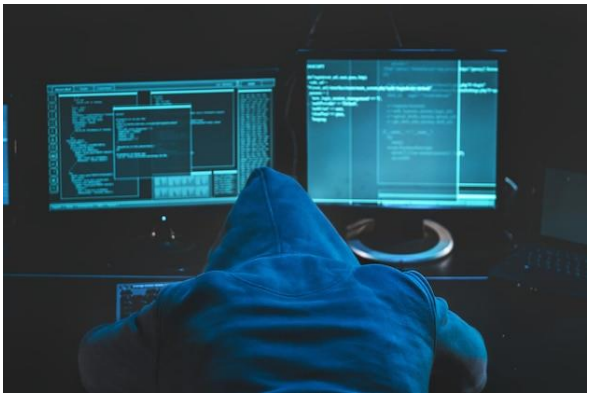
Beyond the Technicalities: Potential Applications:

Font Exploration and Research: Developers, designers, and typography enthusiasts can leverage UNICHARS to delve into the nuances of different fonts, evaluate their Unicode character support, and discover hidden gems within diverse font libraries.

Character Set Research and Development: Linguists, language enthusiasts, and those working with multilingual text can utilize UNICHARS to explore character sets, uncover

relationships between characters, and gain insights into language representation and encoding.

Unicode-Related Development: Software engineers can reference UNICHARS for examples of font selection and Unicode character rendering techniques, applying these concepts in their own projects to create applications that seamlessly handle diverse languages and scripts.



UNDERSTANDING PARAGRAPH FORMATTING:

Goal: Arrange text lines within margins, aligned left, right, centered, or justified (flush with both margins).

DrawText Limitations: While convenient for simple tasks, it lacks flexibility for more complex formatting.

Key Functions for Text Formatting:

GetTextExtentPoint32: Determines the width and height of text in logical units based on the current font.

TextOut: Writes text at a specified location within a device context.

SetTextJustification: Adjusts spacing for justified text.

- Left alignment: Ideal for most body text, offering a natural reading flow.
- Right alignment: Useful for specific elements like dates and page numbers.
- Center alignment: Effective for titles, headings, and quotations to draw attention.
- Justified alignment: Creates a clean, professional look, but use cautiously as excessive stretching can impair 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);
```

GetTextExtentPoint32: A versatile tool for measuring text dimensions, enabling accurate placement within margins.

TextOut: The workhorse for displaying text, offering flexibility in positioning and alignment.

SetTextJustification: The key to achieving justified text, carefully distributing extra space between words for visual harmony.

Remember to clear justification errors: Call SetTextJustification(hdc, 0, 0) before starting a new line to ensure consistent formatting.

Formatting Multiple Lines:

To format multiple lines of text, you can follow these steps:

1.

Calculate Available Width: Determine the available width for the text by subtracting the right margin from the left margin. This will be the maximum width that each line of text can occupy.

```
int leftMargin = 10;   // Example left margin
int rightMargin = 290;   // Example right margin
int availableWidth = rightMargin - leftMargin;
```

2.

Break Text into Lines: Split the text into individual lines using word-wrapping algorithms or manual splitting. This process ensures that each line does not exceed the available width.

```c
#include <string.h>

#define MAX_LINE_LENGTH 80  // Adjust as needed
#define MAX_WORDS 50        // Adjust as needed

int main() {
    char text[] = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.";
    char lines[MAX_WORDS][MAX_LINE_LENGTH];
    int currentLine = 0, wordCount = 0, wordLength;
    char *word;

    word = strtok(text, " ");  // Tokenize the text by spaces

    while (word != NULL && wordCount < MAX_WORDS) {
        wordLength = strlen(word);

        if (strlen(lines[currentLine]) + wordLength + 1 <= MAX_LINE_LENGTH) { //Check if word fits current line
            strcat(lines[currentLine], word);  // Append word to current line
            strcat(lines[currentLine], " ");  // Add a space
        } else {
            currentLine++;  // Move to the next line
            strcpy(lines[currentLine], word);  // Start a new line with the word
            strcat(lines[currentLine], " ");  // Add a space
        }

        wordCount++;
        word = strtok(NULL, " ");  // Get the next word
    }

    // Print the wrapped lines
    for (int i = 0; i <= currentLine; i++) {
        printf("%s\n", lines[i]);
    }

