INTRODUCTION: WHAT ARE CHILD WINDOW CONTROLS?

Up until now, we have been building one big, blank window. But real apps need buttons, checkboxes, lists, and scrollbars. In Windows API terms, these are called **Child Window Controls**.

The Big Secret: A "Button" is not some special graphical drawing. **It is a Window.** It has a Window Class, a Window Procedure, and it processes messages just like your main application window. The only difference is:

1. It is small.
2. It lives *inside* the client area of a "Parent" window.
3. It is pre-programmed to look and act like a button.

1. How to Create Them

You have two options:

- **Option A: The Hard Way (Custom Controls)** You write your own Window Procedure, draw the button yourself using GDI, handle mouse clicks, and register a new class. (We rarely do this unless we want something totally unique).
- **Option B: The Easy Way (Predefined Controls)** Windows comes with built-in Window Classes that are already registered for you. You just call CreateWindow and pass the specific class name.
    - "button"
    - "edit" (Text box)
    - "listbox"
    - "scrollbar"
    - "static" (Labels)

Example: To make a button, you don't call RegisterClass. You just say:
CreateWindow("button", "Click Me", ...)

2. Communication: The "Parent-Child" Talk

Since the button is a separate window, it needs a way to talk to your main window (the Parent).

- **Child to Parent (Notifications):** When you click the button, it sends a message to the Parent's WndProc saying, "Hey! I was clicked!" *(Technical Note: This is usually a WM_COMMAND message).*
- **Parent to Child (Commands):** If the Parent wants to change the button's text, it sends a message to the button saying, "Change your text to 'Submitted'." *(Technical Note: This is done via SendMessage or SetWindowText).*

3. Where do they live?

You will use controls in two main environments:

A. On a Normal Window (The "Manual" Way)

- **What it is:** You place a button directly on your main app screen.
- **The Catch:** You have to do everything yourself. You must calculate the X/Y coordinates. If the user resizes the window, the button stays stuck in place unless you write code to move it. You also have to manage "Focus" (which window receives keyboard input).

B. In a Dialog Box (The "Manager" Way)

- **What it is:** A special popup window (like "File > Open" or "Settings").
- **The Benefit:** Windows includes a **Dialog Manager**. It handles the layout, the tab order, and the focus for you. It is much easier to set up.

4. "Standard" vs. "Common" Controls

- **Standard Controls:** The basics that have been in Windows since version 1.0 (Buttons, Edit boxes, Scrollbars). This chapter focuses on these.
- **Common Controls:** The fancy modern ones (Progress Bars, Tree Views, Sliders). These live in a separate library and are more complex to set up.

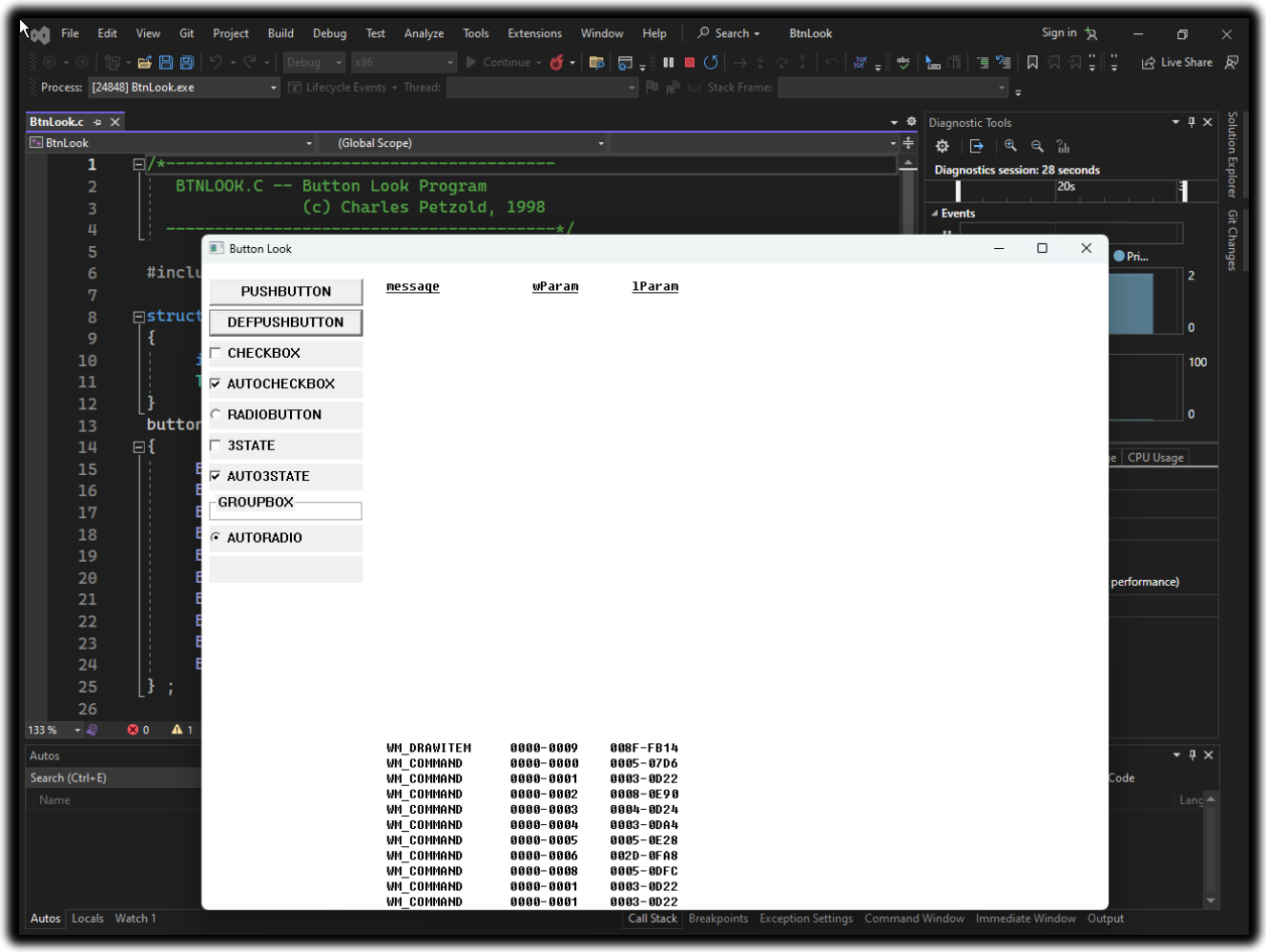
4. Quick Review

Question 1: Is a "Button" inside your application a completely different object type than the Application Window itself? *(Answer: No! They are both just "Windows." The button is just a child window of the application window.)*

Question 2: If you want to create a standard push button, do you need to write a WNDCLASS and register it? *(Answer: No. You use the pre-defined class name "button" inside CreateWindow.)*

Question 3: Why is putting controls on a "Normal Window" harder than in a "Dialog Box"? *(Answer: In a normal window, you have to manually calculate positions and handle resizing code. In a Dialog, the Dialog Manager handles much of that for you.)*

BtnLook program in chapter 9...



The video illustration...

BtnLook
program.mp4

The BTNLOOK Program: A "Button Zoo"

The **BTNLOOK** program is essentially a showcase. It doesn't do any useful work; instead, it displays 10 different types of buttons on the screen so you can see how they look and behave.

1. The Goal

To demonstrate the **10 Standard Button Styles** available in Windows. It also acts as a "spy" tool: whenever you click a button, the program prints the exact details of the message (wParam and lParam) that the button sent to the parent.

2. Key Mechanics

The "Owner-Draw" Button: One of the buttons has the style BS_OWNERDRAW. This means Windows *won't* paint it. The program itself must listen for WM_DRAWITEM and manually draw the button's face (using GDI functions).

Message Spy: When a button is clicked, it sends a WM_COMMAND message. The main window catches this and draws the message details on the right side of the screen.

3. Message Handling Breakdown

MESSAGE	ACTION IN BTNLOOK
`WM_CREATE`	Initializes the UI. Calls `CreateWindow` 10 times to build the various button controls (Push, Radio, Checkbox, etc.).
`WM_SIZE`	Responsive Design. If you stretch the window, this code runs to reposition the buttons so they stay organized and neat.
`WM_PAINT`	Draws the descriptive text labels (e.g., "Push Button", "Checkbox") next to the actual controls.
`WM_DRAWITEM`	Specific to the **Owner-Draw** button. Contains the custom GDI drawing logic for that specific button's appearance.
`WM_COMMAND`	**The Click Handler.** It grabs the ID of the clicked button and displays it to the user.

Creating Child Windows: The Recipe

A child window is just a window that lives inside another one. This includes buttons, text boxes, and lists.

1. The CreateWindow Call

You use the exact same function as you did for your main window, but the parameters are tweaked.

```
hwndButton = CreateWindow(
    TEXT("button"),       // 1. Class Name (Predefined by Windows)
    TEXT("Click Me"),     // 2. Window Text (The label on the button)
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, // 3. Styles
    x, y,                 // 4. Position (Relative to parent's client area)
    width, height,        // 5. Size
    hwndParent,           // 6. Parent Window Handle (Crucial!)
    (HMENU) iButtonID,    // 7. Child ID (Cast to HMENU type)
    hInstance,            // 8. Instance Handle
    NULL                  // 9. Extra Params
);
```

2. Critical Parameters Explained

Class Name ("button"): This tells Windows to use its built-in logic for drawing and handling clicks. You don't need to write a WndProc for this!

Styles (WS_CHILD | WS_VISIBLE):

- WS_CHILD: "I am attached to a parent. If the parent moves, I move. If the parent minimizes, I hide."
- WS_VISIBLE: "Show me immediately." (Without this, the button is created invisible).

Parent Window (hwndParent): This links the button to your main window.

Child ID: In a main window, this is the Menu Handle slot. Since child windows don't have menus, we reuse this slot to store a unique integer ID (like 1, 2, 100). This ID is what you check later in WM_COMMAND.

Understanding the "Button Types" (Styles)

When you create a button, you add a flag to the style parameter to tell Windows what *kind* of button it is.

- **BS_PUSHBUTTON:** A normal "OK" or "Cancel" button.
- **BS_DEFPUSHBUTTON:** A button with a thick black border (usually the "Enter" key trigger).
- **BS_CHECKBOX:** A square box with text.
- **BS_AUTOCHECKBOX:** Same as above, but Windows handles the "check mark" toggling automatically.
- **BS_RADIOBUTTON:** A circle. Used for "one of many" choices.
- **BS_GROUPBOX:** A rectangular frame with a title, used to group other controls visually.
- **BS_OWNERDRAW:** A blank slate. You draw whatever you want.

Quick Review

Question 1: If you create a button but forget WS_VISIBLE, what happens? *(Answer: The button exists in memory and can receive messages, but the user cannot see it on the screen.)*

Question 2: Where does the button send its notification messages (like "I was clicked")? *(Answer: To the Parent Window's WndProc, specifically as a WM_COMMAND message.)*

Question 3: What is the difference between BS_CHECKBOX and BS_AUTOCHECKBOX? *(Answer: BS_CHECKBOX requires you to manually write code to draw the checkmark when clicked. BS_AUTOCHECKBOX toggles the checkmark automatically without extra code.)*

The Button Creation Loop (WM_CREATE)

Instead of writing CreateWindow 10 separate times, the program uses a for loop to create all 10 buttons efficiently. It pulls the text and styles from a data array (button[]).

1. The Code Logic

The goal is to stack the buttons vertically on the left side of the window.

```
for (i = 0; i < 10; i++)
{
    hwndChild[i] = CreateWindow(
        TEXT("button"),                    // 1. Class Name (Always "button")
        button[i].szText,                  // 2. Text (e.g., "Push Button")
        WS_CHILD | WS_VISIBLE | button[i].iStyle, // 3. Style flags
        cxChar,                            // 4. X Position (1 character indent)
        cyChar * (1 + 2 * i),              // 5. Y Position (Calculated row height)
        20 * cxChar,                       // 6. Width (20 characters wide)
        7 * cyChar / 4,                    // 7. Height (1.75 characters tall)
        hwnd,                              // 8. Parent Window Handle
        (HMENU) i,                         // 9. Child ID (0 to 9)
        ((LPCREATESTRUCT) lParam)->hInstance, // 10. Instance Handle
        NULL                               // 11. Extra Params
    );
}
```

2. Analyzing the Parameters

TEXT("button"): This is the magic word. It tells Windows, "Use your internal code to make a button." If you typo this (e.g., "Button" with a capital B in older versions), it fails.

The Style (WS_CHILD | WS_VISIBLE):

- **WS_CHILD**: Mandatory. Without this, the button tries to be a standalone desktop window (and usually fails or looks weird).
- **WS_VISIBLE**: Crucial. If you forget this, the button is created but remains invisible until you manually call ShowWindow.

The Position Math (y coordinate):

- cyChar * (1 + 2 * i)
- This formula spaces them out. i=0 is at line 1. i=1 is at line 3. i=2 is at line 5. It leaves a gap between each button.

The ID ((HMENU) i):

- Notice we cast the integer i to HMENU.
- **Why?** The function expects a Menu Handle here. But for child windows, this slot is repurposed to hold the **Control ID**. We will use this ID (0 through 9) later to identify which button was clicked.

Important Concepts

1. System Metrics (cxChar / cyChar)

The code relies heavily on cxChar and cyChar. These represent the average width and height of a character in the system font.

- **Why?** By using these instead of hard pixels (e.g., "100 pixels"), the buttons automatically scale up if the user has a larger font size or high-DPI screen.

2. The Instance Handle

((LPCREATESTRUCT) lParam)->hInstance

- In WinMain, hInstance is easy to get.
- In WndProc, it's harder. When WM_CREATE fires, lParam points to a structure containing the creation data. We extract hInstance from there to pass it to the child window.

Quick Review

Question 1: Why do we cast i to (HMENU) in the CreateWindow call? *(Answer: Because the function signature demands an HMENU type in that position, even though we are actually passing an integer ID.)*

Question 2: If you change 20 * cxChar to 5 * cxChar, what happens? *(Answer: The buttons become very narrow (5 characters wide), likely cutting off the text inside them.)*

Question 3: Does this loop draw the buttons? *(Answer: No. CreateWindow creates the button logic. Windows then automatically generates a WM_PAINT message for the buttons, causing them to draw themselves.)*

GETTING THE INSTANCE HANDLE (HINSTANCE)

When creating child windows (like buttons), the CreateWindow function demands the **Instance Handle** of your application (hInstance). But how do you get it inside your WndProc? The user's notes outline two popular ways to solve this.

The Problem: In WinMain, you have easy access to hInstance because it is passed as a parameter. In WndProc, however, you are isolated. You don't automatically have access to that variable. Here are the two ways to fix this:

Method 1: The Global Variable (The "Quick & Dirty" Way)

This is the method described in your notes. It is very common in older C programs (like Petzold's examples) because it is simple.

1. Create a Global Variable: Put this at the very top of your .c file, outside any function.

```
HINSTANCE hInst; // Global variable visible to all functions
```

2. Initialize it in WinMain: As soon as the program starts, save the handle.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
    hInst = hInstance; // Save the local parameter to the global variable
    // ... rest of your code ...
}
```

3. Use it in WndProc: Now you can use hInst anywhere.

```
hwndButton = CreateWindow(TEXT("button"), ..., hInst, NULL);
```

Method 2: The API Approach (GetWindowLongPtr)

If you dislike global variables (which is good practice in modern coding), you can ask Windows to look it up for you using the Window Handle (hwnd).

Note: Your notes had a typo "Gggyy.iLdgyyLgng". This refers to GetWindowLong.

The Modern Code (64-bit safe):

```
// Ask Windows: "Who created this window?"
HINSTANCE hInst = (HINSTANCE)GetWindowLongPtr(hwnd, GWLP_HINSTANCE);
```

- **hwnd**: The handle of your main window.
- **GWLP_HINSTANCE**: A flag telling Windows to fetch the Instance Handle associated with this window.

Summary Table

METHOD	PROS	CONS
Global Variable	Very easy to understand. Highly efficient (direct memory access).	"Pollutes" the global namespace. Discouraged in strict/modular C++ architecture.
`GetWindowLongPtr`	Clean. No globals needed. Data is tied directly to the window handle (`HWND`).	Requires a function call (slightly slower). Requires the `HWND` to be valid to access data.
`LPCREATESTRUCT`	The "Purest" way. Allows passing initial data directly during window creation.	Only accessible during the `WM_CREATE` message processing.

Quick Review

Question 1: Why can't you just write hInstance inside WndProc without doing anything else? *(Answer: Because hInstance is a local variable inside WinMain. WndProc cannot see inside WinMain.)*

Question 2: Your notes mentioned GetModuleHandle(NULL). What does that do? *(Answer: It retrieves the instance handle of the currently running file (.exe). It's another way to initialize the global variable if you didn't save the one from WinMain.)*

Question 3: Which method is better if you are writing a large, complex application? *(Answer: Method 2 (API) or passing it via a class structure. Avoiding global variables prevents bugs where different parts of the program overwrite each other's data.)*

Handling Button Clicks (WM_COMMAND): The Notification

When a user clicks a button, the button itself doesn't launch a missile or save a file. It is just a dumb window. Instead, it picks up a telephone and calls its Parent Window.

- **The Caller:** The Child Button.
- **The Receiver:** The Parent Window (WndProc).
- **The Message:** WM_COMMAND.

Decoding the Message Parameters

The WM_COMMAND message packs three vital pieces of information into two variables (wParam and lParam). You have to "unpack" them to understand what happened.

PARAMETER	PART	WHAT IT HOLDS	MEANING
`wParam`	Low Word	Child ID	*"Who called?" (The ID you assigned in `CreateWindow`, e.g., 0 through 9).*
	High Word	Notification Code	*"What happened?" (e.g., `BN_CLICKED`, `BN_DOUBLECLICKED`).*
`lParam`	Full Value	Child Handle	*The actual `HWND` (Window Handle) of the button control.*

The Code Pattern: To extract these values, you use macros:

```
case WM_COMMAND:
    int id = LOWORD(wParam);          // Which button?
    int code = HIWORD(wParam);        // What did it do?
    HWND hButton = (HWND) lParam;     // The button handle

    if (code == BN_CLICKED) {
        // Do something!
    }
    break;
```

The Notification Codes (The "Action" Types)

The "Notification Code" tells you exactly what the user did to the button. In the BTNLOOK program, these are displayed so you can see the internal mechanics.

Here are the standard Button Notifications (BN_):

Notification Code Identifier	Value	Description
BN_CLICKED	0	The button has been clicked.
BN_PAINT	1	The button needs to be repainted.
BN_HILITE or BN_PUSHED	2	The button has been highlighted or pushed.
BN_UNHILITE or BN_UNPUSHED	3	The button has been unhighlighted or unpushed.
BN_DISABLE	4	The button has been disabled.
BN_DOUBLECLICKED or BN_DBLCLK	5	The button has been double-clicked.
BN_SETFOCUS	6	The button has received the input focus.
BN_KILLFOCUS	7	The button has lost the input focus.

Why do we need BN_CLICKED vs BN_PUSHED?

- **BN_PUSHED** happens the moment your finger goes *down*.
- **BN_CLICKED** happens only after your finger goes *down AND up* while still over the button.
- *Rule of Thumb:* Always listen for BN_CLICKED. It allows the user to change their mind (by dragging the mouse away before releasing).

Quick Review

Question 1: If you receive a WM_COMMAND message, how do you know which button triggered it? *(Answer: Check the Low Word of wParam (LOWORD(wParam)). It contains the ID number you assigned in CreateWindow.)*

Question 2: Does lParam contain the ID of the button? *(Answer: No. lParam contains the **Handle** (HWND) of the button. The ID is in wParam.)*

Question 3: In the BTNLOOK program, why do we see BN_PAINT notifications? *(Answer: Because one of the buttons was created with the BS_OWNERDRAW style. This tells Windows "I will draw this button myself," so Windows sends BN_PAINT whenever that button needs updating.)*

The "Focus" Shift

Before looking at the code, it is important to understand what happens to the keyboard when you click a button.

- **Stealing Focus:** When you click a child button, the button grabs the **Input Focus**.
- **The Consequence:** Your main window stops receiving keyboard messages (WM_KEYDOWN). Instead, the *Button* gets them.
- **Button Behavior:** The button control is programmed to ignore most keys, but it listens for the **Spacebar**. Pressing Space while a button has focus pushes the button (same as a mouse click).

The Code: Decoding WM_COMMAND

When the parent receives WM_COMMAND, it needs to unpack the data to answer: "Who called?" and "What did they do?"

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
    case WM_COMMAND:
        {
            // 1. Extract the Data
            int iID   = LOWORD(wParam);           // The Child ID (e.g., 0-9)
            int iCode = HIWORD(wParam);           // The Notification (e.g., BN_CLICKED)
            HWND hwndChild = (HWND) lParam;       // The Button's Handle

            // 2. Format the Message (Safe String Handling)
            // We use TCHAR arrays to support both Unicode and ANSI
            TCHAR szBuffer[256];

            // wsprintf is the standard Windows way to format strings (like printf)
            wsprintf(szBuffer,
                     TEXT("Child Window ID: %d\nNotification Code: %d"),
                     iID, iCode);

            // 3. Display it
            // Note: MessageBox pauses the program until you click OK.
            MessageBox(hwnd, szBuffer, TEXT("Button Notification"), MB_OK);
        }
        return 0;

    // ... handle other messages ...

    default:
        return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

Code Breakdown (Why we fixed it this way)

LOWORD & HIWORD Macros: The wParam is a 32-bit integer. Windows packs two 16-bit integers inside it to save space.

- **Low 16 bits:** The ID.
- **High 16 bits:** The Notification Code.
- *Correction from your notes:* You had LOKORD and HIWRD. The correct macros are LOWORD and HIWORD.

wsprintf vs snprintf: Your snippet tried to use snprintf with TEXT() macros. This often causes errors because snprintf is for standard ASCII C, while TEXT often implies Unicode (Wide Characters).

- **Best Practice:** In Windows API, use wsprintf (simple) or StringCchPrintf (safe) which automatically handle the TCHAR type logic.

The Formatting: We put both values into *one* buffer.

- *Why?* Calling MessageBox twice is annoying for the user. They would have to click "OK" for the ID, and then click "OK" again for the Code. Doing it in one shot is much cleaner.

A Critical Note on MessageBox vs. Real Apps

In the BTNLOOK program described in Petzold's book, the program does **not** actually use MessageBox.

- **Why?** MessageBox is "Modal"—it freezes the entire application until you close the popup.
- **The Better Way:** The actual BTNLOOK program simply saves the values to variables and calls InvalidateRect(hwnd, NULL, TRUE). This triggers a repaint, and the program draws the text directly on the window background. This allows you to click buttons rapidly without being interrupted by popups.

Quick Review

Question 1: When a button has focus, what does the Spacebar do? *(Answer: It triggers a click event (BN_CLICKED).)*

Question 2: Why did we use curly braces { ... } inside the case WM_COMMAND: block? *(Answer: In C/C++, if you declare new variables (like szBuffer) inside a switch case, you must wrap that case in braces to define the scope.)*

Question 3: What is HIWORD(wParam) used for in this message? *(Answer: It retrieves the **Notification Code** (e.g., seeing if the button was clicked vs. double-clicked).)*

HOW PARENT WINDOW TALKS TO ITS CHILD WINDOW IN BTNLOOK:

Think of the **parent window** as a manager and the **child windows (buttons)** as workers.

The manager doesn't click the buttons itself. Instead, it **talks to them by sending messages**.

1. Sending Messages to Child Windows

A parent window can send messages to its child windows to tell them what to do or to ask them questions. For example, it can:

- Ask a button what state it's in
- Tell a button to change how it looks
- Turn a button on or off

It does this by sending messages directly to the child window, kind of like sending a short note that says, *"Change this"* or *"What's your status?"*

2. Button-Specific Messages

Buttons also understand **special messages** that only buttons know how to respond to.

These messages start with **BM**, which stands for **Button Message**. You can think of them as button-only commands—like a remote control that works only on buttons.

Windows defines eight of these button messages in WINUSER.H. Each one lets the parent window control or query a button in a specific way, such as checking it, unchecking it, or changing its behavior.

Button Message	Value	Description
BM_GETCHECK	0x00F0	Retrieves the check mark state of a check box or radio button.
BM_SETCHECK	0x00F1	Sets the check mark state of a check box or radio button.
BM_GETSTATE	0x00F2	Retrieves the state of a button (normal, pushed, or disabled).
BM_SETSTATE	0x00F3	Sets the state of a button (normal, pushed, or disabled).
BM_SETSTYLE	0x00F4	Changes the style of a button.
BM_CLICK	0x00F5	Simulates a mouse click on a button.
BM_GETIMAGE	0x00F6	Retrieves the image associated with a button.
BM_SETIMAGE	0x00F7	Sets the image associated with a button.

3. Check Marks (Check Boxes and Radio Buttons)

Think of a check box or radio button like a light switch.

- **BM_GETCHECK** is how the parent asks:
  "Is this switch on or off?"
- **BM_SETCHECK** is how the parent says:
  "Turn this switch on" or *"Turn it off."*

The parent window sends these messages to the button to check or change whether the mark is there.

---

4. Button State (Normal, Pressed, Disabled)

A button can be in different moods:

- **Normal** (ready to be clicked), **Pressed** (being pushed) or **Disabled** (grayed out and unusable)
- **BM_GETSTATE** is the parent asking:
  "What mood are you in right now?"
- **BM_SETSTATE** is the parent telling the button:
  "Look pressed" or *"Go disabled."*

---

4. Changing How a Button Looks (Style)

The **style** controls how a button looks and behaves.

BM_SETSTYLE is used when the parent wants to change the button's look or behavior.

Think of it like changing a button's outfit.

---

5. Fake a Mouse Click on a Button

Sometimes you want a button to act like it was clicked, even if the user didn't touch the mouse.

BM_CLICK is like the parent saying: *"Pretend you were clicked right now."*

This makes the button run its normal click action automatically.

---

6. Button Images (Icons or Pictures)

Buttons can have pictures on them.

- **BM_GETIMAGE** asks:
  "What picture are you showing?"
- **BM_SETIMAGE** says:
  "Show this new picture instead."

This is useful when you want the button's appearance to change.

---

7. Child Window ID (Name Tag)

Every child window has a unique ID, like a **name tag**.

You can get this ID by using:

- GetWindowLong, or
- GetDlgCtrlID

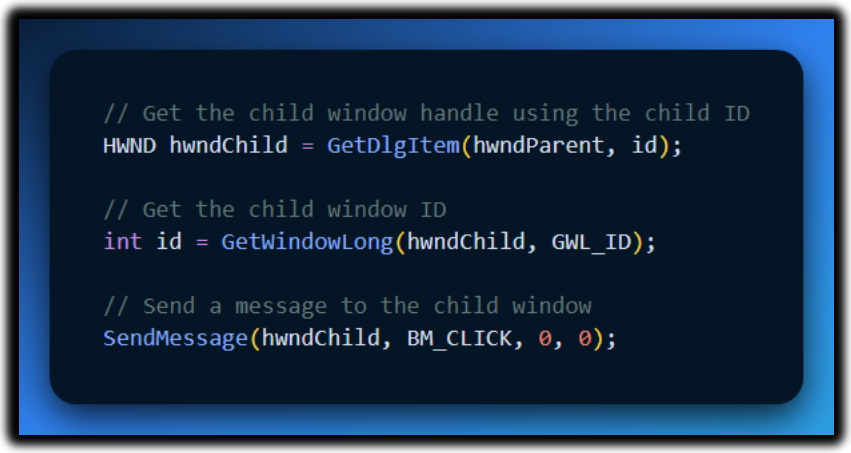
8. Child Window Handle (Phone Number)

If the ID is the name tag, the **handle** is like the phone number.

Once you know:

- the parent window, and
- the child window's ID,

you can get the child's handle using **GetDlgItem**, so you can talk to it directly.



```
// Get the child window handle using the child ID
HWND hwndChild = GetDlgItem(hwndParent, id);

// Get the child window ID
int id = GetWindowLong(hwndChild, GWL_ID);

// Send a message to the child window
SendMessage(hwndChild, BM_CLICK, 0, 0);
```

This example code:

- Finds the child
- Identifies it
- Sends it a message it understands

In short, the parent finds a child window using its ID, gets its handle, and then communicates with it by sending a message.

PUSH BUTTON DEFINITION AND APPEARANCE

A **push button** is the most common kind of button you see in Windows programs. It's a rectangle with some text on it, like **OK**, **Cancel**, or **Submit**.

When a push button is created with CreateWindow, the text you give it becomes the label on the button. The button fills the entire width and height you specify, and the text is automatically centered inside the rectangle.

Push buttons are mainly used to do **one immediate action**. You click the button, the action happens, and that's it. The button does **not** stay on or off. This is why they are often used in dialog boxes for things like accepting or canceling something.

Types of Push Buttons

There are two kinds of push buttons:

- **BS_PUSHBUTTON**
- **BS_DEFPUSHBUTTON**

The **DEF** in BS_DEFPUSHBUTTON means **default**.

In dialog boxes, the default push button is special. It's usually the button that activates when the user presses **Enter** on the keyboard.

However, when these buttons are used as **child window controls**, both types behave the same way. The only visual difference is that a BS_DEFPUSHBUTTON has a **thicker, darker border**, making it stand out more.

How a Push Button Should Look

A push button looks best when its height is about **1¾ times the height of the text** inside it. This gives the button enough space so it doesn't look squished.

The width of the button should be wide enough to hold:

- the text, plus
- a little extra space on both sides

The BTNLOOK program follows these rules so the buttons look clean and balanced.

What Happens with the Mouse

When the mouse cursor is over a push button and the user presses the mouse button, the push button redraws itself with a **3D effect**. This makes it look like the button is being pushed inward.

When the mouse button is released:

- the button goes back to its normal look
- the button sends a WM_COMMAND message to the parent window
- the notification code sent is BN_CLICKED

This is how the parent window knows the button was clicked.

What Happens with the Keyboard

When a push button has keyboard focus, you'll see a **dashed rectangle** around the text on the button.

If the user presses and releases the **Spacebar**, it works exactly the same as clicking the button with the mouse. The button visually presses down and then sends the same BN_CLICKED message to the parent window.

Simple Way to Remember

- Mouse click = action happens
- Spacebar = same action
- Button doesn't stay on or off
- Parent window is notified with BN_CLICKED

Simulating Push Button States

Sometimes you want a push button to **look pressed**, even if the user didn't click it.

You can do this by sending the button a **BM_SETSTATE** message. Think of this as telling the button, *"Pretend someone is holding you down."*

To make the button look pressed (depressed), you send:

```
SendMessage(hwndButton, BM_SETSTATE, 1, 0);
```

This makes the button draw itself as if it's being pushed.

To make the button go back to its normal look, you send:

```
SendMessage(hwndButton, BM_SETSTATE, 0, 0);
```

In both cases, hwndButton is the handle to the push button window that was returned when the button was created with CreateWindow.

Getting the Current State of a Push Button

You can also ask a push button whether it is currently pressed by sending it a **BM_GETSTATE** message.

When you do this:

- The button returns **TRUE** if it is pressed down
- It returns **FALSE** if it is not pressed

That said, most programs don't actually need to check this. Push buttons are usually clicked, handled, and then forgotten.

Additional Notes (Important Ideas)

- Push buttons **do not remember on or off states**. Because of this, messages like **BM_SETCHECK** and **BM_GETCHECK** are not used with push buttons.
- Push buttons are usually connected to **event handlers**, which run some code when the button is clicked.
- **BM_SETSTATE** → *"Look pressed / stop looking pressed"*
- **BM_GETSTATE** → *"Are you pressed right now?"*
- Push buttons = **actions**, not **states**

CHECK BOXES: WHAT THEY ARE AND HOW THEY LOOK

A **check box** is a small square box with text next to it, usually on the right side. You've seen these everywhere—settings screens, options menus, and forms.

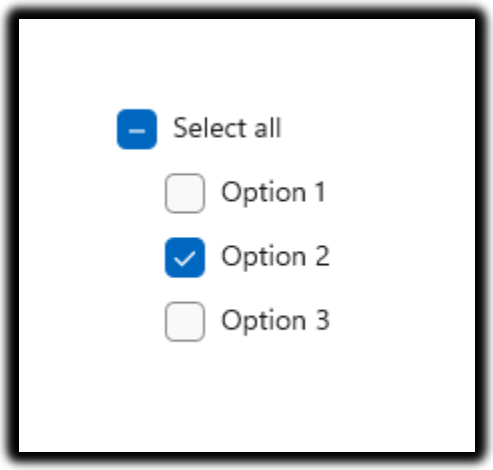
Check boxes are used when you want the user to **turn an option on or off**.

They work like a light switch:

- Click once → a check mark appears
- Click again → the check mark disappears

Types of Check Boxes

There are two main kinds of check boxes, and they behave a little differently.



1. BS_CHECKBOX (Manual Control)

With **BS_CHECKBOX**, Windows does **not** manage the check mark for you.
The programmer is responsible for turning the check mark on and off.

You do this using messages:

- **BM_GETCHECK** → asks, *"Are you checked?"*
- **BM_SETCHECK** → tells the box, *"Check yourself"* or *"Uncheck yourself."*

Example Logic (Simple Explanation)

First, you ask the check box for its current state.
Then, you switch it to the opposite state.

In other words:

- If it's checked → uncheck it
- If it's unchecked → check it

This is how you manually toggle a BS_CHECKBOX.

---

2. BS_AUTOCHECKBOX (Automatic Control)

With **BS_AUTOCHECKBOX**, Windows does the work for you.

When the user clicks the check box:

- Windows automatically adds or removes the check mark
- You don't have to send BM_SETCHECK yourself

All you usually do is ask for the current state using **BM_GETCHECK** when you need to know whether it's checked.

Because of this, you can mostly ignore the WM_COMMAND message for toggling—the system already handled it. Extra Check Box Styles include:

---

3. BS_3STATE

This type of check box has **three states** instead of two:

1. Unchecked
2. Checked
3. Grayed-out (indeterminate)

The grayed-out state is useful when an option is **unknown, mixed, or not relevant**.

You can set this third state by sending BM_SETCHECK with a value of 2.

---

4. BS_AUTO3STATE

This works like BS_3STATE, but again, Windows does the work for you. Each click cycles through: Unchecked → Checked → Indeterminate → back to Unchecked

No manual state management is required.

Check Box Size and Position

The check box square is placed on the **left side** of the control, with the text next to it.

- The minimum height is about the height of one character
- The minimum width is the text width plus a little extra space

Windows automatically centers the check box vertically inside the rectangle you give it in CreateWindow.

User Interaction and Messages

When the user clicks **anywhere inside the check box area**—the box or the text—a WM_COMMAND message is sent to the parent window.

The parent window can use this message to:

- React to the click
- Check the current state
- Update the program's behavior based on the user's choice

Easy Way to Remember

- Check boxes **remember on/off state**
- BS_CHECKBOX → *you* control the check mark
- BS_AUTOCHECKBOX → Windows controls the check mark
- BM_GETCHECK → "Are you checked?"
- BM_SETCHECK → "Check / uncheck yourself"

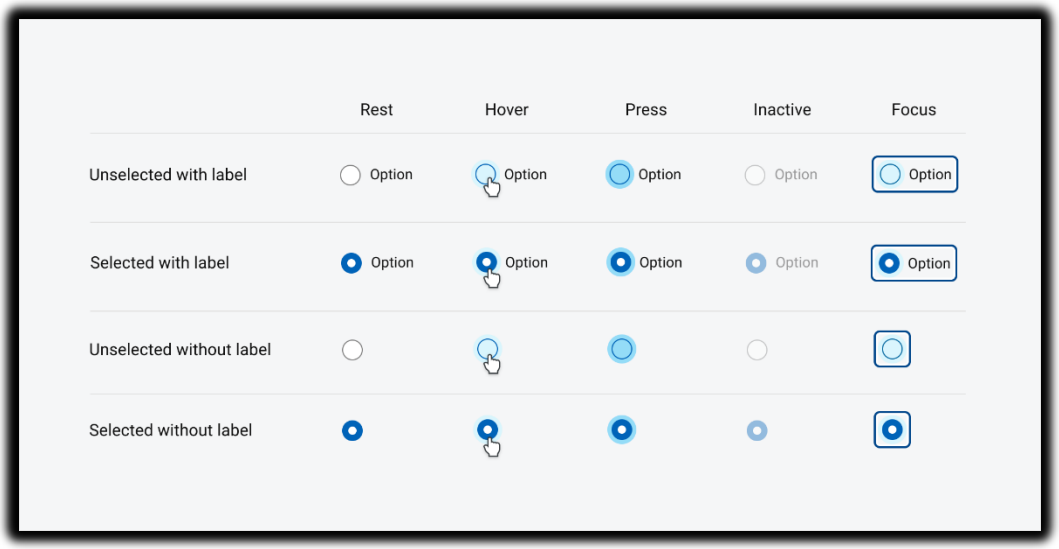
RADIO BUTTONS: WHAT THEY ARE

Radio buttons are used when the user must choose **only one option** from a group.

You usually see them in dialog boxes for choices like:

- Small / Medium / Large
- Yes / No
- One option out of many

Once one option is chosen, the others automatically turn off.



How Radio Buttons Look

Radio buttons look similar to check boxes, but instead of a square, they have a **small circle**.

- An empty circle means “not selected”
- A filled circle means “selected”

Each radio button usually has text next to it that explains the option.

Radio Button Styles

Radio buttons are created using one of these styles:

- **BS_RADIOBUTTON**
- **BS_AUTORADIOBUTTON**

BS_AUTORADIOBUTTON is made especially for dialog boxes and is the most commonly used. It lets Windows handle some of the work for you.

How Radio Buttons Behave

Radio buttons do **not** act like switches.

- Clicking a selected radio button does **nothing**
- You cannot turn it off by clicking it again

Instead, radio buttons work as a **group**:

- When you select one radio button,
- any other radio button in the same group is automatically deselected

This is what makes them “one-choice-only” controls.

Managing Radio Button State

When a radio button is clicked, it sends a WM_COMMAND message to the parent window.

When the parent receives this message, it should:

1. Turn **off** the other radio buttons in the same group
2. Turn **on** the radio button that was clicked

To select the radio button that sent the message, you send it a **BM_SETCHECK** message with wParam set to 1.

In simple terms, this tells the button:

“You are now the selected one.”

Easy Way to Remember Radio Buttons

- **Radio buttons = pick one**
- You **cannot unselect** a radio button by clicking it again
- Selecting one **automatically turns the others off**
- They **always work in groups**

How to Program Them

To **select a radio button**, you send it a BM_SETCHECK message like this:

```
SendMessage(hwndButton, BM_SETCHECK, 1, 0); // 1 = selected
```

To **deselect all the other radio buttons in the same group**, you loop through them and send BM_SETCHECK with wParam = 0:

```
for (int i = 0; i < numRadioButtons; i++) {
    HWND otherRadioButton = GetDlgItem(hwndParent, radioButtonIDs[i]);
    if (otherRadioButton != hwndButton) {
        SendMessage(otherRadioButton, BM_SETCHECK, 0, 0); // 0 = deselected
    }
}
```

Explanation in simple terms:

1. Pick the radio button the user clicked and mark it as **selected**.
2. Go through all the other radio buttons in the same group and **turn them off**.
3. This way, **only one radio button in the group is selected at a time**, which is how radio buttons are supposed to behave.

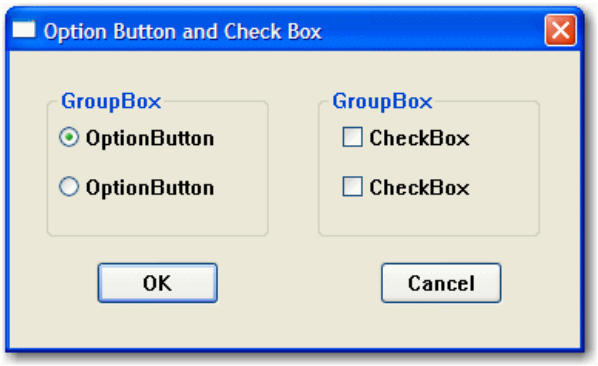
GROUP BOXES

A **group box** is a rectangle with a label at the top. Its job is **not** to do anything when clicked.

Group boxes are **not interactive**. You can't click them, check them, or press them. They are only there to **organize** things on the screen.

Think of a group box like a **labeled box on a form**. It visually tells the user:

"These controls belong together."



What Group Boxes Are Used For

Group boxes are commonly used to surround related controls, such as:

- Radio buttons
- Check boxes

For example, if you have multiple sets of radio buttons, each set would usually be placed inside its own group box so the user can easily see which options go together.

Important Things to Remember

- Group boxes use the **BS_GROUPBOX** style
- They are for **visual grouping only**
- They do **not** send messages or respond to clicks
- Their purpose is to make the interface clearer and easier to understand

---

Group Box Appearance

A **group box** looks like a rectangle with a label at the top.

- The label comes from the window text you give it.
- Unlike check boxes or radio buttons, it **does not have a check mark or any other state**—it's purely visual.

Group Box Function

- Group boxes **do not respond** to mouse clicks or keyboard input.
- They **do not send** WM_COMMAND messages to the parent window.
- Their main job is to **organize related controls** (like radio buttons or check boxes) so the interface is easier to understand and use.

Simple Way to Remember

- Group box = **visual organizer**
- No clicking, no messages, no state
- Helps users see which controls belong together
- Group box = **label + border**
- No clicking, no actions
- Helps users see related options at a glance

CHANGING BUTTON TEXT

You can change the text displayed on a button using the **SetWindowText** function.

- **hwnd** → the handle of the button you want to change
- **pszString** → the new text you want the button to show

```
HWND hwndButton = CreateWindow(
    TEXT("BUTTON"), TEXT("Original Text"),
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
    0, 0, 100, 30, hwndParent, NULL, NULL, NULL
);

// Change the button text
SetWindowText(hwndButton, TEXT("New Text"));
```

Explanation:

- First, we create a button with the text **"Original Text."**
- Then we call SetWindowText to update it to **"New Text."**

Getting Button Text

You can also find out what text is currently displayed on a button using **GetWindowText**.

- **hwnd** → the handle of the button
- **pszBuffer** → where the text will be stored
- **iMaxLength** → the maximum number of characters to copy

The function returns the **length of the text** it copied, or 0 if something went wrong.

```
TCHAR pszText[100];  // Buffer to hold the text
int iLength = GetWindowText(hwndButton, pszText, sizeof(pszText)/sizeof(TCHAR));
```

- We create a buffer (pszText) to hold the text.
- GetWindowText copies the current button text into the buffer.
- iLength tells us how many characters were copied.

Simple Way to Remember

- **SetWindowText** → *"Change the button text"*
- **GetWindowText** → *"Read what the button text is now"*

This works for **all types of buttons**—push buttons, check boxes, and radio buttons.

Getting Button Text

You can find out what text is currently shown on a button using the **GetWindowText** function.

How It Works - GetWindowText needs three things:

1. **hwnd** – the handle to the button you want to get the text from
2. **pszBuffer** – a place (buffer) where the text will be stored
3. **iMaxLength** – the maximum number of characters to copy into the buffer

The function returns the **number of characters copied**. If something goes wrong, it returns 0.

```
TCHAR pszText[100];  // Buffer to store the button text
int iLength = GetWindowText(hwndButton, pszText, sizeof(pszText)/sizeof(TCHAR));
```

Explanation:

We create a buffer pszText to hold the button text.

GetWindowText copies the current text from the button into pszText.

iLength tells us how many characters were copied.

Simple Way to Remember

GetWindowText = "Read what the button says now"

You always need a **buffer** to store the text

The function returns the **length of the text**

Visible and Enabled Buttons

For a button to actually **respond to clicks or keyboard input**, it must be both:

1. **Visible** – the user can see it
2. **Enabled** – the user can interact with it

If a button is visible but **not enabled**, Windows will display its text in **gray** and the button cannot be clicked.

Making a Button Visible

There are two ways to make a button visible:

1. **When creating the button** – include the WS_VISIBLE style in the CreateWindow call.
2. **After creating the button** – call ShowWindow with the SW_SHOWNORMAL flag.

Example (Visible via Window Style)

```
HWND hwndButton = CreateWindow(
    TEXT("BUTTON"), TEXT("Click Me"),
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,  // WS_VISIBLE makes it visible
    0, 0, 100, 30,
    hwndParent, NULL, NULL, NULL
);
```

Here's an example of making a button visible using ShowWindow:

```
HWND hwndButton = CreateWindow(
    TEXT("BUTTON"),          // The type of control: a button
    TEXT("Button Text"),     // The text displayed on the button
    WS_CHILD | BS_PUSHBUTTON,// Styles: child window + push button
    0, 0, 100, 30,          // x, y position and width, height
    hwndParent,              // The parent window
    NULL, NULL, NULL         // Menu, instance, extra parameters
);

ShowWindow(hwndButton, SW_SHOWNORMAL);  // Makes the button appear
```

CreateWindow **creates the button**, but it might not be visible yet if WS_VISIBLE isn't used.

The ShowWindow call with SW_SHOWNORMAL **makes the button appear on the screen**.

After this, the button is **visible and ready to be clicked**, as long as it is also enabled.

Hiding a Button

If you want to **hide a button** so the user can't see it, use the ShowWindow function with the SW_HIDE flag:

```
ShowWindow(hwndButton, SW_HIDE);  // Hides the button
```

Explanation:

- The button still exists in memory, but it is **not visible** on the screen.
- Users **cannot click** it while hidden.

Enabling and Disabling Buttons

By default, buttons are **enabled**, meaning users can click them.

To **disable a button**, so it appears gray and cannot be clicked, use EnableWindow with FALSE:

```
EnableWindow(hwndButton, FALSE);  // Disable the button
```

The button stays visible but cannot respond to clicks or keyboard input.

To **enable a previously disabled button**, use EnableWindow with TRUE:

```
EnableWindow(hwndButton, TRUE);  // Enable the button
```

The button becomes clickable again and looks normal.

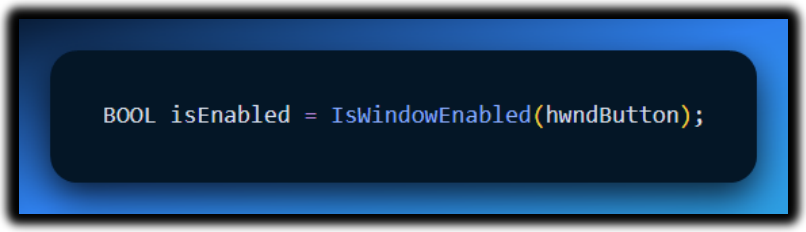
You can also **check the current state** of a button and visibility.

Use **IsWindowVisible** to see if a button is visible:

```
BOOL isVisible = IsWindowVisible(hwndButton);
```

- Returns TRUE → the button is visible
- Returns FALSE → the button is hidden

Use IsWindowEnabled to see if a button is enabled:



- Returns TRUE → the button can be clicked
- Returns FALSE → the button is disabled

Simple Way to Remember

- ShowWindow(hwndButton, SW_HIDE) → hide button
- ShowWindow(hwndButton, SW_SHOWNORMAL) → show button
- EnableWindow(hwndButton, TRUE/FALSE) → enable or disable button
- IsWindowVisible → check if visible
- IsWindowEnabled → check if enabled

INPUT FOCUS AND BUTTONS

The "Focus" Problem

By default, **clicking a button steals the focus.** When you click a button (push button, checkbox, etc.), Windows assumes you want to interact with *that* specific control.

- **Visual Cue:** The button text gets surrounded by a small dotted rectangle.
- **The Consequence:** Your main window (WndProc) stops receiving WM_KEYDOWN messages. Instead, the button receives them.
- **Button Behavior:** The button ignores almost every key *except* the **Spacebar**, which triggers a click.

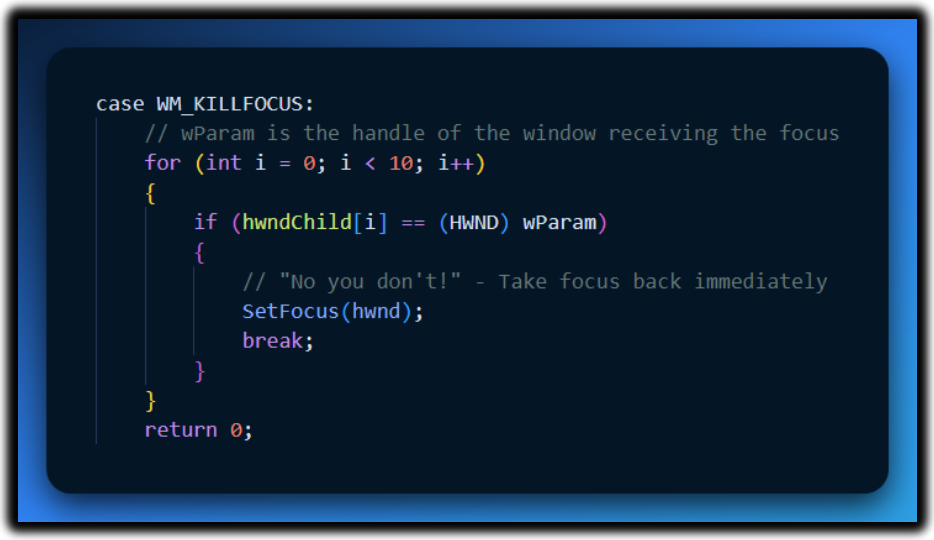
The Hack: "Give it Back!"

If you want your main window to *always* keep the keyboard focus (for example, if you are writing a game or a drawing app where clicking a tool shouldn't stop your hotkeys from working), you have to fight the system.

You do this by trapping the WM_KILLFOCUS message. This message is sent to your window *right before* it loses focus.

Method 1: The Loop Check

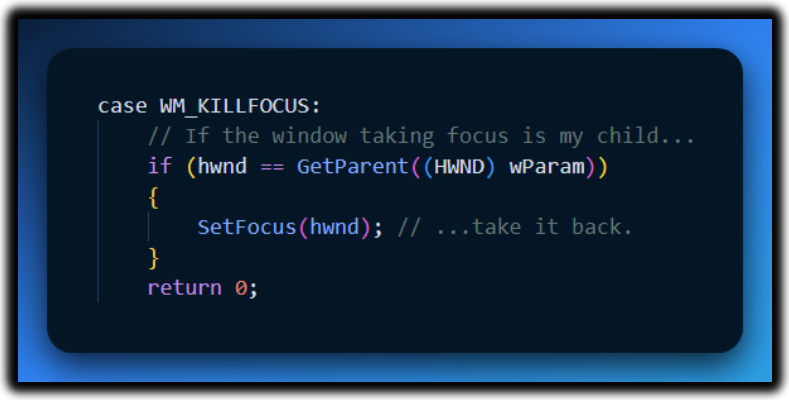
You check if the window "stealing" the focus (wParam) is one of your own buttons. If so, you forcefully reclaim it.



```
case WM_KILLFOCUS:
    // wParam is the handle of the window receiving the focus
    for (int i = 0; i < 10; i++)
    {
        if (hwndChild[i] == (HWND) wParam)
        {
            // "No you don't!" - Take focus back immediately
            SetFocus(hwnd);
            break;
        }
    }
    return 0;
```

Method 2: The Parent Check (Simpler)

Instead of looping through arrays, you just ask, "Is the new focus window a child of mine?"



```
case WM_KILLFOCUS:
    // If the window taking focus is my child...
    if (hwnd == GetParent((HWND) wParam))
    {
        SetFocus(hwnd); // ...take it back.
    }
    return 0;
```

The Drawback (Why this is a hack)

While this keeps your main window active, it breaks standard Windows accessibility:

1. **No Spacebar:** The user can no longer hover over a button and press **Space** to click it repeatedly.
2. **No Visual Feedback:** The user never sees the dotted focus rectangle, so they don't know which button was last touched.
3. **No Tab Navigation:** You cannot press **Tab** to jump between buttons.

The "Correct" Solution: A professional Windows application usually *wants* buttons to have focus. To support advanced navigation (like Tabbing between fields), you normally wouldn't fight WM_KILLFOCUS. Instead, you would use a method called "Subclassing" (Chapter 9/10) to let the parent window "spy" on the button's keyboard input without stealing focus back.

Quick Review

Question 1: What visual indicator shows that a button has the Input Focus? *(Answer: A dashed/dotted line surrounding the button's text.)*

Question 2: If you implement the WM_KILLFOCUS hack above, what happens when you click a button and then press the Spacebar? *(Answer: Nothing happens. The button lost focus immediately after the click, so the Spacebar keystroke goes to the Main Window, not the button.)*

Question 3: In WM_KILLFOCUS, what does the wParam parameter represent? *(Answer: It holds the handle (HWND) of the window that is receiving the focus.)*

The Better Solution: Window Subclassing

