WHAT IS A "STOCK" PEN?

The Concept: The Standard Issue Toolkit.

When you get a fresh Device Context (hdc), Windows hands you a default set of tools. The default pen is **Black, Solid, and 1 Pixel wide**.

However, Windows keeps a small inventory of other standard pens ready for you to use instantly. These are called **Stock Objects**. Because they are always available in memory, you don't need to "create" or "destroy" them; you just ask to borrow them.

The Three Main Stock Pens:

1. **BLACK_PEN**: Draws a solid black line (1 pixel wide). *This is the default.*
2. **WHITE_PEN**: Draws a solid white line (1 pixel wide). *Useful for erasing black lines on a white background.*
3. **NULL_PEN**: The "Invisible" Pen. It draws nothing. *Useful when you want to draw a shape (like a rectangle) with a filled interior but NO outline.*

The Two-Step Process

The Concept: Check Out and Equip.

Using a GDI object requires two distinct functions: finding the tool (GetStockObject) and putting it in your hand (SelectObject).

Step 1: Get the Handle (GetStockObject)

You cannot use the pen directly by name. You must ask Windows for its "ID card," which is called a **Handle** (HPEN).

```
// Ask Windows for the handle to the white pen
HPEN hPen = GetStockObject(WHITE_PEN);
```

Step 2: Select it into the DC (SelectObject)

Getting the handle doesn't change anything yet. You must strictly tell the Device Context to **equip** that pen.

```
// Tell the DC: "Put down the current pen and pick up this one."
SelectObject(hdc, hPen);
```

Pro Tip: You can combine these into a single line, which is very common in GDI programming: SelectObject(hdc, GetStockObject(WHITE_PEN));

The "Save and Restore" Pattern

The Concept: Respect the defaults.

When you use SelectObject, it swaps the tools. It puts your new pen into the DC and **returns the handle of the old pen** that was previously there.

It is considered "good manners" (and prevents bugs) to save that old pen and put it back when you are finished.

The Workflow:

1. **Save:** Equip the White Pen, but catch the old Black Pen in a variable.
2. **Draw:** Do your work with the White Pen.
3. **Restore:** Put the old Black Pen back.

Code Example:

```
HPEN hOldPen;

// 1. Equip White Pen, SAVE the old Black Pen
hOldPen = (HPEN)SelectObject(hdc, GetStockObject(WHITE_PEN));

// 2. Draw a White Line
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);

// 3. RESTORE the old Black Pen
SelectObject(hdc, hOldPen);

// Now we are back to normal (Black)
MoveToEx(hdc, 200, 200, NULL);
LineTo(hdc, 300, 300);
```

Why use NULL_PEN?

You might wonder why you would select a pen that doesn't draw. This is used for **Border-less Shapes**.

If you draw a Rectangle normally, it has a black border.

If you select NULL_PEN first, the Rectangle will have **no border**, only the filled color (the Brush).

The GDI Object Lifecycle.

The Concept:

Unlike Stock objects (which are permanent), Custom objects take up system memory. You are responsible for managing them.

Step 1: Create (CreatePen)

To make a new pen, you use the CreatePen function. It requires three arguments to define the pen's personality:

1. **Style:** Solid, Dashed, Dotted, etc. (e.g., PS_SOLID, PS_DASH).
2. **Width:** How thick the line is (in pixels).
3. **Color:** The RGB color value (e.g., RGB(255, 0, 0) for red).

```
// Create a solid red pen, 1 pixel wide
HPEN hPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
```

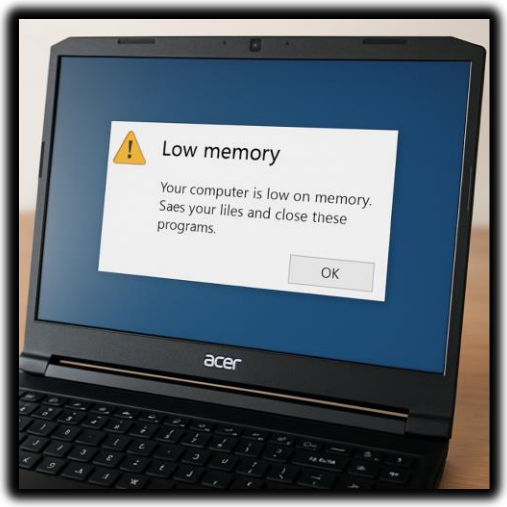
Step 2: Select (SelectObject)

Creating a pen doesn't make it active. It just sits in memory. You must "pick it up" using SelectObject.

- **The Rule:** A Device Context (DC) can only hold **one** pen at a time.
- **The Swap:** When you select your new pen, Windows puts down the old one and returns its handle to you.

Step 3: Delete (DeleteObject)

When you are done, you **must** destroy the pen to free up the memory. If you don't, your program will "leak" memory, eventually crashing Windows.



The Three Golden Rules of GDI

The Concept: Preventing Crashes.

Managing GDI objects is strictly enforced. Violating these rules is the #1 cause of graphics bugs.

- **Delete what you Create:** If you used CreatePen (or any Create... function), you must eventually call DeleteObject.
- **Never delete while active: Crucial:** You cannot delete a pen while it is currently selected into a Device Context. You must select the old pen back in (or select a Stock object) *before* you delete your custom pen.
- *Analogy:* You can't throw away the pen you are currently writing with.
- **Never delete Stock Objects:** Stock objects (like WHITE_PEN) belong to Windows. Do not try to delete them.

The GDI Family

The Concept: The 6 Tools.

A "Logical Pen" is just one member of the GDI Object family. All of these follow the same SelectObject / DeleteObject rules:

1. **Pens:** For drawing lines and borders.
2. **Brushes:** For filling areas.
3. **Fonts:** For text styles.
4. **Bitmaps:** For images.
5. **Regions:** For complex shapes and clipping.
6. **Palettes:** For color management (mostly obsolete now).

Corrected Code Example

The Concept: Putting it all together. The text provided had some typo errors in the code. Here is the corrected, working version of the lifecycle:

```
HPEN hNewPen, hOldPen;

// 1. CREATE: Solid Red Pen, 5 pixels thick
hNewPen = CreatePen(PS_SOLID, 5, RGB(255, 0, 0));

// 2. SELECT: Equip the new pen, SAVE the old one
hOldPen = (HPEN)SelectObject(hdc, hNewPen);

// 3. DRAW: Use the new pen
MoveToEx(hdc, 100, 10, NULL);
LineTo(hdc, 200, 200);

// 4. RESTORE: Put the old pen back (Crucial Step!)
SelectObject(hdc, hOldPen);

// 5. DELETE: Destroy the custom pen to free memory
DeleteObject(hNewPen);
```

THE 7 PEN STYLES

The Concept: Line Terminology. When you create a pen using CreatePen(Style, Width, Color), the first argument determines the pattern. There are 7 standard styles available in GDI.

