

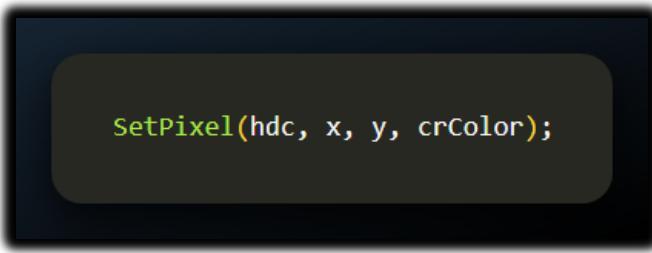
DRAWING LINES AND DOTS

1. SetPixel Function

The Concept: The "Pointillist's Touch."

SetPixel is the most fundamental building block of GDI. It doesn't draw a shape or a line; it changes the color of exactly **one** tiny dot on the screen.

Analogy: Think of a mosaic artist placing a single-colored tile, or an artist touching the canvas once with the very tip of a fine brush.



```
SetPixel(hdc, x, y, crColor);
```

- ✓ **hdc (The Canvas):** The permit telling Windows *where* you are allowed to draw.
- ✓ **x & y (The Coordinates):** The specific address of the pixel (e.g., "5th row, 10th column").
- ✓ **crColor (The Paint):** The specific color ID (COLORREF) you want to use.

The "Nearest Color" Rule: Remember our earlier talk about GetNearestColor? SetPixel uses that logic automatically.

- **Scenario:** You ask for "Neon Ghost Green."
- **The Hardware:** "I can't display that."
- **The Result:** SetPixel won't try to fake it with a pattern (dithering). It snaps to the closest **solid** color available to keep the pixel sharp.

2. GetPixel Function

The Concept: The "Eyedropper Tool."

If SetPixel is writing to the screen, GetPixel is reading from it. It looks at a specific coordinate and tells you what color is currently there.

Analogy: Think of the **Eyedropper tool** in MS Paint or Photoshop. You click a spot on the image to "sample" the color so you can use it somewhere else.



```
crColor = GetPixel(hdc, x, y);
```

- ✓ **x & y:** The address you want to investigate.
 - ✓ **Return Value:** It hands you back the COLORREF (the color ID card) for that specific spot.
-

3. The Trade-off

While these functions are the "atoms" of drawing, they are **slow** if you use them to draw big pictures.

- **Why?** Imagine painting a wall by making millions of tiny dots, one at a time. It takes forever.
- **Usage:** These are best used for precise, small-scale work (like drawing a graph line pixel-by-pixel or reading a specific color value), not for filling huge backgrounds.

4. Limitations of SetPixel and GetPixel

The Concept: The "Micro-Management" Trap.

While having total control over every dot sounds great, in practice, it is often a trap.

- **Performance Overhead:**

- ✓ **Analogy:** Imagine trying to move a pile of sand using **tweezers** (SetPixel). It works, but it takes forever.
- ✓ **The Reality:** Drawing a line with **SetPixel** requires the CPU to wake up, check permissions, and talk to the video card thousands of times for a single line.
- ✓ **The Better Way:** Functions like **LineTo** or Polyline are like using a **bulldozer**. You give one command ("Move this sand"), and the specialized hardware (Graphics Card) does it instantly in one go.

- **Device-Dependent Colors:**

- ✓ **Analogy:** Speaking a local dialect.
- ✓ **The Reality:** A COLORREF is just a raw number. If you bypass Windows' color matching tools, "Green" on your monitor might look like "Teal" on a projector. High-level functions handle this translation for you; direct pixel access often does not.

5. Alternative Graphics Approaches

The Concept: "Delegating to the Experts."

For 99% of tasks, you should stop trying to manipulate individual pixels and let GDI do the heavy lifting.

A. High-Level Functions (The Standard)

- **What they are:** Commands like Rectangle, Ellipse, or TextOut.
- **Why use them:**
 - ✓ **Speed:** They use "Hardware Acceleration" (using the GPU instead of the CPU).
 - ✓ **Consistency:** They ensure colors look the same on all screens (Device Independence).

B. Raster Operations (The "Smart" Way)

- **What they are:** Mathematical rules for combining pixels.
- **Analogy:** Instead of painting over a wall, think of using a **colored filter** or a **stencil**.
- **Usage:** You can tell Windows: "Take this whole block of pixels and *invert* their colors." It modifies thousands of pixels instantly using a single math rule (ROP), which is much faster than a loop of SetPixel.

C. Custom Drawing Functions (The "Expert" Way)

- **What they are:** Writing your own optimized algorithms.
 - **Usage:** If you are building a game engine or a photo editor, the standard GDI tools might be too simple. In this case, developers write custom code (often manipulating memory directly) to achieve specific effects that standard GDI functions can't handle.
-

LINE DRAWING FUNCTIONS IN WINDOWS GDI

1. LineTo Function

The Concept: "Connect the Dots."

The LineTo function is the standard way to draw a straight line. It relies entirely on the **Current Position** (where your pen is currently hovering).

- **How it works:** You tell Windows, "Draw a line from *wherever I am now to this new spot.*"
- **The behavior:**
 1. The pen touches the paper at the **Current Position**.
 2. It draws a straight line to the (x, y) coordinates you provide.
 3. **Crucial Step:** The pen *stays* at that new end point. The "Current Position" is updated to (x, y).
- **Analogy:** It's like walking while dragging a stick in the sand. You stop at point B, and if you start walking again, the next line starts from point B.

```
LineTo(hdc, x, y);
```

2. Polyline and PolylineTo Functions

The Concept: The "Continuous Stroke."

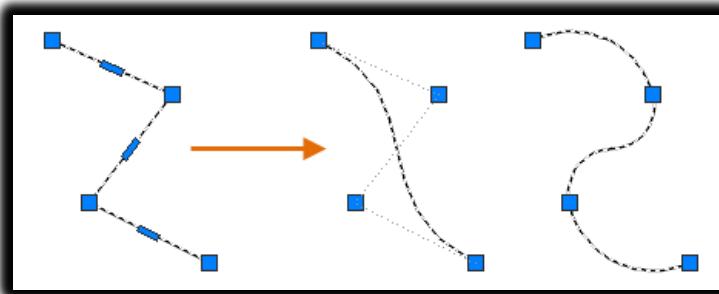
If LineTo is drawing one segment at a time, Polyline functions are for drawing a jagged path (like a lightning bolt or a graph) in one smooth command.

A. Polyline

- **The Rule:** "Ignore where I was; start fresh here."
- **How it works:** You give it a list of points (Coordinate A, B, C, D). It draws lines connecting A→B→C→D.
- **Key Detail:** It does **not** use or update the Current Position. It is a standalone drawing operation.

B. PolylineTo

- **The Rule:** "Continue from where I left off."
- **How it works:** It starts drawing from the **Current Position** to the first point in your list, then continues to the rest.
- **Key Detail:** It **updates** the Current Position to the very last point when it finishes.



```
Polyline(hdc, lpPoints, cCount);  
PolylineTo(hdc, lpPoints, cCount);
```

Syntax Breakdown:

- **lpPoints:** The list of coordinates (The "Map").
- **cCount:** How many points are on that list.

Clarification on "Closed" vs "Open":

- While your notes mention PolylineTo defines a "closed" polyline, technically it just draws a continuous chain of lines.
- If you want the shape to automatically close itself (drawing a line back to the start to form a loop), GDI uses a function called Polygon.
- Polyline and PolylineTo usually leave the shape "open" unless you manually list the starting point again at the end.

Function	Starts From...	Updates Current Position?	Usage
LineTo	Current Position	Yes	Drawing single segments one by one.
Polyline	First point in array	No	Drawing a shape without messing up your "cursor" location.
PolylineTo	Current Position	Yes	Extending an existing drawing with a complex path.

C. PolyPolyline Function

The Concept: The "Batch Job."

If Polyline draws one continuous shape (like a lightning bolt), PolyPolyline draws multiple separate shapes in a single command.

- **The Problem:** Drawing a map with 50 separate roads using Polyline requires calling the function 50 times. This is slow.
- **The Solution:** PolyPolyline lets you hand Windows a single list of all the roads at once.
- **How it works:** You give it a massive list of points, and a second list telling it how to break them up (e.g., "The first 3 points form one line; the next 4 points form a separate line...").
- **Analogy:** Instead of mailing 10 separate letters (using 10 stamps), you put all 10 letters into one big package. It's faster and cheaper.

```
PolyPolyline(hdc, polyPoints, nCount);
```

2. Factors Affecting Line Appearance

The Concept: The "Atmospheric Conditions."

When you draw a line, it doesn't just depend on the coordinates. Five invisible settings in the Device Context (DC) change how that line actually appears on the screen.

A. Current Pen Position

- **What it is:** The "Starting Block."
- **Applies to:** LineTo, PolylineTo, PolyBezierTo, ArcTo.
- **Effect:** For these functions, you don't say *where* to start; they automatically start wherever the last drawing finished. If you don't track this, your lines might start in the wrong place.