```
case WM_KILLFOCUS:
if (hwnd == GetParent((HWND) wParam)) {
    SetFocus(hwnd);
}
return 0;
```

The method we saw before—fighting the button for focus using WM_KILLFOCUS—is kind of a **hack**. It works, but it **breaks keyboard input** for things like the Spacebar or Tab key. The professional way to solve this is **Window Subclassing**.

What is Window Subclassing?

Subclassing is like **inserting a middle manager** between the button and Windows:

- Normally: Key Press → Button
- With Subclassing: Key Press → Your Function → Button

It allows you to **intercept messages** sent to a child window **before the window handles them itself**.

Why Use Subclassing?

By intercepting the messages, you can:

- **Spy on the keyboard** before the button reacts.
- Handle **Tab key** navigation yourself.
- Let the **Spacebar** still trigger the button normally.
- Send all other messages to the button without changing its behavior.

Pro Tip: Later in Chapter 9, the **COLORS1 program** shows how subclassing can create a fully keyboard-navigable interface.

Explained Like You're a Teenager

1. What is Input Focus?

Imagine you are at a **party with 10 people talking at once**. You can hear everyone, but you can **only speak to one person at a time**.

You turn to face **Dave**. Now you and Dave are "locked in."

Translate that to Windows:

- **The Desktop = the party** (many apps open: Chrome, Spotify, Discord, a game…)
- **The Keyboard = your voice**
- **Focus = the person you are talking to (Dave)**

The rule: Your keyboard **only talks to the window that has focus**. If you want to type somewhere else, you must **click that window** first to move the focus spotlight.

---

2. What is a Handle (HWND)?

Imagine ordering pizza for a **LAN party with 50 people**.

You can't say "give this pizza to the guy in the black shirt," because **multiple people might be wearing black shirts**.

Instead, **everyone wears a nametag with a unique ID**: #101, #102, #103…

Translate that to Windows:

- **The Window = the person at the party**
- **The Handle (HWND) = their unique ID tag** (like 0x004F32)

How it works together:

1. You click at coordinate (500, 300).
2. Windows checks its map and finds **Window #9942** at that spot.
3. Windows says: "Shift the **Focus Spotlight** to #9942. All keyboard input now goes here."

---

3. Quick Review

Question 1: If you have 10 buttons on screen, how many can have **input focus** at the same time?

Answer: Only **one**. The spotlight can only be in one place.

Question 2: Does the **handle of a window** ever change while the program is running?

Answer: No. Once a window is created, its **handle stays the same** until it's destroyed. It's like a Social Security Number for the window.

Question 3: Why is **subclassing better than stealing focus back**?

Answer: Subclassing lets you **filter input smartly**—keeping useful keys like Spacebar working while handling navigation keys like Tab yourself.

System Colors in Windows

System colors are a set of colors that Windows uses to paint the interface you see on your screen, like:

- Window borders
- Title bars
- Buttons
- Text

These colors are **predefined by Windows** and can be accessed or changed through code.

- **GetSysColor** → lets you **read** a system color.
- **SetSysColors** → lets you **change** one or more system colors.

Example: You can find out what color Windows uses for the background of active windows, or even change the button color programmatically.

Simple analogy:
Think of system colors like **Windows' default paint palette**. Every GUI element grabs its color from this palette, so the interface looks consistent.

Table of System Colors

GetSysColor and SetSysColors Identifier	Registry Key or WIN.INI Value	Default RGB Value	Description
COLOR_SCROLLBAR	Scrollbar	C0-C0-C0	The color of scrollbars.
COLOR_BACKGROUND	Background	00-80-80	The color of the background behind windows.
COLOR_ACTIVECAPTION	ActiveTitle	00-00-80	The color of the title bar of the active window.
COLOR_INACTIVECAPTION	InactiveTitle	80-80-80	The color of the title bar of inactive windows.
COLOR_MENU	Menu	C0-C0-C0	The color of the background of menus.
COLOR_WINDOW	Window	FF-FF-FF	The color of the background of windows.
COLOR_WINDOWFRAME	WindowFrame	00-00-00	The color of the border of windows.
COLOR_MENUTEXT	MenuText	C0-C0-C0	The color of text in menus.
COLOR_WINDOWTEXT	WindowText	00-00-00	The color of text in windows.
COLOR_CAPTIONTEXT	TitleText	FF-FF-FF	The color of text in title bars.
COLOR_ACTIVEBORDER	ActiveBorder	C0-C0-C0	The color of the border of the active window.
COLOR_INACTIVEBORDER	InactiveBorder	C0-C0-C0	The color of the border of inactive windows.
COLOR_APPWORKSPACE	AppWorkspace	80-80-80	The color of the background of non-client areas, such as the desktop and the Start menu.
COLOR_HIGHLIGHT	Highlight	00-00-80	The color of the highlight when selecting text or items.
COLOR_HIGHLIGHTTEXT	HighlightText	FF-FF-FF	The color of text in the highlight.
COLOR_BTNFACE	ButtonFace	C0-C0-C0	The color of the face of buttons.
COLOR_BTNSHADOW	ButtonShadow	80-80-80	The color of the shadow of buttons.
COLOR_GRAYTEXT	GrayText	80-80-80	The color of grayed-out text.
COLOR_BTNTEXT	ButtonText	00-00-00	The color of text on buttons.

COLOR_INACTIVECAPTIONTEXT	InactiveTitleText	C0-C0-C0	The color of text in inactive title bars.
COLOR_BTNHIGHLIGHT	ButtonHighlight	FF-FF-FF	The color of the highlight on buttons when the mouse is over them.
COLOR_3DDKSHADOW	ButtonDkShadow	00-00-00	The color of the darkest shadow of buttons.
COLOR_3DLIGHT	ButtonLight	C0-C0-C0	The color of the lightest light of buttons.
COLOR_INFOTEXT	InfoText	00-00-00	The color of text in message boxes.
COLOR_INFOBK	InfoWindow	FF-FF-FF	The color of the background of message boxes.
No identifier	ButtonAlternateFace	B8-B4-B8	The color of the alternate face of buttons.
COLOR_HOTLIGHT	HotTrackingColor	00-00-FF	The color of hot tracked items.
COLOR_GRADIENTACTIVECAPTION	GradientActiveTitle	00-00-80	The gradient color of the active title bar.
COLOR_GRADIENTINACTIVECAPTION	GradientInactiveTitle	80-80-80	The gradient color of the inactive title bar.

The exact RGB values of system colors can **change slightly** depending on your computer's display driver.

Analogy: It's like painting the same wall with slightly different shades depending on the brand of paint—you get the same general color, but the exact tone may vary.

CHALLENGES WITH SYSTEM COLORS FOR BUTTONS

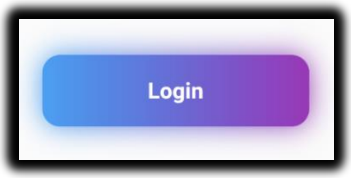
In modern Windows versions, buttons have become **visually more complex**, with 3D effects and shadows. This makes using system colors more tricky for programmers. Here's why:

Inconsistent Color Names

- ✓ Some system colors have intuitive names that match their purpose.
- ✓ Others are less predictable, making it hard to know exactly what color will appear.

Multiple Colors per Button

- ✓ A single button can use **different colors** for its face, shadow, text, and border.
- ✓ This makes managing button colors more complicated.



Color Clash with Client Area

- ✓ If your window's background (client area) is left white, it can **clash** with button colors.
- ✓ This creates an inconsistent or "ugly" look.

Solutions to Address Color Issues

Programmers can use several strategies to make buttons look right:

1. **Yield to System Colors**
    - ✓ Set the **client area background** to COLOR_BTNFACE.
    - ✓ This makes the window background match the default button face, avoiding clashes.
2. **Explicitly Set Text Colors**
    - ✓ By default, text uses white background and black text.
    - ✓ To match buttons, set:
        - Text background → COLOR_BTNFACE
        - Text color → COLOR_WINDOWTEXT
3. **Handle System Color Changes**
    - ✓ Users can **change system colors** while your program is running.
    - ✓ To update your window to match, handle the WM_SYSCOLORCHANGE message.
    - ✓ This will **redraw the client area** with the new colors.

Analogy:
Think of system colors like a **paint set Windows gives you**. Buttons now use **shadows, highlights, and text colors**, so if your background doesn't match, it looks messy. By adjusting your window to "use the same paint set," everything looks clean and consistent.

Alternative Approach: Custom Colors

Instead of using system colors, you can **choose your own custom colors** for:

- The window background (client area)
- Buttons
- Text

Benefits:

- You have **full control** over how everything looks.
- You don't have to worry about the user changing system colors while your program is running.

Drawbacks:

- You need to **manage all your colors** carefully.
- You must make sure the colors **look consistent** across your whole application.

Analogy:
It's like decorating a room yourself instead of using a pre-made paint set. You can pick exactly the colors you want, but you also have to make sure the walls, furniture, and decorations all match.

Code examples:

These two pieces of code makes the window background and button text use system colors (COLOR_BTNFACE and COLOR_WINDOWTEXT) so the buttons and text look consistent and don't clash.

```
wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1);
```

```
SetBkColor(hdc, GetSysColor(COLOR_BTNFACE));
SetTextColor(hdc, GetSysColor(COLOR_WINDOWTEXT));
```

This code catches system color changes and forces the window to redraw so the new colors are applied.

```
case WM_SYSCOLORCHANGE:
InvalidateRect(hwnd, NULL, TRUE);
break;
```

Coloring Specific Buttons (WM_CTLCOLORBTN)

Sometimes you don't want to blend in. You want *specific* control over how a button looks. Windows gives you a chance to interfere right before a button is painted.

The Message: WM_CTLCOLORBTN **Sent To:** The Parent Window (WndProc). **When:** Just before a button draws itself.

What you can do inside this message:

You (the Parent) can tell the button (the Child) three things:

1. **"Use this Text Color"** (SetTextColor)
2. **"Use this Background Color for Text"** (SetBkColor)
3. **"Use this Paint Brush for the Background"** (Return a HBRUSH)

```
case WM_CTLCOLORBTN:
    SetTextColor((HDC) wParam, RGB(0, 0, 255)); // Make text Blue
    SetBkColor((HDC) wParam, RGB(255, 255, 0)); // Make text background Yellow
    return (LRESULT) hBrushYellow;              // Paint the button face Yellow
```

The Catch with WM_CTLCOLORBTN

The name of this message is **confusing**.

Standard Push Buttons (like OK or Cancel) mostly ignore it because Windows draws them with a 3D style. You usually **can't change their background color** using this message.

Owner-Drawn Buttons already paint themselves, so this message does nothing for them.

Real-world use: WM_CTLCOLORBTN is mainly useful for **simple or old-style buttons**. For modern colorful buttons, you usually use **owner-draw**, where you draw the button completely yourself.

WM_CTLCOLORBTN can change button colors, but it has limits. Using system colors or custom-drawn buttons is usually a better way to control colors.


OnDraw.mp4

Owner-Draw Buttons

The **OWNDRAW** program shows how to use owner-draw buttons. Owner-draw buttons let you **fully control how buttons look**, instead of relying on the standard Windows style.

The program has **two main parts**: the WinMain function and the WndProc window procedure.

WinMain function:

- Registers the window class, which defines how the window behaves and looks.
- Creates the main window using the registered class.
- Shows the main window on the screen.
- Enters the **message loop**, which handles messages sent to the window until it is closed.

WndProc window procedure:

- Handles messages sent to the window.
- **WM_CREATE:** Creates two owner-draw buttons when the window is first opened.
- **WM_SIZE:** Resizes the buttons whenever the window size changes.
- **WM_COMMAND:** Handles button clicks by resizing the window.
- **WM_DRAWITEM:** Draws the owner-draw buttons with custom graphics.

Drawing Owner-Draw Buttons

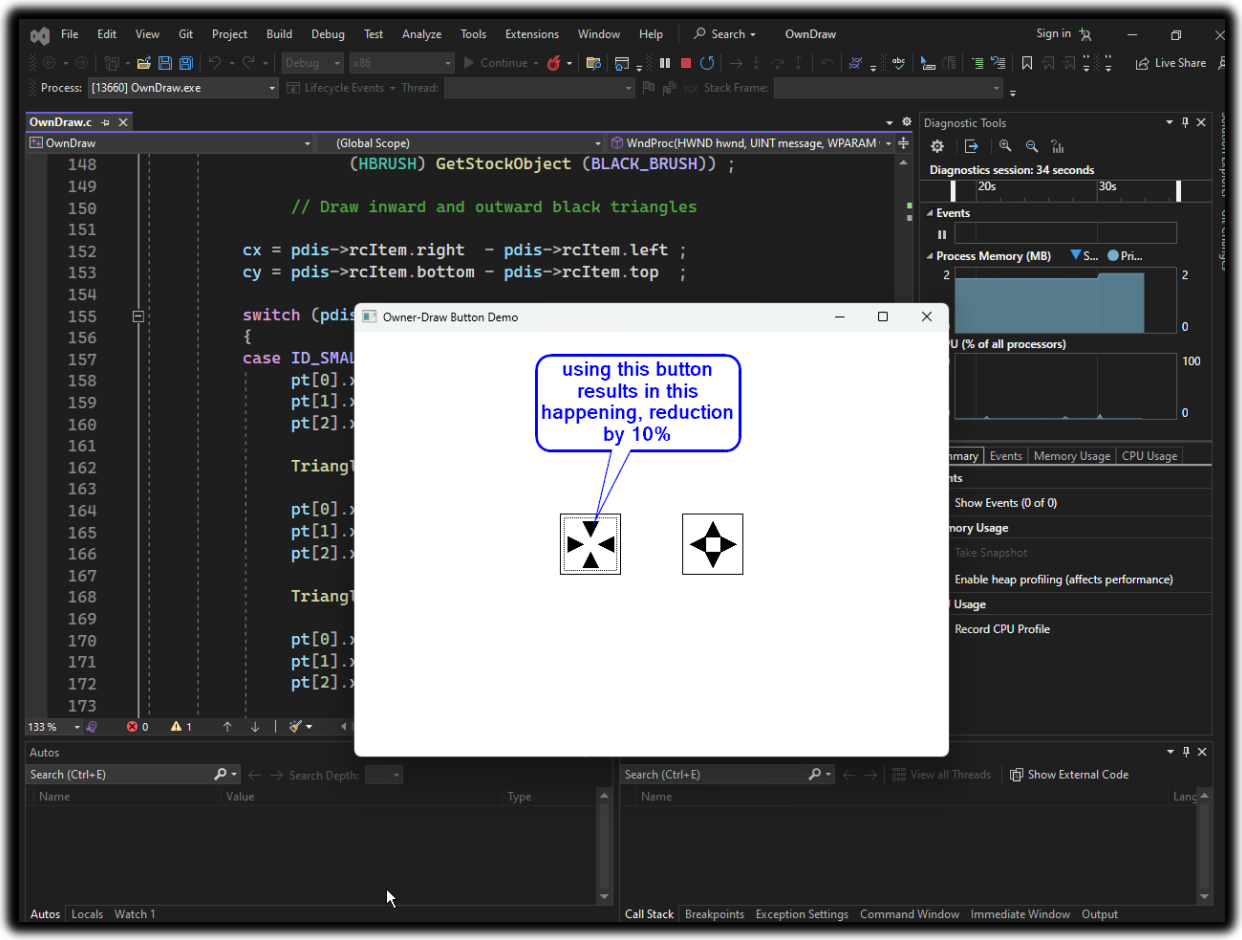
The **WM_DRAWITEM** message is responsible for drawing the buttons. It:

- Uses a Triangle function to draw triangles on the buttons.
- Uses InvertRect and DrawFocusRect to show button selection and focus.

Button Functionality

- The **left button** decreases the window size by 10% when clicked.
- The **right button** increases the window size by 10% when clicked.
- This works by changing the window's rectangle and calling MoveWindow to update the position and size.

The OWNDRAW program shows how owner-draw buttons let you **design custom button appearances**. They are **flexible and powerful**, but require **more coding** than standard buttons.



Owner-Draw Buttons (Summary)

- OWNDRAW creates two buttons with the **BS_OWNERDRAW** style. Their size is based on the system font, and they are positioned in the center of the window whenever the window is resized.
- Clicking the left button **shrinks** the window by 10%, and clicking the right button **grows** it by 10%. The program updates the window size and repositions the buttons automatically.
- Owner-draw buttons send a **WM_DRAWITEM** message when they need to be repainted. The message provides a DRAWITEMSTRUCT containing all the information needed to draw the button, including its size, device context, control ID, and state (pressed or focused).
- The OWNDRAW program draws the button using a white background, a black border, and four triangles. When the button is pressed, it inverts the colors, and when it has focus, it draws a dotted rectangle.
- **Important tips:** Always leave the device context in the state you found it, unselect any GDI objects you used, and do not draw outside the button's rectangle.

STATIC CLASSES IN C AND WINAPI

In C and WinAPI, **static classes** are used to create child window controls that **display content** but do **not interact with the user**. They are commonly used to show text, images, or other static content. Static controls are created using the CreateWindow function with the "static" window class.

Characteristics of Static Controls

- **No mouse or keyboard input:** Static controls cannot be focused and do not respond to mouse clicks or keyboard presses.
- **No WM_COMMAND messages:** They do not send messages to their parent window when interacted with.
- **Mouse click passes through:** Static controls trap WM_NCHITTEST messages and return HTTRANSPARENT, which allows mouse clicks to go through to the window underneath.

Types of Static Controls

1. **Rectangular static controls:** Draw a solid rectangle or frame in the client area. Colors are usually based on system colors.
2. **Text static controls:** Display text, which can be left-aligned, right-aligned, or centered.
3. **Icon static controls:** Display an icon. These are less commonly used.

Creating Static Controls

To create a static control, use CreateWindow with "static" as the window class. Key parameters include:

- **Parent window handle:** The handle of the parent window.
- **Window style:** Specifies whether it is a rectangular, text, or icon control.
- **Window text:** The text to display (ignored for rectangles).
- **X and Y coordinates:** The position of the upper-left corner.
- **Width and Height:** The size of the static control.

Customizing Static Controls

You can change the appearance of static controls by handling the WM_CTLCOLORSTATIC message, which the parent window receives **before the control is painted**. Using this message, you can:

- Change the **text color** with SetTextColor.
- Change the **background color** with SetBkColor.
- Return a **custom brush** to set a custom background pattern.

Example of Static Classes

Example of Static Classes
The code below creates a static control that simply displays the text "Hello, world!" in the client area of a parent window:

```
HWND hStaticControl = CreateWindow(
  "static",
  "Hello, world!",
  WS_VISIBLE | WS_CHILD,
  10,
  10,
  100,
  25,
  hParentWindow,
  (HMENU) 1,
  hInstance,
  NULL
);
```

Static classes are a handy way to add text or images to your WinAPI applications. They're easy to set up and customize, and they won't get in the way of the user interacting with other controls in your app. Here's the full table:

Static Window Style	Description
SS_BLACKRECT	Draws a black rectangle in the client area of the child window.
SS_BLACKFRAME	Draws a black frame in the client area of the child window.
SS_GRAYRECT	Draws a gray rectangle in the client area of the child window.
SS_GRAYFRAME	Draws a gray frame in the client area of the child window.
SS_WHITERECT	Draws a white rectangle in the client area of the child window.
SS_WHITEFRAME	Draws a white frame in the client area of the child window.
SS_ETCHEDHORZ	Creates a shadowed-looking frame with the white and gray colors, with a horizontal emphasis.
SS_ETCHEDVERT	Creates a shadowed-looking frame with the white and gray colors, with a vertical emphasis.
SS_ETCHEDFRAME	Creates a shadowed-looking frame with the white and gray colors, with both horizontal and vertical emphasis.
SS_LEFT	Creates left-justified text.
SS_RIGHT	Creates right-justified text.
SS_CENTER	Creates centered text.
SS_ICON	Not applicable to child window controls.
SS_USERITEM	Not applicable to child window controls.

SCROLL BAR CLASS

The **scroll bar class** is used to create child window scroll bars that can appear anywhere inside the client area of a parent window. Unlike button controls, scroll bars **do not send WM_COMMAND messages**. Instead, they send **WM_VSCROLL** and **WM_HSCROLL** messages when the user interacts with them.

Creating Scroll Bar Controls

To create a scroll bar control, use the CreateWindow function with the predefined "scrollbar" window class. You also specify one of the two scroll bar styles:

- **SBS_VERT** for vertical scroll bars
- **SBS_HORZ** for horizontal scroll bars

Key parameters for CreateWindow include:

- **Parent window handle:** The handle of the parent window.
- **Window style:** Either SBS_VERT or SBS_HORZ.
- **Window text:** Ignored for scroll bars.
- **X and Y coordinates:** The position of the upper-left corner of the scroll bar.
- **Width and Height:** The size of the scroll bar control.

Understanding the lParam Parameter

When processing scroll bar messages, you can tell the difference between a **window scroll bar** and a **scroll bar control** by checking the lParam parameter:

- lParam = 0 → a window scroll bar
- lParam = handle of scroll bar → a scroll bar control

Setting Scroll Bar Range and Position

You can control the scroll bar's range and position using the same functions as window scroll bars:

- **SetScrollRange**: Sets the minimum and maximum positions.
- **SetScrollPos**: Sets the current position.
- **SetScrollInfo**: Sets the minimum, maximum, page size, current position, and optionally other scroll bar information.

Customizing Scroll Bar Colors

You can trap the WM_CTLCOLORSCROLLBAR message to **customize the color** of the area between the two scroll bar buttons. This lets you change the appearance of the scroll bar control to match your app's theme.

Colors chapter 9.mp4

COLORS1 Program Overview

The **COLORS1 program** demonstrates working with child window controls, custom colors, and keyboard input in a WinAPI application. Its key features are:

Creating Child Window Controls

The program creates **10 child window controls**:

- ✓ 3 scroll bars
- ✓ 6 static text windows
- ✓ 1 static rectangle

All child controls are created using the CreateWindow function, which specifies the parent window, class, style, position, and size.

Customizing Scroll Bar Colors

The program traps the WM_CTLCOLORSCROLLBAR message to set the interior of the three scroll bars to red, green, and blue.
This is done by returning a brush created with CreateSolidBrush.

Customizing Static Text Colors

The program traps the WM_CTLCOLORSTATIC message to color static text.
Brushes for the text background are also created using CreateSolidBrush.

Keyboard Navigation (VK_TAB)

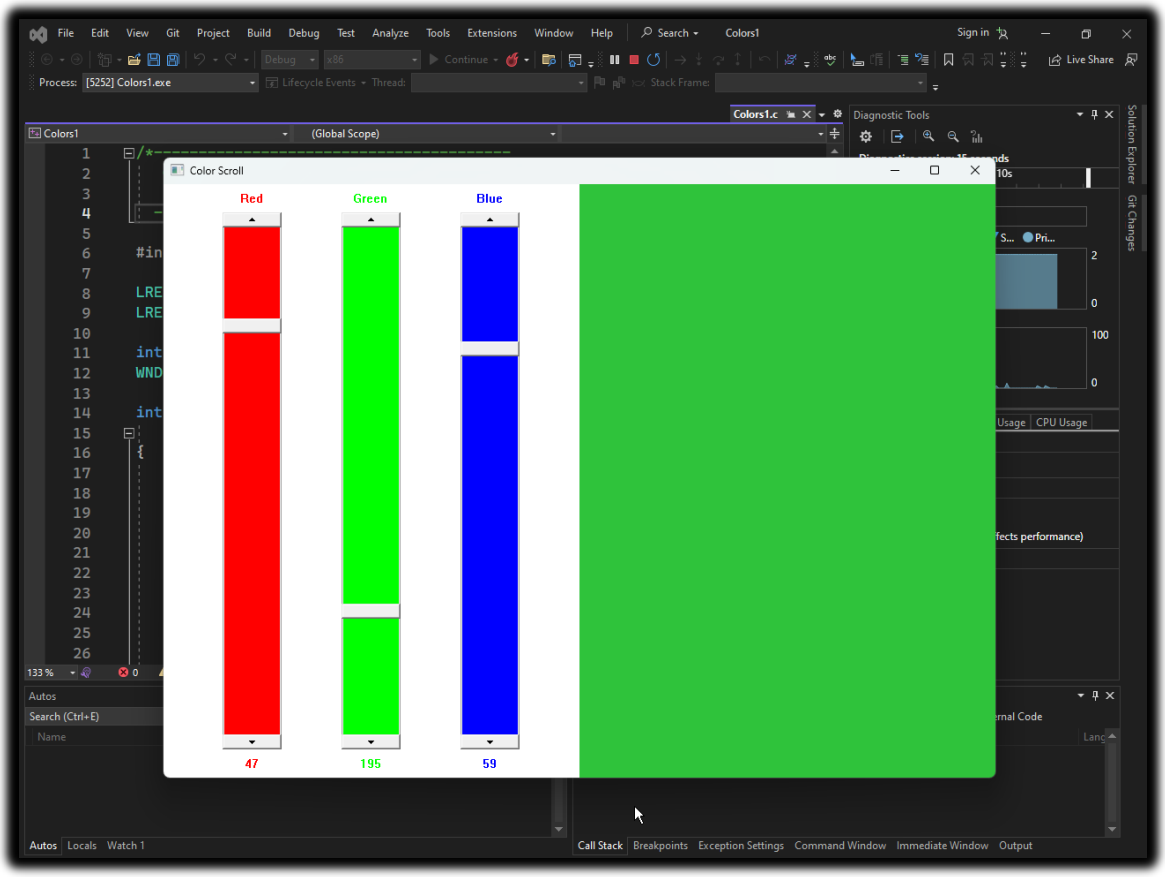
The VK_TAB key is used to switch focus between the three scroll bars.
SetFocus moves the focus to the next scroll bar in the tab order.

Handling Shift+Tab

The program uses GetKeyState to detect if the Shift key is pressed.
This changes the direction of tabbing so focus moves backward when Shift+Tab is used.

Default Message Handling

Messages that the program does not handle are passed to DefWindowProc to ensure the default window behavior is preserved.



Window Procedure (WndProc)

The WndProc handles messages from Windows and manages the program's behavior:

- **WM_CREATE:** Creates the child windows (scroll bars, static text, static rectangle).
- **WM_SIZE:** Resizes and repositions the child windows when the main window changes size.
- **WM_VSCROLL:** Updates the client area color and the text of static controls when a scroll bar is moved.
- **WM_CTLCOLORSCROLLBAR:** Colors the interior of scroll bars (red, green, blue).
- **WM_CTLCOLORSTATIC:** Colors the text and background of static controls.

Child Windows

Child windows are used to display scroll bars, color labels, and color values. They are created inside the main window and updated dynamically based on user interaction.

Window ID	Window Class	Style	Function
0	scrollbar	SBS_VERT	Red scroll bar
1	scrollbar	SBS_VERT	Green scroll bar
2	scrollbar	SBS_VERT	Blue scroll bar
3	static	SS_CENTER	Red label
4	static	SS_CENTER	Green label
5	static	SS_CENTER	Blue label
6	static	SS_CENTER	Red value
7	static	SS_CENTER	Green value
8	static	SS_CENTER	Blue value
9	static	SS_WHITERECT	White rectangle

Scroll Bar Handling (WM_VSCROLL)

When a scroll bar is scrolled:

- Identify which scroll bar sent the message.
- Get its new value.
- Update the client area color accordingly.
- Update the static text showing the scroll bar value.

Coloring Scroll Bars (WM_CTLCOLORSCROLLBAR)

- Identify which scroll bar is being drawn.
- Create a brush of the appropriate color (red, green, or blue).
- Return the brush to Windows to paint the scroll bar.

Coloring Static Controls (WM_CTLCOLORSTATIC)

- Identify which static control is being drawn.
- Set the text and background colors.
- Return a brush to paint the control.

Scroll bars in WinAPI are straightforward to create and customize. They can be used to control colors, values, and positions of other controls, making them versatile for interactive applications like COLORS1.

WINDOW SUBCLASSING

Window subclassing lets you change how an existing window handles messages.
It's useful for adding features or modifying behavior.

Steps to subclass a window:

1. Get the original window procedure using GetWindowLong with GWL_WNDPROC.
2. Replace it with your own procedure using SetWindowLong.
3. In your procedure, handle the messages you want, and pass the rest to the original procedure.

Example: Jumping Between Scroll Bars

You can use subclassing to catch WM_KEYDOWN messages.

If the Tab key is pressed, move the focus to the next scroll bar in the array.

Here's the subclassing procedure code:

```
// --- SECTION 3: Subclassing and Keyboard Interception ---
// From image_11a4a5.png
LRESULT CALLBACK SubclassProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        case WM_KEYDOWN:
        {
            // Intercepting the TAB key to cycle focus between controls
            if (wParam == VK_TAB)
            {
                // Retrieve current ID and calculate the index for the next control
                int idFocus = GetWindowLong(hwnd, GWL_ID);
                int nextFocus = (idFocus + 1) % MAX_SCROLLBARS;
                // Shift focus to the next scroll bar in the array
                SetFocus(hwndScroll[nextFocus]);
                return 0;
            }
        }
        break;
        default:
            // Pass all other messages to the original Window Procedure
            return CallWindowProc((WNDPROC)GetWindowLong(hwnd, GWL_WNDPROC), hwnd, msg, wParam, lParam);
    }

    return 0;
}
```

Subclassing is hijacking a window's "brain" (the Window Procedure) to change how it behaves. You catch messages before the original window can see them.

To take control, use SetWindowLong (or SetWindowLongPtr on x64). You must save the old address to keep the window from crashing.

```
// Save the original 'brain' address and swap it with our SubclassProc
oldWndProc = (WNDPROC)SetWindowLong(hWnd, GWL_WNDPROC, (LONG)SubclassProc);
```

Inside your SubclassProc, handle the messages you want. For everything else, pass the data back to the original procedure so the window stays functional.

When finished, put the original "brain" back where it belongs.

This can be done by calling the SetWindowLong function with the GWL_WNDPROC parameter and the address of the original window procedure.

Here is the code for removing the subclassing procedure:

```
// Restore the original address
SetWindowLong(hWnd, GWL_WNDPROC, (LONG)oldWndProc);
```

- **The Goal:** Make scroll bars respond to the **Tab** key (normally, scroll bars only handle input when focused).
- **The Method:** Use **Subclassing** to intercept the WM_KEYDOWN message.
- **The Mechanism:**
  1. Use GetWindowLong to find the original address.
  2. Use SetWindowLong to inject your own procedure.
- **The Golden Rule:** Always call the original procedure for messages you don't modify. If you don't, the window "dies" because it forgets how to perform its basic duties.

Simple Logic Flow

1. **Intercept:** Catch WM_KEYDOWN.
2. **Check:** Is it the VK_TAB key?
3. **Action:** If yes, shift focus to the next scroll bar.
4. **Pass-through:** Send everything else to the original procedure.

```
LRESULT CALLBACK SubclassProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    if (msg == WM_KEYDOWN && wParam == VK_TAB) {
        // "I'll take it from here."
        // Logic to jump to the next scroll bar goes here.
        return 0;
    }

    // "Not my business."
    // Hand the message back to the original owner so the window doesn't crash.
    return CallWindowProc(oldWndProc, hwnd, msg, wParam, lParam);
}
```

SETTING THE BACKGROUND BRUSH

1. Initial Background Brush

- When COLORS1 defines its window class, it sets the client area background to black.
- A solid black brush is created using CreateSolidBrush(0) and assigned to the hbrBackground member of the WNDCLASSEX structure.

2. Updating the Background Color

- When the scroll bar values change, COLORS1 updates the background color.
- A new brush is created with the color determined by the RGB values of the three scroll bars using CreateSolidBrush(RGB(r, g, b)).
- SetClassLong is used to assign this new brush to hbrBackground.

3. Deleting the Old Brush

- After setting the new brush, the old brush must be deleted to free resources using DeleteObject(oldBrush).

4. Invalidating the Client Area

- After changing the brush, the client area is invalidated using InvalidateRect(hwnd, NULL, TRUE).
- This tells Windows to repaint the client area.
- Passing TRUE ensures the background is erased before repainting.

5. WM_PAINT and WM_ERASEBKGND Messages

- **WM_PAINT:** COLORS1 does not handle it directly; DefWindowProc is called, which uses BeginPaint and EndPaint to validate the window.
- **WM_ERASEBKGND:** COLORS1 ignores it; Windows erases the background using the brush in the window class.

6. Cleaning Up

On WM_DESTROY, COLORS1 deletes the old brush with DeleteObject to free resources. COLORS1 colors its background by creating a new brush, assigning it to hbrBackground, deleting the old brush, and invalidating the client area so Windows repaints it with the new color.

COLORING THE SCROLL BARS

1. Scroll Bar Brushes

- COLORS1 creates three brushes for the scroll bars: red, green, and blue.
- These brushes are created during WM_CREATE using CreateSolidBrush.
- The crPrim array holds the RGB values for the brushes.

2. Applying Brushes

- When WM_CTLCOLORSCROLLBAR is received, WndProc returns the appropriate brush for the scroll bar based on its ID.
- The ID is obtained with GetWindowLong(hwndScrollBar, GWL_ID).

3. Cleanup

The brushes are destroyed in WM_DESTROY using DeleteObject to avoid memory leaks.

COLORING THE STATIC TEXT

1. Setting Text and Background Colors

- The text color matches the color of the corresponding scroll bar using SetTextColor.
- The background color is set to the system color COLOR_BTNHIGHLIGHT using SetBkColor.

2. Using a Brush for Background

- To prevent the background color from changing if the system color changes, COLORS1 creates a brush of COLOR_BTNHIGHLIGHT during WM_CREATE.
- This brush is returned when handling WM_CTLCOLORSTATIC.
- The brush is destroyed during WM_DESTROY to avoid memory leaks.

3. Handling System Color Changes

- COLORS1 processes WM_SYSCOLORCHANGE to recreate the hBrushStatic brush with the updated COLOR_BTNHIGHLIGHT.
- This keeps the static text background color consistent with the system colors.

Here is the code for creating and destroying the brushes:

```
// Create the brushes
for (i = 0; i < 3; i++) {
  hBrush[i] = CreateSolidBrush(crPrim[i]);
}

// Destroy the brushes
for (i = 0; i < 3; i++) {
  DeleteObject(hBrush[i]);
}
```

Here is the code for handling the WM_CTLCOLORSCROLLBAR message:

```
case WM_CTLCOLORSCROLLBAR:
  i = GetWindowLong((HWND)lParam, GWL_ID);
  return (LRESULT)hBrush[i];
```

Here is the code for handling the WM_CTLCOLORSTATIC message:

```
case WM_CTLCOLORSTATIC:
  SetTextColor(hdc, GetTextColor(hdc));
  SetBkColor(hdc, COLOR_BTNHIGHLIGHT);
  return (LRESULT)hBrushStatic;
```

Here is the code for handling the WM_SYSCOLORCHANGE message:

```
case WM_SYSCOLORCHANGE:
  DeleteObject(hBrushStatic);
  hBrushStatic = CreateSolidBrush(COLOR_BTNHIGHLIGHT);
  return 0;
```

Popad1 program inside the chapter 9 folder…

POPPAD1.mp4

POPPAD1 is a small multiline text editor in C using WinAPI.

- It is under 100 lines of code and does **not** handle files.
- Users can type, move the cursor, select text, delete, copy, and paste from the clipboard.
- The code is split into **WinMain** (setup) and **WndProc** (message handling).

WinMain Function

- Registers the window class with style, procedure, icon, cursor, background, and class name.
- Creates the main window using CreateWindow.
- Shows and updates the window using ShowWindow and UpdateWindow.

WndProc Function

Handles messages for the main window:

1. **WM_CREATE** – Creates the edit control using CreateWindow.
2. **WM_SETFOCUS** – Sets input focus to the edit control using SetFocus.
3. **WM_SIZE** – Resizes the edit control using MoveWindow.
4. **WM_COMMAND** – Handles notifications like EN_ERRSPACE and EN_MAXTEXT from the edit control.
5. **WM_DESTROY** – Posts a quit message with PostQuitMessage.

Edit Control Styles

- **Text Justification:** ES_LEFT, ES_RIGHT, ES_CENTER.
- **Multi-line Editing:** ES_MULTILINE for multi-line text.
- **Horizontal Scrolling:** ES_AUTOHSCROLL for single-line edit controls.
- **Vertical Scrolling:** ES_AUTOVSCROLL for multi-line edit controls.
- **Scroll Bars:** WS_HSCROLL and WS_VSCROLL to add scroll bars.
- **Border:** WS_BORDER draws a border around the edit control.
- **Selection Highlighting:** ES_NOHIDESEL keeps text selected even when the control loses focus.

Poppad1 Edit Control

The POPPAD1 program creates an edit control with the following styles:

STYLE	DESCRIPTION
WS_CHILD	Specifies that the window is a child window.
WS_VISIBLE	Specifies that the window is initially visible.
WS_HSCROLL	Specifies a horizontal scroll bar.
WS_VSCROLL	Specifies a vertical scroll bar.
WS_BORDER	Specifies a border window style.
ES_LEFT	Specifies a left-aligned edit control.
ES_MULTILINE	Specifies a multiline edit control.
ES_AUTOHSCROLL	Automatically scrolls horizontally when needed.
ES_AUTOVSCROLL	Automatically scrolls vertically when needed.

Edit Control Summary (Child Window)

- The **edit control** is a **child window** of the main window.
- It is **visible** and includes **horizontal and vertical scroll bars**.
- It has a **border**, is **left-justified**, supports **multi-line editing**, and automatically scrolls both horizontally and vertically.
- It does **not hide the selection** when it loses input focus (ES_NOHIDESEL).

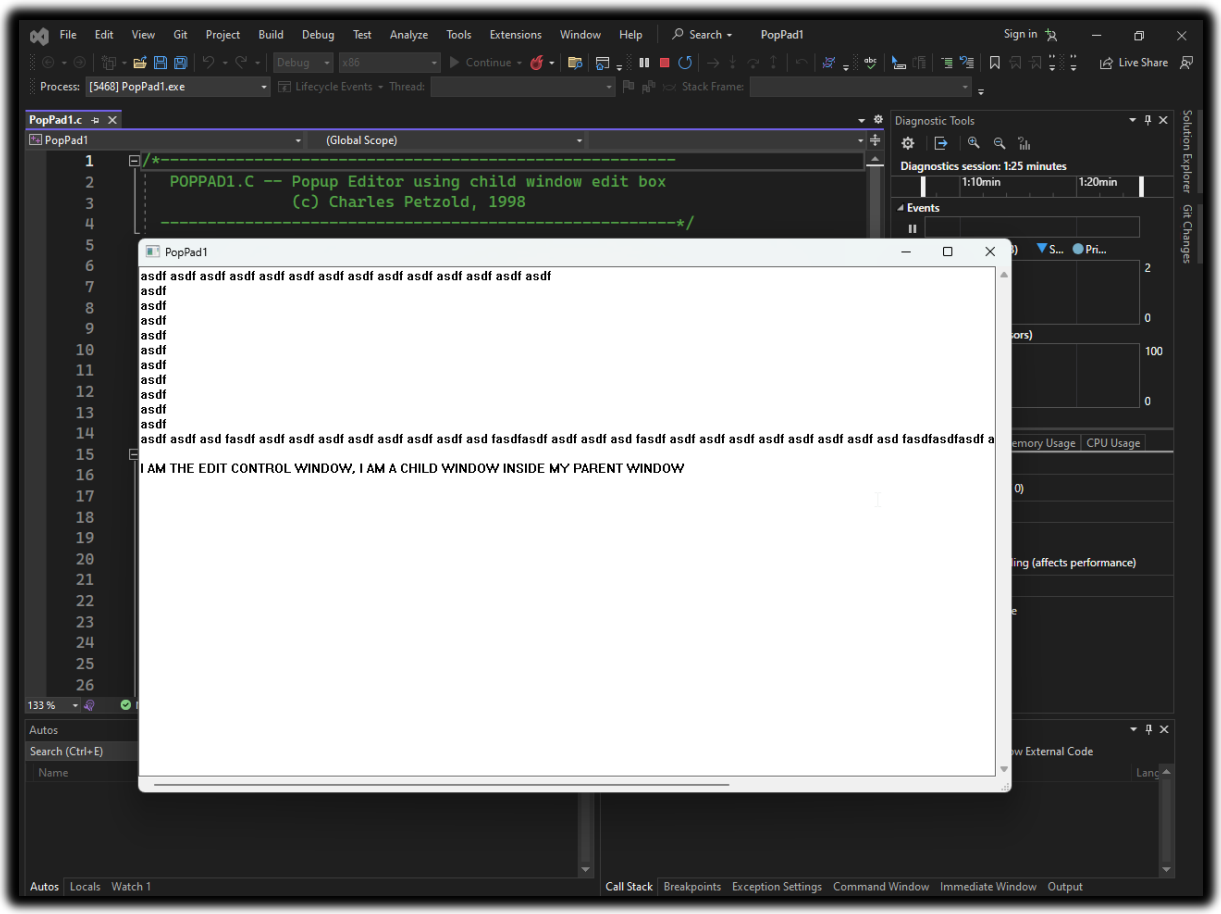
Resizing the Edit Control

- When the main window is resized, the WM_SIZE message is received.
- The edit control is resized to fill the window using:

```
MoveWindow(hwndEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);
```

- (0, 0) sets the top-left corner of the edit control.
- LOWORD(lParam) and HIWORD(lParam) set the width and height to match the main window.
- TRUE tells Windows to repaint the control after resizing.

The **edit control window class** is very flexible, letting you control appearance and behavior through its styles, making it ideal for text editing in Windows applications.



In POPPAD1, the whole window you see is actually the edit control.
The program creates one edit control as a child and resizes it to fill the main window.
This makes the edit control look like the main window itself.

Edit Control Notifications

Edit controls send WM_COMMAND messages to their parent to report events.
These events include text changes or scroll bar actions.
The wParam and lParam of the message carry details about the event.

Notification Codes

The notification code is in the high-order word (HIWORD) of wParam.
Each code tells what kind of event happened (like text changed or scrolling).

Notification Code	Meaning
EN_SETFOCUS	The edit control has gained the input focus.
EN_KILLFOCUS	The edit control has lost the input focus.
EN_CHANGE	The edit control's contents will change.
EN_UPDATE	The edit control's contents have changed.
EN_ERRSPACE	The edit control has run out of space.
EN_MAXTEXT	The edit control has run out of space on insertion.
EN_HSCROLL	The edit control's horizontal scroll bar has been clicked.
EN_VSCROLL	The edit control's vertical scroll bar has been clicked.

lParam Parameter

The lParam of a WM_COMMAND message contains the handle of the edit control that sent the notification.

Poppad1 Notification Handling

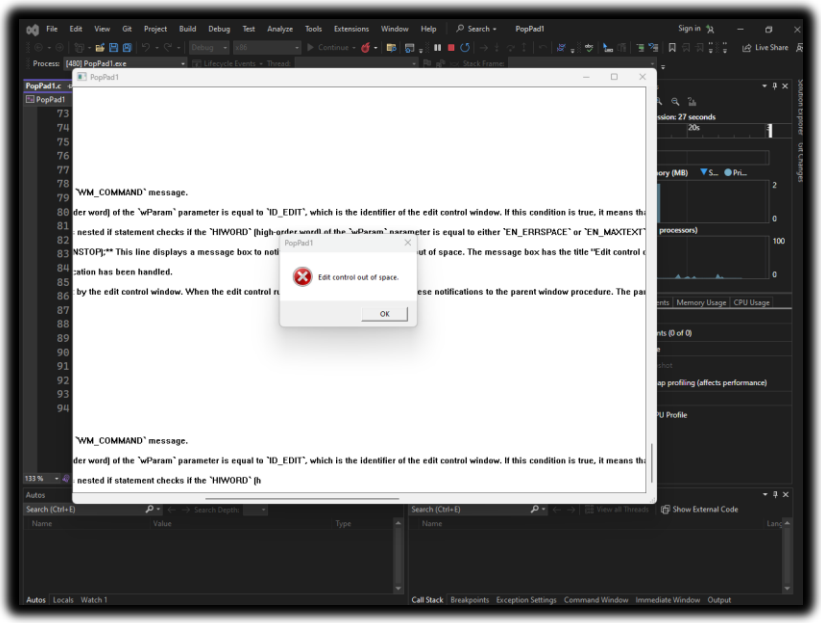
POPPAD1 only handles EN_ERRSPACE and EN_MAXTEXT notifications.
When these occur, the program shows a message box to the user.
This happens if the edit control runs out of memory or exceeds its text limit.

Here’s the code for handling EN_ERRSPACE and EN_MAXTEXT:

```
case WM_COMMAND:
  if (LOWORD(wParam) == ID_EDIT) {
    if (HIWORD(wParam) == EN_ERRSPACE ||
      HIWORD(wParam) == EN_MAXTEXT) {
      MessageBox(hwnd, TEXT("Edit control out of space."),
            szAppName, MB_OK | MB_ICONSTOP);
      return 0;
    }
  }
  return 0;
```

- **Purpose:** Notifications let the program respond to events in an edit control, such as when it runs out of space.
- **WM_COMMAND:** The parent window receives this message whenever a control sends a notification.
- **Identify the source:** Check if the notification comes from the edit control (ID_EDIT).
- **Identify the notification type:** Check if it is EN_ERRSPACE or EN_MAXTEXT, which indicate that the edit control has run out of space.
- **Respond:** Show a message box to notify the user that the edit control is full.
- **Finish handling:** Return 0 to indicate the notification has been handled.

Edit control notifications are essential for tracking events and responding to them, such as handling out-of-space errors in POPPAD1.



The code shows a message box when the edit control runs out of space by handling the EN_ERRSPACE and EN_MAXTEXT notifications.

USING EDIT CONTROLS

Edit Controls and Tab Key Handling

Edit controls: Let users enter and edit text; can be single-line or multi-line.

Multiple edit controls: Tab and Shift-Tab can be used to move input focus between them.

Window subclassing: Allows intercepting key messages (like WM_KEYDOWN) for an edit control.

Tab key handling:

- ✓ Pressing **Tab** moves focus to the next edit control.
- ✓ Pressing **Shift+Tab** moves focus to the previous edit control.

Similar concept: Works like COLORS1 program's tabbing between scroll bars.

Subclassing gives full control over keyboard navigation between edit controls.

```
LRESULT CALLBACK EditSubclassedProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
    case WM_KEYDOWN:
        if (wParam == VK_TAB)
        {
            HWND hNextEdit = GetNextDlgItem(hwnd, wParam);
            if (hNextEdit)
            {
                SetFocus(hNextEdit);
                return 0;
            }
        }
        break;
    }

    return DefSubclassProc(hwnd, msg, wParam, lParam);
}
```

Subclassing Edit Controls for Key Handling

Subclass procedure (EditSubclassedProc):

- Intercepts WM_KEYDOWN messages for the edit control.
- If **Tab** is pressed, moves the input focus to the next edit control.

Handling the Enter key:

- Can be used to move focus to the next edit control.
- Can also act as a signal that all edit fields are complete.

Inserting text:

- Use SetWindowText to set the text of the edit control.

Subclassing allows custom behavior for Tab, Enter, and text manipulation in edit controls.

```
SetWindowText(hwndEdit, TEXT("This is some text."));
```

This code sets the text of the edit control with the handle hwndEdit to "This is some text.".

Retrieving Text from an Edit Control

Get text length: Use GetWindowTextLength to find out how many characters are in the edit control.

Get text content: Use GetWindowText to retrieve the actual text from the edit control.

Always check the text length first so you can allocate enough space to store the text.

```
int length = GetWindowTextLength(hwndEdit);
if (length > 0)
{
    TCHAR buffer[length + 1];
    GetWindowText(hwndEdit, buffer, length + 1);
    MessageBox(hwnd, buffer, TEXT("Text from edit control"), MB_OK);
}
```

Get text length: Use GetWindowTextLength(hwndEdit) to determine how many characters are in the edit control.

Allocate buffer: If the length is greater than zero, allocate a buffer large enough to hold the text.

Retrieve text: Use GetWindowText(hwndEdit, buffer, length + 1) to copy the text into the buffer.

Display text: You can then display the text in a message box or use it in your program.

Key Point: Always allocate space for the null terminator (length + 1) when getting the text.

Messages to Edit Controls

There are many messages that you can send to an edit control using the SendMessage function. Here are some of the most common messages:

WM_CUT: This message removes the current selection from the edit control and sends it to the clipboard.

WM_COPY: This message copies the current selection to the clipboard but leaves it intact in the edit control.

WM_CLEAR: This message deletes the current selection from the edit control without passing it to the clipboard.

WM_PASTE: This message inserts the text from the clipboard at the cursor position in the edit control.

```
SendMessage(hwndEdit, WM_CUT, 0, 0);

SendMessage(hwndEdit, WM_COPY, 0, 0);

SendMessage(hwndEdit, WM_CLEAR, 0, 0);

SendMessage(hwndEdit, WM_PASTE, 0, 0);
```

EM_GETSEL: This message gets the starting and ending positions of the current selection in the edit control.

```
int start, end;
SendMessage(hwndEdit, EM_GETSEL, (WPARAM)&start, (LPARAM)&end);
```

EM_SETSEL: This message sets the selection in the edit control to the specified starting and ending positions.

```
SendMessage(hwndEdit, EM_SETSEL, 0, 10);
```

EM_REPLACESEL: This message replaces the current selection with the specified text.

```
SendMessage(hwndEdit, EM_REPLACESEL, 0, (LPARAM)"This is some text.");
```

EM_REPLACESEL Message in Edit Controls

Purpose: Replaces the currently selected text in an edit control with new text.

Parameters:

- wParam – Determines if the operation can be undone.
  - ✓ TRUE → operation can be undone.
  - ✓ FALSE → operation cannot be undone.
- lParam – Pointer to a null-terminated string containing the replacement text.

Return Value: None.

Undoing the Operation: If wParam is TRUE, you can undo the replacement by sending the EM_UNDO message to the edit control.

This is useful for programmatically inserting or replacing text in an edit control while optionally allowing undo.

```
SendMessage(hwndEdit, EM_UNDO, 0, 0);
```

Using EM_REPLACESEL to Insert Text

Purpose: EM_REPLACESEL can be used to insert text at the current selection or cursor position in an edit control.

How it Works:

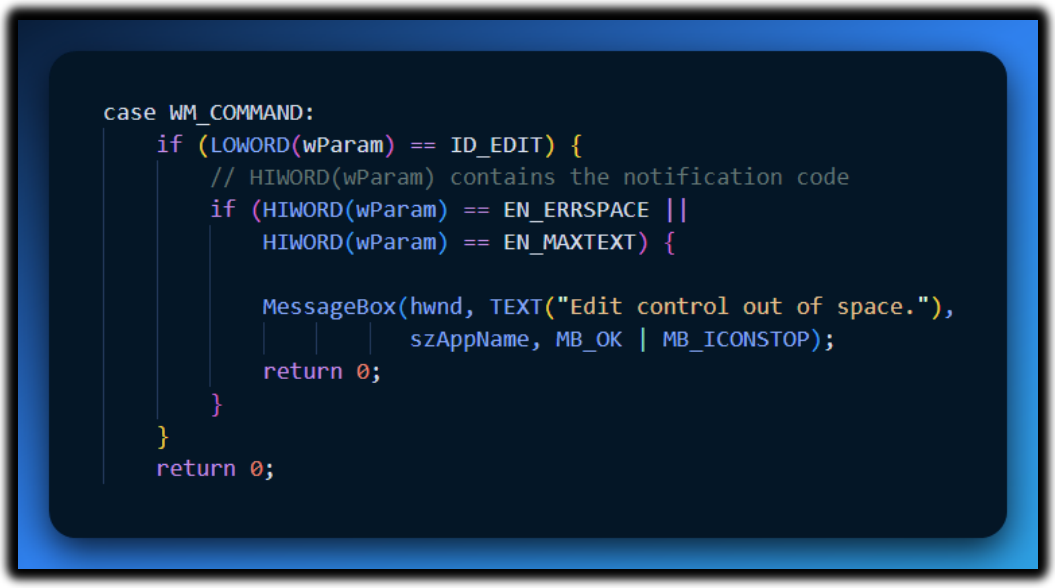
- The current selection (or caret position) is replaced with the specified text.
- If no text is selected, the new text is inserted at the cursor.
- The operation can be made undoable by setting wParam to TRUE.

Parameters:

- wParam – TRUE to allow undo, FALSE to disallow undo.
- lParam – Pointer to the null-terminated string to insert.

Example Use Case: Inserting "This is some text." at the current cursor position in an edit control.

Key Point: EM_REPLACESEL is a convenient way to insert or replace text without manually manipulating the edit control's buffer.



```
case WM_COMMAND:
    if (LOWORD(wParam) == ID_EDIT) {
        // HIWORD(wParam) contains the notification code
        if (HIWORD(wParam) == EN_ERRSPACE ||
            HIWORD(wParam) == EN_MAXTEXT) {

            MessageBox(hwnd, TEXT("Edit control out of space."),
                       szAppName, MB_OK | MB_ICONSTOP);
            return 0;
        }
    }
    return 0;
```

Multiline Edit Controls

Purpose: Allow editing of multiple lines of text in an edit control.

Key Messages:

- EM_GETLINECOUNT – Returns the total number of lines in the edit control.
- EM_LINEINDEX – Returns the starting character index of a specified line.
- EM_LINELENGTH – Returns the length (number of characters) of a specified line.
- EM_GETLINE – Copies the text of a specified line into a buffer.

Key Point: These messages make it easy to access or manipulate individual lines in a multiline edit control.

WHAT IS A LISTBOX?

A **Listbox** is a standard control that displays a scrollable list of strings. It's like a restaurant menu: you look at the options and pick what you want. Key Features:

- **Scrolling:** If the list is too long, it adds a scrollbar automatically (if you ask for it).
- **Highlighting:** When you click an item, it turns blue (inverted colors) to show it is selected.
- **Navigation:** You can use the Mouse or the Keyboard (Arrow keys, Page Up/Down).

Creating a Listbox

You create it just like a button, but you use the class name "listbox".

```
hwndList = CreateWindow(
    TEXT("listbox"),      // Class Name
    NULL,                 // Window Text (Listboxes don't use this title)
    WS_CHILD | WS_VISIBLE | LBS_NOTIFY | WS_VSCROLL | WS_BORDER, // Styles
    x, y, width, height,
    hwndParent, (HMENU)ID, hInstance, NULL
);
```

The Essential Styles

Listboxes are plain by default. You almost *always* need to add these specific styles to make them useful:

1. **LBS_NOTIFY (Crucial):**
    - ✓ **Without this:** The listbox is "mute." It won't tell the parent window when the user clicks something.
    - ✓ **With this:** The listbox sends WM_COMMAND messages to the parent whenever the user selects an item.
2. **WS_VSCROLL:** Adds a vertical scrollbar so users can see items hidden at the bottom.
3. **WS_BORDER:** Draws a thin black line around the box. Without it, the listbox just looks like floating text.

Selection Types: Single vs. Multiple

There are two ways a user can interact with the list:

1. Single Selection (Default)

- **Behavior:** The user clicks one item. If they click a different one, the first one gets deselected.
- **Use Case:** Picking a country, selecting a font size.

2. Multiple Selection (LBS_MULTIPLESEL)

- **Behavior:** The user can click "Apple," then "Banana," then "Orange," and all three stay highlighted.
- **Use Case:** Selecting files to delete, choosing pizza toppings.

Performance Tip: The "Redraw" Problem

Imagine you want to add 1,000 items to your list.

- **The Bad Way:** You add Item 1 (Listbox redraws). You add Item 2 (Listbox redraws)... You add Item 1000 (Listbox redraws). The screen flickers like crazy and it is slow.
- **The Good Way (WM_SETREDRAW):** You tell the listbox "Stop painting!" You add all 1,000 items in the dark. Then you tell it "Okay, paint now."

Code Pattern:

```
// 1. Freeze
SendMessage(hwndList, WM_SETREDRAW, FALSE, 0);

// 2. Add 1000 items (Loop)
// ... processing ...

// 3. Unfreeze
SendMessage(hwndList, WM_SETREDRAW, TRUE, 0);
```

*Note: Your notes mentioned LBS_NOREDRAW. This is a style, not a command. It creates a listbox that **never** updates visually, which is rarely what you want. Using the WM_SETREDRAW message is the correct technique for batch updates.*

Explain Like I'm a Teenager: The Listbox

The Vending Machine Think of a Listbox like a digital Vending Machine window.

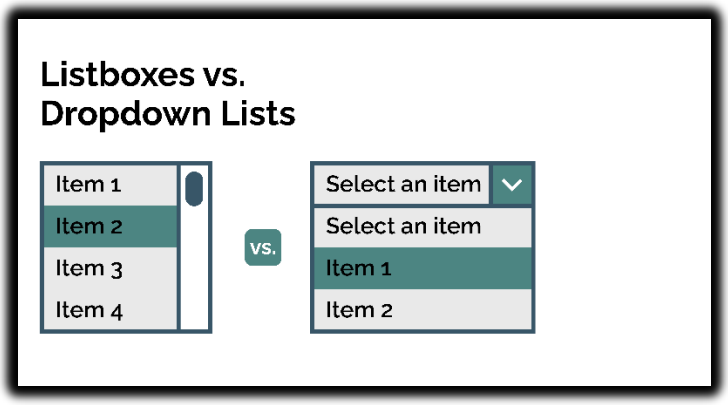
1. **The Window (CreateWindow):** You build the glass box.
2. **The Styles (WS_BORDER, WS_VSCROLL):** You put a frame around it and make sure the rows can rotate so people can see the Cheetos at the back.
3. **LBS_NOTIFY:** This is the most important part. By default, a vending machine doesn't scream "SOMEONE BOUGHT CHIPS!"
    - ✓ If you *don't* use LBS_NOTIFY, the machine takes the money and dispenses the chips silently. You (the Program) have no idea it happened.
    - ✓ If you *do* use LBS_NOTIFY, the machine buzzes your phone every time someone presses a button. "Hey! User selected Item #3!"

Quick Review

Question 1: If you create a Listbox but forget the WS_BORDER style, what does it look like? *(Answer: It looks like a plain list of text floating on the background with no box around it.)*

Question 2: Why is LBS_NOTIFY necessary if you want to handle clicks? *(Answer: Without it, the Listbox does not send WM_COMMAND messages to the parent window when a selection is made.)*

Question 3: You have a loop adding 5,000 files to a listbox and the app freezes and flickers. What message fixes this? *(Answer: WM_SETREDRAW (sending FALSE before the loop and TRUE after).)*



Standard List Box Style

The Windows header files define a list box style called LBS_STANDARD that includes the most commonly used styles. It is defined as:

Style	Description
LBS_NOTIFY	Enables the list box to send notifications to the parent window.
LBS_SORT	Sorts strings in the list box alphabetically.
WS_VSCROLL	Adds a vertical scroll bar to the list box.
WS_BORDER	Creates a border window style.

📦 List Box Styles — what that combo really means

You're basically saying:

- **LBS_NOTIFY** → tell the parent when the user does things (selects, double-clicks, etc.). This is how your window procedure actually *knows* stuff happened.
- **LBS_SORT** → auto-sort items alphabetically. If your items aren't meant to be sorted, don't use this — it *will* rearrange them behind your back.
- **WS_VSCROLL** → force a vertical scrollbar so the user can scroll through items.
- **WS_BORDER** → just gives it a visible border instead of that "flat grey void" look.

That combo basically says:

"This list box talks to the parent, sorts itself, can scroll vertically, and actually looks like a control instead of a ghost."

No magic, just flags stacked together.

🖐️ Resizing and moving list boxes — reality check

You mentioned **WS_SIZEBOX** and **WS_CAPTION**.

Yeah, technically you *can* slap those on and turn the list box into a weird mini window you can drag/resize. But in real UI design, that's just chaos:

- users don't expect list boxes to behave like independent windows
- it breaks consistency with every normal Windows app
- it looks like 1995 shareware

So truthfully:

You *can* — but you really shouldn't. Leave resizing responsibility to the parent dialog/window and handle it in WM_SIZE.

That's the clean way.

📏 List Box Dimensions

To prevent clipped text and UI "vibes," use exact system values.

1. Calculating Width

The total width must fit the text **and** the scrollbar.

Formula: listbox_width = longest_string_pixels + GetSystemMetrics(SM_CXVSCROLL)

The Scrollbar: Never hardcode this. Use GetSystemMetrics(SM_CXVSCROLL) to get the exact pixel width based on the user's current DPI and theme.

The Text: Never count characters (e.g., "W" is wider than "i").

- Get the Device Context (DC).
- Select the List Box font.
- Measure with GetTextExtentPoint32.

2. Calculating Height

The height is determined by the number of rows you want visible.

Formula: listbox_height = char_height * desired_number_of_items

COMPONENT	WINAPI FUNCTION	WHY?
Scrollbar Width	`GetSystemMetrics(SM_CXVSCROLL)`	*Accounts for system scaling, high DPI settings, and custom desktop themes.*
String Width	`GetTextExtentPoint32`	*Gets the exact pixel length of a specific string based on the current font.*
Line Height	`GetTextMetrics`	*Gets the height of a single character (plus internal leading) to space lines correctly.*

Adding Strings to a List Box

This is the most common way to put text into your list.

- **Behavior:** It appends the string to the **end** of the list.
- **Exception:** If your listbox has the LBS_SORT style (which sorts items alphabetically automatically), the string will be inserted in the correct alphabetical position, not necessarily at the end.