    return 0;
}
```

The provided C code snippet is meticulously crafted to implement a word-wrapping algorithm, transforming a raw text into neatly formatted lines. Let's break down the key components and functionalities of the code.

Essential Headers:

The inclusion of stdio.h and string.h sets the foundation for the code's functionality. These headers bring in essential functions for input/output and string manipulation, respectively.

Setting Boundaries:

The constants MAX_LINE_LENGTH and MAX_WORDS establish crucial boundaries for the word-wrapping process. These constants ensure that lines adhere to a specified maximum length, preventing overflow and managing the total number of words processed.

Textual Home:

The raw text to be formatted is housed in a character array named text. This array serves as the starting point for the word-wrapping algorithm.

Lines Awaiting Construction:

The 2D character array lines is designated to hold the formatted text. This array serves as a canvas where the lines will be constructed as the word-wrapping algorithm unfolds.

Guiding Variables:

Variables such as currentLine, wordCount, and wordLength play pivotal roles in tracking progress and making decisions during the word-wrapping process.

Splitting Text into Words:

The strtok function makes its entrance, skillfully tokenizing the text array into individual words based on space delimiters. This sets the stage for the subsequent word-wrapping actions.

Word-by-Word Journey:

The code enters a loop, iteratively processing each word extracted by strtok. Within this loop, a meticulous sequence of actions takes place for each word:

Measuring Word Length: The strlen function precisely determines the length of the current word.

Line-Breaking Decisions: A critical check assesses whether the word can fit within the remaining space on the current line. This decision considers the length of the word, existing text on the line, and space for separation.

Appending or Starting Anew: Depending on the assessment, the word is either appended to the current line using strcat, ensuring a cohesive flow, or a new line is initiated with strcpy, placing the word at the beginning for a balanced presentation.

Unveiling the Wrapped Text:

In a final act, the code iterates through the lines array, which now holds the beautifully wrapped lines. Using a printf loop, each line is elegantly printed to the console, revealing the text in its newly formatted glory. This explanation showcases the transformation of raw text into well-organized lines.

In the realm of text formatting for Windows applications, certain additional considerations and best practices play a pivotal role in enhancing the visual appeal and readability of content.

Additional Considerations:

Font Size and Style:

These factors wield significant influence over text dimensions and formatting, allowing developers to tailor the visual presentation based on the chosen font characteristics.

Paragraph Spacing:

Fine-tuning vertical spacing between paragraphs is a key aspect of controlling the overall layout and improving the visual flow of textual content.

Indentation:

Horizontal spacing at the beginning of lines, achieved through indentation, contributes to a well-organized and aesthetically pleasing text structure.

Tab Stops:

The ability to set tab stops enables precise control over text alignment in tabular layouts, ensuring a clean and structured appearance.

Best Practices:

Font Selection:

Optimal font choices play a crucial role in readability and overall aesthetics. Developers are encouraged to select fonts that not only enhance readability but also complement the content and application theme.

Line Spacing:

Providing adequate vertical spacing between lines contributes to visual comfort and clarity. Striking the right balance ensures an inviting and easily readable text presentation.

Justification:

While text justification can enhance visual appeal, it should be used judiciously. Excessive stretching may hinder readability, so developers are advised to apply justification thoughtfully.

Hyphenation:

Consideration of hyphenation for breaking long words can significantly improve the appearance of justified text. Thoughtful implementation enhances both the visual aesthetics and overall readability.

Mastering Techniques:

By mastering these techniques, developers can create visually appealing and well-formatted text in their Windows apps. The careful consideration of font size, style, paragraph spacing, indentation, tab stops, and adherence to best practices allows developers to craft content that not only meets functional requirements, while providing an engaging and user-friendly experience.

3.

Format Each Line: Apply the desired text alignment (left, right, center, or justified) to each line of text as needed.

```c
#include <stdio.h>
#include <string.h>
#define MAX_LINES 50  // Adjust as needed
#define MAX_LINE_LENGTH 80  // Adjust as needed

enum TextAlignment { LEFT, RIGHT, CENTER, JUSTIFIED };

int main() {
    char lines[MAX_LINES][MAX_LINE_LENGTH];
    int availableWidth = 60;  // Example width
    TextAlignment alignment = LEFT;  // Example alignment
    // ... (Assuming lines are already populated with text)
    for (int i = 0; i < MAX_LINES; i++) {
        int lineLength = strlen(lines[i]);
        int padding = availableWidth - lineLength;
        switch (alignment) {
            case LEFT:
                // No additional padding needed
                break;
            case RIGHT:
                // Add padding to the left
                for (int j = 0; j < padding; j++) {
                    lines[i][j] = ' ';
                }
                lines[i][padding] = '\0';  // Terminate string after padding
                break;
            case CENTER:
                // Add padding to both sides
                int halfPadding = padding / 2;
                memmove(lines[i] + halfPadding, lines[i], lineLength + 1);  // Shift content to the right
                for (int j = 0; j < halfPadding; j++) {
                    lines[i][j] = ' ';
                }
                for (int j = halfPadding + lineLength; j < padding + lineLength; j++) {
                    lines[i][j] = ' ';
                }
                lines[i][padding + lineLength] = '\0';  // Terminate string
                break;
            case JUSTIFIED:
                // Implement justification logic here
                break;
        }
        // Output the formatted line
        printf("%s\n", lines[i]);
    }
    return 0;
}
```

The provided code demonstrates text alignment techniques in C, specifically handling left, right, and center alignment.

It assumes that an array of lines (lines) has already been populated with text content. The code utilizes the stdio.h library for input/output operations and the string.h library for string manipulation.

The main function begins by declaring the lines array, which has a maximum number of lines (MAX_LINES) and a maximum line length (MAX_LINE_LENGTH).

It also declares variables for the available width of the lines and the desired text alignment, represented by the availableWidth and alignment variables, respectively.

The program then proceeds to iterate through each line in the lines array. For each line, it measures the length of the text using strlen and calculates the padding needed by subtracting the line length from the available width.

A switch statement is used to handle different alignment cases.

- For the LEFT alignment, no additional padding is required, so the code moves to the next line without making any adjustments.
- For the RIGHT alignment, the code adds padding spaces to the left of the line. It iterates over the padding length and assigns a space character to each position in the line array. The resulting line is terminated with a null character to ensure proper string termination.
- For the CENTER alignment, the code adds padding spaces to both sides of the line. It calculates half of the padding length and uses memmove to shift the existing content of the line to the right. Then, it iterates over the first half of the padding length and assigns a space character to each position before the shifted content. It also assigns space characters for the remaining half of the padding length after the shifted content. The resulting line is terminated with a null character.
- The JUSTIFIED alignment case is a placeholder for implementing justification logic, which is not implemented in this example. You can customize this case to implement your desired justification algorithm.

After formatting the line according to the specified alignment, the code outputs the formatted line using printf. It iterates through each line in the lines array and prints it to the console.

The code allows you to adjust constants such as MAX_LINES and MAX_LINE_LENGTH to suit your specific requirements.

4.

Adjust Vertical Position: If you have multiple lines of text, add the line height (size.cy) to the vertical position for each subsequent line to ensure proper spacing.

Tell me more. Can you provide an example of how to use the SetTextJustification function for adjusting spacing in justified text? What are some best practices for selecting fonts that enhance readability and complement the content and application theme? Could you explain how to calculate the available width for text and break it into individual lines using word-wrapping algorithms?

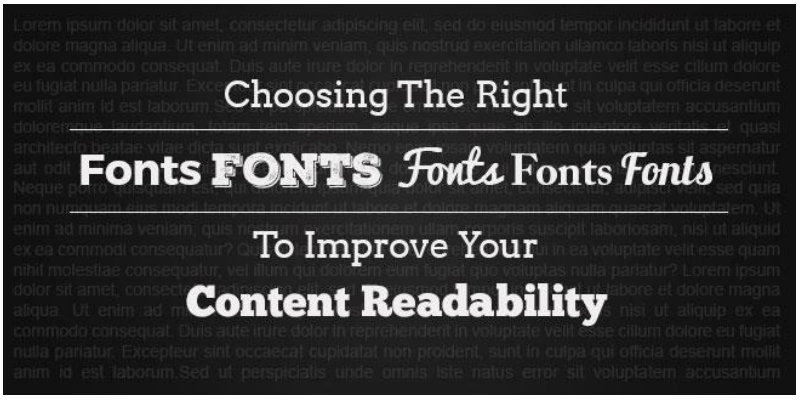
Here's an example of how to use the SetTextJustification function in C programming to adjust spacing for justified text:

```c
#include <windows.h>

int main() {
    HDC hdc = GetDC(NULL); // Get the device context

    // Set up the text attributes
    SetTextAlign(hdc, TA_LEFT);
    SetTextJustification(hdc, 2, 1); // Adjust spacing for justified text

    // Display the justified text
    TextOut(hdc, 100, 100, "Lorem ipsum dolor sit amet", strlen("Lorem ipsum dolor sit amet."));

    // Release the device context
    ReleaseDC(NULL, hdc);

    return 0;
}
```

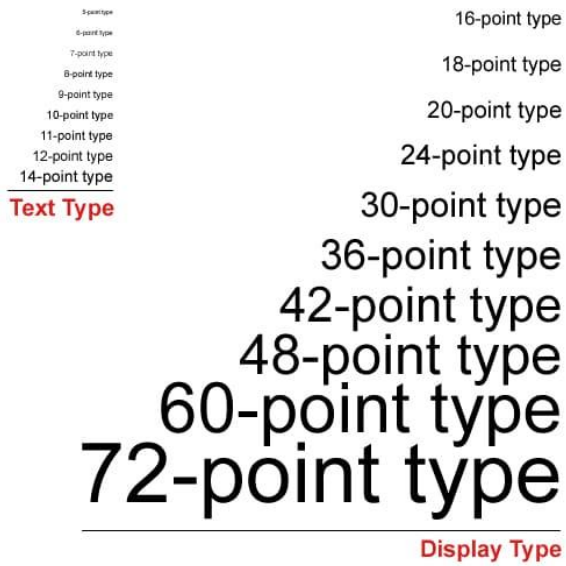
In this example, the SetTextJustification function is used to adjust the spacing for justified text. The second parameter is set to 2, which represents the desired average character spacing. The third parameter is set to 1, which represents the desired minimum space between words.

Regarding font selection, here are some best practices for selecting fonts that enhance readability and complement the content and application theme:

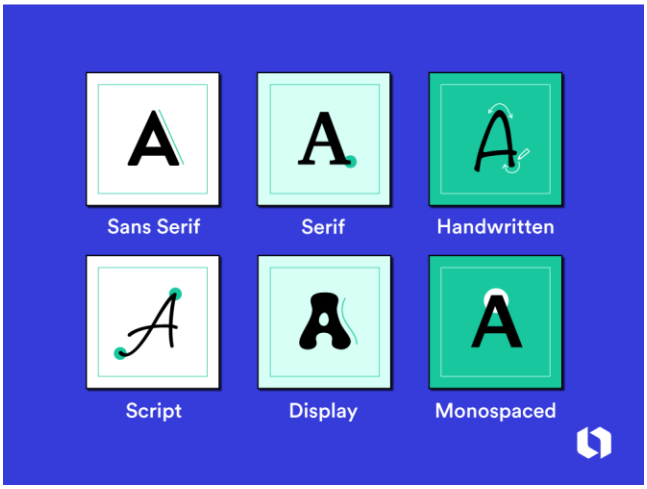
Choose a font with good readability: Select a font that is clear and legible, with distinct letterforms and appropriate spacing between characters. Fonts like Arial, Verdana, and Calibri are known for their readability.



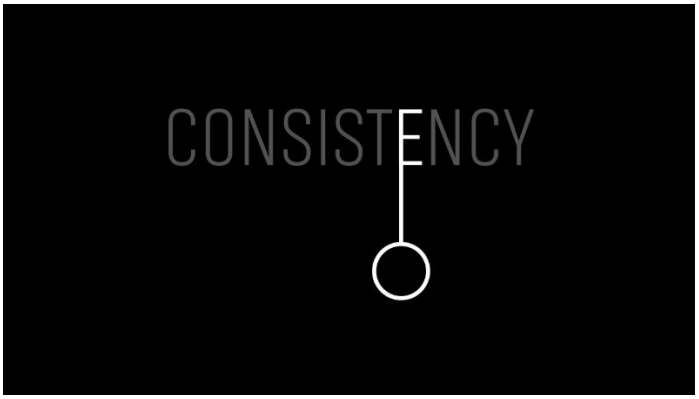
Consider the font size: The font size should be large enough to ensure readability, especially for longer paragraphs of text. Generally, a font size between 10 and 12 points is considered standard for body text.



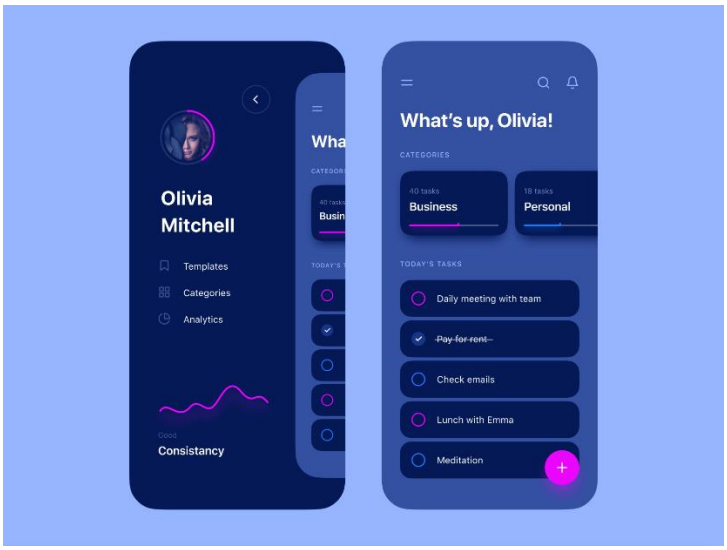
Pay attention to font style: Different font styles (regular, bold, italic) can be used to emphasize specific elements or headings. However, avoid using excessive variations in font styles, as it can make the text harder to read.



Maintain consistency: Consistency in font usage throughout the application creates a cohesive and professional appearance. Limit the use of multiple fonts to avoid visual clutter and confusion.



Consider the application theme: Choose a font that aligns with the overall design and theme of the application. For example, a formal business application may benefit from a classic and traditional font, while a creative app may opt for a more unique and artistic font.



Now, let's discuss how to calculate the available width for text and break it into individual lines using word-wrapping algorithms in C programming.

To calculate the available width for text, you need to determine the width of the container or the area where the text will be displayed.

This could be a window, a specific region, or a predefined size. Once you have the width, you subtract any margins or padding to get the maximum width that each line of text can occupy.

To break the text into individual lines using word-wrapping algorithms, you can use the following approach:

```c
#include <stdio.h>
#include <string.h>
#define MAX_LINE_LENGTH 80  // Maximum line length
#define MAX_LINES 10        // Maximum number of lines

void WrapText(const char* text) {
    char lines[MAX_LINES][MAX_LINE_LENGTH + 1];  // Array to hold formatted lines
    int lineIndex = 0;  // Current line index
    // Tokenize the text into words
    char* word = strtok(text, " ");
    while (word != NULL) {
        // Check if the word can fit within the current line width
        if (strlen(word) <= MAX_LINE_LENGTH - strlen(lines[lineIndex])) {
            // Append the word to the current line
            strcat(lines[lineIndex], word);
            strcat(lines[lineIndex], " ");
        } else {
            // Move to the next line
            lineIndex++;
            // Check if the line index exceeds the maximum number of lines
            if (lineIndex >= MAX_LINES) {
                break;
            }
            // Copy the word to the new line
            strcpy(lines[lineIndex], word);
            strcat(lines[lineIndex], " ");
        }
        // Get the next word
        word = strtok(NULL, " ");
    }
    // Print the formatted lines
    for (int i = 0; i <= lineIndex; i++) {
        printf("%s\n", lines[i]);
    }
}

int main() {
    const char* text = "Lorem ipsum dolor sit amet vitae.";
    WrapText(text);
    return 0;
}
```

In this example, the WrapText function takes the input text and breaks it into individual lines using a word-wrapping algorithm.

The lines array is used to store the formatted lines, with each line having a maximum length of MAX_LINE_LENGTH. The lineIndex variable keeps track of the current line being processed.

The strtok function is used to tokenize the input text into individual words based on space delimiters. Eachword is then checked to see if it fits within the current line width. If it does, the