B. The Pen

What it is: The "Physical Tool."

Effect: This is the most obvious factor. It dictates:

- **Width:** Is it a fine tip or a marker?
- **Color:** Red, Blue, Black?
- **Pattern:** Solid, Dashed, or Dotted?

C. Background Mode

What it is: The "Gap Policy."

Relevance: This is crucial for **Dashed** or **Dotted** lines. It decides what happens in the empty spaces between the dashes.

- **OPAQUE:** "Fill the gaps." Windows paints the spaces between dashes with the Background Color.
- **TRANSPARENT:** "Leave the gaps." Whatever image was behind the line shows through the gaps.

D. Background Color

- **What it is:** The "Gap Filler Paint."
- **Effect:** If (and only if) you are in OPAQUE mode, this is the color used to fill the spaces between dots or dashes.

E. Drawing Mode (ROP2)

What it is: The "Chemical Reaction."

Effect: It determines how the ink mixes with the paper.

- **Standard (R2_COPYPEN):** The new line simply covers up whatever was there.
- **XOR (R2_XORPEN):** The new line inverts the colors underneath it. This is often used for "rubber band" selection boxes that need to remain visible on both white and black backgrounds.

DRAWING STRAIGHT LINES WITH MOVETOEX AND LINETO

1. The Concept: The "Lift" and The "Draw"

In Windows GDI, drawing a line isn't just one command. It mimics how a human draws on physical paper:

1. **Lift** your hand and move it to a starting spot (without making a mark).
2. **Press** down and draw to the end spot.

This is why MoveToEx and LineTo are almost always used together.

2. MoveToEx Function

The Concept: The "Hovering Hand."

MoveToEx is the command to move the pen **without** touching the paper.

- **What it does:** It updates the **Current Position** (the invisible cursor) to a specific set of coordinates.
- **Crucial Note:** It does **not** draw anything. It simply tells Windows, "If I were to start drawing right now, I would start *here*."

```
MoveToEx(hdc, xBeg, yBeg, NULL);
```

- **hdc:** The Canvas/Permit.
- **xBeg & yBeg:** The coordinates where you want to "hover" your pen.
- **NULL:** This last parameter is the "Receipt."

Explanation: If you provide a pointer here, Windows will write down the *old* position before moving to the new one. Most of the time, you don't care where the pen *was*, so you just pass NULL (ignore it).

3. The Workflow (Putting it Together)

The Concept: Connect the dots.

Since LineTo always draws from the "Current Position," you must set that position first.

The Sequence:

1. **Step 1 (MoveToEx):** "Start here." (Sets Current Position to Point A).
2. **Step 2 (LineTo):** "Draw to here." (Draws a line from A → B).
3. **The Aftermath:** The pen is now resting at Point B.
 - ✓ *If you call LineTo again:* It will draw from B → C.

Summary:

- **MoveToEx:** Sets the start.
- **LineTo:** Draws the line and sets the new start for the *next* line.

```
LineTo(hdc, xEnd, yEnd);
```

- **hdc:** The Canvas/Permit.
- **xEnd & yEnd:** The destination coordinates.

Behavior:

- **Starts:** It begins exactly where the "Current Position" was left (usually set by MoveToEx).
 - **Ends:** It stops at the coordinates you provide (xEnd, yEnd).
 - **Updates:** The "Current Position" is moved to the end of this new line, ready for the next segment.
-

4. Practical Example: Drawing a Grid

The Concept: The "Weaving" Technique.

Drawing a single line is easy. Drawing a grid (like a checkerboard or graph paper) requires a systematic loop. You are essentially "weaving" the lines: first all the horizontal ones, then all the vertical ones.

The Logic:

To draw a grid, you need two separate loops:

1. The Rows (Horizontal Lines):

- Start at the top-left edge (\$x=0\$).
- Draw a line straight across to the right edge.
- Move down a specific amount (e.g., 100 pixels).
- Repeat.

2. The Columns (Vertical Lines):

- Start at the top-left edge (\$y=0\$).
- Draw a line straight down to the bottom edge.
- Move right a specific amount. Repeat.

5. The Code in Action

The Concept: Automation.

Here is the code snippet that automates this process. It uses GetClientRect to measure the window size so the grid fits perfectly.

Breakdown of the Code:

Measuring the Canvas (GetClientRect): Before drawing, the code asks the window:

"How big are you right now?"

```
#include <windows.h>

// Assuming hwnd, hdc, and rect are properly defined elsewhere in your code.

GetClientRect(hwnd, &rect);

// Draw horizontal lines
for (int x = 0; x < rect.right; x += 100)
{
    MoveToEx(hdc, x, 0, NULL);
    LineTo(hdc, x, rect.bottom);
}

// Draw vertical lines
for (int y = 0; y < rect.bottom; y += 100)
{
    MoveToEx(hdc, 0, y, NULL);
    LineTo(hdc, rect.right, y);
}
```

It stores the width (rect.right) and height (rect.bottom).

The Horizontal Loop (First for loop):

- **MoveToEx(hdc, 0, y, NULL):** "Lift pen, move to the left edge at height y."
- **LineTo(hdc, rect.right, y):** "Draw straight across to the right edge."
- **y += 100:** "Move down 100 pixels for the next line."

The Vertical Loop (Second for loop):

- **MoveToEx(hdc, x, 0, NULL):** "Lift pen, move to the top edge at position x."
- **LineTo(hdc, x, rect.bottom):** "Draw straight down to the bottom."
- **x += 100:** "Move right 100 pixels for the next line."

To wrap up the code you just saw:

- The snippet effectively draws a mesh of vertical and horizontal lines.
 - It uses a standard "spacing" of 100 pixels to create a clean, graph-paper look within the window's client area.
-

6. The Evolution of Coordinates (16-bit vs. 32-bit)

The Concept: The "Sketchpad" vs. The "Infinite Canvas."

In the old days of Windows (like Windows 3.1 or 95), coordinates were often limited to **16-bit** numbers.

Old Limitation: Think of drawing on a standard sheet of Letter paper. If you tried to draw a line past the edge (beyond 32,767 pixels), the math would break, and the line would wrap around or disappear.

Modern Windows (10 & 11):

- **The Upgrade:** Windows now uses full **32-bit values** for GDI coordinates.
 - **Analogy:** Instead of a sheet of paper, you have a **satellite map** the size of a continent.
 - **The Math:** You can now specify positions ranging into the *billions* of pixels.
-

7. Why This Matters? (Precision & Scale)

The Concept: Micro-Surgery and Skyscrapers.

This expanded range isn't just about making "bigger" pictures; it's about **flexibility**.

- **Precision (Zooming In):** Applications like **CAD software** or **Architectural tools** need to draw tiny details (like a screw on a bridge). With a huge coordinate range, they can treat "1 pixel" as a tiny fraction of a millimeter without running out of numbers.
- **Scale (Zooming Out):** **High-resolution image editors** can handle 8K or 16K images without crashing or glitching because the coordinate system doesn't "hit a wall" at the edge of the screen.

Summary: Windows 10 and 11 have removed the "electric fence" at the edge of the coordinate system. You now have a versatile, expansive canvas that allows you to draw virtually anything, of any size, without worrying about running out of space.

8. Retrieving the Current Position

The Concept: The "GPS Check."

Sometimes, your code draws a complex path and "forgets" where the pen ended up. You can ask Windows for the current coordinates.

- **Function:** GetCurrentPositionEx
- **Usage:** You provide a blank POINT structure, and Windows fills it with the current coordinates.

```
POINT pt;  
GetCurrentPositionEx(hdc, &pt);  
// Now pt.x and pt.y hold the current pen location
```

9. Choosing the Right Function (A Cheat Sheet)

Picking the right tool for the job. Not all lines are created equal.

Here is how to decide which function to use:

Individual Lines (MoveToEx + LineTo):

- *Best for:* Simple, disconnected strokes.
- *Analogy:* Lifting the pen after every single mark.

Series of Connected Lines (Polyline):

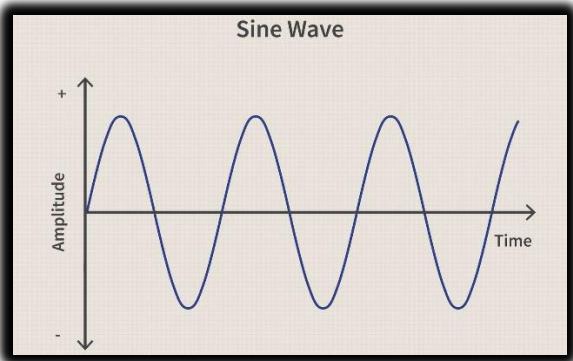
- *Best for:* Complex shapes or graphs with hundreds of points (like a Sine Wave).
- *Benefit:* It is much faster than calling LineTo 1000 times. Windows optimizes the math in one batch.

Continuing a Path (PolylineTo):

- *Best for:* When you want to add a complex shape to the *end* of an existing drawing without breaking the flow.

