MENUS AND OTHER RESOURCES

Icons

Icons are small images that represent things like programs, files, or folders. You can see icons in places like the title bar of windows, the Start menu, the taskbar, and on the desktop. Icons can be colorful or black and white, and they come in different sizes.

Cursors

Cursors are the images that show up as your mouse pointer. They change shape based on what you're doing, like when you hover over a link, select text, or resize a window. Cursors can also be colorful or black and white, and they come in various sizes.

Character Strings

Character strings are text used by programs, like for menus, dialog boxes, or error messages. These text strings can be stored inside the program's **.EXE** file or in a separate resource file.

Custom Resources

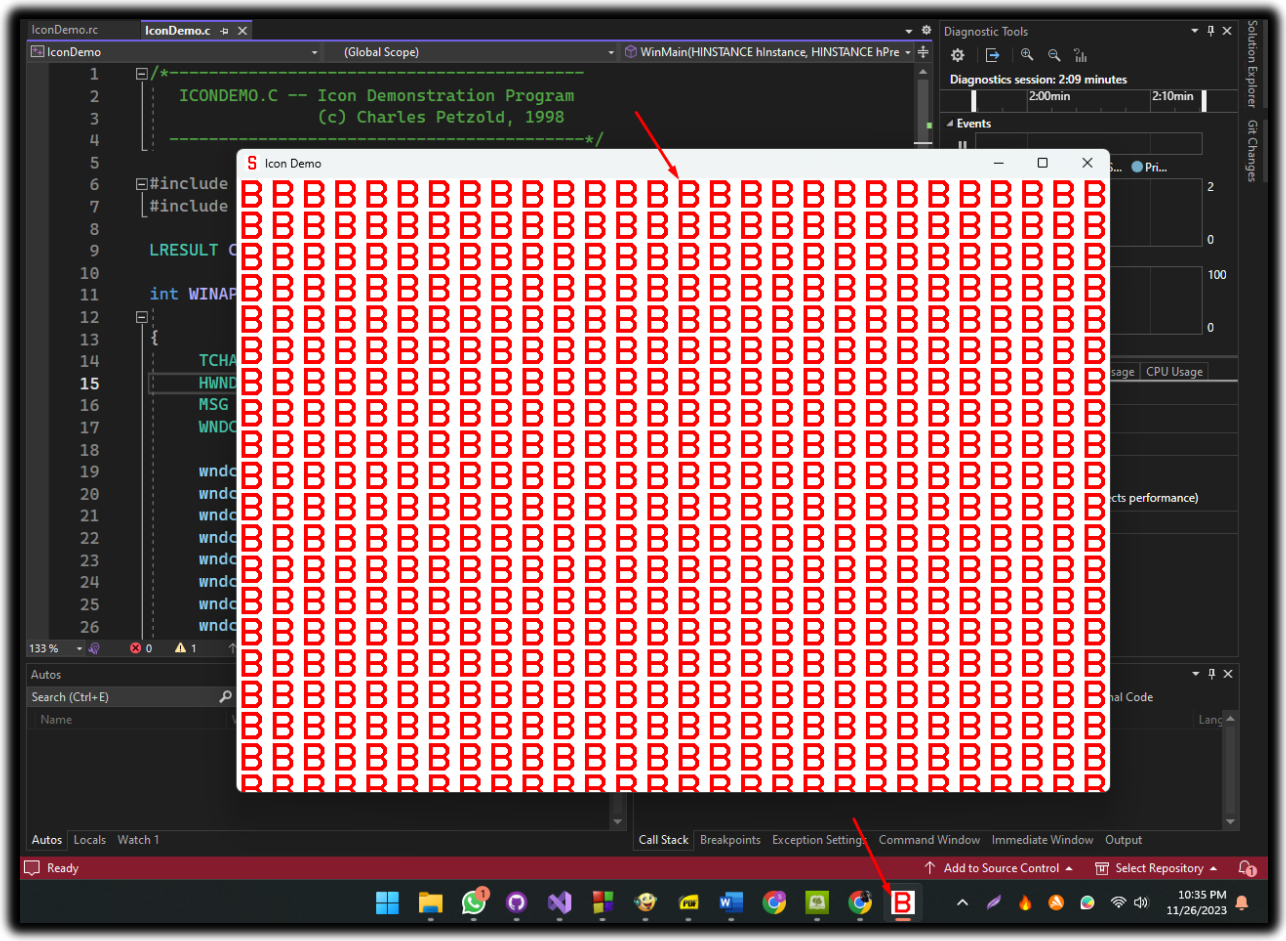
Custom resources are any special files or data used by a program that don't fit into the categories of icons, cursors, text, or menus. For example, these can include things like images, sounds, or videos. Custom resources can be stored inside the **.EXE** file or in a separate file.

Menus

Menus are lists of options that users can select to perform actions in a program. They can appear as pull-down menus, right-click context menus, or toolbars. Menus can be customized to include the options that are most useful for the program.

Keyboard Accelerators

Keyboard accelerators are shortcuts you can use to quickly perform actions in a program, like pressing **Ctrl+S** to save a file. These shortcuts can usually be customized to fit your preferences.



In C programming, **resources** are a way to bundle parts of a program (like icons, cursors, and strings) into the **.EXE** file. This makes it easier to manage and distribute the program since you don't need separate files for each component.

Icons as Resources

A good example of this is **icons**. Normally, icons would be stored in separate files, but by using resources, the icon can be stored inside the program's **.EXE** file. This makes the development process smoother, and ensures the icon becomes a part of the final executable.

Adding an Icon to a Program

To add an icon to your program, you can use the **Image Editor** in **Visual C++ Developer Studio**. This tool lets you create and save an icon in an **.ICO** file. At the same time, Developer Studio creates a **resource script** (with a .RC file extension) that lists all the resources, and a **header file** (RESOURCE.H) to let the program access these resources.

Project Setup: ICONDEMO

Now, let's look at how to set this up in a project called **ICONDEMO**:

After creating the project, Developer Studio generates several files, such as:

- **ICONDEMO.DSW** (project workspace)
- **ICONDEMO.DSP** (project settings)
- **ICONDEMO.MAK** (makefile)

The project also creates a C source code file called **ICONDEMO.C**, where the program's logic will be written.

Example Program Structure

Here's a simplified version of the program structure:

```c
// ICONDEMO.C

#include <windows.h>

// Function declarations
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// Entry point
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
    // ... window and instance setup ...

    // Message loop
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

// Window procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    // ... message handling ...

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

The Source Code in the ICONDEMO Folder

ICONDEMO.C is a Windows program that shows how to use icons in a graphical user interface (GUI) application. It creates a window and fills it with copies of an icon from the program's resources.

- **Windows Header File**: #include <windows.h> includes the Windows header file, which provides important definitions for working with the Windows API.
- **Resource File Inclusion**: #include "resource.h" includes the resource file, where the program's resources (like icons and cursors) are stored.

Key Parts of the Program

- **Window Procedure Function**:
  The function LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) is responsible for handling messages sent to the window by the operating system.
- **Program Entry Point**:
  The function int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) is the entry point of the program. It runs when the program starts.
- **Variable Declarations**:
  Variables like szAppName, hwnd, msg, and wndclass are declared to store essential information, such as the program's name, the window handle, and the current message.
- **Window Class Configuration**:
  The wndclass structure is set up with settings that define how the window looks and behaves.
- **Window Registration**:
  RegisterClass(&wndclass) registers the window class with Windows, allowing the program to create windows using that class.
- **Window Creation**:
  hwnd = CreateWindow(...) creates a window based on the registered class, specifying the window's title, size, and other attributes.
- **Window Display**:
  ShowWindow(hwnd, iCmdShow) makes the window visible to the user.
- **Window Update**:
  UpdateWindow(hwnd) refreshes the window, ensuring it's correctly displayed on the screen.

- **Message Loop**:
  The while (GetMessage(&msg, NULL, 0, 0)) (...) loop keeps retrieving messages from the message queue and sends them to the window procedure function for processing.
- **Message Translation**:
  TranslateMessage(&msg) translates the message into a format the window procedure can understand.
- **Message Dispatching**:
  DispatchMessage(&msg) sends the translated message to the window procedure function to be handled.

Creating a Resource Script

To add an icon to the **ICONDEMO** program, you first need to create a **resource script**. This file defines the program's resources (like icons, cursors, and menus).

Here's how to create it:

1. In **Developer Studio**, go to **File > New**.
2. In the **New** dialog, choose **Resource Script** and click **OK**.
3. In the **File Name** field, type ICONDEMO.RC and click **OK**.
4. This will create two files:
   - ✓ ICONDEMO.RC (the resource script)
   - ✓ RESOURCE.H (a header file that connects the C code with the resources)

Adding an Icon Resource

To add an icon to your resource script:

1. Open the ICONDEMO.RC file in **Developer Studio**.
2. Go to **Insert > Resource**.
3. In the **Resource** dialog, select **Icon** and click **New**.
4. A blank **32x32** icon will appear. Use the painting tools to create your icon.

Saving the Icon Resource

After creating your icon:

1. In the **icon properties** dialog, change the **ID** to IDI_ICON.
2. Set the **Filename** to ICONDEMO.ICO.
3. Click **OK**.
4. The icon is saved as ICONDEMO.ICO in the project folder.

Compiling the Program

To compile the program with the new icon:

1. In **Developer Studio**, go to **Build > Build ICONDEMO**.
2. The program will compile and link with the icon resource.

Running the Program

Once compiled, run the program:

1. Go to **Debug > Start Debugging** in **Developer Studio**.
2. The program will launch, and the icon will be shown in the window.

Additional Tips for Creating Icons

- Use **distinctive colors** to make your icon stand out.
- Keep it **simple** with easy-to-recognize shapes.
- Avoid **too much detail**, which can make the icon look cluttered and hard to see.

Creating Resource Files

Resource files are text files that define a program's resources, like icons, cursors, and menus. These files are compiled into binary files using the **resource compiler** (RC.EXE). Then, the binary files are linked with the program's code to create the final executable.

1. Loading Icons

To load an icon from a resource file, you use the LoadIcon function. It takes two arguments:

- **hInstance**: The instance handle of the program.
- **MAKEINTRESOURCE(IDI_ICON)**: The resource identifier for the icon. The MAKEINTRESOURCE macro converts an integer ID into a resource identifier that LoadIcon can use.

2. Drawing Icons

The DrawIcon function draws an icon on the screen. It takes four arguments:

- **hdc**: The device context (where the icon will be drawn).
- **x, y**: The x and y coordinates for the upper-left corner of the icon.
- **hIcon**: The handle to the icon.

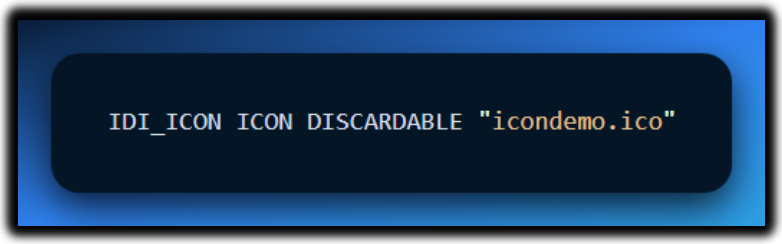
3. Small Icons

Windows will automatically use smaller icons in certain places, like the **title bar** or **taskbar**. The small icon size is typically **16x16** pixels, which you can get using GetSystemMetrics with the SM_CXSMSIZE and SM_CYSMSIZE indices.

To create a small icon, you can select **Small (16x16)** in the icon editor, then design a version of your icon for that size.

4. Understanding Resource Script ICON Statements

In the ICONDEMO.RC file, the line:



This is an **ICON statement** in the resource script, and it defines the icon resource with these properties:

- **Identifier**: IDI_ICON (a unique number that identifies the icon in the project).
- **Type**: ICON (the resource type).
- **Filename**: "icondemo.ico" (the actual icon file).
- **Attribute**: DISCARDABLE (this allows the resource to be discarded from memory when no longer needed).

5. Resource Identifiers

The **identifier** IDI_ICON is a unique number that refers to the icon resource. In this case, the identifier is **101**. These identifiers are used by functions like LoadIcon to retrieve specific resources from the compiled resource file.

6. Resource Types

The **ICON** type tells the resource compiler that the resource is an icon. Resource types help organize different kinds of resources, like icons, cursors, and menus, during the compilation process.

7. Resource Filenames

The filename icondemo.ico specifies where the icon file is located. This can be either a **relative** or **absolute** path, meaning it could be a simple file name or a full path to the file.

8. Resource Attributes

The **DISCARDABLE** attribute means that the icon can be removed from memory by Windows if needed to free up space. This is the default setting, so you don't always need to specify it.

9. Obtaining a Handle to an Icon

To use an icon in your program, you need to get a **handle** to it. This can be done with the LoadIcon function. It takes two arguments:

- **hInstance**: The instance handle of the program.
- **MAKEINTRESOURCE(IDI_ICON)**: The identifier for the icon resource.

The MAKEINTRESOURCE macro converts the integer **IDI_ICON** to a format that can be used by LoadIcon.

Here's an example of how to load the icon defined in the ICONDEMO.RC file:

```
hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON));
```

11. Using the Icon Handle

Once you have a **handle** to an icon, you can draw it on the screen using the DrawIcon function. This function takes four arguments:

- **hdc**: The device context (where the icon will be drawn).
- **x, y**: The x and y coordinates for the upper-left corner of the icon.
- **hIcon**: The handle of the icon to draw.

Here's an example of how to draw the icon at coordinates (100, 100):

```
DrawIcon(hdc, 100, 100, hIcon);
```

12. Getting an Icon Handle

To get a handle to an icon, follow these steps:

1. Define the icon resource in the **resource script**.
2. Compile the resource script into a binary file.
3. Link the binary resource file into the program's **.EXE**.

Once the icon is linked, use the LoadIcon function to obtain a handle to it. Then, you can use this handle to draw the icon on the screen.

13. Loading Icons Using LoadIcon

The LoadIcon function is used to load an icon either from a resource or from a file. It takes two arguments:

- **hInstance**: The program's instance handle.
- **resourceIdentifier**: The identifier for the icon.

The identifier can be a number, a string, or a string with a # symbol in front of it.

14. Loading Icons by Numeric Identifier

To load an icon using a **numeric identifier**, you can use the MAKEINTRESOURCE macro. This macro converts an integer identifier into a format that the LoadIcon function can use.

Here's an example of how to load an icon by numeric identifier:

```
hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(125));
```

In this example, IDI_ICON is the numeric identifier for the icon resource.

15. Loading Icons by Character String

To load an icon by character string, you can simply pass the string to the LoadIcon function. The string can be the name of the icon or the name of the resource file.

Here is an example of how to load an icon by character string:

```
hIcon = LoadIcon(hInstance, "icondemo.ico");
```

16. Loading Icons by String Prefixed with # Character

To load an icon by a string that is prefixed with the # character, you can pass the string to the LoadIcon function. The string should represent a number in **ASCII form**.

Here's an example of how to load an icon using a string prefixed with #:

```
hIcon = LoadIcon(hInstance, TEXT("#125"));
```

17. Using LoadIcon in ICONDEMO

In the **ICONDEMO** program, LoadIcon is called **twice**:

1. **Once when defining the window class**.
2. **Once in the window procedure** to get the handle to the icon for drawing.

In both cases, **ICONDEMO** uses the MAKEINTRESOURCE macro to convert the numeric identifier **IDI_ICON** into a format that can be used with the LoadIcon function.

Here's an example of how **ICONDEMO** calls the LoadIcon function in the window procedure:

```
hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON));
```

18. The LoadIcon Function: Versatile and Flexible

The **LoadIcon** function is very flexible. It can load icons by:

- **Numeric identifier**
- **Character string**
- **String prefixed with the # character**

In **ICONDEMO**, LoadIcon is used to load an icon from a resource file.

Using Icons in Windows Programs: A Deep Dive

1. Setting Icons with WNDCLASS and RegisterClass

When you define a **window class** with the **WNDCLASS** structure and register it with **RegisterClass**, you usually set an icon for the window. This is done through the **hIcon** field in the WNDCLASS structure.

Windows automatically selects the right image size from the icon file when needed (for example, using different sizes for the window title bar or taskbar).

2. RegisterClassEx and WNDCLASSEX

There is an updated version called **RegisterClassEx** that uses the **WNDCLASSEX** structure. This structure adds two extra fields:

- **cbSize**: Specifies the size of the WNDCLASSEX structure.
- **hIconSm**: Holds the small icon handle (for 16x16 icons).

However, using **WNDCLASSEX** may not be necessary, as **Windows** can automatically extract properly sized icons from a single icon file.

3. Dynamic Icon Changes with SetClassLong

If you want to change the program's icon while it's running, you can use the **SetClassLong** function.

For example, if you have a second icon file linked to the identifier **IDI_ALTICON**, you can switch to that icon with the following code:

```
SetClassLong(hwnd, GCL_HICON, LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ALTICON)));
```

This code changes the window's icon to **IDI_ALTICON** dynamically.

4. Alternative: Using GetClassLong with DrawIcon

If you don't want to keep the icon handle in memory but still need to display it, you can get the icon handle at runtime using **GetClassLong**. Then, you can use **DrawIcon** to draw it on the screen.

```
DrawIcon(hdc, x, y, GetClassLong(hwnd, GCL_HICON));
```

5. LoadIcon vs LoadImage

Although some sections of the Windows documentation suggest that **LoadIcon** is "obsolete" and recommend using **LoadImage**, **LoadImage** offers more flexibility.

It is documented in the **Platform SDK** under **User Interface Services/Resources**. However, **LoadIcon** is still simpler and often more than enough for most applications.

In the **ICONDEMO** example, **LoadIcon** is called twice for the same icon without any issues or extra memory usage.

6. Handle Management

LoadIcon is one of the few functions that automatically manages the icon handle, meaning you don't have to worry about manually destroying it. Even though there is a **DestroyIcon** function, it's mainly used with functions like **CreateIcon**, **CreateIconIndirect**, and **CreateIconFromResource**, which allow you to create icons dynamically in your program.

7. Conclusion

Icons are essential in Windows programming. Understanding how to use and manage icons properly is important for creating user-friendly and visually appealing applications.

Customizing Mouse Cursors in Windows Programming

Customizing **mouse cursors** is similar to customizing icons. It can make your Windows applications more interactive and visually appealing. While the default cursors in Windows are fine for most programs, custom cursors can give your app a unique touch.

1. Creating Custom Cursors

Creating a custom cursor is easy and can be done directly in **Developer Studio**. Here's how:

1. Go to the **Insert** menu and select **Resource**.
2. Choose **Cursor** to add a custom cursor.
3. Define the **hotspot** — the point on the cursor where the user's interaction will occur (e.g., where the click will be registered).

2. Setting the Custom Cursor

Once you've created your custom cursor, you can set it for your window class by using the following statement within your class definition:

```
wndclass.hCursor = LoadCursor (hInstance, MAKEINTRESOURCE (IDC_CURSOR)) ;
```

For cursors defined with a text name, use the following statement:

```
wndclass.hCursor = LoadCursor (hInstance, szCursor) ;
```

This will display the customized cursor associated with IDC_CURSOR or szCursor whenever the mouse hovers over a window created based on this class.

For child windows, you can **set different cursors** depending on the child window below the cursor.

If your program defines the window class for these child windows, assign different cursors to each class by setting the hCursor field accordingly.

For predefined child window controls, modify the hCursor field using the following statement:

```
SetClassLong (hwndChild, GCL_HCURSOR,
LoadCursor (hInstance, TEXT ("childcursor")) ;
```

3. Using SetCursor for Custom Cursors

To change the mouse cursor for specific areas within your client area without using child windows, call the SetCursor function:

```
SetCursor (hCursor) ;
```

You can **invoke SetCursor** during **WM_MOUSEMOVE** message processing to change the cursor when the mouse moves over your window. If you don't do this, Windows will just use the cursor defined in the window class.

Efficiency of SetCursor

SetCursor is efficient, especially if the cursor doesn’t change too often or require complex updates.

Here's an example of how to use SetCursor during **WM_MOUSEMOVE**:

```
case WM_MOUSEMOVE:
    SetCursor(LoadCursor(hInstance, MAKEINTRESOURCE(IDR_CUSTOM_CURSOR)));
    break;
```

This way, you can change the cursor based on the mouse movement over your window.

USING CHARACTER STRING RESOURCES IN WINDOWS PROGRAMMING

In Windows programming, you usually define text strings directly in your code. But using **character string resources** is helpful, especially when translating your program into different languages.

This is important for things like **menus** and **dialog boxes**, which are part of the resource script. Instead of putting all your text directly in the code, you can store it in the resource script. This makes translation easier because you only need to change the strings in the resource file, not the code.

Why Use String Resources?

By using string resources, all the text your program uses is stored in one place—the resource script. This helps when you want to translate your program into other languages. You can just translate the strings and relink the program, instead of changing the source code directly.

Creating a String Table

To create a **string table**:

1. Select **"Resource"** from the **Insert** menu.
2. Choose **"String Table"**.
3. The strings will appear on the right side, where you can define identifiers and the text for each string.

In the resource script, the strings are organized in a **multiline statement**, like this:

```
STRINGTABLE DISCARDABLE
BEGIN
    IDS_STRING1, "character string 1"
    IDS_STRING2, "character string 2"
    [other string definitions]
END
```

Manually Creating a String Table

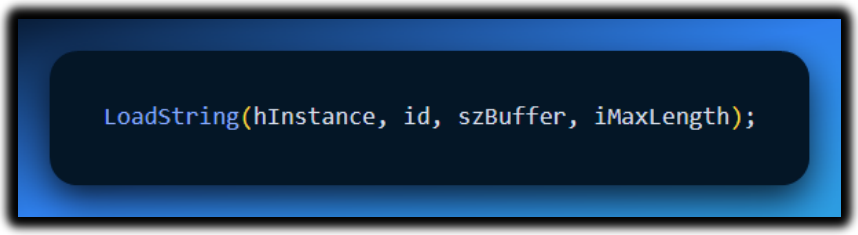
In the past, if you were manually creating a string table in a text editor, you could use **curly brackets** instead of **BEGIN** and **END** statements.

A **resource script** can have multiple string tables, but remember:

- Each **ID** must be unique.
- Each **string** can only be **one line long**, with a maximum of **4097 characters**.
- Control characters like **\t** (tab) and **\n** (newline) can be used in strings, and functions like **DrawText** and **MessageBox** recognize them.

Using LoadString to Retrieve Strings

To use the string resources in your program, you can call the **LoadString** function:



- **id**: The ID number of the string from the resource script.
- **szBuffer**: A character array to store the loaded string.
- **iMaxLength**: The maximum number of characters to copy.

The function returns the number of characters in the string.

Common Practices

- **ID Numbers**: String IDs are usually defined as **macro identifiers** in a header file, often starting with **IDS_**.
- **Formatting Strings**: If you need to embed additional information in the string (like a number or variable), you can use **C formatting characters** with **wsprintf**.

Unicode and String Handling

All the text in your resource files, including the string table, is stored in **Unicode** format in both the compiled **.RES** file and the final **.EXE** file.

- **LoadStringW** loads Unicode text directly.
- **LoadStringA** (available since **Windows 98**) converts from Unicode to the local **code page**.

Example: Displaying an Error Message

Let's look at an example of a function that uses three character strings to show error messages in a **message box**. The **RESOURCE.H** header file contains the string IDs, and the resource script defines the corresponding strings. The C source code uses this header and displays the message.

