MENUS AND OTHER RESOURCES

Icons

Icons are small graphical images that represent programs, files, or folders. They are displayed in the title bar of application windows, in the Start menu, in the taskbar, in Windows Explorer, and as shortcuts on the desktop. Icons can be in color or black and white, and they can be in any size.

Cursors

Cursors are graphical images that represent the mouse pointer. They change shape depending on the context, such as when the mouse is hovering over a link, when it is selecting text, or when it is resizing a window. Cursors can be in color or black and white, and they can be in any size.

Character Strings

Character strings are text strings that are used by programs. They can be used for menus, dialog boxes, error messages, and other purposes. Character strings can be stored in the program's .EXE file or in a separate resource file.

Custom Resources

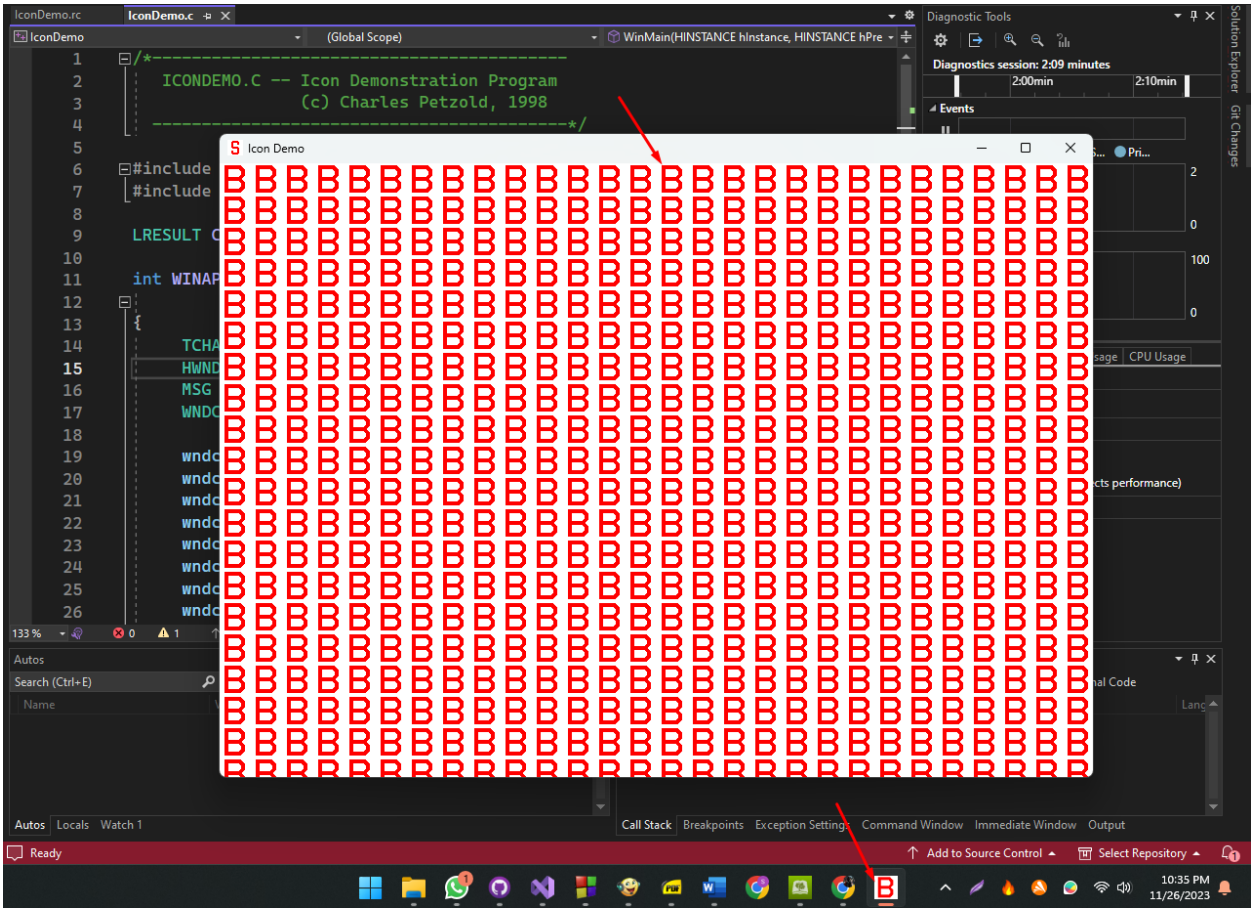
Custom resources are any type of resource that is not an icon, cursor, character string, or menu. They can be used for storing data that is specific to the program, such as images, sounds, or video. Custom resources are stored in the program's .EXE file or in a separate resource file.

Menus

Menus are hierarchical lists of options that users can select to perform actions in a program. They can be displayed as pull-down menus, context menus, or toolbars. Menus can be customized to include the specific options that a program needs.

Keyboard Accelerators

Keyboard accelerators are keyboard shortcuts that allow users to quickly perform actions in a program. They are typically combinations of two or more keys, such as Ctrl+S to save a file. Keyboard accelerators can be customized to the user's preferences.



Introduction

Resources in C programming offer a convenient way to bind various components of a program into the executable file.

This eliminates the need for separate files, making it easier to manage and distribute the application. For instance, icons, cursors, strings, and other custom resources can be included within the program's .EXE file.

Icons as Resources

One notable example is the inclusion of icons. Typically, an icon would require a separate file, but with resources, it can be stored in an editable file on the developer's computer and bound into the .EXE during the build process.

This approach streamlines development and ensures that the icon is an integral part of the executable.

Adding an Icon to a Program

To add an icon to a program, Visual C++ Developer Studio provides the Image Editor, allowing developers to draw an icon that gets saved in an .ICO file.

Simultaneously, Developer Studio generates a resource script (with .RC extension) listing all program resources and a header file (RESOURCE.H) enabling the program to reference these resources.

Project Setup: ICONDEMO

Let's illustrate this process by creating a new project named ICONDEMO in Visual C++ Developer Studio.

After creating the project, the studio generates several files, including ICONDEMO.DSW, ICONDEMO.DSP, and ICONDEMO.MAK. Additionally, a C source code file (ICONDEMO.C) is created, where the program logic will be implemented.