10. Application: Drawing a Sine Wave

The Concept: "Curve via Segments."



Computers can't actually draw true curves; they draw tiny straight lines that *look* like a curve. The **SINEWAVE** program uses Polyline to connect 1000 tiny points, creating a smooth visual.

A. Preprocessor Directives and Constants: Defining the math rules.

The code starts by grabbing the necessary tools (windows.h for graphics, math.h for the sin() function). It also sets up the constants to make the code readable.

- **NUM:** The resolution of the wave (1000 points).
- **TWOPI:** A math helper for the circle calculation ($2 * \pi$)

```
#include <windows.h>
#include <math.h>

#define NUM    1000
#define TWOPI  (2 * 3.14159)
```

B. The WinMain Function

The "Launchpad."

WinMain is the ignition key for any Windows application. It doesn't do the drawing; it builds the workshop where drawing happens.

1. Registers the Window Class:

- *Action:* Fills out a form (WNDCLASS) telling Windows, "I want a window that looks like *this* and behaves like *that*."

2. Creates the Window:

- *Action:* Calls CreateWindow. This allocates the memory and gives the window a handle (hwnd).

3. Shows the Window:

- *Action:* Calls ShowWindow. The window is now visible on the monitor.

4. Updates the Window:

- *Action:* Calls UpdateWindow. This triggers the very first "Paint" command, ensuring the sine wave appears immediately.

5. Message Loop:

- *Action:* Enters a while loop using GetMessage. It sits and waits for the user to click or type, dispatching those events to the right place.

6. Returns Exit Code:

- *Action:* When the user closes the app, the loop breaks, and the program shuts down cleanly.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT("SineWave");
    HWND        hwnd;
    MSG         msg;
    WNDCLASS   wndclass;

    // 1. Register Class (Properties)
    wndclass.style      = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfWndProc = WndProc; // The function that actually draws
    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))
        return 0;

    // 2. Create the Window
    hwnd = CreateWindow(szAppName, TEXT("Sine Wave Using Polyline"),
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL);

    // 3 & 4. Show and Update
    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);

    // 5. Message Loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // 6. Return
    return msg.wParam;
}
```

C. The WndProc Function

The "Brain" of the Window.

While WinMain just built the building, WndProc is the manager sitting at the desk, handling every phone call (message) that comes in. It decides what to do when the window resizes, needs painting, or closes.

D. Handling WM_SIZE

The "Tape Measure."

When you grab the corner of the window and drag it, Windows sends a WM_SIZE message.

- **Action:** The function immediately measures the new width (cxClient) and height (cyClient).
 - **Why:** We need these numbers so we can stretch the sine wave to fit perfectly inside the new window size.
-

E. Handling WM_PAINT (The Core Logic)

The Concept: The "Artist's Routine."

This is the most important part. It runs every time the window needs to be shown.

Step A: The Axis Line

First, we draw a straight horizontal line right through the middle of the screen.

- **MoveToEx:** Start at the far left (0, cyClient / 2).
- **LineTo:** Draw to the far right (cxClient, cyClient / 2).

Step B: The Math Loop (Calculating the Wave)

We don't draw the curve immediately. First, we calculate the coordinates for all 1000 points (NUM) and store them in an array called apt (Array of Points).

- **X-Calculation:** We divide the window width by 1000 to space the points evenly.
- **Y-Calculation:** We use the sin() function.
 - ✓ *Input:* We feed it a value from \$0\$ to \$2\pi\$ (one full circle/wave).
 - ✓ *Scaling:* We multiply the result by cyClient / 2 so the wave is tall enough to fill the screen vertically.

Step C: The Batch Draw

Once the array is full of math data, we call Polyline.

- **Efficiency:** Instead of calling LineTo 1000 times (which would be 1000 separate commands), we send one command: "Connect these 1000 points."
-

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static int cxClient, cyClient;
    HDC hdc;
    PAINTSTRUCT ps;
    POINT apt[NUM]; // Array to hold our 1000 points
    int i;

    switch (message)
    {
        case WM_SIZE:
            // 1. Store the new window dimensions
            cxClient = LOWORD(lParam);
            cyClient = HIWORD(lParam);
            return 0;

        case WM_PAINT:
            // 2. Start Painting
            hdc = BeginPaint(hwnd, &ps);

            // 3. Draw the Center Axis (Horizontal Line)
            MoveToEx(hdc, 0, cyClient / 2, NULL);
            LineTo(hdc, cxClient, cyClient / 2);

            // 4. Calculate the Sine Wave Points
            for (i = 0; i < NUM; i++)
            {
                // Determine X: Spread points evenly across width
                apt[i].x = i * cxClient / NUM;

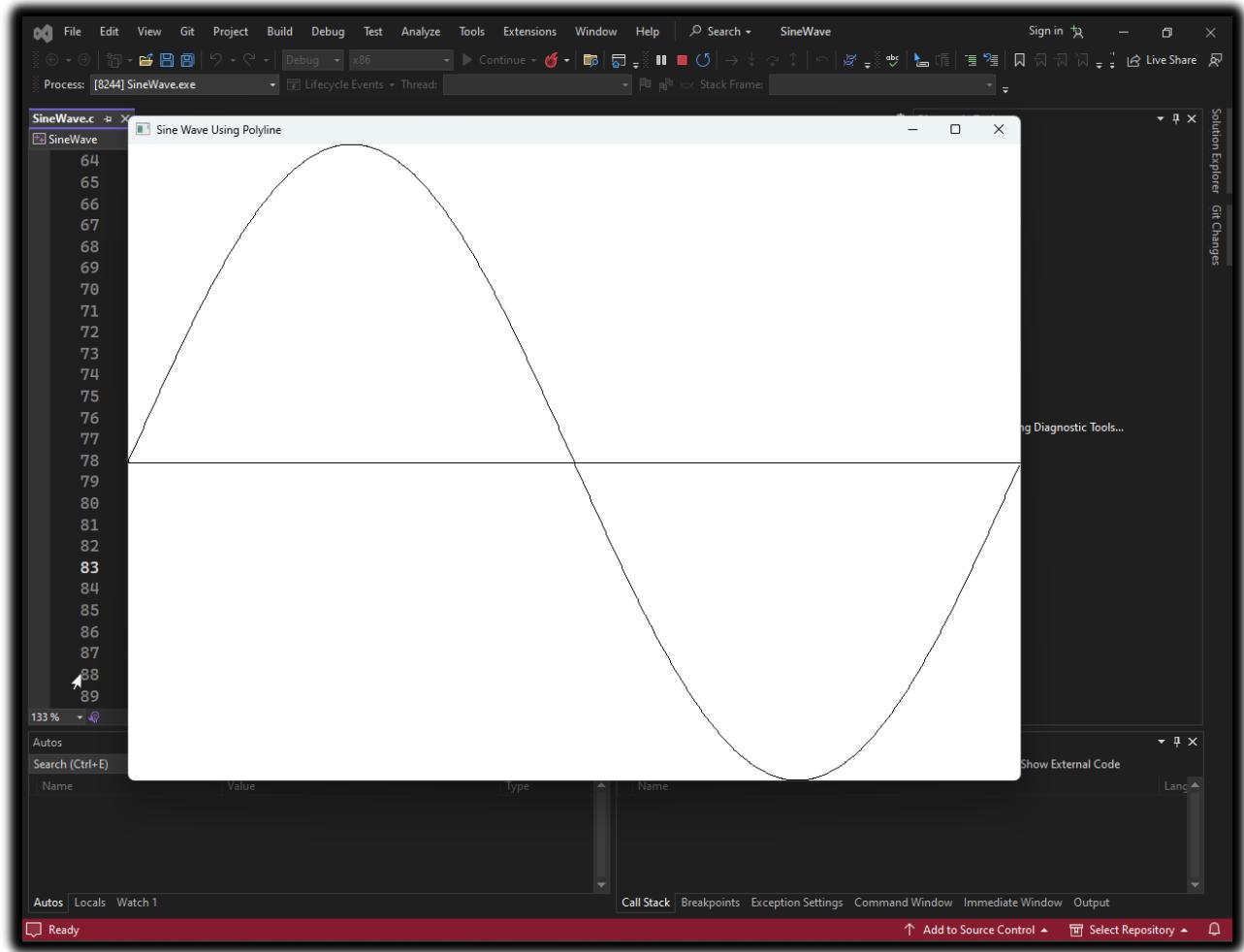
                // Determine Y: Calculate Sine value and scale to height
                // (cyClient / 2) moves the "zero" point to the vertical middle
                apt[i].y = (int) (cyClient / 2 * (1 - sin(TWOPi * i / NUM)));
            }

            // 5. Draw the Wave in one efficient batch command
            Polyline(hdc, apt, NUM);

            EndPaint(hwnd, &ps);
            return 0;

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

    // default handler for messages we don't care about
    return DefWindowProc(hwnd, message, wParam, lParam);
}
```



This snippet is a perfect demonstration of the "GDI Workflow."