word is appended to the current line with a space. If the word exceeds the available space, a new line is created, and the word is placed there.

The process continues until all words have been processed or the maximum number of lines, specified by MAX_LINES, is reached. Finally, the formatted lines are printed.

Note that this example assumes a fixed maximum line length and a maximum number of lines for simplicity. In a real-world scenario, you may need to handle variable line lengths and dynamically allocate memory to accommodate longer texts.

Remember to adjust the values of MAX_LINE_LENGTH and MAX_LINES according to your requirements.

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.

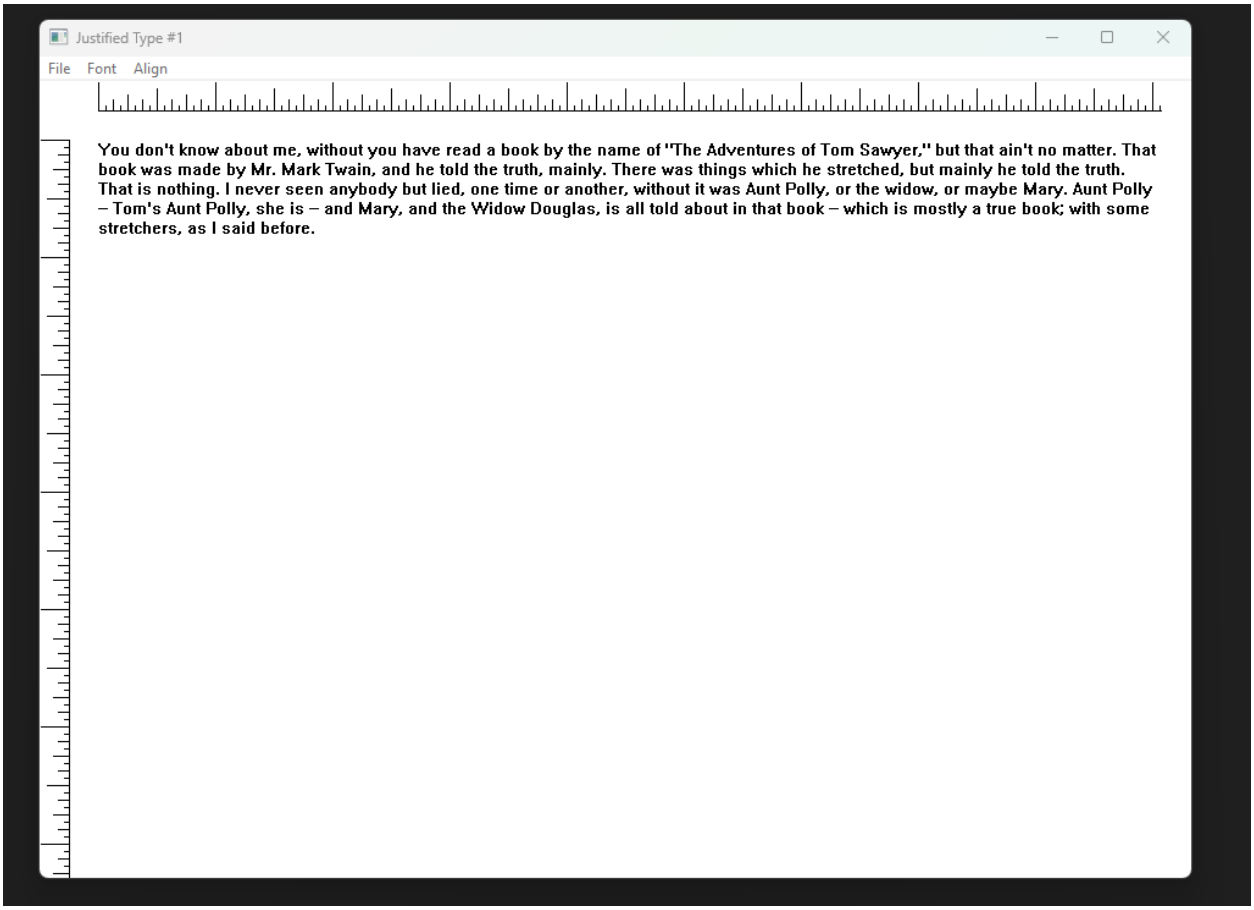
WM_PAINT: This message beckons the program to visually refresh the window's content. It involves:

- 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:

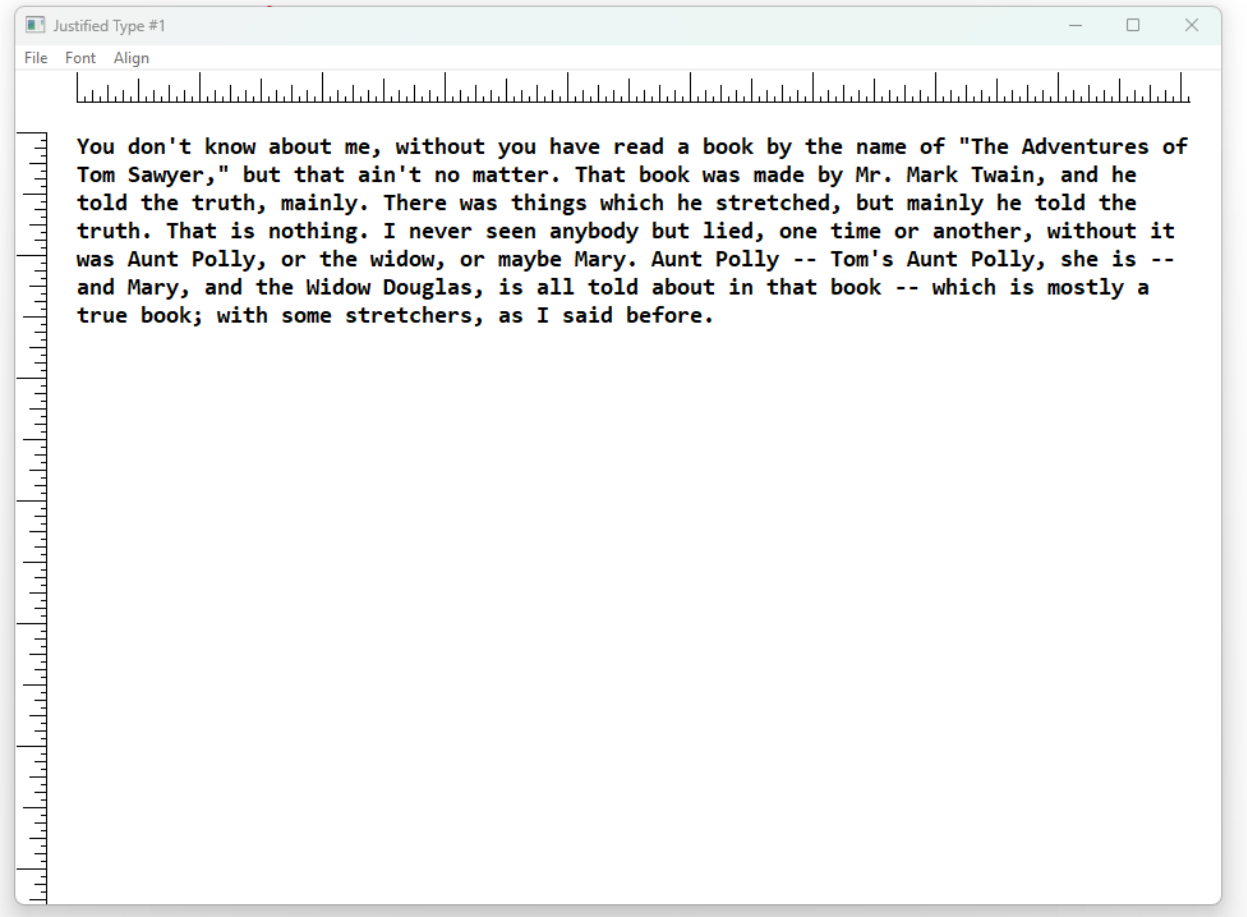