```c
#define IDS_FILENOTFOUND 1
#define IDS_FILETOOBIG 2
#define IDS_FILEREADONLY 3

void OkMessage(HWND hwnd, int iErrorNumber, const TCHAR *szFileName) {
    TCHAR szFormat[40];
    TCHAR szBuffer[60];

    LoadString(hInst, iErrorNumber, szFormat, sizeof(szFormat) / sizeof(szFormat[0]));
    wsprintf(szBuffer, szFormat, szFileName);

    MessageBox(hwnd, szBuffer, szAppName, MB_OK | MB_ICONEXCLAMATION);
}
```

To display a message box containing the "file not found" message, the program calls:

```c
OkMessage(hwnd, IDS_FILENOTFOUND, szFileName);
```

This structure demonstrates how character string resources simplify localization and improve code maintainability in a Windows program.

CUSTOM RESOURCES IN WINDOWS

Custom resources (or user-defined resources) let you store and access various types of data directly within your application, instead of relying on external files.

This is useful for sensitive or frequently accessed data since custom resources are embedded directly in the executable file.

Creating Custom Resources

You create custom resources using a **resource script file** (with a **.RC** extension). This file defines:

- **Resource type**
- **Resource name**
- **Data associated with the resource**

For example, the script below defines a custom resource named **IDR_BINTYPE1** of type **BINTYPE** and links it to the file **BINDATA.BIN**:

```
IDR_BINTYPE1 BINTYPE BINDATA.BIN
```

Loading and Accessing Custom Resources

To load and access a custom resource, use the **LoadResource** and **LockResource** functions:

LoadResource loads the resource into memory:

```
LoadResource(hInstance, lpName, lpType);
```

- ✓ **hInstance**: Application instance handle
- ✓ **lpName**: Resource name or ID
- ✓ **lpType**: Resource type

LockResource locks the resource in memory and gives you a pointer to the data so you can access it.

Freeing Custom Resources

Once you're done with the custom resource, free it from memory with **FreeResource** to prevent memory leaks and ensure proper resource management.

Sample Code

The following code demonstrates how to load and access a custom resource named IDR_BINTYPE1 and display its contents to the console:

```
#include <windows.h>

int main() {
    HMODULE hInstance = GetModuleHandle(NULL);
    HRSRC hResource = FindResource(hInstance, MAKEINTRESOURCE(IDR_BINTYPE1), TEXT("BINTYPE"));

    if (hResource) {
        HGLOBAL hGlobal = LoadResource(hInstance, hResource);
        if (hGlobal) {
            LPVOID pData = LockResource(hGlobal);
            if (pData) {
                DWORD size = SizeofResource(hInstance, hResource);
                char* data = (char*)pData;

                for (DWORD i = 0; i < size; i++) {
                    putchar(data[i]);
                }
            }
            FreeResource(hGlobal);
        }
    }

    return 0;
}
```

This code will load the custom resource IDR_BINTYPE1, lock it into memory, and print its contents to the console.

Benefits of Using Custom Resources

Custom resources have several advantages over using external files to store data in Windows applications:

1. **Embedded Data**: They're stored directly in the executable file, making it easier to manage sensitive or frequently used data.
2. **Memory Management**: Windows automatically handles loading and unloading custom resources, so you don't need to manage file I/O manually.
3. **Portability**: Since custom resources are part of the executable, your application becomes portable and doesn't depend on external files.
4. **Security**: Custom resources are protected by the same permissions as the executable, improving data security.

PROGRAM CODE "POEPOEM" EXPLAINED

The **POEPOEM** program is a Windows application that shows how to use custom resources like an icon and text. Let's go over the code step by step.

Resource Loading and Initialization

WinMain is the program's entry point. It starts by initializing key variables, such as:

- **szAppName**: The application name, loaded from a string resource.
- **szErrMsg**: An error message, also loaded from a string resource.

The program uses **LoadStringA** to load these values from the resource file. Here's how it works:

```
LoadStringA(hInstance, IDS_APPNAME, (char *) szAppName, sizeof(szAppName));
LoadStringA(hInstance, IDS_ERRMSG, (char *) szErrMsg, sizeof(szErrMsg));
```

LoadStringA: Loads the string resource into the **szAppName** and **szErrMsg** variables.

- **hInstance**: The program's instance handle.
- **IDS_APPNAME** and **IDS_ERRMSG**: Resource identifiers for the application name and error message.
- **szAppName** and **szErrMsg**: The variables where the strings will be stored.

Window Creation and Message Loop

After loading the resources, the program registers the window class. If the registration fails, it shows an error message (which was loaded earlier).

Then, the program enters the message loop to handle events like user input.

Creating the Main Window:

The program creates the main window using **CreateWindow**. It also sets the window class name, icon, and cursor during the window class registration.

```
hwnd = CreateWindow(szAppName, szCaption, WS_OVERLAPPEDWINDOW, ...);
```

Message Loop:

Inside the message loop, the program processes messages like user inputs and other events. This is done using **GetMessage**, **TranslateMessage**, and **DispatchMessage**.

```
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Window Procedure (WndProc)

The **window procedure** handles various messages, such as:

- **WM_CREATE**: The window is being created.
- **WM_SIZE**: The window is resized.
- **WM_SETFOCUS**: The window gains focus.
- **WM_VSCROLL**: Vertical scrolling.
- **WM_PAINT**: The window needs to be redrawn.
- **WM_DESTROY**: The window is being destroyed.

When **WM_CREATE** is received, the program sets up a **vertical scrollbar** (using BS_VSCROLL) and loads the text for **"Annabel Lee"** from a custom resource.

```
hResource = LoadResource(hInst, FindResource(hInst, TEXT("AnnabelLee"), ...));
pText = (char *)LockResource(hResource);
```

- **LoadResource**: Loads the resource into memory.
- **LockResource**: Retrieves the pointer to the actual resource data (the text).

Scrollbar and Text Display

Scrollbar Setup:

The program calculates how many lines are in the text resource (like **"Annabel Lee"**) to set the correct range for the scrollbar.

On **WM_SIZE**, the scrollbar is positioned within the window. On **WM_VSCROLL**, the program handles scrolling when the user interacts with the scrollbar.

```
case WM_SIZE:
    MoveWindow(hScroll, ...);  // Move the scrollbar based on the window size
    SetFocus(hwnd);  // Set focus back to the window
    return 0;
case WM_VSCROLL:
    // Handle vertical scroll, adjust the text display based on the scroll position
    return 0;
```

Text Rendering and Cleanup

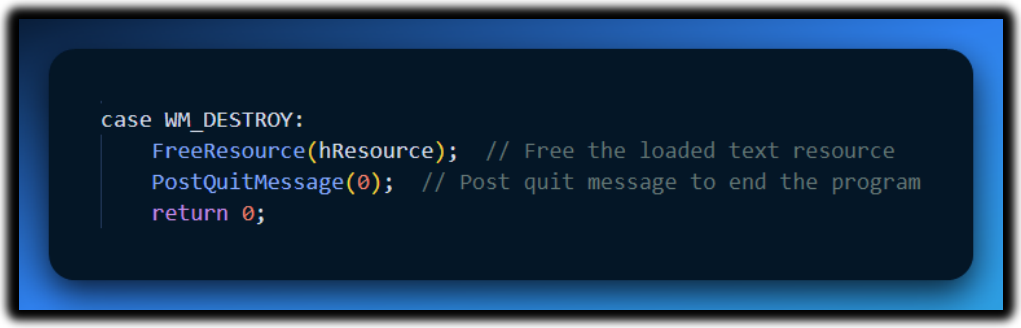
Text Rendering:

On **WM_PAINT**, the program uses **DrawTextA** to display the text based on the current position of the scrollbar.

```
case WM_PAINT:
    // Render the text
    return 0;
```

Resource Cleanup:

On **WM_DESTROY**, when the window is about to be destroyed, the program frees up any resources (like the text) that were loaded earlier using **FreeResource**. It also sends a **PostQuitMessage** to quit the program.



```
case WM_DESTROY:
    FreeResource(hResource);  // Free the loaded text resource
    PostQuitMessage(0);  // Post quit message to end the program
    return 0;
```

The code shows:

- How to initialize and create a window with custom resources (in this case, text).
- It handles **scrolling** the text when the user moves the scrollbar.
- It properly manages **resources**, like loading and freeing them when necessary.
- The message loop is structured to handle user interactions with the scrollbar and ensure smooth text display and window updates.

POEPOEM PROGRAM RESOURCE SECTION IN-DEPTH

This section focuses on how **resources** such as text, icons, and string tables are handled in the **POEPOEM** program. The use of resources helps make the program modular, easier to manage, and more adaptable to different languages. Here's a breakdown of the key points:

Resource Types and Definitions

Resource Script File (POEPOEM.RC):

The **POEPOEM.RC** file defines different types of resources used by the program:

- ✓ **Text Resource**: **ANNABELLEE** – Contains an excerpt from the poem *Annabel Lee* by Edgar Allan Poe.
- ✓ **Icon Resource**: **POEPOEM** – Refers to the icon associated with the application (poepoem.ico).
- ✓ **String Table**: **STRINGTABLE** – Includes various string resources like the app name, caption, and error messages.

Each resource is given a **unique identifier** and **type**, allowing them to be loaded and accessed throughout the program.

String Table Usage

String Table in Resource Script:

The **STRINGTABLE** section in the resource script defines several string resources identified by unique names (e.g., **IDS_APPNAME**, **IDS_CAPTION**, **IDS_ERRMSG**).

These identifiers are referenced in the **RESOURCE.H** header file, which contains the **macro definitions** for each string, making it easier to manage them throughout the code.

Example of the resource script (POEPOEM.RC):

```
STRINGTABLE
BEGIN
    IDS_APPNAME,    "POEPOEM Application"
    IDS_CAPTION,    "Edgar Allan Poe's Poem"
    IDS_ERRMSG,     "Error loading resource"
END
```

Loading String Resources

Loading Strings into the Program:

During the program initialization, strings are loaded from the resource file using the **LoadString** function.

Example code for loading strings:

```
LoadString(hInstance, IDS_APPNAME, szAppName, sizeof(szAppName) / sizeof(TCHAR));
LoadString(hInstance, IDS_CAPTION, szCaption, sizeof(szCaption) / sizeof(TCHAR));
```

This loads the string resources into **szAppName** and **szCaption** variables, which are used later in the program.

Handling Unicode and ANSI Strings

Unicode vs ANSI Support:

The program handles both **Unicode** and **ANSI** strings for compatibility across different Windows versions.

- ✓ **Unicode**: Loaded using **LoadStringW** (wide-character version).
- ✓ **ANSI**: Loaded using **LoadStringA** (narrow-character version).

On **Windows 98**, Unicode is not supported, so the program needs to handle **MessageBoxA** (ANSI version) instead of the Unicode **MessageBoxW**.

This compatibility allows the program to run on both newer and older versions of Windows.

Child Window Scroll Bar Control

Using a Scroll Bar Control:

The application uses a **child window scroll bar control** instead of a standard window scroll bar. This offers an automatic keyboard interface, simplifying input handling.

It eliminates the need for specific **WM_KEYDOWN** events, making the program more user-friendly when interacting with the scroll bar.

Error Handling and Message Display

Error Handling Using String Resources:

When errors occur, appropriate error messages are displayed to the user. These messages are loaded from the **string resources**.

For example, if there's an issue during **class registration** or loading resources, the program will display an error message using the **IDS_ERRMSG** string.

Facilitating Translation

Making the Program Accessible for Translation:

By defining **all text** as resources (like strings and text files), the program is more easily **translated** into different languages.

Translators can simply modify the **string table** in the resource script and relink the program, instead of modifying the source code directly.

This process simplifies localization and ensures that all user-facing text is organized in one place.

Excerpt from "Annabel Lee"

Including Text from "Annabel Lee":

The **Annabel Lee** poem is stored as a **text resource** in **POEPOEM.TXT**. This text is part of the program's content and is loaded at runtime.

The text from the poem is displayed in the program window, and using a resource approach allows this content to be easily replaced or updated without changing the source code.