1. **Math vs. Graphics:** It separates the *calculation* (the for loop) from the *rendering* (Polyline).
2. **Efficiency:** It proves why Polyline is superior to LineTo. If we drew inside the loop, the computer would have to stop and talk to the graphics card 1000 times. By using an array and Polyline, we do all the hard math on the CPU, and then hand the graphics card a single list of instructions.

THE RECTANGLE FUNCTION

The Concept: The "Box Maker."

The Rectangle function is the simplest way to draw a closed shape. Unlike LineTo, which just drags a pen, Rectangle defines an enclosed area.

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom);
```

- ✓ **(hdc)**: The Canvas.
 - ✓ **(xLeft, yTop)**: The coordinates of the **Top-Left** corner.
 - ✓ **(xRight, yBottom)**: The coordinates of the **Bottom-Right** corner.
-

1. The Bounding Box Concept

The Concept: "Fencing the Yard."

You don't need to calculate all four corners or draw four separate lines. You just tell Windows where the fence starts (Top-Left) and where it ends (Bottom-Right).

- **How it works:** Windows automatically calculates the width and height based on these two points and draws the box between them.
 - **Analogy:** It's like clicking and dragging the "Selection Tool" in Photoshop or on your desktop. You define the diagonal, and the computer fills in the rest.
-

2. Crucial New Concept: Pen vs. Brush

The Concept: The "Outline" vs. The "Filler."

This is the most important takeaway for this section. When you used LineTo, you only used a **Pen**. When you use Rectangle, you are using **two tools** at once:

1. The Pen: Draws the **border** (the outline).
2. The Brush: Fills the **interior** (the solid color inside).

The Default Behavior:

- **The Border:** Drawn with the current Pen (default is Black, 1 pixel wide).
- **The Fill:** Drawn with the current Brush (default is **Solid White**).

Why this matters: If you draw a Rectangle over an existing picture, the inside will turn **White**, covering up whatever was underneath. It is *not* transparent by default.

3. The "Up To (But Not Including)" Rule

The Concept: The Exclusive Boundary.

When you tell Windows to draw a Rectangle from 0 to 5, you might expect it to draw pixels at positions 0, 1, 2, 3, 4, and 5. **It does not.**

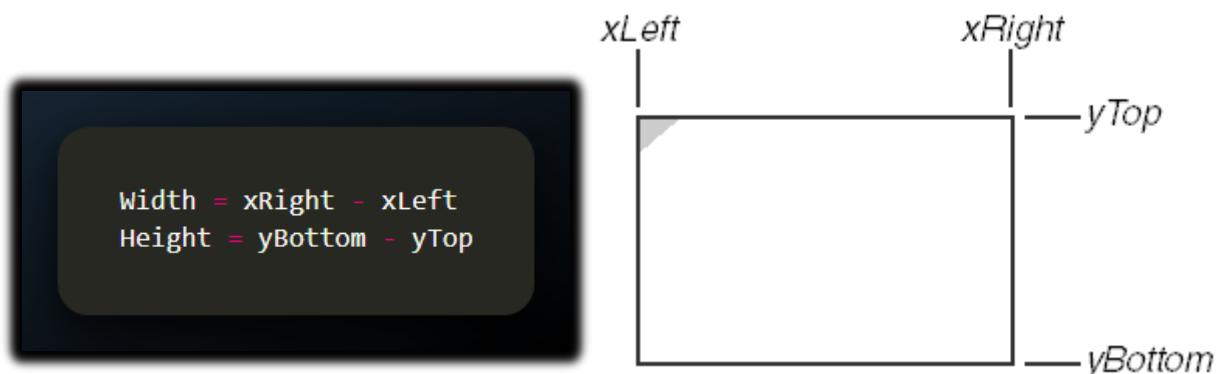
- **The Rule:** GDI includes the Top and Left coordinates, but it excludes the Right and Bottom coordinates.
 - **Analogy:** Think of the coordinates as the **grid lines** between floor tiles, not the tiles themselves. If you measure from line 0 to line 5, you encompass exactly 5 tiles (0, 1, 2, 3, 4). You stop *at* the start of line 5.
-

4. Avoiding "Off-by-One" Errors

The Concept: Trust the Subtraction.

An "Off-by-One" error happens when a programmer tries to "outsmart" the system by manually adding or subtracting 1 pixel, thinking they need to adjust for the border.

The Math: Because of the exclusion rule, the math is actually cleaner:



The Pitfall: If GDI included the last pixel, the width would be $(\text{Right} - \text{Left}) + 1$. By excluding the last pixel, Windows saves you from doing that extra "+1" math everywhere.

5. Example Code

```
// Draw a rectangle  
// Top-Left: (10, 20)  
// Bottom-Right: (100, 150)  
Rectangle(hdc, 10, 20, 100, 150);
```

What actually happens on screen:

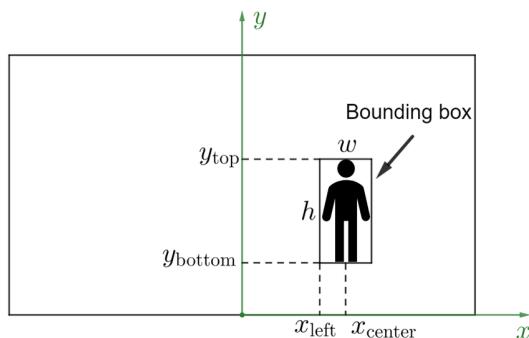
- **X-Axis:** Pixels are drawn from **10 to 99**. Pixel 100 remains untouched.
 - **Y-Axis:** Pixels are drawn from **20 to 149**. Pixel 150 remains untouched.
 - **The Look:** It draws a solid white box (by default) with a black outline.
-

6. Bounding Box & Coordinate Interpretation

The Concept: The "Theoretical Wrapper."

This logic doesn't just apply to rectangles. It applies to **Ellipses** and **Rounded Rectangles** too.

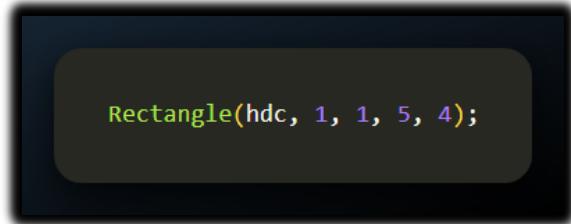
- **The Bounding Box:** Think of this as an invisible crate.
 - ✓ If you want to draw a circle, you first define the square crate it ships in.
 - ✓ Windows draws the circle touching the *inside edges* of that crate.
- **Consistency:** Because Ellipse uses the same bounding box logic as Rectangle, you can easily draw a circle perfectly inside a square by using the exact same coordinates for both functions.



7. Visualizing the "Off-by-One" Logic

The Concept: The Wall, Not the Brick.

Let's look at your specific example to prove how the math works. If you call:



You might expect it to touch pixel 5 and pixel 4. **It does not.**

- **Left (1):** DRAWN.
- **Right (5):** NOT DRAWN (This is the boundary wall).
- **Top (1):** DRAWN.
- **Bottom (4):** NOT DRAWN (This is the boundary wall).

The Result:

- **Width:** $5 - 1 = 4$ pixels.
- **Height:** $4 - 1 = 3$ pixels.
- **Pixels Painted:** Columns 1, 2, 3, 4. Rows 1, 2, 3.

Pro Tip: Always calculate your Right/Bottom coordinates by simple addition: Start + Width. Do not subtract 1. Windows does the subtraction for you.

8. The Ellipse Function

The Concept: The "Ghost Box."

The Ellipse function draws circles and ovals. It is unique because you do not define the center or the radius directly. Instead, you define the **bounding box** (the imaginary rectangle) that the ellipse fits inside.

```
Ellipse(hdc, xLeft, yTop, xRight, yBottom);
```

```
HDC hdc = GetDC(hwnd);
Ellipse(hdc, 50, 50, 150, 100);
ReleaseDC(hwnd, hdc);
```

What happens here?

1. **The Box:** Windows imagines a rectangle from (50, 50) to (150, 100).
2. **The Fit:** It stretches the ellipse to touch the top, bottom, left, and right sides of that imaginary box.
3. **The Math:**
 - ✓ **Width:** 100 pixels right arrow Horizontal Radius: 50.
 - ✓ **Height:** 50 pixels right arrow Vertical Radius: 25.
 - ✓ **Center:** Exactly at (100, 75).

9. The RoundRect Function

The Concept: The "Shaved Corners."

RoundRect (Rounded Rectangle) is a hybrid. It draws a rectangle, but it "shaves off" the sharp corners using small ellipses.

```
RoundRect(hdc, xLeft, yTop, xRight, yBottom, xCornerEllipse, yCornerEllipse);
```

- **xLeft, yTop, xRight, yBottom:** The standard main rectangle.
- **xCornerEllipse:** The width of the small invisible ellipse used to round the corner.
- **yCornerEllipse:** The height of the small invisible ellipse used to round the corner.