- **PS_SOLID**: A continuous, unbroken line. This is the default.
- **PS_DASH**: A series of long dashes. -------
- **PS_DOT**: A series of small dots. .....
- **PS_DASHDOT**: Alternating dashes and dots. _._._._
- **PS_DASHDOTDOT**: Alternating dashes and double-dots. _.._.._
- **PS_NULL**: Invisible. Draws nothing. Used to disable borders.
- **PS_INSIDEFRAME**: A solid line that stays strictly *inside* the bounding box of a shape (more on this below).

---

1. The "One Pixel" Rule (Crucial Technical Detail)

The Concept: Why do my dashed lines look solid?

The text you provided contains a common confusion. In standard Windows GDI (using CreatePen), the patterned styles (PS_DASH, PS_DOT, etc.) **only work if the Pen Width is 1 or less.**

- **If Width = 1:** You get the pattern (dashes/dots).
- **If Width > 1:** Windows forces the style to be PS_SOLID (unless you use the advanced ExtCreatePen function or Windows NT-specific features).

Correction on Source Text: The source text suggests you can specify "dash length" or "spacing" directly in CreatePen (e.g., CreatePen(PS_DASH, 10, 5)). **This is incorrect syntax.** The standard CreatePen function only accepts three arguments: (Style, Width, Color). Custom spacing requires the advanced function ExtCreatePen.

---

2. PS_INSIDEFRAME Explained

The Concept: Avoiding the "Straddle."

This is the most unique style.

- **Normal Pens (PS_SOLID):** When you draw a shape (like a Rectangle) with a thick border (e.g., 10px), the line "straddles" the edge. 5 pixels are inside the box, and 5 pixels are outside.
- **Inside Frame (PS_INSIDEFRAME):** The entire thickness of the border is drawn *inside* the bounding box. This is critical when you need the shape to fit exactly within specific dimensions without the border spilling out.

---

3. What fills the Gaps?

The Concept: The Background Mode.

When you draw a dashed line, what color are the gaps between the dashes?

- **OPAQUE Mode:** The gaps are filled with the current background color (usually white).
- **TRANSPARENT Mode:** The gaps are left alone, showing whatever pixels were already on the screen behind the line.

5. Code Examples

The source text had significant syntax errors. Here is the corrected, functional code for the different styles.

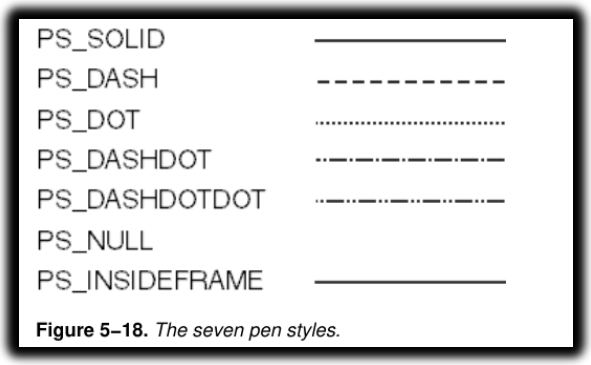


Figure 5–18. *The seven pen styles.*

A. Solid Pen (Thick)

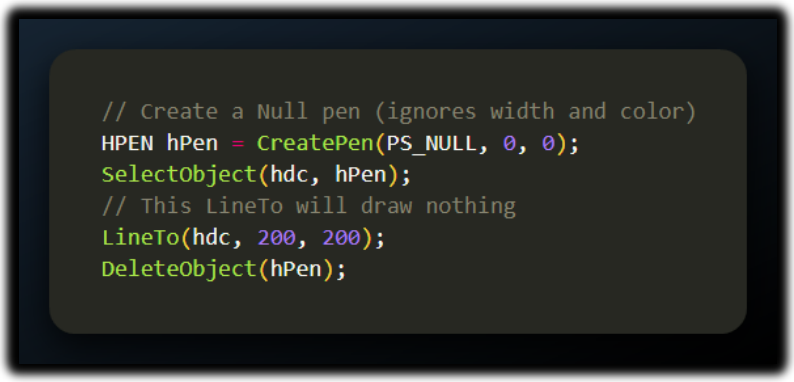
```
// Create a Red, Solid pen, 5 pixels thick
HPEN hPen = CreatePen(PS_SOLID, 5, RGB(255, 0, 0));
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
DeleteObject(hPen);
```

B. Dashed Pen (Standard)

Note: Width must be 1 for the pattern to appear.

```
// Create a Black, Dashed pen, 1 pixel thick
HPEN hPen = CreatePen(PS_DASH, 1, RGB(0, 0, 0));
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
DeleteObject(hPen);
```

C. The Null Pen (Invisible)



```
// Create a Null pen (ignores width and color)
HPEN hPen = CreatePen(PS_NULL, 0, 0);
SelectObject(hdc, hPen);
// This LineTo will draw nothing
LineTo(hdc, 200, 200);
DeleteObject(hPen);
```

---

6. Two Ways to Create a Pen

The Concept: Direct vs. Indirect.

While we previously used CreatePen, Windows offers a second method useful for more complex data handling.

- **CreatePen:** The standard function. You pass the style, width, and color directly as arguments.
- **CreatePenIndirect:** This function takes a single argument: a pointer to a **LOGPEN** ("Logical Pen") structure.
- **Why use it?** It is useful if you are storing pen settings in data structures or reading them from a file, as you can pass the entire structure at once rather than breaking it apart into arguments.

---

7. Pen Attributes: Width and Color

The Concept: Defining the Look.

Line Width (iWidth):

- ✓ **Positive Integer:** Sets the width in logical units (pixels).
- ✓ **Zero:** A special value. It tells Windows to draw a line that is **always 1 pixel wide**, regardless of any zoom level or mapping mode transformations.

Color (crColor):

- ✓ Specified as a COLORREF (a 32-bit value containing Red, Green, and Blue components).

---

8. The Special Case: Dithered Colors

The Concept: Faking colors.