The **POEPOEM program** utilizes resources for the following reasons:

- ✓ **Modularity**: The program keeps resources like icons, text, and strings separate from the main code, making them easier to manage.
- ✓ **Localization**: With all text defined as resources, the program can be easily translated into other languages by modifying the resource files without changing the code.
- ✓ **Error Handling**: String resources are used for error messages, ensuring that all text in the program is consistent and can be easily modified.
- ✓ **Efficiency**: By using child window scroll bars and simplifying error handling with string resources, the program enhances both functionality and user experience.

This approach makes the **POEPOEM** program scalable and maintainable, while also providing an easy path for translation and updates.

MENUS IN WINDOWS

What is a Menu?

A **Menu** is a list of commands hidden behind a title. It saves screen space by organizing hundreds of options into neat little drawers.

- **The User Sees:** Text (e.g., "Open", "Save").
- **The Program Sees:** A numeric **ID** (e.g., IDM_OPEN, IDM_SAVE).
- **The Message:** When clicked, it sends WM_COMMAND—exactly the same as a Button!

---

1. The Three Types of Menus

1. **Main Menu (Top-Level):** The horizontal bar at the top of the window (File, Edit, Help). It is always visible.
2. **Popup Menu (Submenu/Context):** The list that drops down when you click a Main Menu item, or pops up when you Right-Click.
3. **System Menu:** The little icon in the top-left corner of the title bar. It handles "Move," "Size," and "Close." You rarely touch this in code; Windows handles it.

---

2. Creating Menus (The .RC Script)

Unlike buttons, you usually don't create Menus with C code (CreateWindow). Instead, you define them in a **Resource Script** (.rc file). This is a separate text file that acts as a blueprint.

The Anatomy of an Item: Every item in your menu blueprint needs three things:

1. **Name:** What the user reads ("Exit").
2. **ID:** The unique number the program listens for (IDM_EXIT).
3. **Options:** Special flags (Checked, Grayed, etc.).

Formatting Tricks (The Secret Codes):

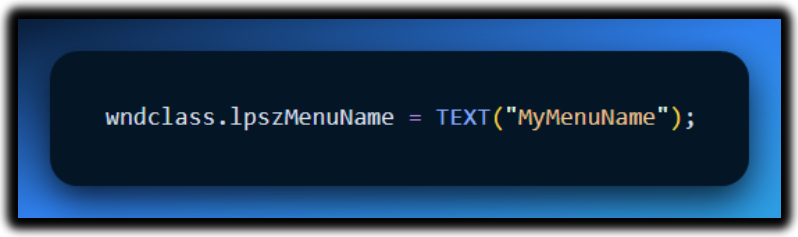
- **The Ampersand (&):** If you write &File, Windows displays it as **<u>F</u>ile**. This creates an automatic keyboard shortcut (Alt + F).
- **The Tab (\t):** If you write Open\tCtrl+O, Windows puts "Open" on the left and "Ctrl+O" on the far right. It looks professional.

---

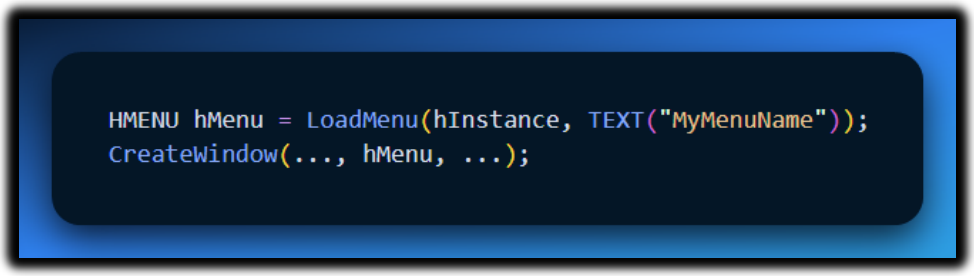
3. Attaching the Menu

Once you have your script, how do you get it onto the window? You have three options:

Method A: The "Class" Way (Most Common) Define it once in your WNDCLASS. Every window of this type will have this menu.



Method B: The "Creator" Way Pass the menu handle when creating a specific window.



Method C: The "Dynamic" Way Change the menu while the program is running (e.g., switching from "User Mode" to "Admin Mode").

```
SetMenu(hwnd, hNewMenu);
```

---

4. Menu States (Attributes)

You can change how menu items behave using **Flags**:

ATTRIBUTE	VISUAL	MEANING
Active	`Normal Text`	The item works normally. The user can interact with it.
Grayed	*Grey Text*	The item is disabled (cannot be clicked). Use this when an option isn't available (e.g., "Paste" when the clipboard is empty).
Checked	`Checkmark (✓)`	A toggle option is **ON** (e.g., "Show Grid").
Separator	———	A horizontal line used to visually group related commands.

---

5. Explain Like I'm a Teenager: The Restaurant

Think of your program as a **Fancy Restaurant**.

- **The Resource Script (.RC):** This is the **Menu Design** printed on paper. It lists "Steak," "Soup," and "Soda," and gives each a number (#1, #2, #3).
- **Grayed Item:** The kitchen ran out of steak. You don't throw the menu away; you just stick a "Sold Out" sticker on it so nobody orders it.
- **The ID:** When you order, you don't scream "I want the Pan-Seared Ribeye with Garlic." You tell the waiter (Windows), "I'll have the **#4**."
- **WM_COMMAND:** The waiter walks to the Chef (The Main Window) and says, "Customer ordered **#4**." The Chef doesn't care what it's called; he just looks up instruction #4 and cooks it.

---

6. Quick Review

Question 1: If you want to create a keyboard shortcut where pressing "Alt+H" opens the Help menu, how do you write the text string? *(Answer: &Help. The & underlines the next character.)*

Question 2: What message does the window receive when a user clicks "File -> Exit"? *(Answer: WM_COMMAND. The specific ID (like IDM_EXIT) will be in the Low Word of wParam.)*

Question 3: If you define a menu in a Resource Script, do you need to write CreateWindow for every single item (like "Open", "Save", "Exit")? *(Answer: No! Windows builds the entire visual structure for you automatically based on the script.)*

---

7. Explanation of Code

Here's an example of defining a menu in a resource script:

```
MENU
BEGIN
    POPUP "&File"           // Defines the File menu with the "&" sign for mnemonic key
    BEGIN
        MENUITEM "&Open", IDM_OPEN     // Open option, with "O" as mnemonic key (Alt + O)
        MENUITEM "&Save", IDM_SAVE     // Save option, with "S" as mnemonic key (Alt + S)
        MENUITEM "Save &As", IDM_SAVEAS    // Save As option, with the ampersand used to display a single & in the menu item
        SEPARATOR                  // Adds a separator line in the menu
        MENUITEM "E&xit", IDM_EXIT   // Exit option, with "X" as mnemonic key (Alt + F, X)
    END
END
```

Pop-up Menu Definition:

POPUP "&File": Defines a pop-up menu titled **"File"**. The **"&"** before **"File"** makes the **F** underlined, which is the **accelerator key**. To activate the **File** menu, you press **Alt + F**.

Menu Items:

MENUITEM "&Open", IDM_OPEN: Defines the **"Open"** option in the **File** menu. The **"&"** before **O** makes the **O** underlined, which is the **mnemonic key**. You can activate this option with **Alt + O**.

MENUITEM "&Save", IDM_SAVE: Defines the **"Save"** option. The **S** is underlined, and it can be activated by pressing **Alt + S**.

MENUITEM "Save &As", IDM_SAVEAS: The **"Save As"** option includes a **double ampersand (&&)** to display a single ampersand **&** in the menu item.

SEPARATOR: Adds a horizontal line in the menu to visually separate the items (useful for grouping related commands).

Exit Option:

MENUITEM "E&xit", IDM_EXIT: Defines the **"Exit"** menu item. The **X** in **Exit** is underlined, and it can be activated using **Alt + F, X** (since it's under the **File** menu, you first press **Alt + F**, then **X**).

Key Points to Note:

Accelerator Keys (Mnemonics):

- The **ampersand (&)** is used to define **mnemonic keys**, which are shortcut keys activated by pressing **Alt** + the underlined letter.
- For example, **Alt + O** opens the **"Open"** option, and **Alt + S** activates **"Save"**.

Double Ampersand (&&):

- In the menu text, **double ampersands (&&)** are used to **display a single ampersand**. This is necessary when you want to include an actual ampersand character in the text, as **&** is a special character in resource scripts.

Separator:

- **SEPARATOR** is used to visually separate groups of menu items, helping organize them in a clear and user-friendly way.

Menu Commands and IDs:

- Each menu item is linked to a **command ID** (e.g., **IDM_OPEN**, **IDM_SAVE**, etc.). These IDs are defined in the program's code and are used to trigger specific actions when the user selects a menu item.

Conclusion:

This example illustrates how to define a **File menu** with options like **Open**, **Save**, **Save As**, and **Exit** in a Windows application. It shows how to use **accelerator keys**, **mnemonics**, and **separators** to make the menu intuitive and easy to use.

Below is a simple program to display a menu:

```
#include <windows.h>

// Define the menu items
MENU
BEGIN
  POPUP "File"
  BEGIN
    MENUITEM "Open", IDM_OPEN
    MENUITEM "Save", IDM_SAVE
    MENUITEM "Save As", IDM_SAVEAS
    SEPARATOR
    MENUITEM "Exit", IDM_EXIT
  END
END

// Create the window class
WNDCLASSEX wc;
wc.cbSize        = sizeof(WNDCLASSEX);
wc.style         = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc   = WndProc;
wc.cbClsExtra    = 0;
wc.cbWndExtra    = 0;
wc.hInstance      = GetModuleHandle(NULL);
wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wc.lpszMenuName  = TEXT("MyAppMenu");
wc.lpszClassName = TEXT("MyAppClass");
wc.hIconSm       = LoadIcon(NULL, IDI_SMALLICON);

// Register the window class
if (!RegisterClassEx(&wc)) {
  MessageBox(NULL, TEXT("Window Registration Failed!"), TEXT("Error!"), MB_ICONEXCLAMATION | MB_OK);
  return 1;
}

// Create the window
HWND hwnd = CreateWindow(
  TEXT("MyAppClass"),
  TEXT("My Application"),
  WS_OVERLAPPEDWINDOW,
  CW_USEDEFAULT, CW_USEDEFAULT,
  640, 480,
  NULL,
  NULL,
  GetModuleHandle(NULL),
  NULL
);

// Check if the window was created successfully
if (!hwnd) {
  MessageBox(NULL, TEXT("Window Creation Failed!"), TEXT("Error!"), MB_ICONEXCLAMATION | MB_OK);
  return 1;
}

// Display the window
ShowWindow(hwnd, SW_SHOW);

// Run the message loop
MSG msg;
while (GetMessage(&msg, NULL, 0, 0)) {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}

// Return the result code
return msg.wParam;
```

This code defines a menu, registers a window class, creates a window, and runs a message loop. The menu is created using the **MENU** macro, which lists menu items defined by the **MENUITEM** macro.

Each item has text and an ID number. The window class is registered with **RegisterClassEx**, using a **WNDCLASSEX** structure.

The window is created with **CreateWindow**, specifying details like class name, title, size, and menu.

The message loop uses **GetMessage** to retrieve messages and **DispatchMessage** to send them to the window procedure for processing.

Key components include defining menu items, registering the window class, creating the window, and handling messages through the message loop.

MENU MESSAGES

The Conversation: You vs. Windows

When a user interacts with a menu, Windows keeps up a constant stream of commentary. You (the WndProc) usually ignore most of it, but sometimes you need to listen.