```
HDC hdc = GetDC(hwnd);
// A rectangle from (30,30) to (120,90)
// Corners rounded by a curve 15px wide and 20px tall
RoundRect(hdc, 30, 30, 120, 90, 15, 20);
ReleaseDC(hwnd, hdc);
```

Function	Shape	Key Input
Rectangle	Square corners	Top-Left & Bottom-Right
Ellipse	Circle/Oval	Top-Left & Bottom-Right of the <i>enclosing box</i>
RoundRect	Soft corners	Box Coordinates + Corner Dimensions

Fun Notes to Read

Yes, when it comes to creating graphical applications in C, especially with the Windows API, you often deal with lower-level concepts and have more manual control over the drawing process. In C, you work with device contexts, pixels, and lower-level drawing functions.

For example, when using the Windows API in C to draw on a window, you might deal with concepts like device contexts (HDC), which represent a drawing surface, and use functions like Rectangle, Ellipse, and RoundRect to draw basic shapes. You might also handle bitmaps directly by creating, modifying, and displaying them manually.

Here's why:

Procedural Nature: C is a procedural programming language, and when you work with graphical programming in C, you often directly call functions that correspond to graphical operations. This can provide more control but might also require more manual management.

Direct Memory Manipulation: In C, you have more direct access to memory, which means you can manipulate data structures and perform operations at a lower level. This is evident when dealing with bitmaps or other pixel-based graphics.

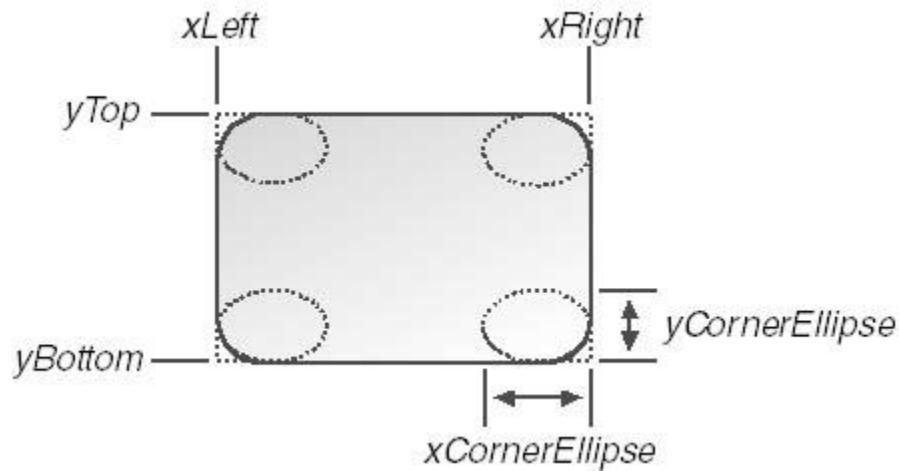
Windows API: When programming in C for Windows, you often use the Windows API, which exposes functions for interacting with the operating system and creating graphical user interfaces. This API is designed to be used with the C programming language.

On the other hand, higher-level languages like C# (especially with Windows Forms, WPF, or UWP) abstract away many of these low-level details. They provide more intuitive, object-oriented frameworks for building graphical applications, making it easier to work with graphical elements without having to deal with the nitty-gritty details of device contexts and manual memory manipulation.

In summary, while C provides more control and lower-level access to system resources, it also requires more manual management, especially when working with graphics. Higher-level languages like C# abstract away many of these details, allowing for more rapid development of graphical applications. The choice between them often depends on the specific requirements and the level of control you need.

ROUNDRECT: THE "CORNER" LOGIC

The Concept: The Invisible Template.

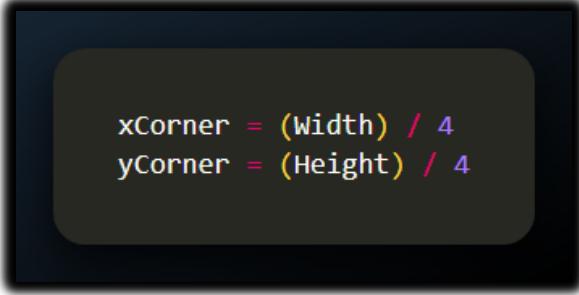


The RoundRect function doesn't just "guess" how to curve the corners. It uses a specific mathematical template: a **small, invisible ellipse** placed at each corner.

- **How it works:** Imagine taking a small ellipse and pushing it into the corner of your rectangle. The RoundRect function traces a quarter of that ellipse to create the curved corner.
- **The Inputs:**
 - ✓ `xCornerEllipse`: The width of that small invisible ellipse.
 - ✓ `yCornerEllipse`: The height of that small invisible ellipse.

1. The "Uneven" Problem

The text mentions a default calculation where we divide the rectangle's dimensions by 4:



```
xCorner = (Width) / 4  
yCorner = (Height) / 4
```

- **The Issue:** If your rectangle is wide and short, your corners will look squashed (elliptical). They won't be perfect quarter-circles.
 - **The Fix:** To get perfectly symmetrical, circular corners, you should ignore the rectangle's size and set xCorner and yCorner to the **same number** (e.g., 20 pixels each).
-

2. Arc, Chord, and Pie Functions

The Concept: Slicing the Ellipse.

These three functions are siblings. They all start by calculating an invisible ellipse (using a bounding box), but they choose to draw only a *part* of it.

Common Arguments: They all require the same inputs:

- **Bounding Box:** (xLeft, yTop, xRight, yBottom) defines the full ellipse size.
- **Start Point:** (xStart, yStart) tells Windows where to begin drawing the curve.
- **End Point:** (xEnd, yEnd) tells Windows where to stop drawing.

A. Arc (The Outline)

- **What it is:** Just the curved line itself.
- **Behavior:** It draws the curve from the start point to the end point.
- **Visual:** Like a trimming from a fingernail. It does **not** connect the ends, and it is **not** filled.

B. Chord (The Bow)

- **What it is:** The curve plus a straight line connecting the ends.
- **Behavior:** It draws the arc, then draws a straight line (the "chord") directly from the start to the end.
- **Visual:** Looks like an archery bow or a semi-circle.
- **Fill:** It is a closed shape, so it is **filled** with the current Brush.

C. Pie (The Slice)

- **What it is:** The curve plus lines connecting to the center.
 - **Behavior:** It draws the arc, then draws straight lines from both ends **to the center** of the ellipse.
 - **Visual:** Looks exactly like a slice of pizza or a Pac-Man shape.
 - **Fill:** It is a closed shape, so it is **filled** with the current Brush.
-

4. The Common Mechanism: "Radial Lines"

The Concept: The Invisible Clock Hands.

A common misconception is that xStart and yStart must be points *exactly on the edge* of the ellipse. They do not.

- **How Windows Thinks:** Windows calculates the **Center** of your bounding box.
- **The Start Ray:** It draws an imaginary line (a ray) from the Center to your (xStart, yStart) coordinates. Where this ray crosses the ellipse is the actual starting point.
- **The End Ray:** It draws a ray from the Center to your (xEnd, yEnd) coordinates. Where this crosses is the actual ending point.

Why this is useful: You don't need to know complex trigonometry to find the exact pixel on the rim of a circle. You just need to point in the general direction (e.g., "Start at the top-right corner").

5. The Golden Rule: Counterclockwise

The Concept: The One-Way Street.

Windows always draws the curve in a **Counterclockwise** direction.

- **The Path:** It starts at the "Start Ray" intersection and travels counterclockwise until it hits the "End Ray" intersection.
 - **The Pitfall:** If you mix up your start and end points, Windows won't draw the "short" arc you wanted; it will draw the "long way around" (the rest of the circle).
-

6. The Three Variants (Arc, Chord, Pie)

The Concept: Open vs. Closed Shapes.

All three functions accept the exact same 9 arguments, but they finish the drawing differently.

A. Arc (The "Open" Curve)

- **Behavior:** Draws **only** the curved line on the rim.
- **Connection:** It does *not* connect the endpoints.
- **Fill:** It is **not filled** (transparent inside), regardless of the current Brush.

B. Chord (The "Bow")

- **Behavior:** Draws the arc, then snaps a straight line directly from the Start point to the End point.
- **Visual:** It creates a shape resembling a D or a bow.
- **Fill:** It is a closed shape, so the interior is **filled** with the current Brush.

C. Pie (The "Slice")

- **Behavior:** Draws the arc, then draws straight lines from both endpoints **to the center**.
- **Visual:** It creates a classic "Pie Chart" slice or a Pac-Man shape.
- **Fill:** It is a closed shape, so the interior is **filled** with the current Brush.

7. Code Syntax

The Concept: The Argument List.

Here is the standardized syntax for all three functions. Note that the first 4 numbers define the *shape* (the ellipse), and the last 4 numbers define the *slice* (the angles).