```
// Syntax: SendMessage(hwndList, LB_ADDSTRING, Unused, StringPointer);
SendMessage(hwndList, LB_ADDSTRING, 0, (LPARAM)TEXT("This is a string"));
```

- **wParam:** Set to 0 (unused).
- **lParam:** The actual text string. (You must cast it to LPARAM).

Inserting Strings (LB_INSERTSTRING)

Use this when exact placement matters. You want to put an item **at a specific line number**, pushing everything else down.

```
// Syntax: SendMessage(hwndList, LB_INSERTSTRING, Index, StringPointer);
// Insert at Index 2 (The 3rd slot)
SendMessage(hwndList, LB_INSERTSTRING, 2, (LPARAM)TEXT("I'm cutting in line!"));
```

Understanding "Index 2": Listboxes are **0-indexed**.

- **Index 0:** First item
- **Index 1:** Second item
- **Index 2:** Third item (This is where your new string goes).

Deleting Strings (LB_DELETESTRING)

To remove a single item, you must know its **Index Number**.

```
// Syntax: SendMessage(hwndList, LB_DELETESTRING, Index, Unused);
// Delete the item at Index 3 (The 4th item visually)
SendMessage(hwndList, LB_DELETESTRING, 3, 0);
```

- **Warning:** Once you delete Item 3, the old Item 4 slides up to become the *new* Item 3. The indices shift immediately!

Clearing the List (LB_RESETCONTENT)

This is the "Nuke" button. It deletes **every single item** in the listbox instantly.

```
SendMessage(hwndList, LB_RESETCONTENT, 0, 0);
```

Explain Like Ur a Teenager: The Playlist

Think of a Listbox like a **Spotify Playlist**.

1. **LB_ADDSTRING (Add to Queue):** You click "Add to Queue." The song goes to the very bottom of the list. It waits its turn.
2. **LB_INSERTSTRING (Play Next):** You click "Play Next." You are forcing this song to jump the line and insert itself right at the top (or specifically where you dragged it), pushing all the other songs down.
3. **LB_DELETESTRING (Remove Song):** You accidentally added a country song to your rap playlist. You hit delete. It vanishes, and the songs below it slide up to fill the gap.
4. **LB_RESETCONTENT (Clear Queue):** The party is over. You hit "Clear Queue." The list is now empty and silence fills the room.

Quick Review

Question 1: If you have a list with items "A", "B", "C" (Indices 0, 1, 2), and you send LB_DELETESTRING with index 1, what happens? *(Answer: "B" is deleted. "C" slides up and becomes the new index 1.)*

Question 2: What value do you pass for wParam when using LB_ADDSTRING? *(Answer: 0. It is unused.)*

Question 3: Does LB_RESETCONTENT destroy the Listbox window itself? *(Answer: No. It only removes the text inside it. The window frame remains on screen, empty.)*

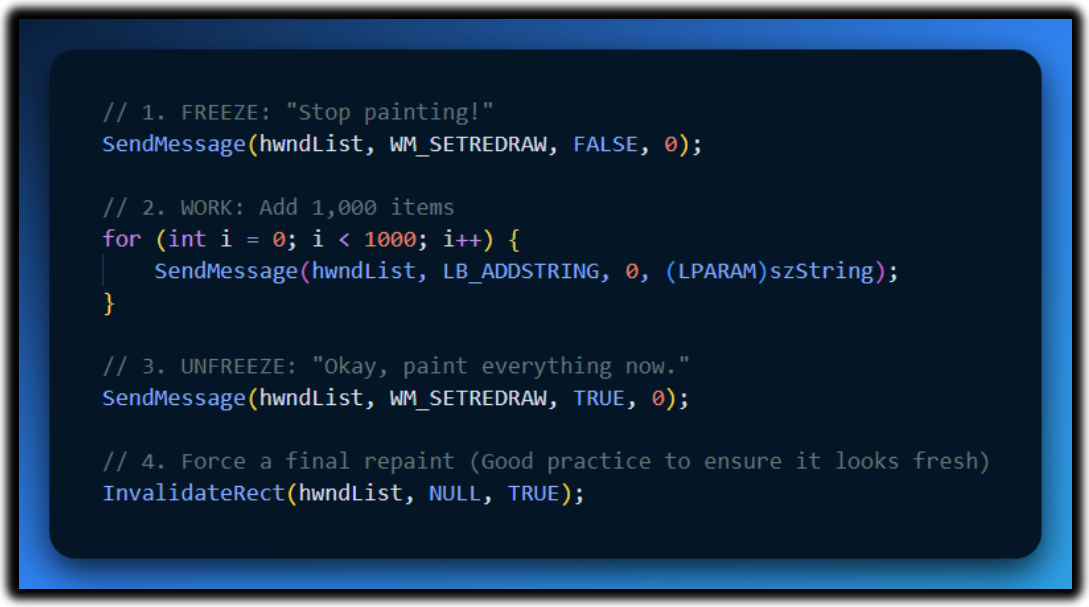
💥💥💥💥💥✅✅✅✅✅✅✅✅✅✅💥💥💥💥💥💥

MANAGING LISTBOX PERFORMANCE & SELECTION

1. Performance: The "Freeze" Trick (WM_SETREDRAW)

If you are adding 1,000 items to a list, the default behavior is slow because the Listbox tries to repaint itself after *every single item*.

- **Result:** The screen flickers, and the app feels laggy.
- **Solution:** You "freeze" the window, do the work, and then "unfreeze" it.



```
// 1. FREEZE: "Stop painting!"
SendMessage(hwndList, WM_SETREDRAW, FALSE, 0);

// 2. WORK: Add 1,000 items
for (int i = 0; i < 1000; i++) {
    SendMessage(hwndList, LB_ADDSTRING, 0, (LPARAM)szString);
}

// 3. UNFREEZE: "Okay, paint everything now."
SendMessage(hwndList, WM_SETREDRAW, TRUE, 0);

// 4. Force a final repaint (Good practice to ensure it looks fresh)
InvalidateRect(hwndList, NULL, TRUE);
```

Analogy: Imagine you are stocking a vending machine. You don't open the door, put in one Snickers, close the door, lock it, unlock it, open it, put in one Twix... You open the door (**FALSE**), stock everything at once, and then close it (**TRUE**).

---

2. Error Handling: Did it work?

Unlike void functions, SendMessage returns a value. When working with Listboxes, you should occasionally check this value to make sure your command worked.

Common Return Values:

- **LB_ERR (-1):** General error (e.g., invalid index).
- **LB_ERRSPACE (-2):** Out of memory (Rare on modern PCs, but possible if you add millions of strings).

Example Check:

```
int iResult = SendMessage(hwndList, LB_ADDSTRING, 0, (LPARAM)"New Item");
if (iResult == LB_ERRSPACE) {
    MessageBox(hwnd, TEXT("List is full!"), TEXT("Error"), MB_OK);
}
```

---

3. Essential Listbox Commands

Here are the three most useful commands for managing the list state.

A. Counting Items (LB_GETCOUNT)

How many items are currently in the list?

```
int iCount = SendMessage(hwndList, LB_GETCOUNT, 0, 0);
// Returns the total number (e.g., 5)
```

B. Setting the Selection (LB_SETCURSEL)

You can force the highlight to jump to a specific item.

- **index:** The 0-based position (0 is the top item).
- **-1: Deselect All.** (Removes the highlight completely).

```
// Select the 2nd item (Index 1)
SendMessage(hwndList, LB_SETCURSEL, 1, 0);

// Clear selection (nothing highlighted)
SendMessage(hwndList, LB_SETCURSEL, -1, 0);
```

C. Searching (LB_SELECTSTRING)

This is a powerful "Search" feature built right into Windows. It finds the first item that **starts with** your string.

- **wParam (Start Index):** The index *before* where you want to start searching. Use **-1** to search from the very top.
- **lParam (Search String):** The text to look for.

Example: You have a list of fruits: "Apple", "Banana", "Cherry". You want to jump to "Banana".

```
// "Find the first item starting with 'B', searching from the start (-1)"
int iIndex = SendMessage(hwndList, LB_SELECTSTRING, -1, (LPARAM)TEXT("B"));

// Result:
// Windows finds "Banana", highlights it automatically, and returns its index.
// If not found, it returns LB_ERR.
```

Quick Review

Question 1: Why do we use -1 as the search index for LB_SELECTSTRING? *(Answer: It tells Windows to start searching from the very beginning of the list.)*

Question 2: If you want to clear the highlight so *nothing* is selected, what index do you pass to LB_SETCURSEL? *(Answer: -1)*

Question 3: Does WM_SETREDRAW only work on Listboxes? *(Answer: No! It works on almost any window or control to stop flickering during complex updates.)*

EXTRACTING DATA FROM LISTBOXES

1. Getting the Current Selection (Single Mode)

First, you need to know *which* row the user clicked.

The Message: LB_GETCURSEL

- **Returns:** The zero-based index of the selected item (0, 1, 2...).
- **If Nothing Selected:** It returns LB_ERR (which is -1). **Always check for this!**

```
int iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);

if (iIndex == LB_ERR) {
    // Nothing is selected, don't crash!
}
```

2. Getting the Text (The "Two-Step Dance")

This is a classic "C Programming" hurdle. You cannot just ask for the text immediately because you don't know how big the string is. You might allocate a buffer of 10 characters, but the user selected a string with 50 characters. Crash!

To fix this, we perform a two-step dance: **Measure, then Copy.**

Step A: Measure (LB_GETTEXTLEN)

Ask the listbox: "How long is the string at Index 2?"

```
// Get length of item at iIndex
int iLength = SendMessage(hwndList, LB_GETTEXTLEN, iIndex, 0);
```

Step B: Copy (LB_GETTEXT)

Now that you know the length, allocate a buffer (array) that is big enough (plus one for the null terminator \0), and *then* ask for the text.

```
// Create a buffer big enough (Length + 1 for null terminator)
TCHAR *szBuffer = malloc((iLength + 1) * sizeof(TCHAR));

// Copy the text into your buffer
SendMessage(hwndList, LB_GETTEXT, iIndex, (LPARAM)szBuffer);

// ... Use the string ...

// Clean up memory
free(szBuffer);
```

Note: For simple apps where you know strings are short, you can often just use TCHAR szBuffer[256] and skip the dynamic allocation (malloc), but the method above is the safest.

---

3. Multiple-Selection Listboxes

If you created the listbox with the LBS_MULTIPLESEL style, LB_GETCURSEL stops working reliably (because 5 things might be selected). You need different commands.

A. Selecting/Deselecting (LB_SETSEL)

You can manually toggle items on or off.

- **wParam:** TRUE to select (highlight), FALSE to deselect.
- **lParam:** The index.
- **The "Select All" Trick:** If lParam is **-1**, it applies the action to **every item** in the list.

```
// Select Item 2
SendMessage(hwndList, LB_SETSEL, TRUE, 2);

// Deselect EVERYTHING (Clear all selections)
SendMessage(hwndList, LB_SETSEL, FALSE, -1);
```

B. Checking State (LB_GETSEL)

You have to loop through the items and ask "Are you selected?" one by one.

```
// Check if Item 1 is selected
int iState = SendMessage(hwndList, LB_GETSEL, 1, 0);

if (iState > 0) {
    // Yes, it is highlighted
}
```

ACTION	SINGLE SELECTION	MULTIPLE SELECTION
Get Highlighted Index	`LB_GETCURSEL`	`LB_GETSELITEMS` (Requires a loop and an integer array buffer)
Set Highlight	`LB_SETCURSEL`	`LB_SETSEL` (Can select/deselect specific items)
Get Text	`LB_GETTEXT`	`LB_GETTEXT`
Get Count	`LB_GETCOUNT`	`LB_GETCOUNT`

4. Quick Review

Question 1: Why do you need to add +1 when allocating memory for LB_GETTEXT? *(Answer: To make space for the hidden Null Terminator \0 at the end of the string.)*

Question 2: If LB_GETCURSEL returns -1 (LB_ERR), what does that mean? *(Answer: No item is currently selected.)*

Question 3: In a Multiple Selection listbox, how do you quickly select **all** items? *(Answer: Send LB_SETSEL with wParam = TRUE and lParam = -1.)*

RECEIVING MESSAGES FROM LIST BOXES

When you interact with a list box (like selecting an item), it sends a message to its parent window to let it know what happened. This message contains details about the item you selected and what type of action you did (like clicking on it).

1. Understanding the WM_COMMAND Message

The WM_COMMAND message has two parts: **wParam** and **lParam**. Here's what each part means:

wParam: This is split into two pieces:

- **LOWORD(wParam)**: This tells you the ID of the list box (child window).
- **HIWORD(wParam)**: This shows what kind of action happened (the notification code).

lParam: This contains the handle (or reference) to the list box itself.

2. Notification Codes

List boxes can send different codes to the parent window depending on the action. These codes tell the parent window exactly what happened. The next section would list these codes and what they mean.

NOTIFICATION CODE	MEANING
`LBN_ERRSPACE`	The list box has run out of space (memory allocation failure).
`LBN_SELCHANGE`	The current selection has changed (the user clicked a different item).
`LBN_DBLCLK`	A list box item has been double-clicked with the mouse.
`LBN_SELCANCEL`	The current selection has been canceled (often occurs in multi-select).
`LBN_SETFOCUS`	The list box has received the input focus.
`LBN_KILLFOCUS`	The list box has lost the input focus.

3. Handling Selection Changes

Whenever the selected item in a list box changes, the list box sends a message called **LBN_SELCHANGE**. This happens when:

- You move the highlight through the list.
- You press the **Spacebar** to select or deselect an item.
- You click on an item with the mouse.

To handle these changes, you can add a check in your WM_COMMAND message handler that looks for the **LBN_SELCHANGE** message.

For example, the code below shows how to get the index of the selected item when the **LBN_SELCHANGE** message is received:

```
switch (HIWORD(wParam)) {
  case LBN_SELCHANGE:
    int iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
    // Handle selection change
    break;
}
```

4. Handling Double-Clicks

The **LBN_DBLCLK** notification code is sent when a list box item is **double-clicked** with the mouse.

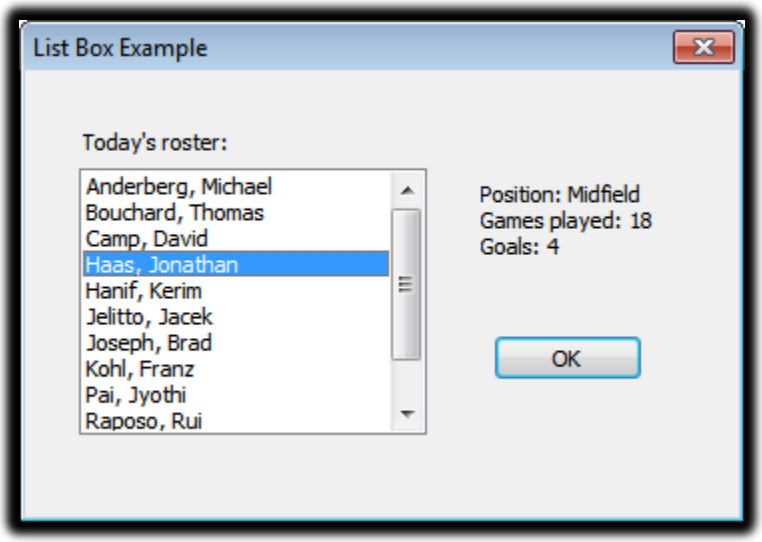
To handle double-clicks, you can add another case in your WM_COMMAND message handler that checks for the **LBN_DBLCLK** message.

For example, the code below shows how to get the index of the item that was double-clicked when the **LBN_DBLCLK** message is received:

```
switch (HIWORD(wParam)) {
  case LBN_SELCHANGE:
    int iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
    // Handle selection change
    break;
  case LBN_DBLCLK:
    int iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
    // Handle double-click
    break;
}
```

5. Receiving and Handling Messages from List Boxes

Handling messages from list boxes is key to making interactive programs. By understanding the different notification codes, you can respond to user actions and create a better user experience.



THE ENVIRON PROGRAM

The **ENVIRON** program is a simple app that shows a list of environment variables and their values. It uses a list box to display the variable names and a text box for the values. The program is written in C and uses the Windows API to create windows and handle user actions.

The Main Function

The main function of the ENVIRON program is **WinMain**. This function:

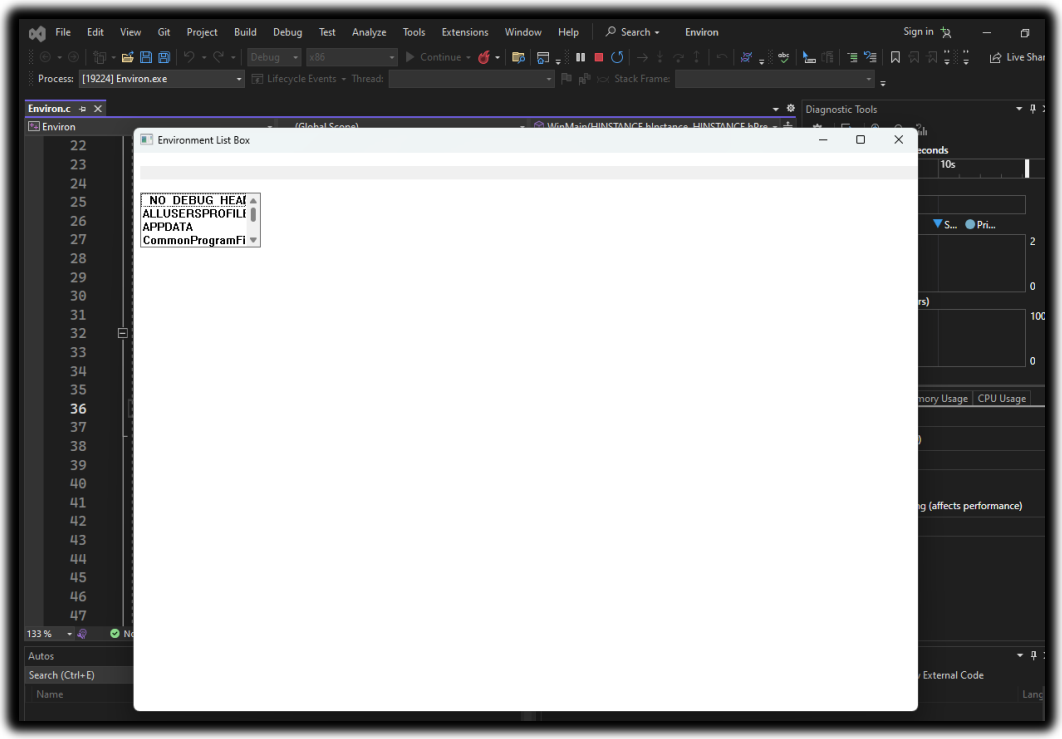
- Initializes the application.
- Creates the main window.
- Runs the main message loop to handle user interactions.

In **WinMain**, the first thing it does is register the window class using the **RegisterClass** function. This defines the window's style and other settings.

Then, **CreateWindow** is used to create the main window. This function takes parameters like the window style, size, and position, and links to the window procedure.

After creating the window, **ShowWindow** makes it visible on the screen and focuses on it.

Finally, **WinMain** enters the message loop with **GetMessage**, which picks up messages from the system and sends them to the right place in the program.



The Logic Flow (The Chain Reaction)

The ENVIRON program is just a glorified browser for your computer's hidden settings (Environment Variables). It has two controls:

1. **Left Side (Listbox):** Shows the *Names* (e.g., "PATH", "USERNAME").
2. **Right Side (Static Text):** Shows the *Value* of the selected name (e.g., "C:\Windows\System32", "User1").

Here is what the WndProc actually does, stripped down to the essentials:

1. Setup (WM_CREATE)

- **Action:** Create the **Listbox** and the **Static Text** control.
- **Fill It:** (In the actual code) It loops through the environment block and fills the listbox with names using LB_ADDSTRING.

2. The Handoff (WM_SETFOCUS)

- **The Problem:** When the app starts, the *Main Window* has the focus. But the Main Window doesn't type; the Listbox does.
- **The Fix:** As soon as the main window gets focus, it immediately passes it to the Listbox.
- **Code:** SetFocus(hwndList);
- **Result:** You can start using arrow keys immediately without clicking first.

3. The Interaction (WM_COMMAND)

This is the core logic. It listens for LBN_SELCHANGE (Listbox Selection Change).

The Chain Reaction:

I) User Clicks: "USERNAME" in the listbox.

II) Notification: Listbox yells LBN_SELCHANGE to the Parent.

III) Step A (Get Selection): Parent asks Listbox, "What text is at the current index?" (LB_GETTEXT). $\rightarrow$ Returns "USERNAME".

IV) Step B (Lookup): Parent asks Windows, "What is the environment value for 'USERNAME'?" (GetEnvironmentVariable). $\rightarrow$ Returns "Dave".

V) Step C (Display): Parent tells the Static control, "Change your text to 'Dave'." (SetWindowText).

4. Cleanup (WM_DESTROY)

- **Action:** PostQuitMessage(0). Standard shutdown.

The Code Logic (Simplified)

Here is how that "wordy" paragraph looks in clean C code:

```c
case WM_COMMAND:
    // Check if the message came from our Listbox (ID_LIST)
    // AND if the user actually changed the selection (LBN_SELCHANGE)
    if (LOWORD(wParam) == ID_LIST && HIWORD(wParam) == LBN_SELCHANGE)
    {
        // 1. Get the Index of the clicked item
        int iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);

        // 2. Get the Text (The Variable Name, e.g., "PATH")
        int iLength = SendMessage(hwndList, LB_GETTEXTLEN, iIndex, 0);
        TCHAR *szName = malloc(iLength + 1);
        SendMessage(hwndList, LB_GETTEXT, iIndex, (LPARAM)szName);

        // 3. Get the Value from Windows
        TCHAR szValue[1024]; // Buffer for the result
        GetEnvironmentVariable(szName, szValue, 1024);

        // 4. Show it in the Static Text box
        SetWindowText(hwndText, szValue);

        free(szName);
    }
    return 0;
```

Explain Like I'm a Teenager: The Jukebox

Think of the ENVIRON program like a digital Jukebox or Karaoke machine.

The Listbox (The Menu): This is the list of Song Titles ("Shape of You", "Bohemian Rhapsody").

The Static Text (The Screen): This is the screen that shows the Lyrics.

The User: You select a song title.

The WndProc (The DJ):

- You press "Bohemian Rhapsody" (LBN_SELCHANGE).
- The DJ looks at the button you pressed (LB_GETTEXT).
- The DJ digs through his hard drive to find the lyrics file for that song (GetEnvironmentVariable).
- The DJ throws the lyrics onto the main screen (SetWindowText).

If you didn't have this logic, clicking the song title would do nothing. The screen would just stay blank.

Quick Review

Question 1: Why do we handle WM_SETFOCUS in this program? *(Answer: To force the keyboard focus onto the Listbox so you can scroll with arrow keys immediately upon startup.)*

Question 2: Does the Listbox store the "Value" (e.g., "C:\Windows")? *(Answer: No. The Listbox only holds the "Name" (e.g., "PATH"). The program has to ask Windows for the value separately using GetEnvironmentVariable.)*

Question 3: What triggers the chain reaction to update the text? *(Answer: The LBN_SELCHANGE notification code inside WM_COMMAND.)*

LISTING FILES WITH LB_DIR

The **LB_DIR** message is a useful tool for filling a list box with a list of files from a directory. You can use it to show files, directories, or even drives.

1. Understanding the iAttr Parameter

The **iAttr** parameter is used to control which files and directories appear in the list. It's a code that defines the file attributes, and the least significant byte of **iAttr** tells you what types of files to include. This can be a combination of the following values:

Value	Attribute
0x0000	Normal file
0x0001	Read-only file
0x0002	Hidden file
0x0004	System file
0x0010	Subdirectory
0x0020	File with archive bit set

The next highest byte of iAttr provides some additional control over the items desired:

Value	Option
0x4000	Include drive letters
0x8000	Exclusive search only

The DDL prefix stands for "dialog directory list."

Using File Attribute Codes

Here are some examples of how to use file attribute codes with the **LB_DIR** message:

To list **normal files**, **read-only files**, and files with the **archive bit** set, use this code:

```
SendMessage(hwndList, LB_DIR, DDL_READWRITE, (LPARAM) szFileSpec);
```

To list **subdirectories** only, use this code:

```
SendMessage(hwndList, LB_DIR, DDL_DIRECTORY, (LPARAM) szFileSpec);
```

To list all valid **drives** and their **subdirectories**, use this code:

```
SendMessage(hwndList, LB_DIR, DDL_DRIVES | DDL_DIRECTORY, (LPARAM) szFileSpec);
```

To list only files that have been modified since the last backup, use the following code:

```
SendMessage(hwndList, LB_DIR, DDL_EXCLUSIVE | DDL_ARCHIVE, (LPARAM) szFileSpec);
```

THE MAGIC COMMAND: LB_DIR

Usually, adding items to a listbox is tedious (one by one). However, Windows has a built-in shortcut to fill a listbox with files from the hard drive instantly.

The Message: LB_DIR Instead of adding strings manually, you tell the Listbox: *"Go look at the hard drive and list everything that matches these rules."*

```
// Syntax: SendMessage(hwndList, LB_DIR, Attributes, (LPARAM)FilterString);
int iStatus = SendMessage(hwndList, LB_DIR, DDL_DIRECTORY | DDL_DRIVES, (LPARAM)TEXT("*.*"));
```

A. The Filter String (lParam)

This works just like the command prompt.

- "*.*": Lists everything.
- "*.c": Lists only C source files.
- "rep*.txt": Lists text files starting with "rep".

B. The Attributes (wParam)

You use flags to tell Windows what *types* of items to include along with normal files.

FLAG	MEANING
`DDL_READWRITE`	Standard normal files. Always include this if you want to display basic files.
`DDL_DIRECTORY`	Include Subdirectories (Folders) in the list.
`DDL_DRIVES`	Include Disk Drives (e.g., A:, C:, D:) in the list.
`DDL_EXCLUSIVE`	Only show the specified attributes; exclude normal files unless `DDL_READWRITE` is also specified.

2. Visual Formatting (How it looks)

When you use LB_DIR, Windows automatically formats special items so the user can tell them apart from files.

1. **Subdirectories:** They are enclosed in brackets.
    - Example: [WINDOWS], [SYSTEM32]
2. **The "Parent" Directory:** A special entry representing "Go Up One Level."
    - Format: [..]
    - *Note:* This does not appear if you are already at the Root (e.g., C:\).
3. **Drives:** They are enclosed in brackets and dashes.
    - Example: [-a-], [-c-]

---

3. Sorting (LBS_SORT)

Correction from your notes: LBS_SORT is a Window Style, not a message.

When you create the listbox with the LBS_SORT style, LB_DIR automatically sorts the results.

Order: Files are listed alphabetically first, followed by subdirectories.

4. Case Study: The HEAD Program

The HEAD program is a simple file viewer. It mimics the Unix command head, which shows the top few lines of a file.

The Goal

1. **Navigate:** Users can double-click folders to change directories.
2. **View:** Users can double-click files to see the first 8KB of text.

The Logic Flow

Scenario A: User Double-Clicks a File (e.g., "notes.txt")

1. **Event:** WM_COMMAND receives LBN_DBLCLK.
2. **Check:** Is the selected item a directory? (Check for brackets []).
3. **Action:** No, it's a file.
4. **Process:**
    - Open the file (CreateFile / _lopen).
    - Read the first **8KB** (8,192 bytes) into a buffer.
    - Paste that text into the Static Text / Edit control on the right (SetWindowText).
    - Close the file.

Scenario B: User Double-Clicks a Directory (e.g., "[WINDOWS]")

1. **Event:** WM_COMMAND receives LBN_DBLCLK.
2. **Check:** Is it a directory? Yes.
3. **Action:** Change the System's "Current Directory."
    - Command: SetCurrentDirectory("WINDOWS").
4. **Refresh:**
    - Clear the listbox: SendMessage(hwndList, LB_RESETCONTENT, 0, 0).
    - Fill it again with the new folder's contents: SendMessage(hwndList, LB_DIR, ... "*.*").

---

5. Explain Like I'm a Teenager: The File Browser

Think of LB_DIR like a **Search Filter** on a shopping site.

LB_DIR: You don't type in every product manually. You just select the filter "Shoes" (*.c) and "Show Out of Stock items" (DDL_DIRECTORY), and the site fills the list automatically.

The Brackets []:

This is how Windows differentiates "Content" from "Containers."

- **File:** homework.docx $\rightarrow$ This is content (a page).
- **Folder:** [School] $\rightarrow$ This is a container (a book). You can't "read" a container; you have to open it to see what's inside.

The "Head" Program:

It's a "Preview" tool.

Imagine a bookshop.

- You walk to a shelf ([Fiction]).
- You pick up a book (HarryPotter.txt).
- You read the first chapter (The first 8KB).
- You put it back. You don't read the whole book (it's faster to just read the beginning to see if you like it).

---

6. Quick Review

Question 1:

If you want your listbox to show C:, D:, and E: drives, which flag must you use in LB_DIR?

(Answer: DDL_DRIVES)

Question 2:

What does the entry [..] do in the HEAD program?

(Answer: It represents the Parent Directory. Double-clicking it moves you "up" one folder (e.g., from C:\Windows back to C:\).)

Question 3:

Why does the HEAD program only read 8KB?

(Answer: For speed. Reading a 500MB video file just to show a text preview would crash or freeze the program. 8KB is enough to identify what the file contains.)

---

7. The HEAD Program

The **HEAD** program is a tool that shows the beginning of a file. It has a list box to display a directory and a text box to show the file contents.

Main Function (WinMain)

- **WinMain** is the entry point of the program. It initializes the app, creates the main window, and starts the message loop.
- **RegisterClass** defines the window's style and procedure.
- **CreateWindow** creates the main window with several parameters like window class, title, size, and other settings.
- **ShowWindow** displays the window on the screen.
- **GetMessage** handles messages sent to the program (like clicks or key presses).

Window Procedure (WndProc)

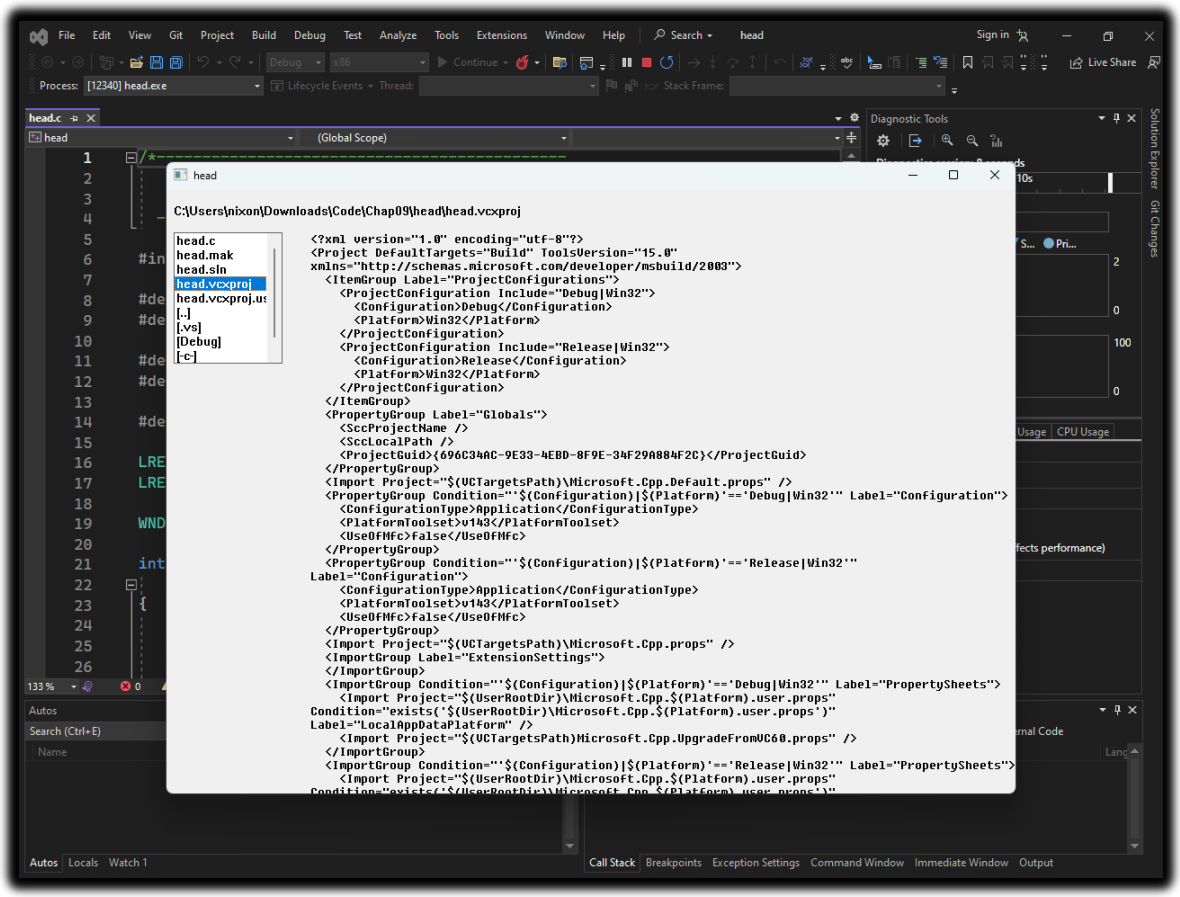
WndProc is the main function that handles all messages for the window.

- **WM_CREATE**: When the window is created, WndProc sets up the list box and text box.
- **SendMessage**: Fills the list box with a directory list.
- **WM_SIZE**: Adjusts the child windows (list box, text box) if the main window is resized.
- **WM_SETFOCUS**: Focuses on the list box when the window is active.
- **WM_COMMAND**: Checks if a list item is double-clicked and opens the file’s contents.
- **WM_PAINT**: Displays the file's content in the text box when the window needs to be repainted.
- **WM_DESTROY**: Posts a **WM_QUIT** message when the window closes, ending the program.

List Box Procedure (ListProc)

ListProc handles messages for the list box.

WM_KEYDOWN: If the **Enter** key is pressed while the list box is focused, it sends a **WM_COMMAND** message to open the file.



8. Handling Double-Clicks

In the **ENVIRON** program, users click on an environment variable to see its value. But for the **HEAD** program, where files are opened by selecting items from a list, this approach wouldn't work because constantly opening and closing files would make the program slow.

Instead, in **HEAD**, users must **double-click** a file or subdirectory to open it. However, list boxes don’t have a built-in way to handle double-clicks with the keyboard, so the **HEAD** program uses **window subclassing** to solve this.

Window Subclassing

Window subclassing is a technique where you modify the behavior of a window by creating a subclass. In **HEAD**, the list box is subclassed using a procedure called **ListProc**.

- **ListProc** checks if the user presses the **Enter** key (VK_RETURN). If they do, it simulates a double-click by sending a **WM_COMMAND** message to the main window.

Processing WM_COMMAND Message

The **WndProc** function handles the **WM_COMMAND** message. When the user selects a file:

- It uses **CreateFile** to check if the selection is a file.
- If it's not a file (likely a subdirectory), it uses **SetCurrentDirectory** to change the directory.
- If the user selects an invalid drive, the program ignores the selection.

Processing WM_PAINT Message

When the window needs to repaint, **WndProc** handles the **WM_PAINT** message. It opens the selected file using **CreateFile**, then reads and displays the contents.

Unicode Considerations

The program assumes all files are ASCII text and uses the **DrawTextA** function to display the content. However, this doesn't work for **Unicode** files, which would appear as garbled text.

To handle **Unicode** properly, the program should check the file's **Byte Order Mark (BOM)** to see if it's Unicode. If it is, it should use **DrawTextW** instead of **DrawTextA**.

In this simplified version of the **HEAD** program, it just uses **DrawTextA** for everything, even if it results in garbled characters for Unicode files.

Conclusion

The **HEAD** program shows how to handle double-clicks in a list box using window subclassing and how to open files. It also discusses the challenge of working with Unicode text and the simpler approach the program uses to handle it.

End of Chapter 9...