1. "I'm about to show the bar" (WM_INITMENU)

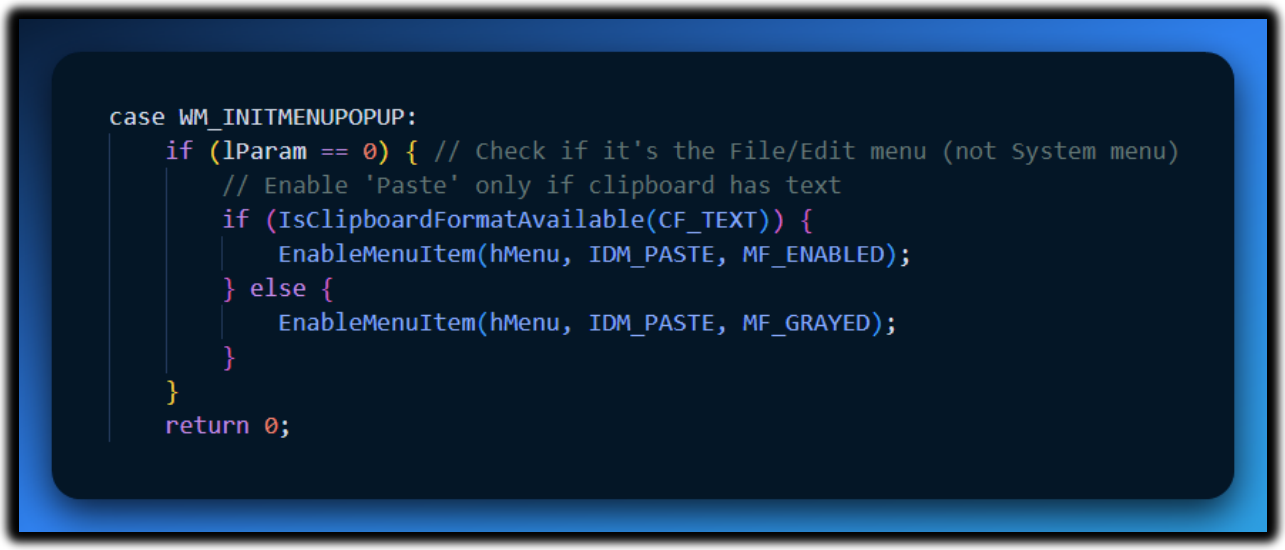
- **When:** The user clicks the top menu bar, but *before* the list drops down.
- **Purpose:** It gives you a split second to modify the top-level bar.
- **Verdict:** Rarely used. Changing the main bar right as someone clicks it feels glitchy.

2. "I'm showing a list" (WM_INITMENUPOPUP) (Important!)

When: The user hovers over "Edit," and the dropdown is about to appear.

Purpose: This is the best time to gray out items.

- *Example:* If the user clicks "Edit", you check: "Is there text in the Clipboard?" If No, you disable the "Paste" button instantly.
- *Why here?* It ensures the menu always looks up-to-date the moment the user sees it.



```
case WM_INITMENUPOPUP:
    if (lParam == 0) { // Check if it's the File/Edit menu (not System menu)
        // Enable 'Paste' only if clipboard has text
        if (IsClipboardFormatAvailable(CF_TEXT)) {
            EnableMenuItem(hMenu, IDM_PASTE, MF_ENABLED);
        } else {
            EnableMenuItem(hMenu, IDM_PASTE, MF_GRAYED);
        }
    }
    return 0;
```

3. "The user is hovering..." (WM_MENUSELECT)

When: The user is moving their mouse up and down the list, highlighting different items.

Purpose: Used to show "Help Text" at the bottom of the screen (in a Status Bar).

- *User hovers 'New':* Status Bar says "Create a new document."
- *User hovers 'Save':* Status Bar says "Save current file."

4. "THEY CLICKED IT!" (WM_COMMAND) (The Big One)

When: The user actually clicks an enabled item.

Purpose: Do the work.

Parameters:

- LOWORD(wParam): The **ID** of the item (e.g., IDM_SAVE).
- HIWORD(wParam): **0** (This tells you it came from a Menu, not a Button).
- lParam: **0** (This confirms it is a Menu).

5. "They clicked the System Menu" (WM_SYSCOMMAND)

- **When:** The user clicks "Close" or "Maximize" in the title bar icon.
- **Purpose:** Usually, you pass this to DefWindowProc so Windows handles the window resizing/closing.
- **Danger:** If you trap this message and don't pass it on, your window will become "frozen" (unmovable and unclosable).

6. "They pressed a weird key" (WM_MENUCHAR)

- **When:** The menu is open, and the user presses a letter that *doesn't* match any item.
- **Purpose:** Very rare. Used for custom keyboard shortcuts inside complex menus.

Comparison Table: The Critical Messages

MESSAGE	CONTEXT	YOU USE IT TO...
`WM_INITMENUPOPUP`	Just BEFORE menu opens	**Enable/Disable (Gray)** items like "Paste" or "Save" based on current app state.
`WM_MENUSELECT`	WHILE hovering	**Update a Status Bar** description to explain what the highlighted menu item does.
`WM_COMMAND`	AFTER clicking	**Execute the actual function** (Open file, Exit, etc.) associated with the Menu ID.

1. Explain Like I'm a Teenager: The Waiter

Imagine you are at a restaurant (The Program).

WM_INITMENUPOPUP (The Daily Special): You (the User) pick up the menu.

- ✓ *The Waiter (Windows):* "Wait! Before you open it... let me cross out the 'Soup of the Day' because we just ran out."
- ✓ *Result:* You open the menu, and 'Soup' is already grayed out.

WM_MENUSELECT (The Finger Point): You run your finger down the list: "Steak... Pasta... Salad..."

- ✓ *The Waiter:* "That steak is spicy... That pasta is vegan... That salad contains nuts."
- ✓ *Result:* He gives you details as you browse.

WM_COMMAND (The Order): You tap the menu: "I want the Pasta."

- ✓ *The Waiter:* "Coming right up!" (He runs to the kitchen to execute command IDM_PASTA).

2. Quick Review

Question 1: If you want to disable the "Save" button because the file hasn't changed, which message is the best place to do it? *(Answer: WM_INITMENUPOPUP.)*

Question 2: In WM_COMMAND, how does the program know if the message came from a Menu or a Button? *(Answer: It checks lParam. If lParam is 0, it's a Menu. If it's a Handle, it's a Control/Button.)*

Question 3: What happens if you forget to pass WM_SYSCOMMAND to DefWindowProc? *(Answer: The standard window controls (Minimize, Maximize, Move, Close) stop working.)*

3. Code Example:

Here are some code examples demonstrating the use of menu-related messages:

Disabling a Menu Item Based on Clipboard Content.

Handling Menu-Related Messages.

```
// --- Handling WM_INITMENUPOPUP for Dynamic Menu Updates ---
case WM_INITMENUPOPUP: // Handle WM_INITMENUPOPUP message

if (HIWORD(lParam) == 1) { // Check if it's a popup menu

    if (LOWORD(lParam) == IDR_POPUP_EDIT) { // Check if it's the Edit popup menu

        // Check if the clipboard contains text using IsClipboardFormatAvailable() function
        if (IsClipboardFormatAvailable(CF_TEXT)) {

            // Enable the Paste menu item using EnableMenuItem() function
            EnableMenuItem(GetMenu(hwnd), IDM_EDIT_PASTE, MF_ENABLED);

        } else { // If no text is available in the clipboard

            // Disable the Paste menu item using EnableMenuItem() function
            // Set both MF_DISABLED and MF_GRAYED flags to disable and gray out the menu item
            EnableMenuItem(GetMenu(hwnd), IDM_EDIT_PASTE, MF_DISABLED | MF_GRAYED);

        }
    }
}
break;
```

Part 1: The "Just-in-Time" Logic (WM_INITMENUPOPUP)

Static menus are boring. Good menus react to the world. The classic example: **The Paste Button.** You cannot "Paste" if there is nothing to paste. But you shouldn't disable the button permanently—what if the user copies something 5 seconds later?

The Solution: Check the clipboard *at the exact moment* the user clicks the "Edit" menu, right before the list appears.

The Logic Flow

Trigger: User clicks "Edit".

Message: Windows sends WM_INITMENUPOPUP.

Check 1: Is this the "Edit" menu? (We don't want to run this logic for the "File" menu).

Check 2: Is there text in the Clipboard? (IsClipboardFormatAvailable)

Action:

- ✓ **Yes:** Enable the Paste button.
- ✓ **No:** Gray out (Disable) the Paste button.

```
case WM_INITMENUPOPUP:
    // lParam LOWORD contains the index of the menu (0=File, 1=Edit, etc.)
    // Let's assume the 'Edit' menu is at index 1.
    if (lParam == 1)
    {
        // Ask Windows: "Is there text in the clipboard?"
        // CF_TEXT = Standard Text Format
        if (IsClipboardFormatAvailable(CF_TEXT))
        {
            // YES: Enable the button
            EnableMenuItem((HMENU)wParam, IDM_PASTE, MF_ENABLED);
        }
        else
        {
            // NO: Gray it out
            EnableMenuItem((HMENU)wParam, IDM_PASTE, MF_GRAYED);
        }
    }
    return 0;
```

Part 2: The MENUDEMO Program

This is a practice application designed to teach you **Menu State Management**. It demonstrates the three most common menu tricks: **Placeholders**, **Checkmarks**, and **Graying**.

I. The Structure

MENUDEMO.RC (The Blueprint): Defines the menu tree (File, Edit, Background, Timer, Help).

MENUDEMO.C (The Brains): The WndProc that reacts to the clicks.

II. The Features & Tricks

A. The Placeholders (File & Edit)

- **Behavior:** When clicked, they just pop up a message: "Not Implemented."
- **Lesson:** How to wire up a menu ID to a simple MessageBox.

B. The "Radio" Style (Background Menu)

Behavior: Lets you pick a background color (White, Light Gray, Gray, Black, etc.).

The Trick: Mutual Exclusion.

When you pick "White", the program must:

- Paint the background White.
- **Uncheck** the previously selected item.
- **Check** the new "White" item.

Code Insight: You typically store the currentSelectionID in a variable so you know which one to uncheck next time.

C. The "Toggle" Style (Timer Menu)

Behavior: Contains "Start" and "Stop".

The Logic: You cannot start a timer that is already running.

Initially: "Start" is Enabled, "Stop" is Grayed.

User Clicks Start:

- SetTimer() starts the beep.
- **Gray out** "Start" (prevent double-clicking).
- **Enable** "Stop".

User Clicks Stop:

- KillTimer() stops the beep.
- **Enable** "Start".
- **Gray out** "Stop".

Explain Like I'm a Teenager: Smart Menus

I. The "Edit" Menu is like a Fridge Light.

- It doesn't stay on all day. It only turns on *the exact second* you open the door.
- WM_INITMENUPOPUP is the sensor that says, "Door opening! Check status!"
- It checks: "Do we have food?" (Clipboard text).
- If empty, it turns off the light (Grays out Paste).

The MENUDEMO Program is a Car Dashboard.

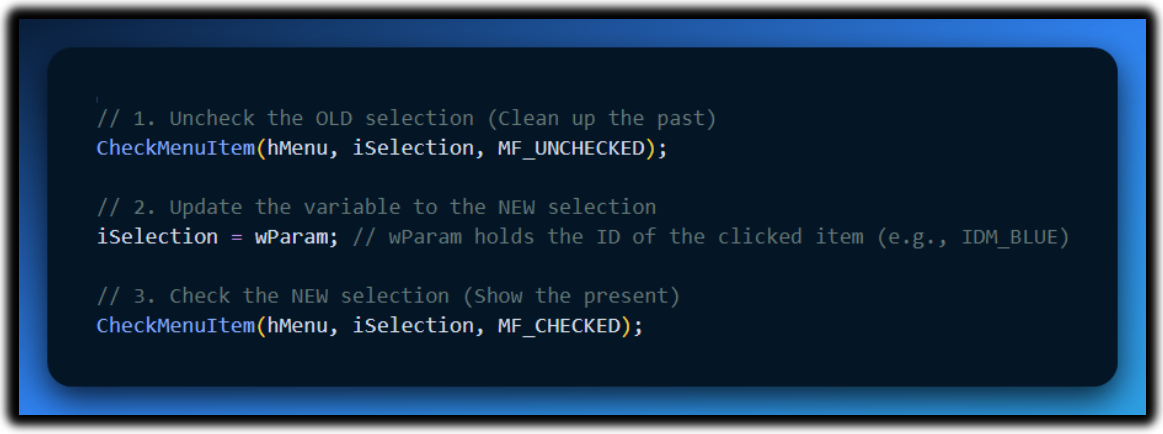
Background Menu (Gear Shift): You can be in Park **OR** Drive. You can't be in both. When you shift to Drive (Check), the stick moves out of Park (Uncheck).