```
// 1. The Arc (Line only)
Arc(hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);

// 2. The Chord (Filled "D" shape)
Chord(hdc, xLeft, yTop, xRight, yBottom, xstart, yStart, xEnd, yEnd);

// 3. The Pie (Filled "Slice" shape)
Pie(hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);
```

```
#include <windows.h>
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            // Draw a rectangle | Coordinates for the rectangle
            Rectangle(hdc, 50, 50, 150, 100);
            // Draw an ellipse | Coordinates for the ellipse
            Ellipse(hdc, 100, 150, 200, 250);
            // Draw a rounded rectangle | Coordinates and corner radii for the rounded rectangle
            RoundRect(hdc, 250, 50, 350, 100, 20, 20);
            // Draw an arc | Coordinates, starting point, and ending point for the arc
            Arc(hdc, 400, 50, 500, 100, 425, 75, 475, 75);
            // Draw a chord | Coordinates, starting point, and ending point for the chord
            Chord(hdc, 400, 150, 500, 200, 425, 175, 475, 175);
            // Draw a pie slice | Coordinates, starting point, and ending point for the pie slice
            Pie(hdc, 400, 250, 500, 300, 425, 275, 475, 275);
            EndPaint(hwnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}
```

8. Visualizing the Output

The Concept: What appears on screen.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HDC         hdc;
    PAINTSTRUCT ps;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);

            // 1. Rectangle: A simple box
            // Top-Left: (50, 50), Bottom-Right: (150, 100)
            Rectangle(hdc, 50, 50, 150, 100);

            // 2. Ellipse: A circle/oval
            // Bounding Box: (100, 150) to (200, 250)
            Ellipse(hdc, 100, 150, 200, 250);

            // 3. Rounded Rectangle: Box with soft corners
            // Bounding Box: (250, 50) to (350, 100)
            // Corner Ellipse: 20px wide, 20px tall
            RoundRect(hdc, 250, 50, 350, 100, 20, 20);

            // 4. Arc: The curved line (not filled)
            // Bounding Box: (400, 50) to (500, 100)
            // Start Ray: (425, 75), End Ray: (475, 75)
            Arc(hdc, 400, 50, 500, 100, 425, 75, 475, 75);

            // 5. Chord: The "Bow" shape (filled)
            // Same dimensions, but closed with a straight line
            Chord(hdc, 400, 150, 500, 200, 425, 175, 475, 175);

            // 6. Pie: The "Slice" shape (filled)
            // Same dimensions, but closed at the center
            Pie(hdc, 400, 250, 500, 300, 425, 275, 475, 275);

            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY:
            // Ensures the application closes properly
            PostQuitMessage(0);
            return 0;
    }

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

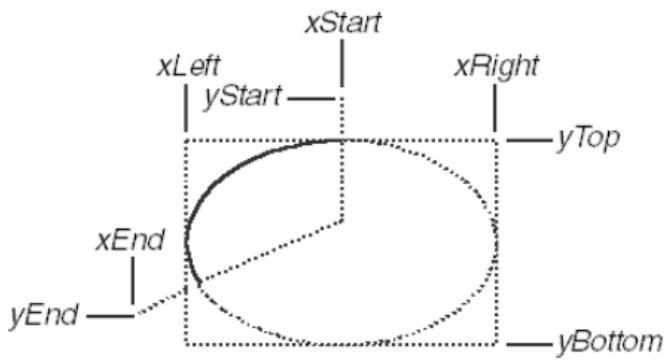


Figure 5–11. A line drawn using the Arc function.

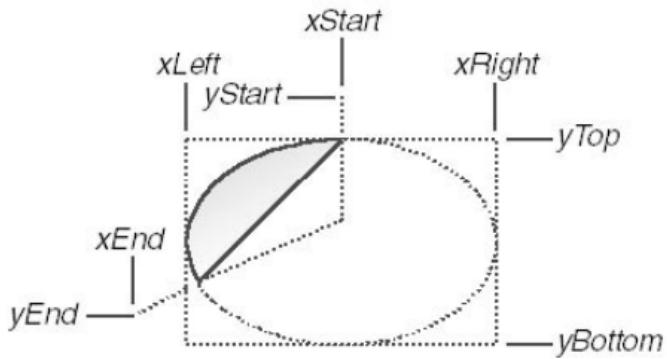


Figure 5–12. A figure drawn using the Chord function.

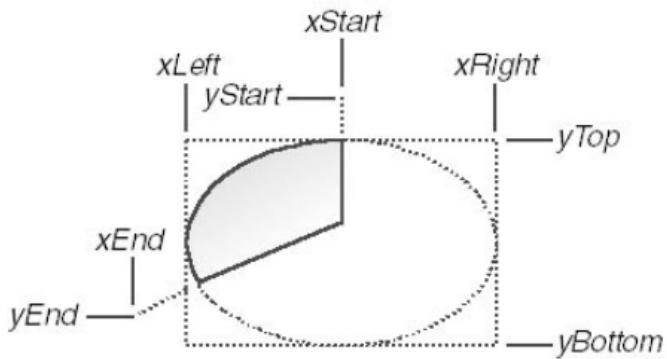
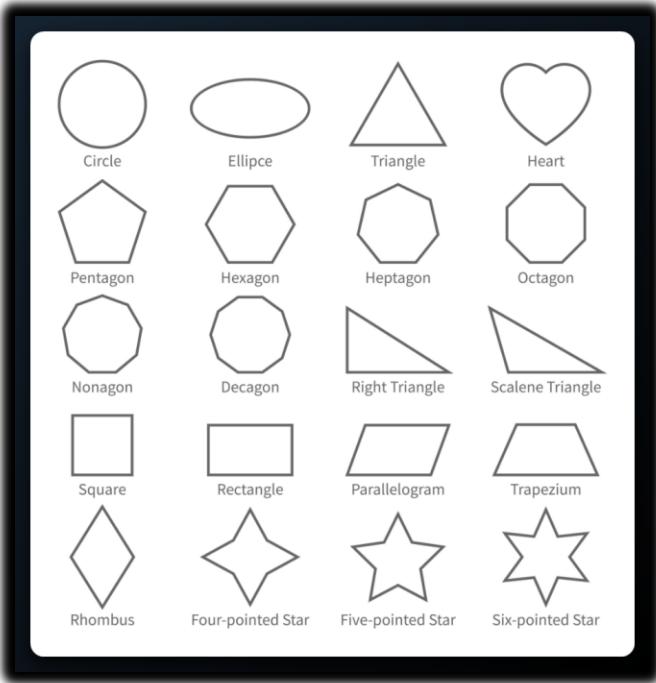


Figure 5–13. A figure drawn using the Pie function.



The code organizes the shapes into distinct rows or columns based on the coordinates:

- **The Solid Shapes:**

- ✓ **Rectangle:** A standard box in the top-left area.
- ✓ **Ellipse:** A circle (since the width 100 matches the height 100) drawn below the rectangle.
- ✓ **RoundedRect:** Sits to the right of the first rectangle, with soft edges.

- **The Partial Shapes (The Trio):**

- ✓ **Arc:** Located at the top right. It draws a curve. Since the start/end points are horizontally aligned through the middle, it likely draws a half-oval curve.
- ✓ **Chord:** Located below the Arc. It looks like the same curve but with a flat line connecting the ends (like a semi-circle filled with color).
- ✓ **Pie:** Located at the bottom right. It looks like a wedge, connecting the arc ends back to the center point.

9. The "Ray" Relationship

The Concept: The Center is the Anchor.

As we discussed, Arc, Chord, and Pie do not simply look at where your xStart and yStart coordinates are on the screen. They care about the **relationship** between those points and the **center of the ellipse**.

- **The Mechanism:** Windows draws an imaginary line (a ray) from the **Center right-arrow Your Point**.
- **The Result:** The intersection where this ray hits the ellipse's rim determines the actual start/end of the drawing.

Visualizing it: Think of the center of the ellipse as the center of a clock. You don't need to touch the numbers on the rim; you just need to point the hour hand in the right direction (e.g., towards 2 o'clock).

10. Precision vs. Convenience

The Concept: Why this design matters.

If Windows required you to provide coordinates that were *exactly* on the edge of the ellipse, you would have to perform complex trigonometry (\sin , \cos , \tan) for every single curve you drew to calculate the exact pixel on the rim.

- **The "Hard" Way (Precision):** Calculating $x = r \cdot \cos(\theta)$ and $y = r \cdot \sin(\theta)$ just to draw a simple corner.
- **The Windows Way (Convenience):** You can just pick a point roughly in the direction you want.
- *Example:* To start at the "Top," you can just use the coordinates of the top-middle of the bounding box. You don't need to calculate where the curve actually hits.

Windows prioritizes **intuition over raw coordinates**. By using relative starting and ending points, it allows developers to define complex curves with simple, readable coordinates, letting the GDI engine handle the heavy mathematical lifting.

11. The LINEDEMO Experiment

The Concept: The "Painter's Algorithm."

The LINEDEMO program isn't just drawing random shapes; it demonstrates a crucial rule of Windows GDI called **Ordering**.

- **How it works:** Windows draws things in the exact order you tell it to.
 - **The Experiment:**
 1. The program draws a crisscross of lines (like a net).
 2. Then, it draws an Ellipse *on top* of those lines.
 - **The Result:** The lines "disappear" behind the ellipse.
-