Example Program Structure

Here's a simplified version of the program structure:

```
// 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 is in the icondemo folder….

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

Windows Header File: The #include <windows.h> statement includes the Windows header file, which contains essential definitions for interacting with the Windows API.

Resource File Inclusion: The #include "resource.h" statement incorporates the resource file, which holds the program's resources, including icons and cursors.

Window Procedure Function: The LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) function serves as the window procedure, responsible for handling messages sent to the window by the operating system.

Program Entry Point: The int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) function acts as the program's entry point, executed when the program starts.

Variable Declarations: Variables are declared to store essential program information, including the program's name (szAppName), window handle (hwnd), current message (msg), and window class structure (wndclass).

Window Class Configuration: The window class structure (wndclass) is configured with settings that define the window's appearance and behavior.

Window Registration: The RegisterClass (&wndclass) statement registers the window class with the system, allowing the program to create windows based on that class.

Window Creation: The hwnd = CreateWindow (...) statement creates a window using the registered window class, specifying the window's name, position, size, and other attributes.

Window Display: The ShowWindow (hwnd, iCmdShow) function displays the created window, making it visible to the user.

Window Update: The UpdateWindow (hwnd) function refreshes the window's contents, ensuring it is properly rendered on the screen.

Message Loop: The while (GetMessage (&msg, NULL, 0, 0)) (...) loop continuously retrieves messages from the message queue and dispatches them to the window procedure function.

Message Translation: The TranslateMessage (&msg) statement translates the retrieved message into a format compatible with the window procedure function.

Message Dispatching: The DispatchMessage (&msg) statement sends the translated message to the window procedure function for processing.

1. Creating a Resource Script

To create an icon for the ICONDEMO program, you first need to create a resource script. A resource script is a text file that contains the definitions of the program's resources, such as icons, cursors, and menus.

To create a resource script, follow these steps:

- In Developer Studio, select File > New.
- In the New dialog box, select Resource Script and click OK.
- In the File Name field, type ICONDEMO.RC and click OK.
- Developer Studio will create two new files: ICONDEMO.RC, the resource script, and RESOURCE.H, a header file that allows the C source code file and the resource script to refer to the same defined identifiers.

2. Adding an Icon Resource

To add an icon resource to the resource script, follow these steps:

- In Developer Studio, open the ICONDEMO.RC file.
- Select Insert > Resource.
- In the Resource dialog box, select Icon and click New.
- A blank 32-pixel-by-32-pixel icon will appear in the resource editor. You can use the painting tools and colors to create your icon.

3. Saving the Icon Resource

Once you have created your icon, you need to save it as an ICO file. To do this, follow these steps:

- In the icon properties dialog box, change the ID to IDI_ICON.
- Change the Filename to ICONDEMO.ICO.
- Click OK.
- Developer Studio will save the icon as ICONDEMO.ICO in the project directory.

4. Compiling the Program

Now that you have created the icon resource, you can compile the program. To do this, follow these steps:

- In Developer Studio, select Build > Build ICONDEMO.
- Developer Studio will compile the program and link it with the icon resource.

5. Running the Program

Once the program has been compiled, you can run it by following these steps:

- In Developer Studio, select Debug > Start Debugging.
- The program will run and the icon will be displayed in the window.

Here are some additional tips for creating icons:

- Use a distinctive color palette so that your icon will stand out.
- Use simple shapes and colors so that your icon is easy to understand and remember.
- Avoid using too much detail, as this can make your icon appear cluttered and difficult to see.

Creating Resource Files

Resource files are text files that contain the definitions of the program's resources, such as icons, cursors, and menus.

Resource files are compiled into binary resource files using the resource compiler RC.EXE. The binary resource files are then linked with the program's object files and libraries to create the final executable file.

Loading Icons

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

- hInstance: The instance handle of the program
- MAKEINTRESOURCE(IDI_ICON): The resource identifier of the icon
- The MAKEINTRESOURCE macro takes an integer resource identifier and converts it to a resource identifier that can be used with the LoadIcon function.

Drawing Icons

The DrawIcon function is used to draw an icon on the screen. The function takes four arguments:

- hdc: The device context of the window in which to draw the icon
- x: The x-coordinate of the upper-left corner of the icon
- y: The y-coordinate of the upper-left corner of the icon
- hIcon: The handle of the icon to draw

Small Icons

Windows will automatically use a smaller version of an icon when it is more appropriate, such as in the title bar and the taskbar.

The small icon size can be obtained from GetSystemMetrics with the SM_CXSMSIZE and SM_CYSMSIZE indices. For most display adapters in current use, the small icon size is 16 by 16 pixels.

To create a small icon, you can select Small (16x16) from the Device combo box in the icon editor. You can then draw a different icon for the small size.

Understanding Resource Script ICON Statements

The line IDI_ICON ICON DISCARDABLE "icondemo.ico" in the ICONDEMO.RC file is a resource script ICON statement. It defines an icon resource with the following properties:

- Identifier: IDI_ICON
- Type: ICON
- Filename: icondemo.ico
- Attribute: DISCARDABLE
- Resource Identifiers

The identifier IDI_ICON is a numeric identifier that uniquely identifies the icon resource within the project. In this case, the identifier is 101. Resource identifiers are used by the LoadIcon function to retrieve specific resources from the compiled resource file.

Resource Types

The type ICON indicates that the resource is an icon. Resource types are used by the resource compiler to organize and manage different types of resources.

Resource Filenames

The filename icondemo.ico specifies the location of the icon file that contains the icon image. The filename can be a relative or absolute path.

Resource Attributes

The attribute DISCARDABLE indicates that the icon can be discarded from memory by Windows if necessary to free up space. This attribute is the default and does not need to be specified.

Obtaining a Handle to an Icon

A program can obtain a handle to an icon by calling the LoadIcon function. The LoadIcon function takes two arguments:

- hInstance: The instance handle of the program
- MAKEINTRESOURCE(IDI_ICON): The resource identifier of the icon
- The MAKEINTRESOURCE macro converts the integer resource identifier IDI_ICON to a resource identifier that can be used with the LoadIcon function.

Here is an example of how to obtain a handle to the icon defined in the ICONDEMO.RC file:

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

Using the Icon Handle

The icon handle can be used to draw the icon on the screen using the DrawIcon function. The DrawIcon function takes four arguments:

- hdc: The device context of the window in which to draw the icon
- x: The x-coordinate of the upper-left corner of the icon
- y: The y-coordinate of the upper-left corner of the icon
- hIcon: The handle of the icon to draw

Here is an example of how to draw the icon defined in the ICONDEMO.RC file at the coordinates (100, 100):

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

The process of getting a handle to an icon involves defining the icon resource in the resource script, compiling the resource script into a binary resource file, and linking the binary resource file into the program's executable file.

Once the icon is linked into the executable file, the program can obtain a handle to the icon by calling the LoadIcon function. The icon handle can then be used to draw the icon on the screen using the DrawIcon function.

Loading Icons Using LoadIcon

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

- hInstance: The instance handle of the program
- resourceIdentifier: The identifier of the icon

The identifier can be a numeric identifier, a character string, or a string prefixed with the # character.

Loading Icons by Numeric Identifier

To load an icon by numeric identifier, you can use the MAKEINTRESOURCE macro.

The MAKEINTRESOURCE macro takes an integer identifier and converts it to a resource identifier that can be used with the LoadIcon function.

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

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

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");
```

Loading Icons by String Prefixed with # Character

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

Here is an example of how to load an icon by string prefixed with the # character:

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

Using LoadIcon in ICONDEMO

ICONDEMO calls the LoadIcon function twice:

- Once when defining the window class.
- Once in the window procedure to obtain a handle to the icon for drawing.

In both cases, ICONDEMO uses the MAKEINTRESOURCE macro to convert the numeric identifier IDI_ICON to a resource identifier.

Here is an example of how ICONDEMO calls the LoadIcon function in the window procedure:

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

The LoadIcon function is a versatile function that can be used to load icons by numeric identifier, character string, or string prefixed with the # character. ICONDEMO demonstrates how to use the LoadIcon function to load an icon from a resource file.

Using Icons in Windows Programs: A Deep Dive

1. Setting Icons with WNDCLASS and RegisterClass

When defining a window class using the WNDCLASS structure and registering it with RegisterClass, it's common to specify an icon. This is typically done through the hIcon field of the WNDCLASS structure. Windows intelligently selects the appropriate image size from a single icon file when needed.

2. RegisterClassEx and WNDCLASSEX

There exists an enhanced version, RegisterClassEx, which utilizes the WNDCLASSEX structure. This structure introduces two additional fields: cbSize and hIconSm.

The cbSize field denotes the size of the WNDCLASSEX structure, while hIconSm is intended for the small icon handle. However, using WNDCLASSEX doesn't seem necessary since Windows can extract correctly sized icons from a single file.

3. Dynamic Icon Changes with SetClassLong

To dynamically change the program's icon during runtime, you can use the SetClassLong function. For example, if you have a second icon file associated with the identifier IDI_ALTICON, you can switch to that icon using:

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

Alternatively, if you prefer not to retain the program's icon handle but instead display it using the DrawIcon function, you can retrieve the handle through GetClassLong. Here's an example:

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

While some sections of the Windows documentation suggest that LoadIcon is "obsolete" and favor LoadImage instead, LoadImage, documented in /Platform SDK/User Interface Services/Resources/Resources, undoubtedly offers greater flexibility.

However, it has not yet surpassed LoadIcon's simplicity. In the provided ICONDEMO example, LoadIcon is called twice for the same icon without issues or additional memory consumption.

LoadIcon is one of the few functions that acquire a handle without explicit handle destruction. It's worth noting that while a DestroyIcon function exists, it is primarily used in conjunction with functions like CreateIcon, CreateIconIndirect, and CreateIconFromResource, which enable the dynamic creation of icon images algorithmically within a program.

In conclusion, icons are fundamental elements of Windows programming, and understanding their proper implementation is crucial for crafting user-friendly and aesthetically pleasing applications.

Customizing Mouse Cursors in Windows Programming

Similar to customizing icons, customizing mouse cursors enhances the visual appeal and interactivity of your Windows applications.

While most programmers find the default cursors provided by Windows to be sufficient, customizing cursors can add a unique touch to your program.