II. Timer Menu (Engine Start Button):

- ✓ When the car is off, the "Start Engine" button lights up.
- ✓ Once the engine is running, the "Start Engine" button turns off (Grayed), and the "Stop Engine" button lights up. You can't start a car that's already running.

III. The "Radio Button" Logic (Background Menu)

This code handles the **Mutual Exclusion** we talked about. You can only have one background color active at a time. If you pick "Blue," "Red" must turn off.



```
// 1. Uncheck the OLD selection (Clean up the past)
CheckMenuItem(hMenu, iSelection, MF_UNCHECKED);

// 2. Update the variable to the NEW selection
iSelection = wParam; // wParam holds the ID of the clicked item (e.g., IDM_BLUE)

// 3. Check the NEW selection (Show the present)
CheckMenuItem(hMenu, iSelection, MF_CHECKED);
```

Key Variable: iSelection This is a static or global variable. It remembers "What was picked *last time*?" If you didn't have this variable, you wouldn't know which item to uncheck, and eventually, every color in your menu would have a checkmark next to it.

IV. The "Toggle" Logic (Timer Menu)

This code prevents the user from doing impossible things, like starting a timer that is already running.

a. When the User clicks "Start":

You must disable the "Start" button (so they can't double-click it) and enable the "Stop" button.

```
// Gray out "Start" (It's already running!)
EnableMenuItem(hMenu, IDM_TIMER_START, MF_GRAYED);

// Light up "Stop" (Now you are allowed to stop it)
EnableMenuItem(hMenu, IDM_TIMER_STOP, MF_ENABLED);
```

b. When the User clicks "Stop":

You flip the switch back.

```
// Light up "Start" (Ready to go again)
EnableMenuItem(hMenu, IDM_TIMER_START, MF_ENABLED);

// Gray out "Stop" (Nothing to stop anymore)
EnableMenuItem(hMenu, IDM_TIMER_STOP, MF_GRAYED);
```

V. Explain Like I'm a Teenager: The Logic

The Background Logic (iSelection) is like a Dating Status.

The Rule: You can only date one person at a time.

The Action: You want to date "Person B" (wParam).

The Code:

- Break up with "Person A" (CheckMenuItem ... MF_UNCHECKED).
- Update your status to "Dating Person B" (iSelection = wParam).
- Post about "Person B" (CheckMenuItem ... MF_CHECKED).
- *If you skip step 1, things get messy.*

The Timer Logic (EnableMenuItem) is like a Video Game Boss.

Boss is Alive (Timer Running):

- Can you "Summon Boss"? No. (Grayed out).
- Can you "Attack Boss"? Yes. (Enabled).

Boss is Dead (Timer Stopped):

- Can you "Summon Boss"? Yes. (Enabled).
- Can you "Attack Boss"? No. (Grayed out - there is nothing to attack).

RESOURCE FILES

The **MENUDEMO.RC** file defines the program's resources, especially the menu structure. It specifies the menu items, their associated IDs, and the initial checked state for the Background menu items.

This file contains the resource definitions for the **MENUDEMO** program, specifically detailing the menu structure. It defines various menus, menu items, and their associated IDs.

I. Key Elements in MENUDEMO.RC:

MENUDEMO MENU DISCARDABLE:

This line declares the start of the menu definition. The **DISCARDABLE** keyword indicates that the resources are not essential for the program's function but are used to provide a user interface.

BEGIN:

Marks the start of the menu definition. It encloses the menu items and popups.

POPUP "&File":

Defines a popup menu with the label **"&File"**. The **&** before "F" signifies that "F" is the hotkey for the menu, enabling users to press **Alt + F** to open the "File" menu.

BEGIN (for &File popup):

Begins the definition of the **"&File"** popup menu, which contains its individual menu items.

MENUITEM Definitions:

Each menu item is defined using **MENUITEM**. For example:

MENUITEM "&New", IDM_FILE_NEW: Defines the "New" item with its associated identifier **IDM_FILE_NEW**.

Other menu items like **"&Open"**, **"&Save"**, **"Save &As"**, and **"E&xit"** are similarly defined.

SEPARATOR: Inserts a separator line between menu items for visual distinction.

POPUP "&Edit", "&Background", "&Timer", and "&Help":

Similar to the "&File" popup, the other popups are defined:

"&Background" defines background color options, where items like **"&White"**, **"&Light Gray"**, etc., are listed. The **CHECKED** keyword indicates that the "White" option is checked by default.

"&Timer" defines timer-related actions, including the **"&Start"** and **"S&top"** options, with the "Stop" option initially disabled using the **GRAYED** keyword.

"&Help" defines a help section, typically containing help-related options.

END:

Each popup menu and the overall menu structure end with an **END** block.

RESOURCE.H:

This file contains the resource header, which defines constant identifiers for the menu items and other resources used in the program.

#define IDM_FILE_NEW 40001:

Defines a constant identifier named **IDM_FILE_NEW** and assigns it the value **40001**. This constant is used to uniquely identify the **"&New"** menu item in the program.

Other Identifiers:

Similar constants are defined for all other menu items, allowing the program to refer to them by their IDs rather than hardcoded strings.

Menu Design Considerations

File and Edit Menu Consistency:

The **File** and **Edit** menus should follow established conventions, such as using **Alt + F** for the **File** menu and **Alt + E** for the **Edit** menu. This ensures users are familiar with how to navigate the program.

Unique Menus for Specific Programs:

Menus outside of **File** and **Edit** can vary depending on the program's functionality. However, it's a good practice to maintain consistency with common design patterns to reduce confusion.

Flexibility in Menu Design:

The menu design is mainly stored in the resource script. This means you can revise and update the menu easily by modifying the resource script without requiring significant changes to the program code.

This structure of the **MENUDEMO.RC** file ensures that the program's menu is both well-organized and easily modifiable, improving maintainability and user experience.

MENUITEM Statements on Top Level

While it's possible to use **MENUITEM** statements at the top level of a menu, it's usually discouraged. This is because top-level items can be accidentally selected, which could confuse users.

If you must use **MENUITEM** at the top level, you can add an exclamation point (!) after the menu item's text. This helps make it clear that the item is not a popup menu.

Defining a Menu Using CreateMenu and AppendMenu

Instead of defining a menu in the resource file, you can also create a menu directly in your program using the **CreateMenu** and **AppendMenu** functions.

- **CreateMenu** creates an empty menu.
- **AppendMenu** adds items to the menu.

This approach gives you more flexibility because you can create and customize the menu programmatically. However, it requires you to manually handle the structure of the menu, unlike defining the menu in a resource file.

```
hMenu = CreateMenu(); // Creates a new menu and returns its handle
AppendMenu(hMenu, MF_POPUP, hSubMenu, "&File"); // Creates the File popup and adds it to the main menu
AppendMenu(hMenu, MF_POPUP, hSubMenu2, "&Edit"); // Creates the Edit popup and adds it to the main menu
// ... Add more menu items and popups
// Set the window's menu
SetMenu(hwnd, hMenu);
```

2. Menu Creation Using CreateMenu and AppendMenu

The **CreateMenu** and **AppendMenu** functions allow you to manually create a menu, but this method is more complex and less preferred than using resource scripts. Here's a breakdown of what happens:

- **CreateMenu** initializes the menu structure.
- **AppendMenu** adds items to the menu. Each item has an identifier, label, and flags (e.g., **MF_POPUP** for popups, **MF_STRING** for regular items, **MF_CHECKED** for checked items, and **MF_GRAYED** for disabled items).
- **SetMenu** assigns the created menu to the window.

Alternative Methods for Menu Creation

- **Array of MENUITEMTEMPLATE Structures**: You can use an array to reduce code repetition and better organize the menu items.
- **LoadMenuIndirect**: This function loads a menu from memory and provides more flexibility than using resource scripts.

Comparison of Methods

- **Resource Scripts**: The easiest and most common approach for menu creation, offering simplicity and visual structure.
- **CreateMenu and AppendMenu**: Manual creation gives more control but is more complex and prone to errors.
- **LoadMenuIndirect**: A middle ground between resource scripts and manual creation.

Recommendation

Use **resource scripts** for menu creation as they are simpler, visual, and integrated with the development environment. Manual methods like **CreateMenu** and **AppendMenu** are only necessary for advanced cases.

Additional Notes

- For testing, you can use the **MenuCustomCode.c** from the chapter 10 folder. Make sure to check the libraries (**libwinmm** and **libgdi32**) are configured correctly.
- If you're working with **Unicode**, ensure your project settings reflect this, and check the code for any necessary modifications (like setting **hIconSm** only for Unicode).

The video…

MenuCustomCode.mp4

You can then continue to popmenu program in windows full source code folder chapter 10.

Explanation of the POPMENU Program

The **POPMENU** program demonstrates how to create a **popup menu** in a Windows application without a top-level menu bar. Here's a simplified breakdown of the key parts of the program:

- **Window Class and Window Creation**: The program defines and registers a window class and then creates the main application window.
- **WndProc Function**: This function handles messages, including creating the popup menu and responding to user actions.
- **WM_CREATE**: When the window is created, the program loads the menu resource and extracts the submenu that will be used as the popup menu.
- **WM_RBUTTONUP**: When the user right-clicks, the program retrieves the mouse coordinates and shows the popup menu at that location using TrackPopupMenu.
- **WM_COMMAND**: Handles the selection of menu items. For example, if a background color is selected, the window's background brush is updated, and the window is redrawn.
- **File Operations**: The program supports basic file operations (e.g., New, Open, Save) and includes a simple "About" dialog.
- **Help**: The "Help" option displays a message saying that help functionality isn't implemented.
- **WM_DESTROY**: Cleans up and closes the program when the window is closed.

This program is a good example of working with popup menus and handling different user interactions in a Windows GUI application.

POPMENU Resource File Breakdown

The **POPMENU.RC** file defines the resources for the **POPMENU** program, particularly focusing on the **menu structure**. Here's a simple breakdown of its key components:

- **POPMENU MENU DISCARDABLE**: This starts the menu definition. The **DISCARDABLE** keyword means the resources are not critical for the program to function, but they are used for the user interface.
- **BEGIN**: Marks the start of the menu structure and encloses all the menus and menu items.
- **POPUP "MyMenu"**: Defines a top-level popup menu called "MyMenu". A popup is a menu that appears when you right-click or trigger a specific action.
- **POPUP "&File"**: This defines a popup menu called **File**. The **&** before "File" sets the **F** as the **hotkey** (activated by pressing **Alt + F**).
- **MENUITEM "&New", IDM_FILE_NEW**: This defines the menu item "New" within the **File** menu and associates it with an identifier **IDM_FILE_NEW**. Other items like "Open", "Save", and "Exit" are also defined in a similar way.
- **SEPARATOR**: Adds a horizontal line in the menu to visually separate different menu items.
- **POPUP "&Edit"**: This defines an **Edit** menu with additional items, similar to the File menu.
- **POPUP "&Background"**: Defines a menu for **background colors**, such as **White**, **Light Gray**, and **Black**, with the option for each color to be checked initially. This allows the user to change the window's background color.
- **POPUP "&Help"**: This defines a Help menu, which could include options like "About" or help documentation.
- **END**: Marks the end of the menu definition.

The **POPMENU.RC** file defines a hierarchical structure of popup menus, such as **File**, **Edit**, **Background**, and **Help**, with each containing various menu items.

The **"MyMenu"** menu serves as a container for all other popups, and the resource file defines the interactions for actions like changing the background color or performing file operations.

Example 2

The code you provided incorrectly uses the **MENUITEMTEMPLATE** structure, which is not part of the Windows API. To use the **LoadMenuIndirect** function, you should instead use the **MENUITEMTEMPLATEHEADER** structure.