Sometimes, you ask for a color that the screen cannot display perfectly (e.g., a specific shade of teal on a limited-color display). Windows normally approximates this using **Dithering** (a pattern of available pixel colors that looks like the desired color from a distance).

- **The Limitation:** Standard pens (Solid, Dashed, etc.) generally cannot use dithered colors; they snap to the nearest pure solid color.
- **The Exception: PS_INSIDEFRAME** is the **only** pen style that supports dithered colors. If you need a smooth, non-standard color for a border, you must use this style.

---

9. Performance Optimization: Static Pens

The Concept: Create once, use many times.

Creating and destroying pens (CreatePen / DeleteObject) every time you draw a line takes processing power. If your program uses a consistent set of pens (e.g., a "Grid Pen" and a "Border Pen"), you should optimize.

- **The Strategy:** define your pen handles as **static** variables and create them only once during your program's initialization.
- **The Benefit:** You avoid the overhead of repeated creation and deletion, making your application faster and more efficient.

```
static HPEN hPenGrid, hPenBorder; // Stored permanently

// In WM_CREATE (Initialization):
hPenGrid = CreatePen(PS_DOT, 1, RGB(200, 200, 200));
hPenBorder = CreatePen(PS_SOLID, 5, RGB(0, 0, 0));

// In WM_PAINT (Drawing):
SelectObject(hdc, hPenGrid);
// Draw grid...
SelectObject(hdc, hPenBorder);
// Draw border...

// In WM_DESTROY (Cleanup):
DeleteObject(hPenGrid);
DeleteObject(hPenBorder);
```

```
55  HPEN hPen1, hPen2, hPen3;  // Declare three pen handles
56
57  // Create three pens with different styles and attributes
58  hPen1 = CreatePen(PS_SOLID, 1, RGB(0, 0, 0));  // Solid black pen with width 1 pixel
59  hPen2 = CreatePen(PS_SOLID, 3, RGB(255, 0, 0));  // Solid red pen with width 3 pixels
60  hPen3 = CreatePen(PS_DOT, 0, RGB(0, 0, 0));  // Dotted black pen with default width 0 pixel
61
62  // Select the second pen into the device context (hPen2) for drawing
63  SelectObject(hdc, hPen2);
64  // Draw lines using the selected pen (hPen2)
65  [Line-drawing functions]
66
67  // Select the first pen into the device context (hPen1) for drawing
68  SelectObject(hdc, hPen1);
69  // Draw lines using the selected pen (hPen1)
70  [Line-drawing functions]
71
72  // Release the device context before deleting the pens
73  ReleaseDC(hWnd, hdc);
74
75  // Delete the pens that are no longer needed
76  DeleteObject(hPen1);
77  DeleteObject(hPen2);
78  DeleteObject(hPen3);
79
80  // Destroy the window
81  DestroyWindow(hWnd);
```

10. Two Strategies for Deleting Pens

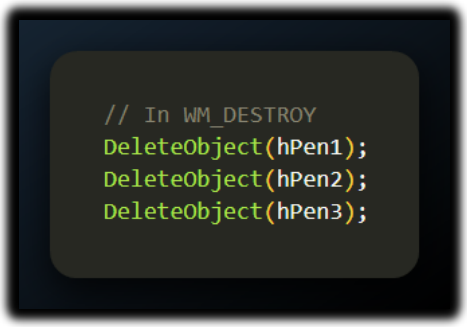
The Concept: When to clean up.

Since you must delete every custom pen you create, you need a strategy for *when* to do it.

Method A: The "Long Haul" (Global/Static)

Create your pens once when the program starts (or on first use) and keep them until the program closes.

- **Where to Delete:** In the WM_DESTROY message.
- **Pros:** Most efficient. You avoid the CPU overhead of creating/destroying tools 60 times a second.
- **Cons:** Uses more memory permanently. Requires you to track global variables for every pen.



Method B: The "Just-in-Time" (Local)

Create the pen right when you need it inside WM_PAINT, use it, and delete it immediately before finishing.

- **Where to Delete:** Inside WM_PAINT, just before EndPaint.
- **Pros:** Keeps memory usage low (the pen only exists for milliseconds). Flexible.
- **Cons:** Risk of "Leaking" if you forget to un-select the pen before deleting it.

11. The "One-Liner" Technique

The Concept: Compact Coding.

Advanced GDI programmers often combine creation, selection, and deletion into dense one-liners. This creates a "temporary" pen that exists only for the current operation.

The Creation: Instead of saving the handle in a variable, pass the creation function directly into the selection function.

```
// Create a Red Dashed pen and immediately equip it
SelectObject(hdc, CreatePen(PS_DASH, 0, RGB(255, 0, 0)));
```

The Deletion (The Magic Line): This looks confusing, but it is a standard GDI idiom.

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

How it works:

1. GetStockObject(BLACK_PEN): Grabs the safe, default pen.
2. SelectObject(...): Puts the Black Pen into the DC and **returns the handle of the previous pen** (the custom Red Dashed one we just used).
3. DeleteObject(...): Takes that returned handle (the Red Dashed one) and destroys it.

12. Retrieving Pen Information

The Concept: Introspection.

Sometimes you have a handle to a pen (or a DC) but you don't know what settings it has. Windows provides functions to inspect them.

GetObject

Extracts the physical details (Style, Width, Color) of a pen handle into a structure called LOGPEN (Logical Pen).

```
LOGPEN logpen;
GetObject(hPen, sizeof(LOGPEN), &logpen);
// Now you can read logpen.lopnColor, logpen.lopnWidth, etc.
```

GetCurrentObject

If you have a Device Context (hdc) but don't know which pen is currently equipped, this retrieves the handle.

```
// "What pen is this DC currently holding?"
HPEN hCurrentPen = GetCurrentObject(hdc, OBJ_PEN);
```

---

13. Advanced: ExtCreatePen

The Concept: The Geometric Pen.

The text briefly mentions ExtCreatePen. While standard pens are "Cosmetic" (fast, simple), ExtCreatePen allows for "Geometric" pens.

- **Features:** End caps (flat vs. round), Joins (beveled vs. mitered corners), and custom patterns.
- **Note:** This is an advanced topic reserved for **Chapter 17**.

PEN GAPS AND BACKGROUND MODES

1. The Question of "Empty" Space

The Concept: The gaps aren't empty by default.

When you draw a dashed line (e.g., -------), you are technically drawing two things:

1. **The Dashes:** Drawn using the Pen color.
2. **The Gaps:** Drawn using the **Background Color**.