Creating customized cursors is straightforward and can be done within the Developer Studio.

Follow the same steps as creating icons: select "Resource" from the "Insert" menu and choose "Cursor." Remember to define the hotspot, which is the point on the cursor where interactions occur.

To set a customized cursor for your window class, use 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")) ;
```

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

```
SetCursor (hCursor) ;
```

Invoke SetCursor during WM_MOUSEMOVE message processing. Otherwise, Windows will use the cursor specified in the window class when the cursor is moved. Documentation suggests that SetCursor is efficient if the cursor doesn't require significant changes.

Utilizing Character String Resources in Windows Programming

In Windows programming, the integration of character string resources might initially seem unconventional since regular character strings defined in source code are commonly employed.

However, character string resources serve a distinct purpose, primarily facilitating the translation of programs into different languages. This becomes especially relevant when dealing with menus and dialog boxes as part of the resource script, as demonstrated later in this chapter and the next.

By using character string resources instead of embedding strings directly into the source code, all the text utilized by your program consolidates into one file—the resource script.

This proves advantageous for translation efforts; if the text in the resource script is translated into another language, creating a foreign-language version of your program simply involves relinking, providing a safer alternative to modifying the source code directly.

To create a string table, you can select "Resource" from the Insert menu and then choose "String Table."

The strings appear in a list on the right side of the screen, allowing you to select and define identifiers and corresponding strings for each entry.

In the resource script, the strings are organized within a multiline statement, as illustrated below:

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

Historically, if you were manually creating this string table in a text editor, you could use left and right curly brackets instead of the BEGIN and END statements.

While a resource script can incorporate multiple string tables, each ID must uniquely identify a single string, and each string can be only one line long with a maximum of 4097 characters.

Control characters like \t and \n for tabs and line breaks are recognized by functions like DrawText and MessageBox.

To utilize string resources in your program, the LoadString function can be employed:

```
LoadString(hInstance, id, szBuffer, iMaxLength);
```

Here, id refers to the ID number preceding each string in the resource script, szBuffer is a pointer to a character array that receives the string, and iMaxLength is the maximum number of characters to transfer. The function returns the number of characters in the string.

Commonly, string ID numbers are macro identifiers defined in a header file, often prefixed with IDS_. In scenarios where additional information must be embedded in the string when displayed, C formatting characters can be used, treating the string as a formatting string in wsprintf.

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

The LoadStringW function loads Unicode text directly, while the LoadStringA function (available under Windows 98) performs a conversion from Unicode to the local code page.

Now, let's explore a function example that employs three character strings to display error messages in a message box. The RESOURCE.H header file contains identifiers for these messages, and the resource script defines a corresponding string table.

The C source code includes this header file and implements a function to display a message box.

```
#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:

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

This structure exemplifies the seamless integration of character string resources in a Windows program, streamlining the localization process and enhancing code maintainability.

Custom Resources in Windows

Custom resources, also known as user-defined resources, are a powerful feature of the Windows development platform that allows programmers to store and access miscellaneous data within their applications.

Unlike external files, custom resources are embedded directly into the executable file, making them convenient for storing sensitive or frequently accessed data.

Creating Custom Resources

Custom resources are typically created using a resource script file, which is a text file with a .RC extension.

The resource script file defines the resource type, resource name, and the data associated with the resource.

For example, the following resource script defines a custom resource named IDR_BINTYPE1 of type BINTYPE and associates it with the file BINDATA.BIN:

```
IDR_BINTYPE1 BINTYPE BINDATA.BIN
```

Loading and Accessing Custom Resources

To load and access a custom resource within your application, you can use the LoadResource and LockResource functions. The LoadResource function takes three parameters:

- hInstance: The handle to the application instance
- lpName: The name or ID of the resource
- lpType: The type of the resource

The LoadResource function returns a handle to the resource, which can then be passed to the LockResource function to lock the resource into memory. The LockResource function returns a pointer to the resource data, which can be used to access the data.

Freeing Custom Resources

Once you have finished accessing a custom resource, you should free it from memory using the FreeResource function. This will prevent memory leaks and ensure that the resource data is properly released.

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;
}
```

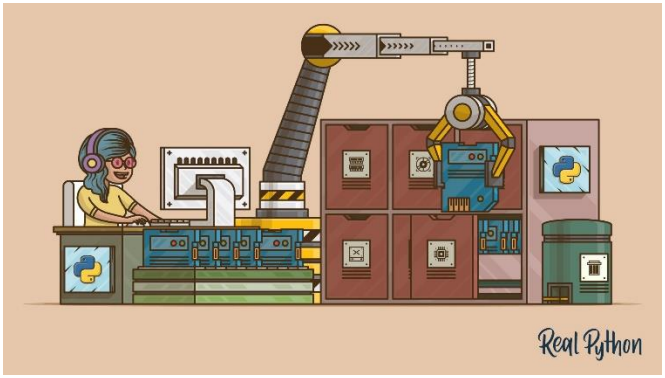
This code will load the custom resource IDR_BINTYPE1, lock it into memory, and print its contents to the console.

Benefits of Using Custom Resources

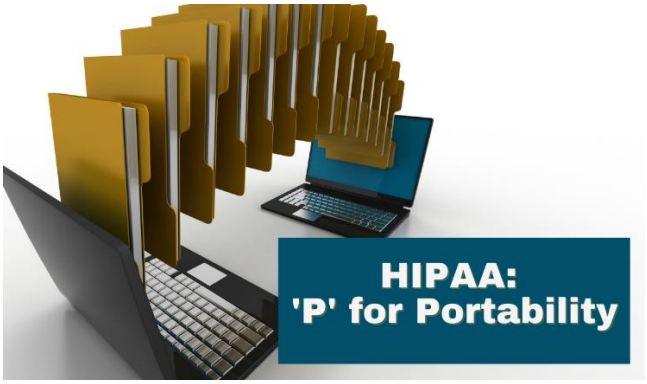
Custom resources offer several advantages over external files for storing data in Windows applications:

Embedded Data: Custom resources are embedded directly into the executable file, making them convenient for storing sensitive or frequently accessed data.

Memory Management: Windows handles the loading and unloading of custom resources, eliminating the need for explicit file I/O operations.



Portability: Custom resources are embedded in the executable file, making applications portable without relying on external files.



Security: Custom resources are protected by the executable file's permissions, enhancing data security.

PROGRAM CODE POEPOEM EXPLAINED:

The provided code is part of a Windows program named "POEPOEM," which demonstrates the use of custom resources, including an icon and text. Let's break down the code in depth:

Resource Loading and Initialization:

The WinMain function is the program's entry point. It initializes necessary variables, including the application name (szAppName) and caption (szCaption), by loading them from string resources using LoadString.

The window class is registered, and if registration fails, an error message is displayed using MessageBoxA. This message is loaded from the IDS_ERRMSG resource.

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

Window Creation and Message Loop:

The program creates the main window using CreateWindow and enters the message loop. The main window's class name, icon, and cursor are set during window class registration.

Inside the message loop, messages are processed using TranslateMessage and DispatchMessage.

```
hwnd = CreateWindow(szAppName, szCaption, WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN, ...);
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Window Procedure (WndProc):

The window procedure handles various messages, including WM_CREATE, WM_SIZE, WM_SETFOCUS, WM_VSCROLL, WM_PAINT, and WM_DESTROY.

On WM_CREATE, the program sets up a vertical scrollbar (hScroll) and loads the text resource (AnnabelLee) using LoadResource and LockResource.

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

Scrollbar and Text Display:

The program calculates the number of lines in the text resource to set the scrollbar range.

On WM_SIZE, the scrollbar is positioned, and on WM_VSCROLL, the text is scrolled accordingly.

```
case WM_SIZE:
    MoveWindow(hScroll, LOWORD(lParam) - xScroll, 0, xScroll, cyClient = HIWORD(lParam), TRUE);
    SetFocus(hwnd);
    return 0;

case WM_VSCROLL:
    // Scroll handling logic
    return 0;
```

Text Rendering and Cleanup:

On WM_PAINT, the text is rendered using DrawTextA based on the current scrollbar position.