This structure is required for creating a menu from memory. The original code needs to be corrected by replacing **MENUITEMTEMPLATE** with the correct **MENUITEMTEMPLATEHEADER** structure.

```
#include <Windows.h>

// Define menu item IDs
#define IDM_FILE_NEW    101
#define IDM_FILE_OPEN   102
#define IDM_FILE_SAVE   103
#define IDM_FILE_SAVE_AS 104
#define IDM_APP_EXIT    105

// Menu template
const MENUITEMTEMPLATE menuItems[] = {
    { 0, 0, 0, 0, MF_POPUP, 0, "File" },
    { 0, IDM_FILE_NEW, 0, 0, MF_STRING | MF_CHECKED, 0, "New" },
    { 0, IDM_FILE_OPEN, 0, 0, MF_STRING, 0, "Open..." },
    { 0, IDM_FILE_SAVE, 0, 0, MF_STRING, 0, "Save" },
    { 0, IDM_FILE_SAVE_AS, 0, 0, MF_STRING, 0, "Save &As..." },
    { 0, 0, 0, 0, MF_SEPARATOR, 0, 0 },
    { 0, IDM_APP_EXIT, 0, 0, MF_STRING, 0, "E&xit" },
    { 0, 0, 0, 0, MF_END, 0, 0 }
};

int main() {
    // Load menu
    HMENU hMenu = LoadMenuIndirect(menuItems);

    // Check if the menu was loaded successfully
    if (hMenu == NULL) {
        // Handle error
        return 1;
    }

    // Use the menu as needed

    // Clean up resources
    DestroyMenu(hMenu);

    return 0;
}
```

Menu Structure and Hierarchy

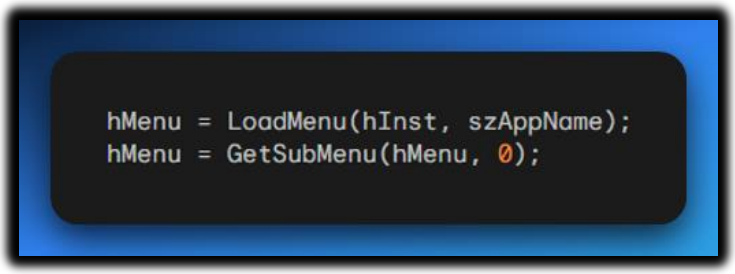
The **POPMENU** program uses a vertical menu structure, unlike **MENUDEMO**, which has a horizontal layout.

POPMENU defines a single top-level popup menu named "MyMenu," which contains submenus for **File**, **Edit**, **Background**, and **Help**.

Menu Handle Acquisition in WM_CREATE:

In the **WM_CREATE** message handler, **POPMENU** acquires the handle to the "MyMenu" popup menu in two steps:

1. **LoadMenu:** Loads the menu resource and retrieves the top-level menu handle.
2. **GetSubMenu:** Retrieves the handle of the "MyMenu" popup by using **GetSubMenu(0)**, as it's the first item in the top-level menu.



Popup Menu Tracking in WM_RBUTTONUP:

```
point.x = LOWORD(lParam);
point.y = HIWORD(lParam);
ClientToScreen(hwnd, &point);
TrackPopupMenu(hMenu, TPM_RIGHTBUTTON, point.x, point.y, 0, hwnd, NULL);
```

I. Mouse Position Retrieval:

- The x and y coordinates of the mouse are extracted from the lParam parameter of the **WM_RBUTTONUP** message.
- x = LOWORD(lParam) and y = HIWORD(lParam) are used to retrieve the coordinates.

II. Coordinate Conversion:

- The **ClientToScreen** function converts the mouse's position from **client coordinates** (relative to the window) to **screen coordinates** (absolute on the screen). This ensures the popup menu appears in the correct location on the screen.

III. TrackPopupMenu Invocation:

The **TrackPopupMenu** function is called to display the popup menu at the mouse position:

- **hMenu:** The handle to the "MyMenu" popup menu.
- **TPM_RIGHTBUTTON:** Specifies that the menu is triggered by a right-click.
- **point.x, point.y:** The x and y coordinates of the mouse position.
- **0:** A reserved parameter.
- **hwnd:** The window handle.
- **NULL:** No menu event handler is used.

IV. Popup Menu Display:

- **TrackPopupMenu** displays the "MyMenu" popup menu at the specified coordinates, with the **File**, **Edit**, **Background**, and **Help** options arranged vertically.

V. Nested Menu Behavior:

- If any of the top-level menu items (e.g., **File**, **Edit**) are selected, the corresponding nested submenu will appear to the right, maintaining the vertical menu layout.

VI. Main Menu and TrackPopupMenu Compatibility:

- Using the same menu for both the main menu and **TrackPopupMenu** is tricky because **TrackPopupMenu** needs a specific popup menu handle. There are workarounds in the Microsoft Knowledge Base article **ID Q99806** to help with this issue.

VII. Modifying the System Menu:

- **System Menu**: This is the menu in the top-left corner of the window (in the caption bar) and is usually part of the **WS_SYSMENU** style. You can add custom menu items to this system menu.
- Modifying the system menu is a quick way to add menus to simple programs without needing to define them in a resource script.

VIII. ID Number Restriction:

- Menu items added to the system menu must have ID numbers **lower than 0xF000** to avoid conflicts with the system's standard menu commands.

IX. WM_SYSCOMMAND Message Handling:

- When handling **WM_SYSCOMMAND** messages (which are triggered by the system menu), make sure to pass other messages to **DefWindowProc**. If you don't, it will disable standard system menu options (like Minimize, Maximize).

X. POORMENU Example:

- **POORMENU** (Poor Person's Menu) shows how to add a separator bar and three custom commands to the system menu. One of the commands removes the added items from the system menu.

XI. Implementation Details:

- **WM_CREATE Message**: In the **WM_CREATE** message handler, the program loads the system menu with **GetSystemMenu(hWnd, FALSE)** and adds custom items using **AppendMenu**.
- **WM_SYSCOMMAND Message**: The program checks the **wParam** value to determine which custom menu item was selected and performs the appropriate action. For standard system menu commands, it uses **DefWindowProc**.
- **Removing Additions**: A menu item called **Remove Additions** removes the custom menu items using **DeleteMenu**.

XII. Advantages of System Menu Modification:

- **Simplicity**: This method is simple and quick, ideal for adding menus to small programs.
- **No Resource Script Needed**: You don't need a resource script, which makes the code simpler.

XIII. Disadvantages of System Menu Modification:

- **ID Number Restrictions**: Custom menu items must have IDs lower than **0xF000**, limiting your options.
- **WM_SYSCOMMAND Handling**: You need to properly handle **WM_SYSCOMMAND** messages to ensure the system menu works correctly.

POORMENU.C - THE POOR PERSON'S MENU:

This program demonstrates how to modify the system menu of a window to add custom menu items, providing a simple method to add menus without using a resource script.

I. Initialization:

Header File: The program includes windows.h for access to Windows functions.

Constant Definitions: Constants are defined for custom menu item IDs:

- IDM_SYS_ABOUT: "About..." menu item.
- IDM_SYS_HELP: "Help..." menu item.
- IDM_SYS_REMOVE: "Remove Additions" menu item.

Window Procedure Declaration: Declares WndProc as the window procedure.

Global Variable: szAppName stores the application's name.

II. WinMain Function:

- **Window Class Registration**: Registers the window class, specifying styles like CS_HREDRAW and CS_VREDRAW for resizing.
- **Window Creation**: Calls CreateWindow to create the main window, specifying its position, style, and instance handle.
- **System Menu Modification**: Retrieves the system menu using GetSystemMenu and adds menu items with AppendMenu: Separator, "About...", "Help...", and "Remove Additions".
- **Window Display**: Uses ShowWindow and UpdateWindow to display the window.
- **Message Loop**: The loop calls GetMessage to retrieve messages and dispatch them to WndProc.
- **Exit Message Processing**: When WM_QUIT is received, PostQuitMessage exits the message loop.

III. WndProc Function:

WM_SYSCOMMAND Handling: Handles system menu item selections:

- IDM_SYS_ABOUT: Displays an "About" message box.
- IDM_SYS_HELP: Displays a "Help" message box.
- IDM_SYS_REMOVE: Removes custom menu items by restoring the original system menu with GetSystemMenu(TRUE).

WM_DESTROY Handling: Posts a WM_QUIT message when the window is destroyed.

DefWindowProc: For other messages, calls DefWindowProc for standard window handling.

Menu ID Definitions: Defines constants for custom menu items at the beginning of the file: IDM_ABOUT, IDM_HELP, and IDM_REMOVE.

Retrieving the System Menu Handle: After window creation, GetSystemMenu is called with FALSE to modify the system menu.

Modifying the System Menu:

AppendMenu is used to add the following items to the system menu:

- **Separator**: Visually separates custom items.
- **"About..."**: Displays an "About" message box.
- **"Help..."**: Displays a message box indicating help is not implemented.
- **"Remove Additions"**: Restores the original system menu by calling GetSystemMenu(TRUE).

IV. Handling WM_SYSCOMMAND Messages

The standard system menu generates WM_SYSCOMMAND messages with specific wParam values corresponding to each menu item. These values are:

WM_SYSCOMMAND Message	wParam Value	Description
SC_RESTORE	0xF120	Restore the window to its normal size and position.
SC_MOVE	0xF010	Move the window.
SC_SIZE	0xF000	Resize the window.
SC_MINIMUM	0xF020	Minimize the window.
SC_MAXIMUM	0xF030	Maximize the window.
SC_CLOSE	0xF060	Close the window.

These messages are handled by DefWindowProc by default, but you can process them yourself to customize behavior, such as disabling or modifying certain system menu actions.

In addition to the basic window commands, there are a few other standard system menu items you can work with:

WM_SYSCOMMAND Message	wParam Value	Description
SC_NEXTWINDOW	0xF040	Switch to the next window in the taskbar.
SC_PREVWINDOW	0xF001	Switch to the previous window in the taskbar.
SC_VSCROLL	0xF013	Scroll the window vertically.
SC_HSCROLL	0xF014	Scroll the window horizontally.
SC_ARRANGE	0xF122	Arrange the windows on the desktop.

By handling WM_SYSCOMMAND messages, you can customize the behavior of these system menu options, as well as add or remove custom options.

V. Menu Modification Options

The **POORMENU** program demonstrated how to add menu items to the system menu using AppendMenu. However, before **Windows 3.0**, the ChangeMenu function was the main method for modifying menus, although it was more complex. Today, Windows provides a set of specialized functions to simplify menu modifications:

- **AppendMenu**: Adds a new menu item to the end of an existing menu.
- **DeleteMenu**: Removes an existing menu item and destroys its associated resources.
- **InsertMenu**: Adds a new menu item at a specific position within an existing menu.
- **ModifyMenu**: Changes the attributes (text, flags, data) of an existing menu item.
- **RemoveMenu**: Removes a menu item but keeps its associated popup menu and resources intact.

VII. DeleteMenu vs. RemoveMenu for Popup Menus

The key difference between DeleteMenu and RemoveMenu is how they handle popup menus:

- **DeleteMenu**: Completely removes a menu item and destroys the associated popup menu and resources.
- **RemoveMenu**: Removes a menu item but **retains** the popup menu, leaving its resources intact.

Understanding these distinctions is important when dealing with popup menus and managing their resources.

Example Usage

To add a new menu item named "New" to the end of the existing menu:

```
AppendMenu(hMenu, MF_STRING, IDM_NEW, TEXT("New"));
```

To remove or delete the menu item with IDM_OPEN and destroy its associated resources:

```
DeleteMenu(hMenu, IDM_OPEN);
```

To insert a new menu item named "Save" at position 3 of the existing menu:

```
InsertMenu(hMenu, 3, MF_STRING, IDM_SAVE, TEXT("Save"));
```

To modify the text of the menu item with IDM_HELP:

```
ModifyMenu(hMenu, IDM_HELP, MF_STRING, NULL, TEXT("Help & Support"));
```

To remove an existing menu item while retaining its popup menu:

```
RemoveMenu(hMenu, IDM_SETTINGS);
```

These specialized functions provide a more organized and efficient approach to modifying menus in modern Windows applications, offering a simpler and more maintainable alternative to the legacy ChangeMenu function.