A loop iterates through ruler divisions, rendering tick marks at designated intervals. These tick marks enhance readability and provide more granular reference points for precise text alignment.

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:

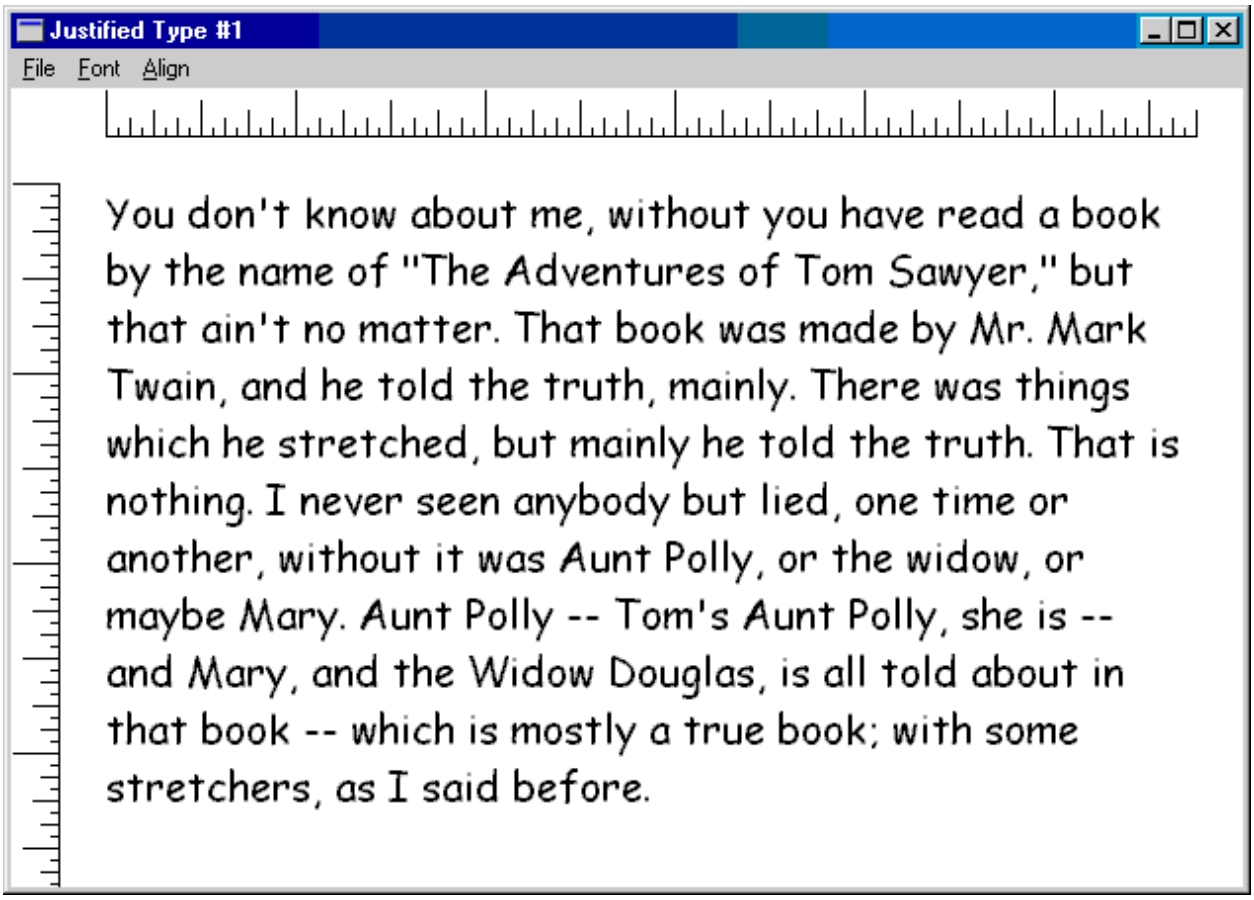


Justify function explained in depth:

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.

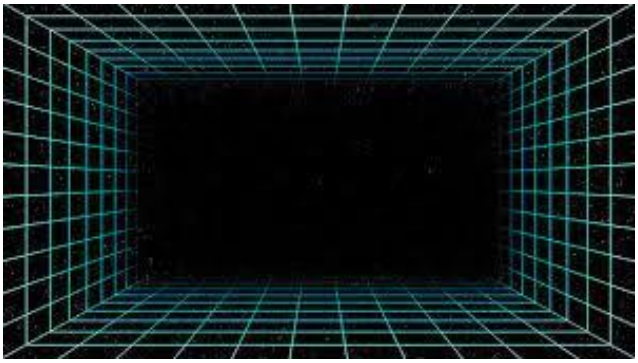
JUSTIFY2 offers valuable insights into techniques for precise text formatting, printer output previewing, TrueType font handling, and optimized text measurement. It serves as a practical example for Windows API developers seeking to address similar challenges in their projects.

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:

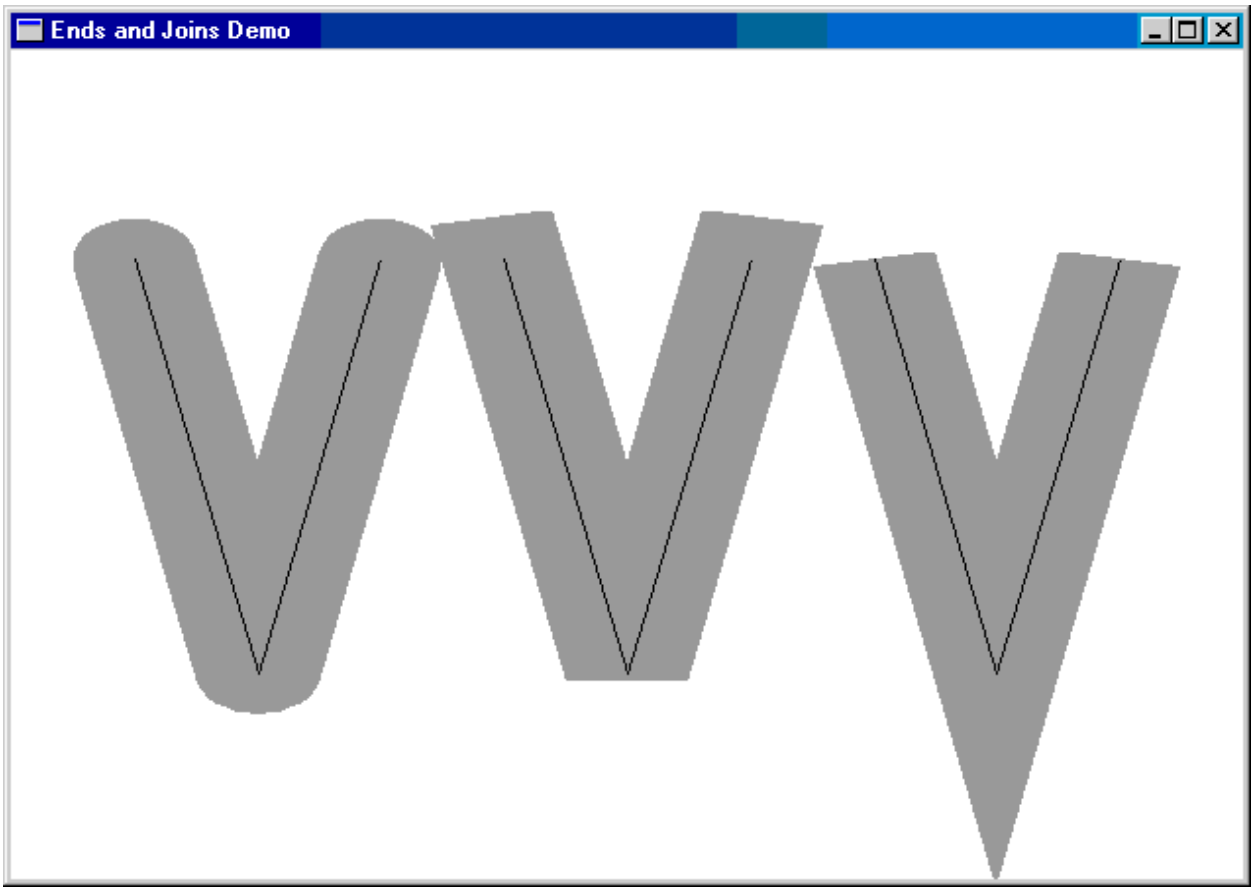