On WM_DESTROY, resources are freed.

```
case WM_PAINT:
    // Text rendering logic
    return 0;

case WM_DESTROY:
    FreeResource(hResource);
    PostQuitMessage(0);
    return 0;
```

In summary, the code demonstrates the initialization, creation, and message processing of a Windows application. It emphasizes the use of custom resources, particularly a text resource (AnnabelLee), and incorporates a scrollbar for text scrolling within the main window. The code structure ensures efficient handling of messages and resource management.

POEPOEM PROGRAM RESOURCE SECTION IN-DEPTH

Resource Types and Definitions:

The resource script file (POEPOEM.RC) defines various resource types, such as text (ANNABELLEE), an icon (POEPOEM), and string tables (STRINGTABLE). These resources are identified by unique names and types.

The POEPOEM icon and ANNABELLEE text are associated with external files (poepoem.ico and poepoem.txt, respectively).

String Table Usage:

The string table defined in the resource script (STRINGTABLE) includes string resources with identifiers (IDS_APPNAME, IDS_CAPTION, and IDS_ERRMSG).

The RESOURCE.H header file contains macro definitions for these string identifiers.

Loading String Resources:

During program initialization, the LoadString function is used to load string resources into memory. For example:

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

Handling Unicode and ANSI Strings:

There's a note about loading strings in both Unicode (LoadStringW) and ANSI (LoadStringA) formats. This is relevant for compatibility with different Windows versions. For instance, under Windows 98, LoadStringW is not supported.

Child Window Scroll Bar Control:

The application uses a child window scroll bar control instead of a window scroll bar. This control provides an automatic keyboard interface, eliminating the need for specific keyboard event processing (WM_KEYDOWN).

Error Handling and Message Display:

The program performs error handling during class registration, loading appropriate error messages from the string resources. If the Unicode version of the program is run under Windows 98, it handles the loading and display of strings using both Unicode and ANSI functions (MessageBoxA).

Facilitating Translation:

By defining character strings as resources, the program becomes more accessible for translators aiming to convert it into a foreign-language version.

Excerpt from "Annabel Lee":

The program includes an excerpt from the poem "Annabel Lee" by Edgar Allan Poe, stored in the resource file (POEPOEM.TXT).

These main points highlight the resource-centric approach of the application, leveraging resources for text, icons, and string handling to enhance modularity, ease of translation, and error handling.

MENUS IN WINDOWS PROGRAMMING

Menus are an essential part of the user interface in Windows applications. They provide users with a clear and concise way to interact with the program and perform various actions.

Menus are typically displayed as a list of options, each with a corresponding command or action. When a user selects a menu item, the application receives a notification and executes the associated command.

Menu Structure

The structure of menus in Windows applications is hierarchical. Each menu can contain submenu items, which are nested within the main menu. This allows for a more organized and manageable presentation of options, especially when dealing with a large number of commands.

Types of Menus

There are several types of menus commonly used in Windows applications:

Main Menu: The main menu is the top-level menu, typically located at the top of the application window. It provides access to the primary functions and features of the program.

Popup Menu: Popup menus are context-sensitive menus that appear when the user right-clicks on an object or control within the application. They provide options specific to the object or context.

System Menu: The system menu is a special menu associated with the window's title bar. It provides options for managing the window, such as moving, resizing, minimizing, maximizing, and closing.

Menu Creation

Menus are created and defined using a resource script, which is a text file with a .RC extension. The resource script specifies the structure, content, and properties of each menu item.

Menu Handling

When a user selects a menu item, the application receives a WM_COMMAND message from Windows. The message contains the ID of the selected menu item, which the application can use to identify the corresponding command or action.

Keyboard Accelerators

Keyboard accelerators are key combinations that can be used to quickly activate menu items. They are typically defined in the resource script and consist of a combination of the Alt key and a letter or number.

Benefits of Menus

Menus offer several benefits for Windows applications:

- User Interface Consistency: Menus provide a consistent way for users to interact with applications, making it easier to learn and use different programs.
- Organized Presentation: Menus allow for a structured presentation of options, preventing clutter and making it easier for users to find the desired commands.
- Accessibility: Menus are accessible to users who rely on keyboard input, providing an alternative to mouse interaction.
- Context-Sensitivity: Context-sensitive popup menus provide relevant options based on the user's current context, improving usability.

Defining Menus in Windows Programming

Menus are a crucial component of Windows applications, providing users with a structured and organized way to interact with the program. To create menus, developers utilize resource scripts, which are text files with the .RC extension. These scripts define the menu's structure, content, and properties of each menu item.

Creating Menu Items

Each menu item is defined by three essential characteristics:

- Text String: The text displayed to the user, typically representing the action or option associated with the menu item.
- Command ID: A unique identifier assigned to the menu item. When the user selects the item, Windows sends a WM_COMMAND message to the application, containing this ID.
- Attributes: Properties that determine the appearance and behavior of the menu item, such as enabled, disabled, grayed, or checked.

Menu Item Properties Dialog Box

Developer Studio provides a Menu Items Properties dialog box that allows you to configure the properties of each menu item. This dialog box includes options for:

- Pop-up: If checked, the menu item invokes a submenu.
- Command ID: The identifier associated with the menu item.
- Grayed: If selected, the menu item is inactive and its text appears grayed.
- Inactive: If selected, the menu item is inactive but its text appears normally.
- Checked: If selected, a check mark is displayed next to the menu item.
- Separator: If selected, a horizontal separator bar is drawn on popup menus.

Keyboard Accelerators

Keyboard accelerators are key combinations that can be used to quickly activate menu items. They are typically defined in the resource script and consist of a combination of the Alt key and a letter or number. For instance, "Alt+F" might be the accelerator for the "File" menu.

Menu Item Text Formatting

To enhance menu readability, two special characters can be used in the menu item text:

- Ampersand (&): Places an underline beneath the following character, indicating the Alt key shortcut for that menu item.
- Columnar Tab (\t): In popup menus, it aligns text in two columns, with the longest text string in the first column determining the width.

Specifying the Menu in the Program

To associate the menu with the window, you can either:

- Window Class: Specify the menu name in the window class structure. This is the most common approach.
- LoadMenu Function: Load the menu resource into memory using LoadMenu and pass the returned handle to CreateWindow.
- SetMenu Function: Assign the menu to a window after it has been created.

Destroying Menus

When a window is destroyed, any attached menus are also destroyed. However, menus not attached to windows should be explicitly destroyed using DestroyMenu before the program terminates.

Example Code

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

```
MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Open", IDM_OPEN
        MENUITEM "&Save", IDM_SAVE
        MENUITEM "Save &As", IDM_SAVEAS
        SEPARATOR
        MENUITEM "E&xit", IDM_EXIT
    END
END
```

This code defines a main menu with a "File" popup menu containing options for opening, saving, saving as, and exiting. Each menu item has a unique identifier and is associated with a command ID.

In this version:

- The ampersand (&) before a letter in the menu items indicates an accelerator key, which is activated by pressing Alt along with the underlined letter.
- The use of double ampersands (&&) in "Save &As" is to display a single ampersand in the menu item.
- The mnemonic keys are set for quick keyboard navigation.
- The SEPARATOR adds a horizontal line to visually separate menu items.
- "Exit" has the mnemonic key 'x' underlined and is activated by pressing Alt + F, X

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 the message loop.

The menu is defined using the MENU macro, which takes a list of menu items.

Each menu item is defined using the MENUITEM macro, which takes the text of the menu item and an ID number.

The window class is registered using the RegisterClassEx function, which takes a pointer to a WNDCLASSEX structure.

The window is created using the CreateWindow function, which takes the name of the window class, the title of the window, the window style, the x and y coordinates of the window, the width and height of the window, a handle to the parent window, a handle to the menu, the instance handle, and a pointer to any additional data.

The message loop is run using the GetMessage and DispatchMessage functions.

GetMessage retrieves a message from the message queue and stores it in the MSG structure.

DispatchMessage sends the message to the window procedure for the window that created the message. The window procedure is responsible for processing the message.

Here are some additional explanations of the code:

- The MENU macro is used to define a menu.
- The POPUP macro is used to define a popup menu.
- The MENUITEM macro is used to define a menu item.
- The IDM_OPEN, IDM_SAVE, IDM_SAVEAS, and IDM_EXIT macros are used to define the ID numbers for the menu items.
- The WNDCLASSEX structure is used to define a window class.
- The RegisterClassEx function is used to register a window class.
- The CreateWindow function is used to create a window.
- The GetMessage function is used to retrieve a message from the message queue.
- The DispatchMessage function is used to send a message to the window procedure.