By default, the Background Color is **White**. This is why if you draw a dotted line over a gray rectangle, you will see a "white box" around your dots, blocking out the gray behind it.

---

2. The Solution: Background Modes

The Concept: Opaque vs. Transparent.

You can control this behavior using the **Background Mode**. This setting tells Windows whether to "fill in" the gaps or leave them alone.

- **OPAQUE (Default):** Windows explicitly fills the gaps between dashes (and the background of text) with the current Background Color. It erases whatever was underneath.
- **TRANSPARENT:** Windows draws *only* the dashes (or the text characters). The gaps are left untouched, allowing any existing drawing behind the line to show through.

---

3. The Functions

To change these settings, you use the following functions:

- **SetBkMode(hdc, iMode)**:
  - ✓ iMode can be OPAQUE or TRANSPARENT.
- **SetBkColor(hdc, crColor)**:
  - ✓ Sets the color used for gaps when in OPAQUE mode.
  - ✓ *Note: This color is ignored if the mode is TRANSPARENT.*

4. Code Example

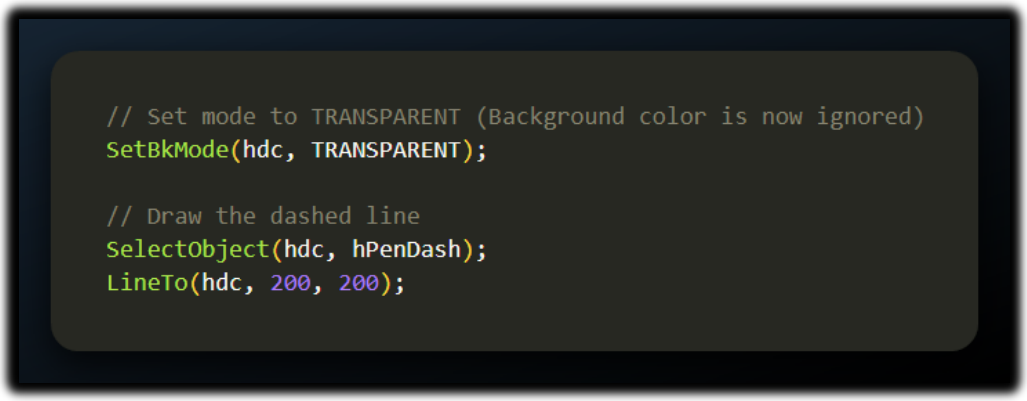
Scenario A: Red Gaps (Opaque) This draws a black dashed line, but the spaces between the dashes are filled with Red.

```
// Set background to RED
SetBkColor(hdc, RGB(255, 0, 0));

// Set mode to OPAQUE (force the red to appear)
SetBkMode(hdc, OPAQUE);

// Draw the dashed line
SelectObject(hdc, hPenDash);
LineTo(hdc, 200, 200);
```

Scenario B: See-Through Gaps (Transparent) This draws a black dashed line, and the spaces are invisible (showing whatever is behind).



```
// Set mode to TRANSPARENT (Background color is now ignored)
SetBkMode(hdc, TRANSPARENT);

// Draw the dashed line
SelectObject(hdc, hPenDash);
LineTo(hdc, 200, 200);
```

Pro Tip: This SetBkMode setting applies to **Text** as well! If you write text and it has an ugly white box around it, it's because your Background Mode is OPAQUE. Set it to TRANSPARENT to make the text blend seamlessly with the background.

DRAWING MODES AND RASTER OPERATIONS (ROPS)

Windows provides **16** standard ROP2 codes. They are named based on the Boolean logic they perform between the **Pen (P)** and the **Destination Screen (D)**.

ROP2 Code	Boolean Operation	Description
R2_BLACK	D & 0	Always draws black.
R2_WHITE	D & 1	Always draws white.
R2_NOTMERGEPEN	~ (D & P)	Inverts the pen color.
R2_MERGEPENDP	D	P
R2_NOTCOPYPEN	~P	Inverts the pen color.
R2_COPYPEN	P	Draws using the pen color.
R2_NOXORPEN	0	Does not draw.
R2_XORPEN	D ^ P	Performs an exclusive-OR operation.
R2_MASKPEN	D & P	Draws using the pen color, but only where the destination color is black.
R2_INVERT	~D	Inverts the destination color.
R2_NOMIRRORPEN	D & ~P	Draws using the pen color, but excludes the pen color where the destination color is black.
R2_MERGECOPYPEN	D & P	Draws using the pen color, but only where the destination color is white.
R2_MIRRORPEN	~P	Inverts the pen color.

1. The "Magic Eraser": R2_XORPEN

The Concept: Rubber Banding.

The most famous ROP2 code is R2_XORPEN. It has a unique mathematical property:

If you XOR a value twice, you get the original value back.

$$(A \oplus B) \oplus B = A$$

Why is this useful?

Imagine you are clicking and dragging the mouse to draw a selection rectangle (a "Rubber Band" box).

1. **Draw 1:** You draw the rectangle using R2_XORPEN. The colors look weird (inverted/mixed), but the shape is visible.
2. **Move Mouse:** To move the box, you must "erase" the old one.
3. **Draw 2:** You draw the **exact same rectangle** again at the old position using R2_XORPEN.
4. *Result:* The second XOR cancels out the first one. The screen is perfectly restored to its original image without you needing to repaint the whole window.

---

2. Setting the Drawing Mode

The Concept: Changing the Rules.

To switch from the simple "Overwriter" mode (R2_COPYPEN) to the "Magic Inverter" mode (R2_XORPEN), use the SetROP2 function.

```
int SetROP2(HDC hdc, int iDrawMode);
```

Returns: The previous drawing mode (so you can restore it later).

Code: Rubber Banding

```
// 1. Set the mode to XOR (Magic Erase Mode)
int oldROP = SetROP2(hdc, R2_XORPEN);

// 2. Select a WHITE pen (XORing with White inverts the colors nicely)
SelectObject(hdc, GetStockObject(WHITE_PEN));

// 3. Draw a line (This appears as inverted colors on screen)
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);

// ... Pause ...

// 4. Draw the SAME line again to "Erase" it
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
// The line is now gone, and the original background is back!

// 5. Restore the default drawing mode
SetROP2(hdc, oldROP);
```

Setting and Getting the Mode

The Concept: Controlling the Interaction.