The function not only draws the V-shaped lines with varying styles but also provides a basis for comparison. It draws identical lines using the stock black pen, allowing the viewer to observe and compare the impact of wide lines with different end cap and join styles against regular thin lines.

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).

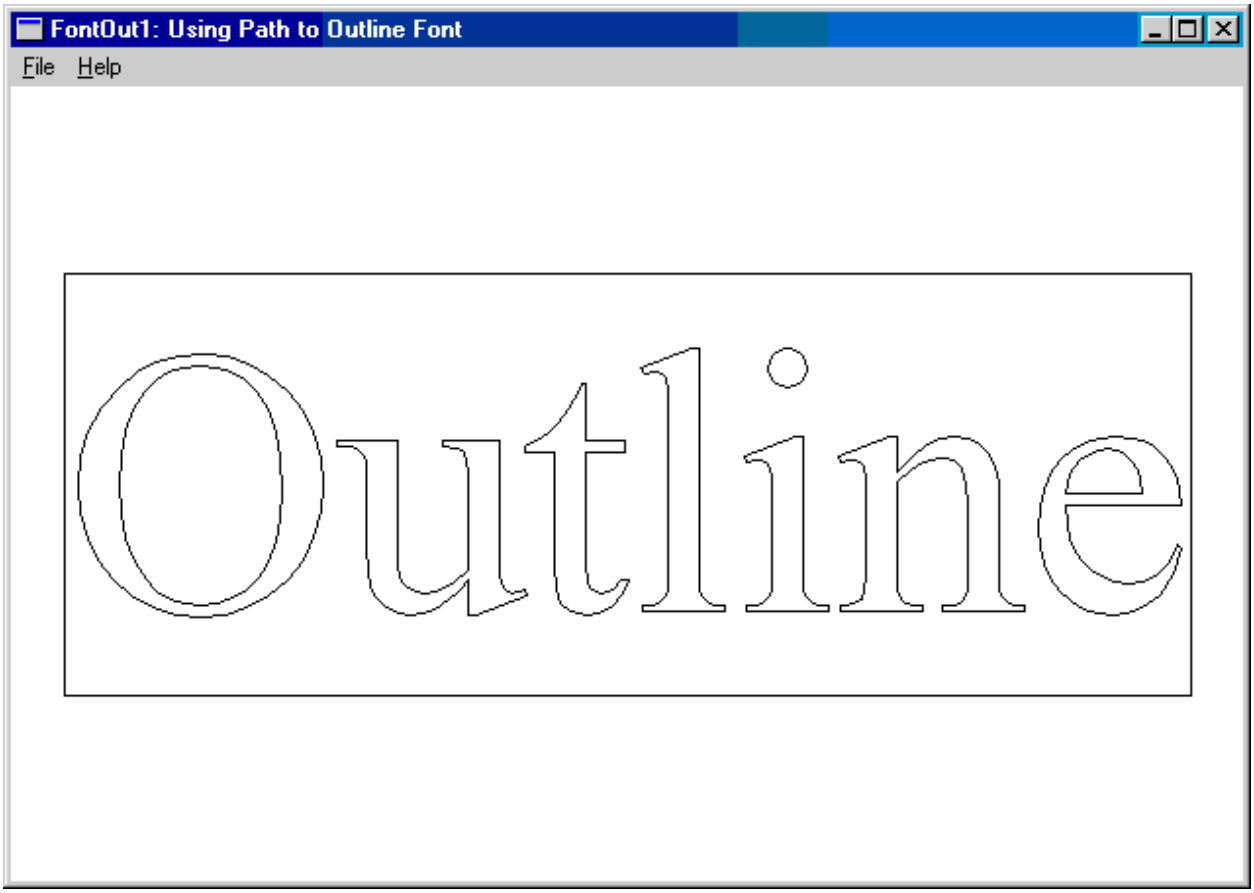
The EndPath function is called to finalize the path definition.

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.

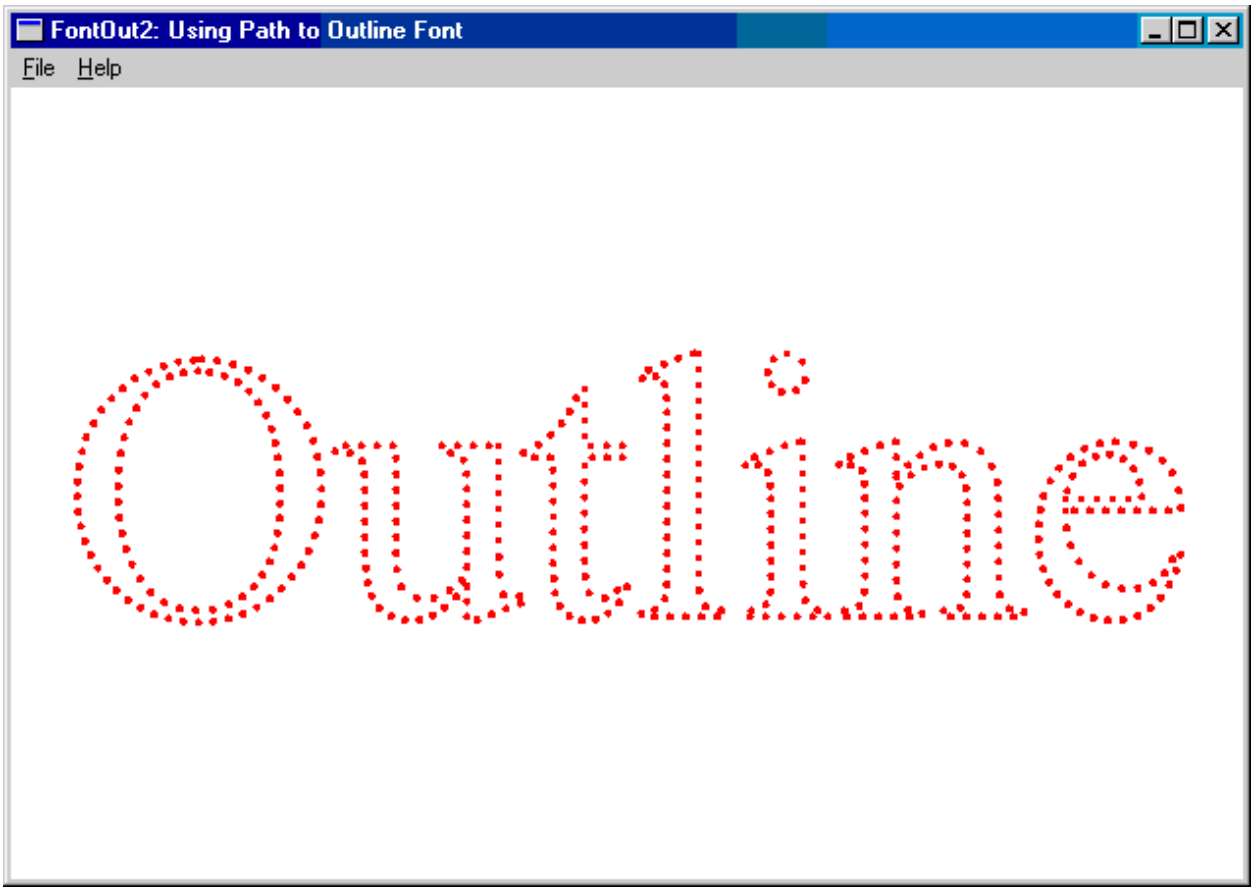
Preserving Resources: Once the display is complete, the default pen and font are restored, and temporary objects are released, ensuring a smooth transition for future artistic endeavors.

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:

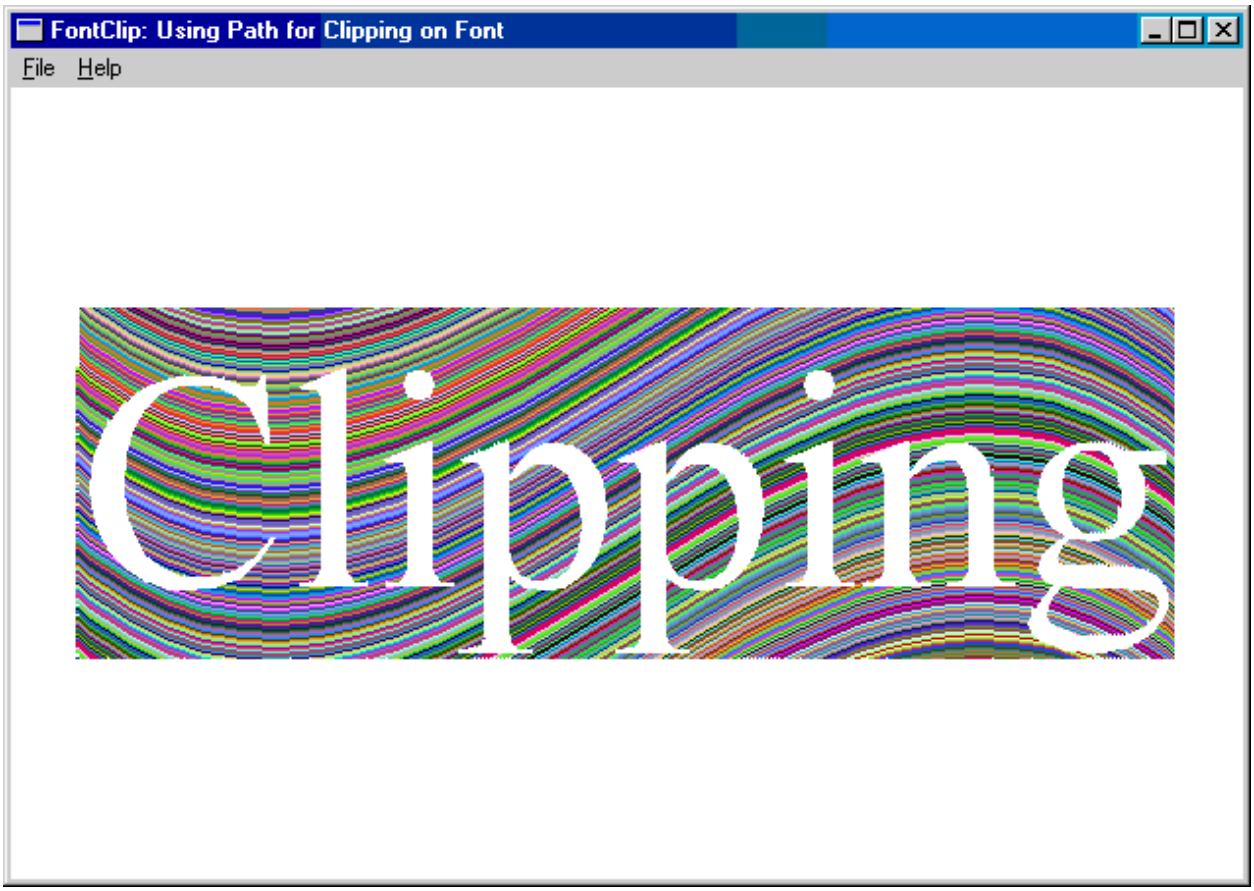
While other programs use clipping primarily for masking or hiding unwanted content, FONTCLIP.C highlights the clipping region itself as a central element of the artwork.

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….