MENU-RELATED MESSAGES

Windows communicates with an application's window procedure using messages. Among these messages, several are specifically related to menus and provide information about the user's interactions with the menu. Here's a detailed breakdown of these messages:

WM_INITMENU

This message is sent to the window procedure before the menu is displayed. The message parameters are:

- wParam: Handle to the main menu.
- lParam: Always 0.

The purpose of this message is to allow the application to modify the menu before it is displayed to the user. However, it is generally recommended to avoid making significant changes to the top-level menu at this point, as it might confuse the user.

WM_MENUSELECT

This message is sent repeatedly as the user moves the cursor or mouse among the menu items. It provides information about the currently selected menu item. The message parameters are:

wParam:

1. LOWORD: Selected item ID or popup menu index.
2. HIWORD: Selection flags.

lParam: Handle to the menu containing the selected item.

The selection flags in the high word of wParam indicate various properties of the selected menu item, such as whether it is grayed, disabled, checked, or a popup menu. This message is useful for implementing features like status bars that display a description of the highlighted menu option.

WM_INITMENUPOPUP

This message is sent when Windows is about to display a popup menu. It provides information about the popup menu and its context. The message parameters are:

- wParam: Handle to the popup menu.
- lParam:
    1. LOWORD: Popup menu index.
    2. HIWORD: 1 for system menu, 0 otherwise.

The LOWORD of lParam indicates the index of the popup menu within the parent menu.

The HIWORD indicates whether the popup menu is part of the system menu or a regular popup menu. This message is useful for enabling or disabling items in a popup menu based on the current context, such as the availability of data in the clipboard for the Paste command.

This message is useful for enabling or disabling items in a popup menu based on the current context. For instance, you can disable the Paste command if the clipboard is empty.

WM_COMMAND

This message indicates that the user has selected an enabled menu item. It is the most important menu-related message. The message parameters are:

wParam:

1. LOWORD: Menu ID or control ID (for child window controls).
2. HIWORD: Notification code (0 for menu items).

lParam:

1. For menu items: 0.
2. For child window controls: Child window handle.

WM_SYSCOMMAND

This message is similar to WM_COMMAND but indicates that the user has selected an item from the system menu. The message parameters are:

- wParam: Menu ID.
- lParam: 0.

For predefined system menu items, the low word of wParam contains the masked menu ID. For added menu items, it contains the user-defined menu ID.

WM_MENUCHAR

This message is sent when the user presses Alt and a character key that does not correspond to a menu item or when a character key is pressed in a displayed popup menu that does not correspond to a menu item. The message parameters are:

wParam:

1. LOWORD: Character code (ASCII or Unicode).
2. HIWORD: Selection code (0 for no popup, MF_POPUP for popup, MF_SYSMENU for system menu popup).

lParam: Handle to the menu.

This message is typically used to display a help message or perform a custom action based on the pressed character.

Handling Menu-Related Messages

Most applications simply pass these menu-related messages to the default window procedure (DefWindowProc) for handling.

DefWindowProc provides the basic functionality for processing menu interactions, such as highlighting the selected item, displaying popup menus, and generating WM_COMMAND messages when a menu item is selected.

However, if you need to implement custom behavior or dynamic menu changes, you can override the handling of these messages in your application's window procedure.

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.

```
610   case WM_INITMENUPOPUP: // Handle WM_INITMENUPOPUP message
611
612     if (HIWORD(wParam) == 1) { // Check if it's a popup menu
613
614       if (LOWORD(wParam) == IDR_POPUP_EDIT) { // Check if it's the Edit popup menu
615
616         // Check if the clipboard contains text using IsClipboardFormatAvailable() function
617         if (IsClipboardFormatAvailable(CF_TEXT)) {
618
619           // Enable the Paste menu item using EnableMenuItem() function
620           EnableMenuItem(GetMenu(hwnd), IDM_EDIT_PASTE, MF_ENABLED);
621
622         } else { // If no text is available in the clipboard
623
624           // Disable the Paste menu item using EnableMenuItem() function
625           // Set both MF_DISABLED and MF_GRAYED flags to disable and gray out the menu item
626           EnableMenuItem(GetMenu(hwnd), IDM_EDIT_PASTE, MF_DISABLED | MF_GRAYED);
627
628         }
629       }
630     }
631     break;
```

WM_INITMENUPOPUP Message Handling:

The code snippet begins by handling the WM_INITMENUPOPUP message using a case statement. This message indicates that a popup menu is about to be displayed.

Checking Popup Menu Type:

Inside the case block, the code checks if the popup menu is indeed a popup menu by examining the HIWORD of the wParam parameter. If HIWORD(wParam) is 1, it confirms that the message is for a popup menu.

Identifying the Edit Popup Menu:

Next, the code checks if the popup menu is specifically the Edit popup menu (IDR_POPUP_EDIT) by examining the LOWORD of the wParam parameter. If LOWORD(wParam) is IDR_POPUP_EDIT, it indicates that the current message is for the Edit popup menu.

Checking Clipboard Content:

If the popup menu is the Edit popup menu, the code checks whether the clipboard contains text using the IsClipboardFormatAvailable() function. If IsClipboardFormatAvailable(CF_TEXT) returns TRUE, it means the clipboard contains text.

Enabling Paste Menu Item:

If the clipboard contains text, the code enables the Paste menu item (IDM_EDIT_PASTE) using the EnableMenuItem() function. This function sets the menu item to an enabled state, allowing users to select it.

Disabling Paste Menu Item:

If the clipboard does not contain text, the code disables the Paste menu item (IDM_EDIT_PASTE) using the EnableMenuItem() function. Additionally, it sets the MF_DISABLED and MF_GRAYED flags to both disable the menu item and make it appear grayed out, indicating that the option is currently unavailable.

The MENUDEMO program demonstrates the basic functionality of menu handling in Windows applications. It creates a simple window with five main menu options: File, Edit, Background, Timer, and Help.

Each of these options has a corresponding popup menu with additional options. The program handles user interactions with the menus by processing WM_COMMAND messages and modifying the menu items accordingly.

The MENUDEMO program consists of two main source files: MENUDEMO.C and MENUDEMO.RC.

MENUDEMO.C contains the program code that defines the window procedure, handles messages, and implements the menu logic. MENUDEMO.RC defines the resources for the program, including the menu structure.

Window Procedure

The window procedure (WndProc) is responsible for handling messages sent to the program's window.

It processes various messages, including WM_CREATE, WM_COMMAND, WM_TIMER, and WM_DESTROY. The most relevant part of the window procedure is the handling of WM_COMMAND messages, which are generated when users select menu items.

Menu Handling

The MENUDEMO program handles menu interactions by extracting the menu ID from the wParam parameter of WM_COMMAND messages. Based on the menu ID, it performs different actions:

File Menu: For File menu items, it displays a message box indicating that the feature is not yet implemented.

Edit Menu: For Edit menu items, it displays a message box indicating that the feature is not yet implemented.

Background Menu: For Background menu items, it updates the window's background color based on the selected option. It also checks and unchecks the corresponding menu items to reflect the current selection.

Timer Menu: For Timer menu items, it starts or stops a timer using SetTimer() and KillTimer() functions, respectively. It also enables and disables the Start/Stop menu items accordingly.

Help Menu: For Help menu items, it displays message boxes with appropriate help or about information.

Timer Handling

The MENUDEMO program uses a timer to generate periodic beeps. It starts the timer when the Start menu item is selected and stops it when the Stop menu item is selected. The timer message handler (WM_TIMER) simply generates a beep using MessageBeep(0).