Just like selecting a Pen, you must explicitly tell the Device Context (DC) which mathematical rule to use when drawing.

SetROP2(hdc, iDrawMode): Sets the new drawing mode.

- ✓ *Returns:* The previous mode (useful for undoing changes).

GetROP2(hdc): Checks what mode is currently active.

Summary of Key ROP2 Codes

The Concept: The Logic Table.

Here is the breakdown of the codes mentioned in your text, based on the Boolean logic symbols (P = Pen, D = Destination Screen):

ROP2 Code	Boolean Logic	Result
`R2_COPYPEN`	`P`	**(Default)** Draws the pen color directly. Overwrites everything.
`R2_NOTCOPYPEN`	`~P`	Draws the **inverse** of the Pen color. (e.g., White Pen draws Black).
`R2_BLACK`	`0`	Always draws **Black**.
`R2_WHITE`	`1`	Always draws **White**.
`R2_NOP`	`D`	**"No Operation"**. Does nothing. (Useful for debugging).
`R2_NOT`	`~D`	**Inverts the Screen**. Ignores the Pen color entirely.

ROP2 on Color Systems

The Concept: Bitwise Math.

On a monochrome (black and white) screen, "Inverting" is simple: Black becomes White. On a modern color screen, Windows performs the bitwise operation on **each bit** of the RGB value.

Example: The R2_NOT Mode This mode calculates ~Destination. It flips every bit of the screen's color.

- **Cyan Background:** 00000000 11111111 11111111 (Green + Blue)
- **Inverted (~):** 11111111 00000000 00000000 (Red only)
- **Result:** A Red line.

*(Note: Your source text mentions Cyan becoming Magenta. Mathematically, Cyan inverts to Red, and Green inverts to Magenta. The core concept remains: you get the **complementary** color).*

Why is R2_NOT useful? It guarantees visibility. If you draw with R2_COPYPEN (Black) on a Black background, the line is invisible. If you draw with R2_NOT, it inverts the background, ensuring the line always contrasts against what is behind it (unless the background is perfectly medium gray 128,128,128, where inverting changes nothing).

DRAWING FILLED AREAS

1. The Two-Tool System

The Concept: Outline + Interior.

Unlike drawing a simple line, drawing a geometric shape (like a square or circle) requires **two** separate tools at the same time:

- ✓ **The Pen:** Draws the **Border** (Outline).
- ✓ **The Brush:** Fills the **Interior** (Area).

When you call a shape function (like Rectangle), Windows looks at the Device Context (DC) to see which Pen and which Brush are currently selected, and it uses both.

2. Functions for Drawing Filled Shapes

Windows provides 7 standard functions that automatically use the current Pen and Brush.

WinAPI Function	Shape Type	Description
`Rectangle`	**Box**	Draws a standard rectangle using the current Pen and Brush. *(Note: `FillRect` is distinct—it only fills and requires a specific Brush handle).*
`Ellipse`	**Oval / Circle**	Draws an ellipse bounded by a specific rectangle coordinate set.
`RoundRect`	**Smooth Box**	A rectangle with rounded corners. Requires setting the width/height of the "rounding ellipse."
`Chord`	**Bow**	A closed curve with a flat side (like a bowstring). It connects the start and end of the arc with a straight line.
`Pie`	**Wedge**	A slice of pie. Connects the start and end of the arc to the **center point** of the ellipse.
`Polygon`	**Multi-sided**	A closed shape with straight edges (e.g., triangle, hexagon). Connects the last point to the first automatically.
`PolyPolygon`	**Complex**	Draws multiple distinct polygons in a single function call. Useful for complex, non-contiguous shapes.

Crucial Distinction: FillRect vs. Rectangle

- Rectangle() uses **both** the Pen and the Brush. It draws a border and fills it.
- FillRect() uses **only** the Brush. It paints the area but draws **no border**.

3. Mastering Visibility with "Null" Objects

Sometimes you want a shape with *only* a border (hollow), or *only* a fill (no outline). You achieve this using **Stock Objects**.

Scenario A: Hollow Shape (Border Only)

You want a black frame, but you want to see the background through the middle.

The Trick: Select the **NULL_BRUSH**.

Why? The default brush is White. If you draw a rectangle over an image, you get a big white block. Using NULL_BRUSH makes the inside transparent.

```
SelectObject(hdc, GetStockObject(NULL_BRUSH)); // Make inside transparent
Rectangle(hdc, 10, 10, 100, 100);
```

Scenario B: Borderless Shape (Fill Only)

You want a solid red square with no black outline.

The Trick: Select the **NULL_PEN**.

```
SelectObject(hdc, GetStockObject(NULL_PEN)); // Disable the border
SelectObject(hdc, hRedBrush);                // Select Red Fill
Rectangle(hdc, 10, 10, 100, 100);
```

4. Stock Brushes

Just like Pens, Windows keeps a stash of ready-to-use Brushes so you don't always have to create your own.

- **WHITE_BRUSH** (The Default)
- **BLACK_BRUSH**
- **LTGRAY_BRUSH** (Light Gray)
- **GRAY_BRUSH** (Medium Gray)
- **DKGRAY_BRUSH** (Dark Gray)
- **NULL_BRUSH** (Invisible)

To use one:

```
SelectObject(hdc, GetStockObject(LTGRAY_BRUSH));
```

Code Examples

Here are some code examples for drawing filled areas with borders:

```
// Draw a filled rectangle with a black outline
HBRUSH hBrush = GetStockObject(RED_BRUSH);
SelectObject(hdc, hBrush);

HPEN hPen = CreatePen(PS_SOLID, 2, RGB(0, 0, 0));
SelectObject(hdc, hPen);

FillRect(hdc, 100, 100, 200, 200);

// Draw a filled ellipse with a blue outline
hBrush = GetStockObject(GREEN_BRUSH);
SelectObject(hdc, hBrush);

hPen = CreatePen(PS_SOLID, 3, RGB(0, 0, 255));
SelectObject(hdc, hPen);

Ellipse(hdc, 300, 300, 400, 400);
```