12. Why do the lines disappear? (The Fill)

This confirms that the Ellipse is **not transparent**. Even though we haven't set a color, the default **White Brush** is active.

- **Solid vs. Wireframe:** If the Ellipse were just a wireframe (like a hula hoop), you would see the lines through it.
- **The Reality:** The Ellipse is a solid disk (like a dinner plate). When placed on top of the lines, its white interior paints over the pixels that were previously black lines.

13. Summary of Module: Drawing Basics

We have now covered the fundamental tools for creating geometry in Windows GDI. Here is the roadmap of what we just learned:

Tool Category	WinAPI Function	Key Concept & Analogy
The Cursor	<code>MoveToEx</code>	The "Lift and Move" technique. Updates the current position without drawing anything. Essentially picking up the pen and placing it at a new start point.
The Line	<code>LineTo</code>	The "Draw" technique. Drags the pen from the current position to the specified point, leaving a trail (line) behind.
The Box	<code>Rectangle</code>	The "Fill & Border" wrapper. Defines a rectangular area using Top-Left and Bottom-Right coordinates. It draws the border and fills the inside.
The Curve	<code>Ellipse / RoundRect</code>	The "Bounding Box" fit. You don't draw the circle directly; you define a rectangular box, and the system stretches a curve to fit perfectly inside it.
The Slice	<code>Arc</code> , <code>Chord</code> , <code>Pie</code>	The "Radial Cut". Uses "Radial Rays" (imaginary lines from the center) to slice sections of an ellipse, like cutting a pizza or defining a specific curve segment.

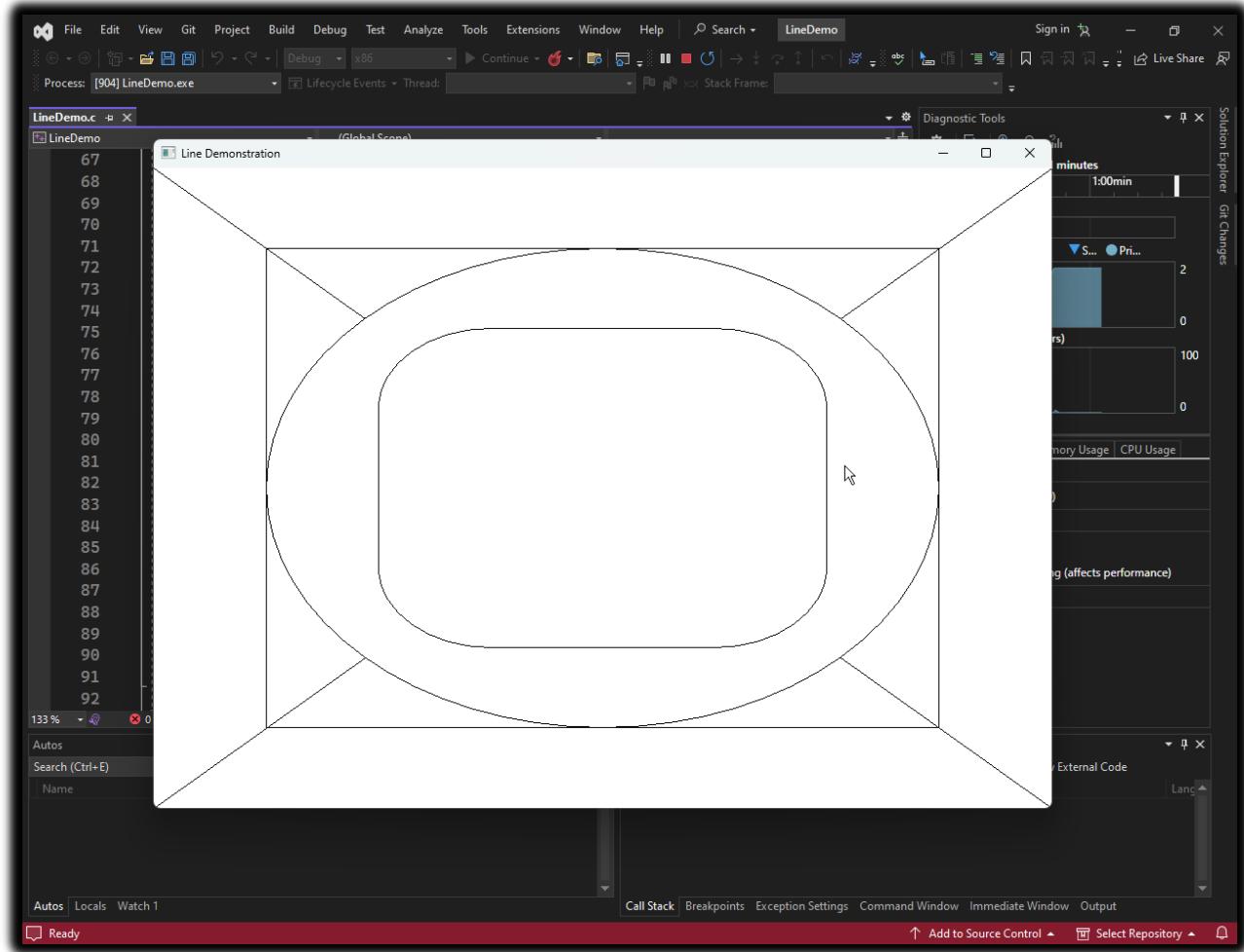
14. What's Missing?

Right now, everything we draw is **Black Outlines** with **White Interiors**.

- We can't change the line thickness or color (Green lines? Dashed lines?).
- We can't change the fill color (Red circles? Hatched patterns?).

Next Step: To fix this, we need to unlock the next level of GDI objects: **PENS** (for the outline) and **BRUSHES** (for the fill).

The code is in Chapter 5 LineDemo folder. Run the .sln file in visual studio community.



WHAT IS A BEZIER SPLINE?

The Concept: The "Magnetic" Curve.

A Bezier spline is a parametric curve defined not by a radius or a center, but by **Control Points**.

- **How it works:** Imagine a rubber band stretched between two points. Now, imagine magnets placed nearby that pull the rubber band towards them *without* actually touching it.
 - **The Control Points:** These "magnets" determine the shape. The curve starts at the first point, ends at the last point, and is influenced (pulled) by the middle points.
 - **Usage:** This is the math behind almost every computer font (TrueType), vector logo, and the "Pen Tool" in Photoshop.
-

1. How Do They Work? (The Math)

The Concept: Weighted Influence.

The mathematical engine driving these curves is called **Bernstein Polynomials**.

- **Weighted Sum:** The curve is a calculation of "weights." As the line travels from Start to End, the influence of the first control point fades while the influence of the second control point grows.
- **The Rule:** The complexity of the curve depends on the number of points.
 - ✓ **Linear (Degree 1):** 2 points (Just a straight line).
 - ✓ **Quadratic (Degree 2):** 3 points (One curve, like a parabola).
 - ✓ **Cubic (Degree 3):** 4 points (The standard "S" curve used in GDI).

2. Advantages of Bezier Splines

The Concept: Why we use them over other curves.

Bezier splines are preferred over older methods (like B-splines or Hermite splines) for three main reasons:

- **Smooth & Continuous:** Unlike connecting jagged lines, Bezier curves remain perfectly smooth, even when you chain multiple segments together. There are no sharp corners at the joints.
 - **Easy to Control:** They are intuitive. You don't need to type in math equations; you simply click and drag a control point, and the curve updates instantly (visual editing).
 - **Computationally Efficient:** Despite looking complex, the math (Bernstein polynomials) is very fast for computers to calculate. This allows us to render complex fonts and graphics in milliseconds.
-

3. Applications of Bezier Splines

The Concept: Where do we see them?

Bezier splines are the industry standard for creating smooth curves in digital environments.

- **Computer Graphics:** Used in vector software like **Adobe Illustrator** and **Inkscape** to draw smooth paths that don't pixelate when zoomed in.
- **Font Design:** They define the outlines of **PostScript (Type 1)** and TrueType fonts. Every letter 'S' you read on a screen is a series of Bezier curves.
- **Animation:** Tools like **Adobe After Effects** and **Maya** use them to control timing and movement (e.g., easing an object to a stop).
- **Robotics:** They smooth out the motion of robot arms, ensuring the machine moves fluidly from Point A to Point B without jerky, mechanical stops.

4. The BEZIER.C Demonstration

The Concept: Interactive Curves.

The BEZIER.C program (from Chapter 5) is an interactive tool that lets you drag control points with your mouse to see how they warp the curve in real-time.

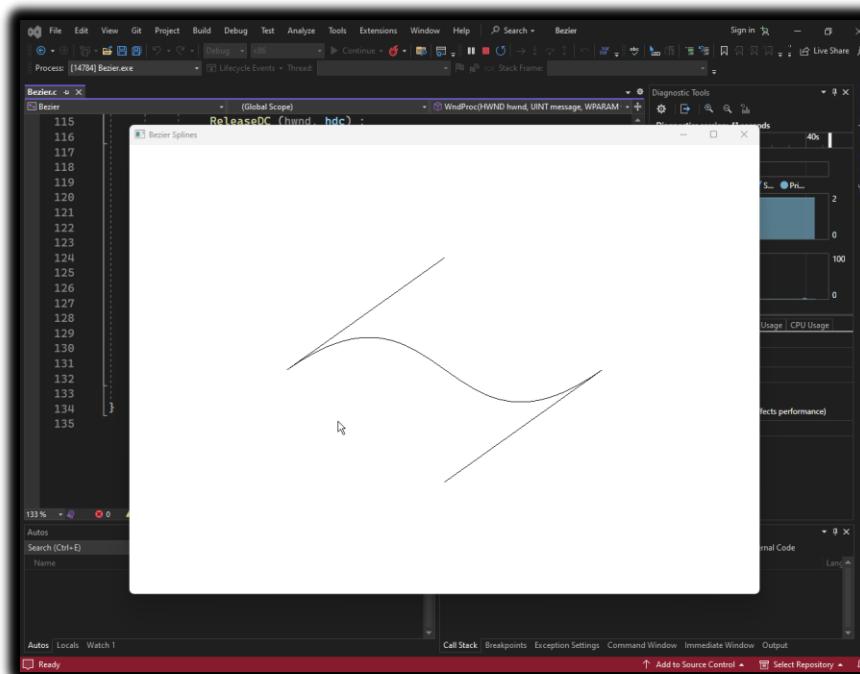
WinMain: The entry point. It creates the window and initializes the four points (Start, End, and two Control Points).

DrawBezier Function:

- **The Core:** Calls PolyBezier to draw the actual curve.
- **The Guide:** Draws straight lines connecting the Control Points to the End Points. This helps the user visualize the "tangents" or "strings" pulling the curve.

WndProc (The Interaction Logic):

- **WM_SIZE:** If the window shrinks, the points are recalculated to stay visible.
- **WM_LBUTTONDOWN / WM_MOUSEMOVE:** If you click and drag the **Left** mouse button, you move the **First Control Point**.
- **WM_RBUTTONDOWN / WM_MOUSEMOVE:** If you click and drag the **Right** mouse button, you move the **Second Control Point**.
- **WM_PAINT:** Redraws the curve whenever the points move.



Program in Chapter 5 Bezier folder.

The BEZIER.C program uses the following code to draw the Bezier curve:

```
DrawBezier (hdc, apt);
```

5. Defining a Bezier Spline

The Concept: The 4-Point Rule.

A standard cubic Bezier curve is defined by exactly **four points** in the apt array.

1. **P0 (Start Point):** The curve begins here (Anchor).
2. **P1 (Control Point 1):** The curve leaves P0 heading towards this point.
3. **P2 (Control Point 2):** The curve arrives at P3 coming from this point.
4. **P3 (End Point):** The curve ends here (Anchor).

Key Characteristic: The curve typically does *not* touch P1 and P2. They act like magnets, pulling the curve towards them. The curve is contained within the "Convex Hull" (the polygon formed by connecting all four points), preventing it from shooting off into infinity.

6. Why are they used in CAD?

The text highlights three specific characteristics that make Bezier splines perfect for design work:

- **Easy to Manipulate:** You don't need to be a mathematician to use them. By simply dragging a handle (the control point), a user can intuitively bend the curve into the desired shape.
- **Well Controlled (The Convex Hull):** This is a critical safety feature for software. The curve is guaranteed to stay inside the polygon formed by connecting the four points (Start ➤ Control1 ➤ Control2 ➤ End). It will never shoot off into infinity or cause calculation errors.
- **Aesthetically Pleasing:** They naturally produce smooth, organic curves rather than mechanical or jagged lines.

7. The Code: DrawBezier Function

The Concept: Visualizing the Mechanics.

The program uses a custom function called DrawBezier to render the scene. It does two things:

1. **Draws the Curve:** It calls the Windows API function PolyBezier.
2. **Draws the Strings:** It draws straight lines connecting the Control Points to the Anchors. This visualizes the "tangents" so you can see exactly how the control points are pulling the curve.

The PolyBezier Arguments:

- **hdc:** The drawing surface (Device Context).
 - **apt:** An array containing exactly **4 points** (Start, Control 1, Control 2, End).
 - **iCount:** Set to **4**.
-

8. The Math Behind the Curve

The Concept: Parametric Equations.

$$\begin{aligned}x(t) &= (1 - t)^3 x_0 + 3t(1 - t)^2 x_1 + 3t^2(1 - t)x_2 + t^3 x_3 \\y(t) &= (1 - t)^3 y_0 + 3t(1 - t)^2 y_1 + 3t^2(1 - t)y_2 + t^3 y_3\end{aligned}$$

t: A time value ranging from **0 to 1**.

- ✓ When $t = 0$, the pixel is at the **Start Point**.
- ✓ When $t = 1$, the pixel is at the **End Point**.

x0 / y0: The Start Point.

x1 / y1: The First Control Point.

x2 / y2: The Second Control Point.

x3 / y3: The End Point.

Key Assumptions: These equations ensure three things:

1. The curve is firmly anchored at the Start and End points.
 2. At the **Start**, the curve travels in the exact direction of the first control point.
 3. At the **End**, the curve arrives from the exact direction of the second control point.
-

9. Technical Note: Point Structures

In both PolyBezier and PolyBezierTo, the apt parameter is an array of POINT structures.

- **Structure:** struct POINT { LONG x; LONG y; };
 - **Usage:** These x and y coordinates map directly to pixels on your screen.
-

10. Windows GDI Graphics Practice Questions

1. In modern Windows operating systems (Windows 10/11), what is the bit-depth used for GDI coordinate values?
2. Which specific industry benefits most from the expanded 32-bit coordinate range due to the need for high precision?
3. Which GDI function is used to retrieve the current position of the drawing pen?
4. What is the primary efficiency advantage of using Polyline over a loop of LineTo calls?
5. In the Sine Wave example discussed, which mathematical function was used to calculate the Y-coordinates?
6. Does the Polyline function update the Device Context's current position after drawing?
7. Which function should be used if you want to draw a series of connected lines starting from the current position and updating it at the end?
8. When drawing a Rectangle, what two tools are used to render it?
9. What is the default brush color used when drawing a Rectangle if no other brush is selected?
10. Which arguments define a Rectangle in GDI?
11. How does Windows GDI handle the 'Right' and 'Bottom' coordinates when drawing a rectangle?

12. If you call Rectangle(hdc, 0, 0, 10, 10), what is the width of the drawn rectangle in pixels?
13. What is an 'Off-by-One' error in the context of GDI graphics?
14. What is a 'Bounding Box'?
15. How do you define the size and position of an Ellipse in GDI?
16. Which function draws a rectangle with curved corners?
17. In the RoundRect function, what do the last two arguments (xCornerEllipse, yCornerEllipse) represent?
18. If you want the corners of a RoundRect to be perfect quarter-circles, how should you set the corner arguments?
19. Which of the following functions creates an OPEN shape (not filled)?
20. What visual shape does the Chord function produce?
21. What visual shape does the Pie function produce?
22. How does Windows determine the start and end points of an Arc, Chord, or Pie?
23. In which direction does Windows GDI always draw an arc?
24. How many arguments do Arc, Chord, and Pie all share?
25. Why does the LINEDEMO program draw the Ellipse after the lines?
26. What is the 'Painter's Algorithm' demonstrated by LINEDEMO?
27. What is a Bezier Spline?
28. How many points are required to define a single standard GDI Cubic Bezier curve?
29. In a Bezier curve, what is the role of the Control Points?
30. What is the 'Convex Hull' property of a Bezier curve?
31. Which function is used to draw Bezier curves in GDI?
32. In the BEZIER.C example, what happens when you drag the mouse?
33. What mathematical polynomials form the basis of Bezier curves?
34. If you want to draw 2 connected Bezier curves using PolyBezier, how many points must be in the array?
35. To ensure a smooth connection between two Bezier curves, what condition must be met?
36. What is the variable 't' in the Bezier parametric equation?

37. Which of these is a common application of Bezier splines?
 38. The PolyBezierTo function differs from PolyBezier because:
 39. Which message does WndProc process to handle drawing requests?
 40. What function must be called to start painting in WM_PAINT?
 41. What function allows you to draw straight lines to visualize the Bezier control handles (the strings)?
 42. If xStart and yStart in an Arc function are (0,0) and the center of the bounding box is (100,100), where does the drawing start?
 43. Which header file is required to use GDI functions?
 44. What data type is used for coordinates in the POINT structure?
 45. In the SINEWAVE program, WM_SIZE was used to:
 46. Why is the Pie function useful for charts?
 47. What is the result of Rectangle(hdc, 10, 10, 10, 10)?
 48. If you want to draw a perfect circle using Ellipse, what must be true about the bounding box?
 49. Does PolyBezier automatically fill the area inside the curve?
 50. Why are BeginPaint and EndPaint paired?
-