Code snippets:

This code snippet removes the checkmark from the previously selected background color and adds a checkmark to the newly selected color.

```
CheckMenuItem (hMenu, iSelection, MF_UNCHECKED) ;
iSelection = wParam ;
CheckMenuItem (hMenu, iSelection, MF_CHECKED) ;
```

This code snippet grays out the Start option and enables the Stop option when the timer is started.

```
EnableMenuItem (hMenu, IDM_TIMER_START, MF_ENABLED) ;
EnableMenuItem (hMenu, IDM_TIMER_STOP, MF_GRAYED) ;
```

This code snippet enables the Start option and grays out the Stop option when the timer is stopped.

```
EnableMenuItem (hMenu, IDM_TIMER_START, MF_ENABLED) ;
EnableMenuItem (hMenu, IDM_TIMER_STOP, MF_GRAYED) ;
```

RESOURCE FILES

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

MENUDEMO.RC:

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

MENUDEMO MENU DISCARDABLE: This line declares the start of the menu definition. The keyword DISCARDABLE indicates that the resources are not required for the program to function, but they are used to provide a user interface.

BEGIN: This block marks the beginning of the menu definition. It encloses the various menu items and popups that make up the menu structure.

POPUP "&File": This line defines a popup menu with the label "&File". The ampersand (&) before the letter "F" indicates that the letter "F" should be the hotkey for the menu.

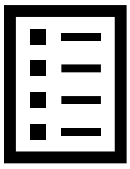
BEGIN: This block marks the beginning of the "&File" popup menu. It encloses the individual menu items within this popup.

MENUITEM "&New", IDM_FILE_NEW: This line defines a menu item with the label "&New" and associates it with the IDM_FILE_NEW identifier.

MENUITEM "&Open", IDM_FILE_OPEN: This line defines a menu item with the label "&Open" and associates it with the IDM_FILE_OPEN identifier. Similarly, subsequent lines define menu items for "&Save", "Save &As...", and "E&xit", each with their respective IDs.

SEPARATOR: This line inserts a separator line in the popup menu, providing visual separation between menu items.

END: This block marks the end of the "&File" popup menu.



POPUP "&Edit": Similar to the "&File" popup menu, this line defines a popup menu with the label "&Edit" and encloses its individual menu items.

POPUP "&Background": This line defines a popup menu with the label "&Background" and encloses its individual menu items.

MENUITEM "&White", IDM_BKGND_WHITE, CHECKED: This line defines a menu item with the label "&White", associates it with the IDM_BKGND_WHITE identifier, and sets the initial checked state to true.

MENUITEM "&Light Gray", IDM_BKGND_LTGRAY: Similar to the previous line, this line defines menu items for "Light Gray", "Gray", "Dark Gray", and "Black", each with their respective IDs.

END: This block marks the end of the "&Background" popup menu.

POPUP "&Timer": Similar to the previous popups, this line defines a popup menu with the label "&Timer" and encloses its individual menu items.

MENUITEM "&Start", IDM_TIMER_START: This line defines a menu item with the label "&Start" and associates it with the IDM_TIMER_START identifier.

MENUITEM "S&top", IDM_TIMER_STOP, GRAYED: This line defines a menu item with the label "S&top", associates it with the IDM_TIMER_STOP identifier, and sets the initial state to grayed out, indicating it is initially disabled.

END: This block marks the end of the "&Timer" popup menu.

POPUP "&Help": Similar to the previous popups, this line defines a popup menu with the label "&Help" and encloses its individual menu items.

END: This block marks the end of the menu definition.

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: This line defines a constant identifier named IDM_FILE_NEW and assigns it the value 40001. This constant is used to identify the "&New" menu item.

Similarly, subsequent lines define constant identifiers for all the menu items and other resources used in the program, each with their respective numeric values.

These code excerpts provide a detailed explanation of the menu structure and resource identifiers used in the MENUDEMO program.

Menu Design Considerations

The design of menus in Windows applications should follow established conventions to provide a consistent and familiar user experience.

File and Edit Menu Consistency: The File and Edit menus should adhere to standardized formats and hotkey combinations (Alt + F, Alt + E) to maintain user familiarity across different programs.

Unique Menus for Specific Programs: Menus beyond File and Edit can vary depending on the program's functionality. However, it's recommended to follow common design patterns to minimize user confusion.

Flexibility in Menu Design: Menu revisions primarily involve modifying the resource script, leaving the program code largely unaffected. This allows for easy menu updates without significant code changes.

MENUITEM Statements on Top Level

While MENUITEM statements can be used at the top level of a menu, it's generally discouraged due to the increased risk of accidental selection. If top-level menu items are necessary, consider using an exclamation point (!) after the text string to indicate that it's not a popup menu item.

Defining a Menu Using CreateMenu and AppendMenu

An alternative approach to menu creation involves using the CreateMenu and AppendMenu functions directly within the program code. This method provides more flexibility but requires manual menu construction.

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

Menu Creation Using CreateMenu and AppendMenu

The provided code in Figure 10-7 demonstrates the manual creation of a menu using the CreateMenu and AppendMenu functions. While it serves as an alternative to resource scripts, it's generally not recommended due to its complexity and verbosity.

Step-by-Step Breakdown:

CreateMenu: The code initializes the menu structure by creating two top-level menus (hMenu and hMenuPopup) and additional popup menus.

AppendMenu: Individual menu items are added to the respective menus using AppendMenu. Each item includes its identifier, text label, and flags (MF_POPUP for popups, MF_STRING for normal items, MF_CHECKED for initially checked items, and MF_GRAYED for disabled items).

SetMenu: Finally, the main menu (hMenu) is assigned to the window using SetMenu.

Alternative Menu Creation Methods

Array of MENUITEMTEMPLATE Structures: To reduce code repetition, one could use an array of MENUITEMTEMPLATE structures, each containing the menu item's information. This approach provides better organization and reduces code size.

LoadMenuIndirect: The LoadMenuIndirect function allows loading a menu from a memory buffer containing a MENUITEMTEMPLATE structure. This method is similar to resource script loading but provides more flexibility.

Comparison of Methods

Resource Scripts: Resource scripts offer a straightforward and visual approach to menu definition, making them the preferred method.

CreateMenu and AppendMenu: Manual menu creation using CreateMenu and AppendMenu is more complex and error-prone but provides greater control over the menu structure.

LoadMenuIndirect: LoadMenuIndirect offers a balance between resource scripts and manual creation, allowing menu definition from memory.

Recommendation

Resource scripts remain the recommended approach for menu creation due to their ease of use, visual representation, and integration with the development environment.

Manual methods like CreateMenu and AppendMenu or LoadMenuIndirect are generally reserved for advanced scenarios or specific requirements.

There is a sample application code in chapter 10 folder called MenuCustomCode.c.

Run it to see some menu. Ensure you have the libraries libwinmm and libgdi32 configured properly as we said in chapter 1/2/3/4, somewhere in these chapters…

The code has two codes within it, for unicode and all that, so cross-check it first.

This modification explicitly sets hIconSm only when UNICODE is defined. Please check your project settings to ensure that the character set is consistent with your code.

If you are working with Unicode, make sure your project settings are configured for Unicode. If you are working with ANSI, remove the #ifdef UNICODE and #endif lines.

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 is a demonstration of creating a popup menu in a Windows application without a top-level menu bar. Let's break down the code and explore its functionality in detail.

The program starts by defining a window class, registering it, and creating the main application window. The WndProc function serves as the window procedure handling various messages.

In the WM_CREATE message case, the program loads the menu resource defined in the resource file associated with the application. It then extracts the submenu from the loaded menu using GetSubMenu. This submenu will be the popup menu displayed when the user right-clicks on the window.

The WM_RBUTTONUP message case responds to a right-click event. It retrieves the mouse coordinates, converts them to screen coordinates, and uses TrackPopupMenu to display the popup menu at the specified location.

The WM_COMMAND message case handles menu item selections. It checks which menu item was selected and takes appropriate actions. Notably, it changes the background color of the window based on the selected color option. This is achieved by updating the window's background brush using SetClassLong and triggering a redraw with InvalidateRect.

The program includes options for handling file operations (New, Open, Save, etc.) and a basic "About" dialog. The "Help" option displays a message indicating that help functionality is not yet implemented.

Finally, the WM_DESTROY message case handles the termination of the program when the window is closed.

RESOURCE FILE:

POPMENU MENU DISCARDABLE

This line defines the start of the menu definition. The keyword DISCARDABLE indicates that the resources are not required for the program to function, but they are used to provide a user interface.

BEGIN

This block marks the beginning of the menu definition. It encloses the various menu items and popups that make up the menu structure.