These examples demonstrate the use of **FillRect** and **Ellipse** to draw filled shapes with borders. The code first selects the desired brush and pen into the device context, and then calls the respective drawing function to create the shape.

```
// Draw a filled rectangle with a black outline
HBRUSH hBrush = CreateSolidBrush(RGB(255, 0, 0)); // Create a solid red brush
SelectObject(hdc, hBrush); // Select the red brush into the device context

HPEN hPen = CreatePen(PS_SOLID, 2, RGB(0, 0, 0)); // Create a black pen with a width of 2 pixels
SelectObject(hdc, hPen); // Select the black pen into the device context

// Draw a filled rectangle with coordinates (100, 100), (200, 100), (200, 200), and (100, 200)
Rectangle(hdc, 100, 100, 200, 200);

// Draw a filled ellipse with a blue outline
DeleteObject(hBrush); // Delete the red brush to free up resources

hBrush = CreateSolidBrush(RGB(0, 255, 0)); // Create a solid green brush
SelectObject(hdc, hBrush); // Select the green brush into the device context

DeleteObject(hPen); // Delete the black pen to free up resources

hPen = CreatePen(PS_SOLID, 3, RGB(0, 0, 255)); // Create a blue pen with a width of 3 pixels
SelectObject(hdc, hPen); // Select the blue pen into the device context

// Draw a filled ellipse with coordinates (300, 300), (400, 300), (400, 400), and (300, 400)
Ellipse(hdc, 300, 300, 400, 400);

// Clean up resources
DeleteObject(hBrush); // Delete the green brush to avoid memory leaks
DeleteObject(hPen); // Delete the blue pen to avoid memory leaks
```

DRAWING POLYGONS AND FILLING MODES

1. The Polygon Function

The Concept: Connect the dots and fill. The Polygon function is the workhorse for creating custom shapes (triangles, stars, hexagons). It performs two actions automatically:

1. **Connects:** Draws lines connecting all points in your array.
2. **Closes:** If the last point is not the same as the first, it automatically draws a line back to the start to close the loop.
3. **Fills:** Fills the interior with the current Brush.

```
POINT pts[5] = { {100, 0}, {200, 100}, ... }; // 5 points defining a shape
Polygon(hdc, pts, 5);
```

---

2. The PolyPolygon Function

The Concept: Multiple shapes, one call.

If you need to draw a map (e.g., the islands of Hawaii), you have multiple distinct shapes that are part of one logical object. Instead of calling Polygon 5 times, you call PolyPolygon once.

Arguments:

lpPolyCounts: An array of integers telling Windows how many vertices belong to each shape.

Example: {3, 4, 5} means "The first 3 points make a triangle, the next 4 make a square, the next 5 make a pentagon."

---

3. The Big Question: What is "Inside"?

The Concept: Handling self-intersecting lines.

If you draw a simple square, the "inside" is obvious. But what if you draw a **5-pointed Star** (Pentagram) where the lines cross over each other? Is the center of the star "inside" the shape (filled) or "outside" (hollow)?

This depends on the **Polygon Filling Mode**.

---

4. The Two Filling Modes

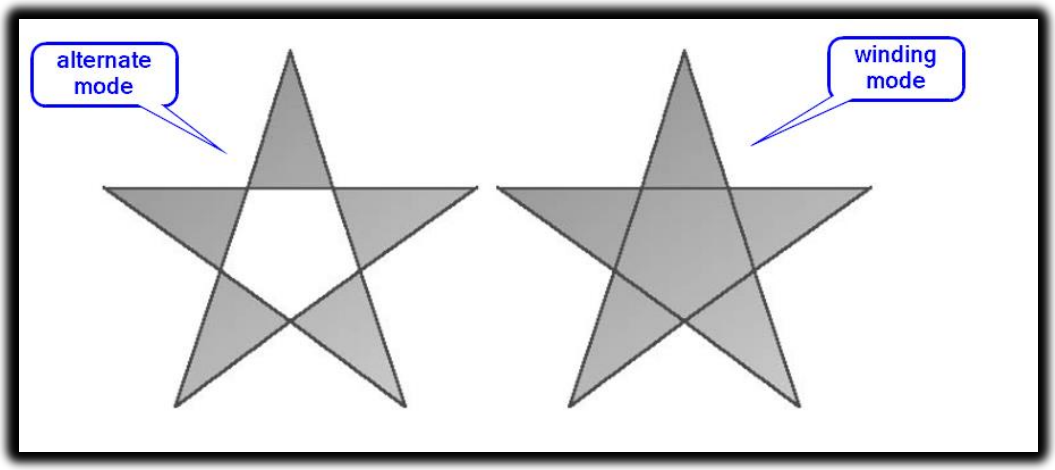
You can toggle between these modes using SetPolyFillMode(hdc, iMode).

Mode A: ALTERNATE (The Default)

Logic: The "Odd-Even" Rule. Imagine a ray of light shooting from the left of the screen to the right.

1. When the ray hits the first line of your shape, it enters the "Inside." (Turn Fill **ON**).
2. When it hits the next line, it exits to the "Outside." (Turn Fill **OFF**).
3. When it hits a third line, it enters the "Inside" again.

Visual Result (The Star): The center of a 5-pointed star will remain **HOLLOW**. The ray hits the first arm (On), hits the inner crossing (Off), and passes through the center unfilled.



Mode B: WINDING

Logic: The "Non-Zero" Rule. This is smarter but more computationally expensive. It tracks the **direction** the lines were drawn.

- Every time the imaginary ray crosses a line drawn **UP**, add 1.
- Every time it crosses a line drawn **DOWN**, subtract 1.
- **Rule:** If the total count is anything other than Zero, it is "Inside."

Visual Result (The Star): The center of the star is **FILLED**. Even though the lines cross, the math determines that the center is enclosed by the shape.

Here is how to switch modes to draw a filled star.

```
POINT apt[5] = { ... }; // Coordinates for a 5-pointed star

// 1. Set mode to WINDING (Fill the entire star, including center)
SetPolyFillMode(hdc, WINDING);
Polygon(hdc, apt, 5);

// 2. Set mode to ALTERNATE (Leave the center hollow)
SetPolyFillMode(hdc, ALTERNATE);
Polygon(hdc, apt, 5);
```

In most cases, winding mode will fill all enclosed areas of a single polygon. However, for complex polygons with self-intersections or holes, alternate mode may be more appropriate.

Winding Mode for Filling Enclosed Areas

While winding mode generally fills all enclosed areas of a single polygon, there are exceptions. To determine whether an enclosed area is filled in winding mode, follow these steps:

- Imagine a line drawn from a point inside the enclosed area to infinity.
- Count the number of times the polygon boundary lines cross this imaginary line.
- If the number of boundary line crossings is odd, the area is filled.
- If the number of boundary line crossings is even, the area is filled if the number of boundary lines going in one direction is not equal to the number of boundary lines going in the other direction.