OTHER MENU COMMANDS

Forcing Menu Bar Redraw

After modifying a top-level menu item, the changes may not be immediately reflected on the menu bar until Windows redraws it. To force an immediate redraw, use the DrawMenuBar function:

```
DrawMenuBar(hwnd);
```

The argument to DrawMenuBar is the handle to the window, not the menu itself. This redraws the entire menu bar, including the updated top-level menu item.

Retrieving Popup Menu Handle

To obtain the handle to a popup menu, use the GetSubMenu function:

```
hMenuPopup = GetSubMenu(hMenu, iPosition);
```

Here, hMenu is the handle to the top-level menu, and iPosition is the index (starting at 0) of the popup menu within the top-level menu. The returned handle, hMenuPopup, can be used with other menu functions, such as AppendMenu, to manipulate the popup menu.

Counting Menu Items

To determine the number of items in a top-level or popup menu, use the GetMenuItemCount function:

```
iCount = GetMenuItemCount(hMenu);
```

This function takes the handle to the menu as its argument and returns the total number of items in the menu.

Obtaining Menu ID from Popup Menu

To retrieve the menu ID for an item in a popup menu, use the GetMenuItemID function:

```
id = GetMenuItemID(hMenuPopup, iPosition);
```

Here, hMenuPopup is the handle to the popup menu, and iPosition is the index (starting at 0) of the item within the popup menu. The returned value, id, is the unique menu ID associated with the item.

Checking or Unchecking Menu Items

In **MENUDEMO**, the CheckMenuItem function is used to check or uncheck an item in a popup menu. The hMenu is the handle to the menu, id is the menu item's ID, and iCheck specifies whether to check (MF_CHECKED) or uncheck (MF_UNCHECKED) the item.

```
CheckMenuItem(hMenu, id, iCheck);
```

For popup menus, you can also use the item's position (starting at 0) instead of its menu ID:

```
CheckMenuItem(hMenu, iPosition, MF_CHECKED | MF_BYPOSITION);
```

This alternative allows you to check or uncheck an item using its position within the popup menu.

Enabling or Disabling Menu Items

The EnableMenuItem function, similar to CheckMenuItem, enables or disables menu items. Its third argument can be MF_ENABLED, MF_DISABLED, or MF_GRAYED:

```
EnableMenuItem(hMenu, id, MF_ENABLED); // Enable the item
EnableMenuItem(hMenu, id, MF_DISABLED); // Disable the item
EnableMenuItem(hMenu, id, MF_GRAYED); // Gray out the item
```

For top-level menu items with popup menus, use MF_BYPOSITION instead of a menu ID:

```
EnableMenuItem(hMenu, iPosition, MF_ENABLED | MF_BYPOSITION);
```

An example of using EnableMenuItem is demonstrated in the POPPAD2 program discussed later.

Highlighting Menu Items

The HiliteMenuItem function, like CheckMenuItem and EnableMenuItem, controls the highlighting of menu items. It uses MF_HILITE and MF_UNHILITE:

```
HiliteMenuItem(hMenu, id, MF_HILITE); // Highlight the item
HiliteMenuItem(hMenu, id, MF_UNHILITE); // Unhighlight the item
```

This highlighting is the reverse video effect used when selecting menu items. Typically, you don't need to manually manage menu highlighting, as Windows handles it automatically.

Retrieving Menu Text

To retrieve the character string associated with a menu item, use the GetMenuString function:

```
iCharCount = GetMenuString(hMenu, id, pString, iMaxCount, iFlag);
```

The iFlag parameter indicates whether to use a menu ID (MF_BYCOMMAND) or a positional index (MF_BYPOSITION). The function copies up to iMaxCount characters into pString and returns the actual number of characters copied.

Obtaining Menu Item Flags

To determine the current flags of a menu item, use the GetMenuState function:

```
iFlags = GetMenuState(hMenu, id, iFlag);
```

Similar to GetMenuString, iFlag specifies whether to use a menu ID (MF_BYCOMMAND) or a positional index (MF_BYPOSITION).

The returned iFlags value represents the combination of current flags, which can be checked against the MF_DISABLED, MF_GRAYED, MF_CHECKED, MF_MENUBREAK, MF_MENUBARBREAK, and MF_SEPARATOR flags.

Destroying Menus

When you no longer need a menu in your program, you can destroy it using the DestroyMenu function:

```
DestroyMenu(hMenu);
```

This function invalidates the menu handle and frees the associated resources.

The nopopup program…

NoPopups.mp4

NOPOPUPS Program Overview

The **NOPOPUPS** program uses multiple top-level menus instead of nested ones, switching between them using the SetMenu function, similar to the character-mode style of Lotus 1-2-3.

Main Function
The WinMain function initializes the application by registering the window class, creating the window, and handling the message loop.

Window Procedure

- **WM_CREATE**: Loads three top-level menus (hMenuMain, hMenuFile, hMenuEdit) and sets the initial menu to hMenuMain using SetMenu.
- **WM_COMMAND**: Switches between top-level menus (IDM_MAIN, IDM_FILE, IDM_EDIT) or triggers actions for individual commands (e.g., IDM_FILE_NEW).
- **WM_DESTROY**: Restores the main menu and releases resources by destroying unused menus.

Menu Switching Mechanism
The program dynamically switches between top-level menus using SetMenu, avoiding nested menus.

Menu Resource Creation
The program uses three separate menu resources: MenuMain, MenuFile, and MenuEdit, defined in the NOPOPUPS.RC file with MENU and MENUITEM keywords.

Menu Loading and Handling

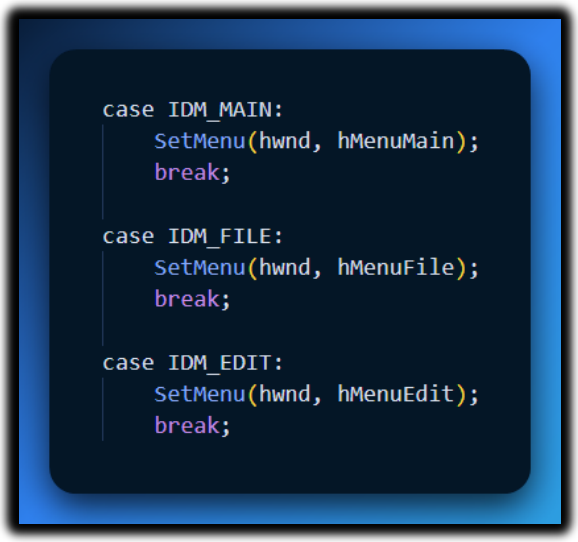
In the window procedure, when the **WM_CREATE** message is received, the program loads each menu resource into memory using the LoadMenu function:

```
hMenuMain = LoadMenu(hInstance, TEXT("MenuMain"));
hMenuFile = LoadMenu(hInstance, TEXT("MenuFile"));
hMenuEdit = LoadMenu(hInstance, TEXT("MenuEdit"));
```

This creates separate menu handles (hMenuMain, hMenuFile, and hMenuEdit) for each menu resource.

Dynamic Menu Switching

The program switches between menus by dynamically assigning a menu handle to the window using SetMenu. When a menu option (like IDM_MAIN, IDM_FILE, or IDM_EDIT) is selected, the corresponding menu handle is assigned:



```
case IDM_MAIN:
    SetMenu(hwnd, hMenuMain);
    break;

case IDM_FILE:
    SetMenu(hwnd, hMenuFile);
    break;

case IDM_EDIT:
    SetMenu(hwnd, hMenuEdit);
    break;
```

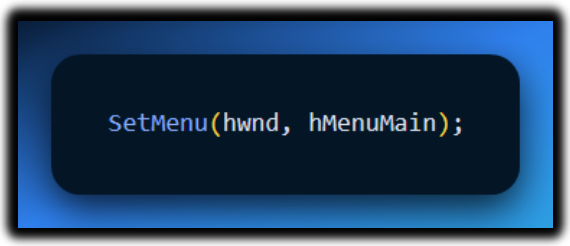
Menu Resource Advantages

Using separate menu resources for each top-level menu has several advantages:

- **Clarity**: Each menu is clearly defined in its own resource, making it easier to understand and manage.
- **Isolation**: Changes to one menu do not affect others, minimizing the risk of unintended consequences.
- **Flexibility**: Each menu can be customized independently, allowing for different layouts or styles.

Initial Menu Setup

The program starts by displaying the **main menu** using SetMenu:



```
SetMenu(hwnd, hMenuMain);
```

This assigns hMenuMain to the window, making it the visible menu. The main menu contains three options: "MAIN:", "File...", and "Edit...". "MAIN:" is disabled using the **INACTIVE** flag, preventing it from generating WM_COMMAND messages when selected.

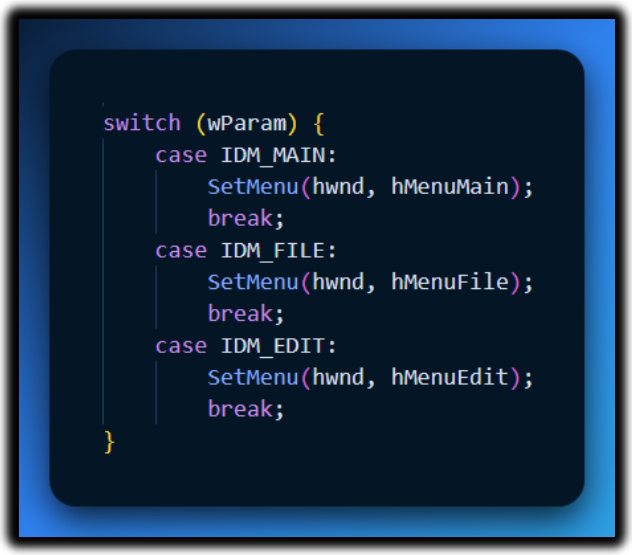
Submenu Identification

The **File** and **Edit** menus are labeled with "FILE:" and "EDIT:", respectively. These labels help distinguish them as submenus.

The last option in each menu is "(Main)", which, when selected, triggers a return to the **main menu**. This is handled by the program logic based on the menu ID associated with "(Main)".

Menu Switching Mechanism

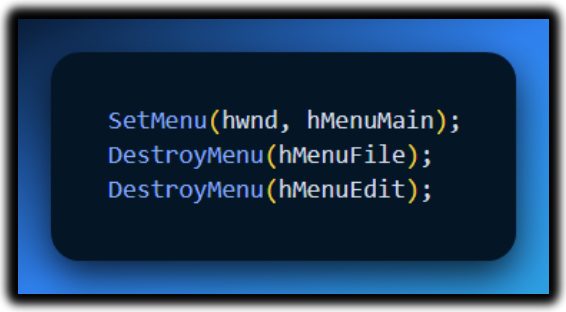
Menu switching occurs in the WM_COMMAND handler. When a menu option is selected, the corresponding menu handle (hMenuMain, hMenuFile, or hMenuEdit) is assigned to the window using SetMenu, effectively switching the visible menu:



```
switch (wParam) {
    case IDM_MAIN:
        SetMenu(hwnd, hMenuMain);
        break;
    case IDM_FILE:
        SetMenu(hwnd, hMenuFile);
        break;
    case IDM_EDIT:
        SetMenu(hwnd, hMenuEdit);
        break;
}
```

Menu Destruction

During the **WM_DESTROY** message, the program cleans up by setting the menu back to the main menu and destroying the File and Edit menus:



This ensures proper resource management and prevents memory leaks. The main menu is automatically destroyed when the window closes.

The **NOPOPUPS** program uses separate menu resources, dynamic menu switching, and explicit destruction to create a menu system that avoids traditional nested menus.

The program allows the user to switch between top-level menus using SetMenu and ensures proper cleanup of resources when the window is closed.