POPUP "MyMenu"

This line defines a top-level popup menu with the label "MyMenu".

BEGIN

This block marks the beginning of the "MyMenu" popup menu. It encloses the individual popups within this top-level menu.

POPUP "&File"

Similar to the "MyMenu" popup menu, this line defines a popup menu with the label "&File" and encloses its individual menu items.

MENUITEM "&New", IDM_FILE_NEW

This line defines a menu item with the label "&New" and associates it with the **IDM_FILE_NEW identifier.**

Similarly, subsequent lines define menu items for "&Open", "&Save", "Save &As...", and "E&xit", each with their respective IDs.

SEPARATOR

This line inserts a separator line in the popup menu, providing visual separation between menu items.

POPUP "&Edit"

Similar to the "&File" popup menu, this line defines a popup menu with the label "&Edit" and encloses its individual menu items.

POPUP "&Background"

Similar to the previous popups, this line defines a popup menu with the label "&Background" and encloses its individual menu items.

MENUITEM "&White", IDM_BKGND_WHITE, CHECKED

This line defines a menu item with the label "&White", associates it with the IDM_BKGND_WHITE identifier, and sets the initial checked state to true.

MENUITEM "&Light Gray", IDM_BKGND_LTGRAY

Similar to the previous line, this line defines menu items for "Light Gray", "Gray", "Dark Gray", and "Black", each with their respective IDs.

POPUP "&Help"

Similar to the previous popups, this line defines a popup menu with the label "&Help" and encloses its individual menu items.

END

This block marks the end of the "MyMenu" popup menu.

END

This block marks the end of the menu definition.

The POPMENU.RC resource file defines a hierarchical menu structure with multiple popup menus, each containing menu items. The "&File", "&Edit", "&Background", and "&Help" popups provide options for common tasks, while the "MyMenu" top-level menu serves as an organizational element.

In summary, the POPMENU program showcases the creation of a floating popup menu that appears in response to a right-click. It dynamically changes the background color of the window based on the user's selection and includes basic menu options for file operations and information about the program.

Example 2:

The code you provided is attempting to create a menu using the LoadMenuIndirect function by specifying an array of MENUITEMTEMPLATE structures. However, there is a mistake in the code. The MENUITEMTEMPLATE structure is not a predefined structure in the Windows API.

To create a menu using LoadMenuIndirect, you need to use the MENUITEMTEMPLATEHEADER structure, which is a part of the menu template. Here is a corrected version of your code:

```
#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 differs from MENUDEMO in its menu structure. Instead of individual menu items directly under the top-level menu, it has a single popup menu named "MyMenu" that contains the File, Edit, Background, and Help options. This creates a vertical menu organization instead of a horizontal one.

Menu Handle Acquisition

In the WM_CREATE message handler, POPMENU obtains the handle to the "MyMenu" popup menu using a two-step process:

LoadMenu: It first loads the menu resource using LoadMenu, retrieving the top-level menu handle.

GetSubMenu: Since the "MyMenu" popup is the first item in the top-level menu, POPMENU retrieves its handle using GetSubMenu with the index 0.

Menu Handle Acquisition in WM_CREATE:

```
hMenu = LoadMenu(hInst, szAppName);
hMenu = GetSubMenu(hMenu, 0);
```

- LoadMenu: This line loads the menu resource associated with the application name (szAppName) and returns the top-level menu handle.
- GetSubMenu: Since the "MyMenu" popup is the first item in the top-level menu, GetSubMenu retrieves its handle using the index 0.

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);
```

Mouse Position Retrieval: These lines extract the mouse position (x and y coordinates) from the lParam parameter of the WM_RBUTTONUP message.

Coordinate Conversion: The ClientToScreen function converts the mouse position from client coordinates (relative to the window) to screen coordinates (absolute on the screen).

TrackPopupMenu Invocation: This line invokes the TrackPopupMenu function, passing the following parameters:

- hMenu: The handle to the "MyMenu" popup menu
- TPM_RIGHTBUTTON: Flag indicating a right-click context menu
- point.x: X-coordinate of the mouse position
- point.y: Y-coordinate of the mouse position
- 0: Reserved parameter
- hwnd: The window handle
- NULL: Pointer to a menu event handler (not used in this case)

Popup Menu Display: As a result of calling TrackPopupMenu, Windows displays the "MyMenu" popup menu at the specified location, with the File, Edit, Background, and Help options arranged vertically.

Nested Menu Behavior: Selecting any of these options triggers the corresponding nested popup menus to appear to the right, maintaining the vertical menu structure.

Main Menu and TrackPopupMenu Compatibility

Using the same menu for both the main menu and TrackPopupMenu poses a challenge due to the requirement of a popup menu handle for TrackPopupMenu. The Microsoft Knowledge Base article ID Q99806 provides workarounds to address this issue.

Modifying the System Menu

Parent windows with the WS_SYSMENU style feature a system menu box in the caption bar's left corner.

This menu can be modified by adding custom menu commands. While less common today, modifying the system menu provides a quick and straightforward way to add a menu to simple programs without defining it in a resource script.

ID Number Restriction

When adding commands to the system menu, the ID numbers assigned must be lower than 0xF000 to avoid conflicts with the IDs reserved for standard system menu commands.

WM_SYSCOMMAND Message Handling

In the window procedure, when processing WM_SYSCOMMAND messages for these custom menu items, it's crucial to pass all other WM_SYSCOMMAND messages to DefWindowProc. Failure to do so effectively disables the standard options on the system menu.

POORMENU Example

POORMENU, or "Poor Person's Menu," demonstrates adding a separator bar and three commands to the system menu. The last command removes these additions.

Implementation Details

WM_CREATE Message: In WM_CREATE, the program retrieves the system menu using GetSystemMenu(hWnd, FALSE) and adds the desired menu items using AppendMenu.

WM_SYSCOMMAND Message: In WM_SYSCOMMAND, the program checks the wParam value to determine the selected menu item and performs the corresponding action. For the custom menu items, it handles the actions itself. For other WM_SYSCOMMAND messages, it calls DefWindowProc to maintain standard menu functionality.

Removing Additions: The "Remove Additions" menu item triggers the removal of the custom menu items using DeleteMenu.

Advantages of System Menu Modification

Simplicity: Modifying the system menu is a straightforward approach for adding menus to simple programs.

No Resource Script Dependency: It eliminates the need to define the menu in a resource script, reducing the code complexity.

Disadvantages of System Menu Modification

Restricted ID Numbers: The ID numbers for custom menu items must be lower than 0xF000 to avoid conflicts.

Handling WM_SYSCOMMAND Messages: Proper handling of WM_SYSCOMMAND messages is essential to maintain standard system menu behavior.

POORMENU.C - THE POOR PERSON'S MENU

This program, titled "Poor Person's Menu," demonstrates how to modify the system menu of a window to add custom menu items. It provides a simple and straightforward approach to adding menus without defining them in a resource script.

Initialization

Header File Inclusion: The code includes the necessary Windows header file (windows.h) to access Windows functions and data structures.

Constant Definitions: Three constants are defined to represent the ID numbers for the custom menu items: IDM_SYS_ABOUT for the "About..." menu item, IDM_SYS_HELP for the "Help..." menu item, and IDM_SYS_REMOVE for the "Remove Additions" menu item.

Window Procedure Declaration: The WndProc function is declared as the window procedure for the main window.

Global Variable Declaration: A global variable, szAppName, stores the application's name as a string.

WinMain Function

Window Class Registration: The WinMain function begins by registering the window class using the RegisterClass function. It sets the window class style to include CS_HREDRAW and CS_VREDRAW for redrawing on resize, and assigns the WndProc function as the window procedure.

Window Creation: The CreateWindow function is called to create the main window of the application. It specifies the window class name, window title, window style, window position, and window instance handle.

System Menu Modification: The GetSystemMenu function retrieves the system menu of the main window. The AppendMenu function is then used to add four menu items to the system menu: a separator, "About...", "Help...", and "Remove Additions".

Window Display: The ShowWindow and UpdateWindow functions are used to display the main window and update its contents.

Message Loop: The main message loop is entered using the GetMessage function. It retrieves messages from the message queue and dispatches them to the appropriate window procedures, including WndProc.