Filling Enclosed Areas in Figure 5-20

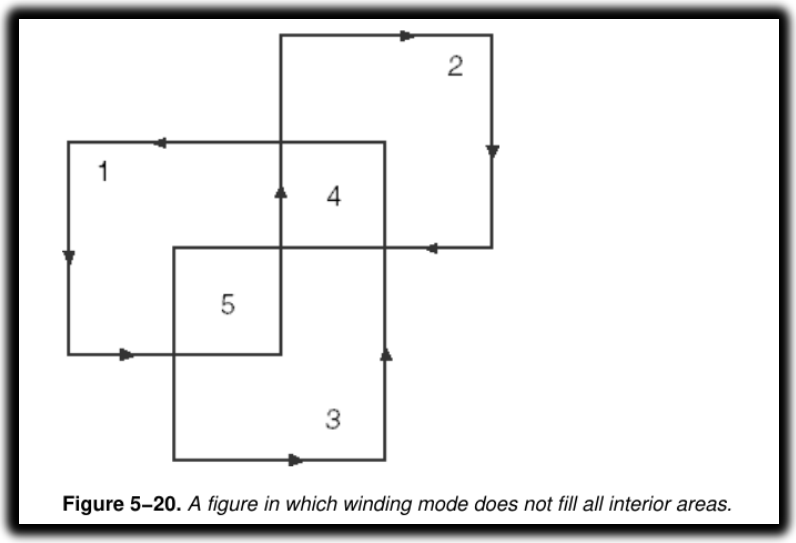
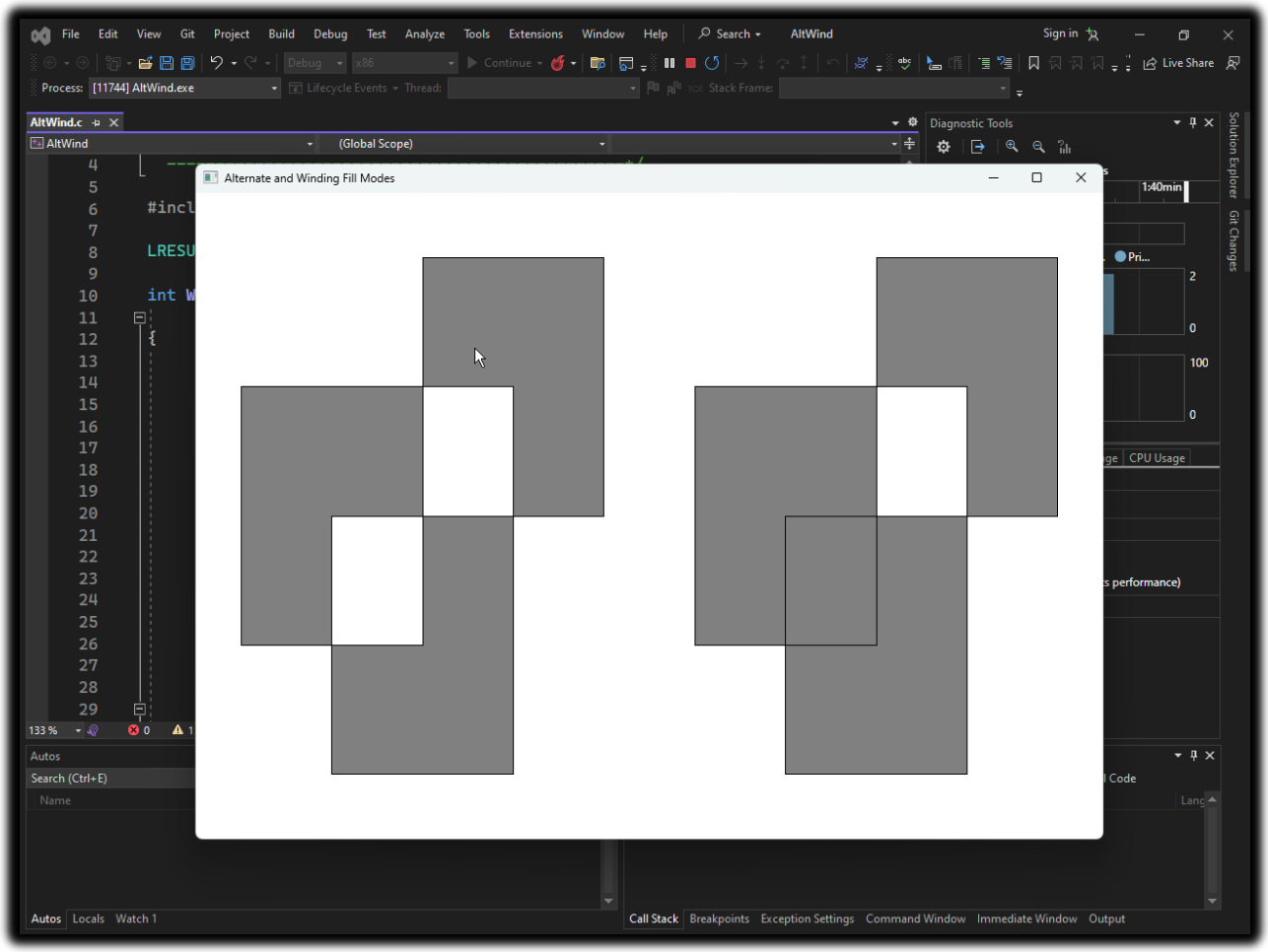


Figure 5–20. *A figure in which winding mode does not fill all interior areas.*

Applying these rules to the figure in Figure 5-20, we get the following results:

- **Enclosed areas 1, 2, and 3:** Both winding mode and alternate mode will fill these areas.
- **Enclosed area 4:** Winding mode will not fill this area. While the number of boundary line crossings is even (two), the two lines go in opposite directions, resulting in a winding number of zero.
- **Enclosed area 5:** Winding mode will fill this area. The number of boundary line crossings is even (two), but both lines go in the same direction, resulting in a winding number of two.

ALTWIND program in Figure 5-21 demonstrates that Windows correctly handles winding mode for filling enclosed areas. *Program source code in Chapter 5, altwind folder.*



BRUSHING THE INTERIORS OF SHAPES

1. What is a GDI Brush?

The Concept: The Wallpaper Roll.

When you fill a shape (like a rectangle or circle), Windows doesn't just "pour paint" into it. Instead, it uses a **Brush**.

- **Technically:** A Brush is a small **8x8 pixel bitmap**.
- **Behavior:** Windows "tiles" this small bitmap horizontally and vertically to fill the entire shape, much like wallpapering a wall.
- **Dithering:** On screens with limited color depth, Windows uses a pattern of different colored pixels (dithering) to simulate a color that doesn't exist in the hardware.

2. The 5 Ways to Create a Brush

Windows provides specific functions depending on whether you want a solid color, a line pattern, or a complex image.

A. Solid Brushes (CreateSolidBrush)

The simplest brush. It fills the shape with a single, uniform color.

Note: If the exact color isn't available on the hardware, Windows may dither it (mix pixels) to approximate the look.

```
// Create a solid Blue brush
HBRUSH hBrush = CreateSolidBrush(RGB(0, 0, 255));
```

B. Hatch Brushes (CreateHatchBrush)

Creates a pattern of lines (hatches). This is useful for charts, architectural drawings, or distinguishable areas in black-and-white prints.

Important: A Hatch Brush has two colors:

1. **The Lines:** Controlled by the crColor argument.
2. **The Gaps:** Controlled by SetBkColor and SetBkMode (Transparent/Opaque) of the Device Context.

The 6 Hatch Styles:

- HS_HORIZONTAL: -----
- HS_VERTICAL: |||||
- HS_FDIAG: //// (Forward Diagonal)
- HS_BDIAG: \\\\ (Backward Diagonal)
- HS_CROSS: ++++ (Grid)
- HS_DIAGCROSS: XXXX (Diagonal Grid)

```
// Create a Red Cross-Hatch brush
HBRUSH hBrush = CreateHatchBrush(HS_CROSS, RGB(255, 0, 0));
```

C. Pattern Brushes (CreatePatternBrush)

Allows you to use a custom image (Bitmap) as the brush.

- **Requirement:** You must first load or create a GDI Bitmap (HBITMAP).
- **Limitation:** The bitmap should ideally be 8x8 pixels. If larger, Windows usually only uses the top-left 8x8 chunk (depending on the OS version).

```
// Assumes hBitmap was loaded previously
HBRUSH hBrush = CreatePatternBrush(hBitmap);
```

D. DIB Pattern Brushes (CreateDIBPatternBrushPt)

Similar to Pattern Brushes, but uses a **Device Independent Bitmap (DIB)**. This allows you to specify the pixel bits directly in memory rather than using a GDI object handle. This is an advanced function used when carrying brush patterns between different computers or printers.

E. The "Master" Function (CreateBrushIndirect)

Instead of passing arguments directly, you fill out a structure called LOGBRUSH (Logical Brush) and pass it to this function. This is the most versatile method.

The LOGBRUSH Structure:

```
typedef struct tagLOGBRUSH {
  UINT     lbStyle;   // BS_SOLID, BS_HATCHED, BS_PATTERN, etc.
  COLORREF lbColor;   // Color of the brush (or hatch lines)
  ULONG_PTR lbHatch;  // Hatch style (HS_VERTICAL) or Handle to Bitmap
} LOGBRUSH;
```

```
void DrawBrushes(HDC hdc) {
    HBRUSH hBrush, hOldBrush;

    // --- 1. SOLID BRUSH ---
    // Create a Solid Blue Brush
    hBrush = CreateSolidBrush(RGB(0, 0, 255));

    // Select it into the DC (Save the old one!)
    hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);

    // Draw a rectangle (Filled Blue)
    Rectangle(hdc, 10, 10, 100, 100);

    // Clean up: Restore old brush, Delete new brush
    SelectObject(hdc, hOldBrush);
    DeleteObject(hBrush);

    // --- 2. HATCH BRUSH ---
    // Create a Red Diagonal Cross-Hatch Brush
    hBrush = CreateHatchBrush(HS_DIAGCROSS, RGB(255, 0, 0));

    // Set Background to YELLOW so the gaps aren't just white
    SetBkColor(hdc, RGB(255, 255, 0));
    SetBkMode(hdc, OPAQUE);

    SelectObject(hdc, hBrush);

    // Draw an Ellipse (Filled with Red X's on Yellow background)
    Ellipse(hdc, 120, 10, 220, 100);

    SelectObject(hdc, hOldBrush);
    DeleteObject(hBrush);

    // --- 3. INDIRECT BRUSH ---
    LOGBRUSH logBrush;
    logBrush.lbStyle = BS_HATCHED;
    logBrush.lbColor = RGB(0, 255, 0); // Green Lines
    logBrush.lbHatch = HS_VERTICAL;    // Vertical Lines

    hBrush = CreateBrushIndirect(&logBrush);

    SelectObject(hdc, hBrush);
    RoundRect(hdc, 240, 10, 340, 100, 20, 20);

    SelectObject(hdc, hOldBrush);
    DeleteObject(hBrush);
}
```

Summary of Brush Styles (lbStyle)

When using CreateBrushIndirect, you use these constants:

- **BS_SOLID**: Solid color.
- **BS_HOLLOW** (or BS_NULL): Transparent brush.
- **BS_HATCHED**: Hatch pattern.
- **BS_PATTERN**: Bitmap pattern.
- **BS_DIBPATTERN**: DIB pattern.

MANAGING BRUSHES: THE GDI LIFECYCLE

The Concept: The Golden Rule of Cleanup.

As with Pens, Brushes are GDI objects that consume system memory (GDI Heap).

If you create them but fail to delete them, your application will suffer from **Memory Leaks**, eventually crashing the graphics system.

Step 1: Selection (SelectObject)

Creating a brush doesn't automatically apply it. You must select it into the Device Context (DC).

```
// 1. Create
HBRUSH hBrush = CreateSolidBrush(RGB(255, 0, 0));
// 2. Select (and save the old one!)
HBRUSH hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
```

Step 2: Deletion (DeleteObject)

When you are done, you **must** delete the brush.

Critical Warning: Never delete a brush while it is currently selected in the DC.

```
// 1. Put the old brush back (Un-select your custom brush)
SelectObject(hdc, hOldBrush);
// 2. NOW it is safe to delete your custom brush
DeleteObject(hBrush);
```

Step 3: Introspection (GetObject)

If you have a handle to a brush (hBrush) but don't know what it looks like, you can retrieve its data into a LOGBRUSH structure.

```
LOGBRUSH lb;
GetObject(hBrush, sizeof(LOGBRUSH), &lb);
// Now you can check lb.lbColor or lb.lbStyle
```