Exit Message Processing: When a WM_QUIT message is received, the PostQuitMessage function is called to post a quit message to the message queue, causing the main message loop to terminate.

WndProc Function

WM_SYSCOMMAND Message Handling: The WndProc function handles WM_SYSCOMMAND messages, which are sent when a menu item is selected from the system menu. It checks the LOWORD of wParam to determine the selected menu item ID.

IDM_SYS_ABOUT: For "About...", it displays a message box with information about the program.

IDM_SYS_HELP: For "Help...", it displays a message box indicating that help is not yet implemented.

IDM_SYS_REMOVE: For "Remove Additions", it restores the original system menu using GetSystemMenu with TRUE as the second parameter.

WM_DESTROY Message Handling: The WndProc function handles WM_DESTROY messages, which are sent when the window is destroyed. It posts a WM_QUIT message to the message queue to trigger the main message loop's termination.

DefWindowProc: For all other messages, the WndProc function calls DefWindowProc to ensure standard window behavior.

Menu ID Definitions

At the beginning of POORMENU.C, three constants are defined: IDM_ABOUT, IDM_HELP, and IDM_REMOVE. These constants represent the ID numbers for the custom menu items that will be added to the system menu.

Retrieving the System Menu Handle

Once the program's window has been created, the GetSystemMenu function is called to obtain a handle to the system menu. The second parameter of GetSystemMenu is set to FALSE, indicating that the program intends to modify the menu.

Modifying the System Menu

The system menu is modified using four calls to the AppendMenu function. These calls add the following items to the menu:

A separator bar: This bar visually separates the custom menu items from the standard system menu items.

"About...": This menu item, identified by IDM_ABOUT, triggers the display of an "About" message box when selected.

"Help...": This menu item, identified by IDM_HELP, triggers the display of a "Help" message box when selected.

"Remove Additions": This menu item, identified by IDM_REMOVE, removes the custom menu items that were added earlier. When selected, it simply calls GetSystemMenu again with the second parameter set to TRUE, restoring the original system menu.

Handling WM_SYSCOMMAND Messages

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

- SC_RESTORE: Restore the window to its normal size and position
- SC_MOVE: Move the window
- SC_SIZE: Resize the window
- SC_MINIMUM: Minimize the window
- SC_MAXIMUM: Maximize the window
- SC_CLOSE: Close the window

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.

While Windows programs typically pass these messages to DefWindowProc for default handling, you can also process them yourself. This allows you to customize the behavior of the standard menu items or disable or remove them entirely.

Standard Additions to the System Menu

The Windows documentation describes additional standard menu items that can be added to the system menu:

- SC_NEXTWINDOW: Switch to the next window in the taskbar
- SC_PREVWINDOW: Switch to the previous window in the taskbar
- SC_VSCROLL: Scroll the window vertically
- SC_HSCROLL: Scroll the window horizontally
- SC_ARRANGE: Arrange the windows on the desktop

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.

Adding these menu items can be appropriate for specific applications.

Conclusion

The POORMENU program demonstrates a simple and effective method for adding custom menu items to a window by modifying the system menu.

It utilizes the GetSystemMenu and AppendMenu functions to add the desired items and handles WM_SYSCOMMAND messages to provide custom behavior for the menu items.

This approach is particularly useful for small programs that do not require elaborate menu structures or resource scripts.

Menu Modification Options

The POORMENU program demonstrated adding menu items to the system menu using AppendMenu. However, prior to Windows 3.0, the ChangeMenu function was the primary tool for modifying menus. While ChangeMenu was versatile, it was also complex. Modern Windows provides a set of specialized functions for modifying menus:

- AppendMenu: Appends a new menu item to the end of an existing menu.
- DeleteMenu: Deletes an existing menu item from a menu and destroys the item's resources.
- InsertMenu: Inserts a new menu item into a specific position within an existing menu.
- ModifyMenu: Modifies the attributes (text, flags, data) of an existing menu item.
- RemoveMenu: Removes an existing menu item from a menu.

DeleteMenu vs. RemoveMenu for Popup Menus

The distinction between DeleteMenu and RemoveMenu is crucial for dealing with popup menus. While both functions remove the menu item, DeleteMenu also destroys the associated popup menu and its resources, whereas RemoveMenu retains the popup menu. This behavior is important when managing popup menus and their associated 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

As demonstrated in MENUDEMO, the CheckMenuItem function can be used to check or uncheck an item in a popup menu:

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

In this case, hMenu can be the handle to either a top-level menu or a popup menu. The id parameter is the menu ID for the item, and the iCheck parameter specifies whether to check (MF_CHECKED) or uncheck (MF_UNCHECKED) the item.

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 demonstrates an unconventional approach to menu creation by using multiple top-level menus without drop-down options.

Instead of traditional nested menus, the program switches between these top-level menus using the SetMenu function, emulating the menu style found in the character-mode application Lotus 1-2-3.

Main Function

The main function (WinMain) initializes the application by defining the window class, registering the window class, creating the window, and displaying it. It also handles the message loop that processes user interactions.

Window Procedure

The window procedure (WndProc) handles various window messages, including WM_CREATE, WM_COMMAND, and WM_DESTROY.

WM_CREATE: Upon window creation, it loads the three top-level menus: hMenuMain, hMenuFile, and hMenuEdit. It then sets the initial menu to hMenuMain using the SetMenu function.

WM_COMMAND: It handles menu selections by switching between the top-level menus based on the selected option (IDM_MAIN, IDM_FILE, or IDM_EDIT). For individual menu commands (IDM_FILE_NEW, IDM_FILE_OPEN, etc.), it produces a beep to indicate menu selection.

WM_DESTROY: Before destroying the window, it restores the main menu (hMenuMain) and destroys the unused menus (hMenuFile and hMenuEdit) to release resources.

Menu Switching Mechanism

The NOPOPUPS program achieves menu switching by dynamically changing the window's menu using the SetMenu function.

When a menu option is selected, the corresponding top-level menu is assigned to the window, effectively switching the visible menu. This approach provides a straightforward way to manage multiple top-level menus without the complexity of nested menus.

Menu Resource Creation

Instead of creating a single menu with nested menu items, the NOPOPUPS program utilizes three separate menu resources: MenuMain, MenuFile, and MenuEdit.

These resources are defined in the NOPOPUPS.RC file using the MENU keyword. Each menu defines its menu items using the MENUITEM keyword, specifying the item's text and identifier (IDM_FILE, IDM_EDIT, etc.).

Menu Loading and Handling

In the window procedure (WndProc), upon receiving the WM_CREATE message, 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 resource.

Dynamic Menu Switching

The program achieves menu switching by dynamically assigning a menu handle to the window using the SetMenu function. When a menu option is selected (IDM_MAIN, IDM_FILE, or IDM_EDIT), the corresponding menu handle is assigned to the window, effectively changing the visible menu:

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

0;
case IDM_FILE:
    SetMenu(hwnd, hMenuFile);
    return

0;
case IDM_EDIT:
    SetMenu(hwnd, hMenuEdit);
    return 0;
```

This approach allows the program to switch between the three defined menus based on user interaction.

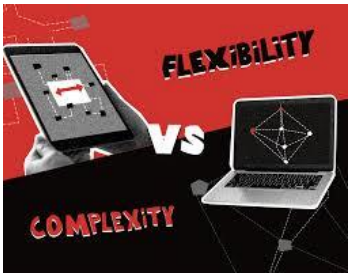
Menu Resource Advantage

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

Clarity: Each menu's structure and contents are clearly defined within its own resource, making it easier to understand and manage the menu system.

Isolation: Changes to one menu do not affect the others, minimizing the risk of unintended consequences.

Flexibility: Each menu can be customized independently, allowing for different menu layouts or styles.



Initial Menu Setup

The program initially displays the main menu using the SetMenu function:

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

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

Submenu Identification

The File and Edit menus begin with "FILE:" and "EDIT:", respectively. These labels serve as visual cues that these options represent submenus. This distinction is important for the program's logic and user interaction.

"Return to Main Menu" Option

The last item in each menu is the character string "(Main)". When this option is selected, it triggers a return to the main menu. This functionality is handled by the program's logic based on the menu ID associated with the "(Main)" option.

Menu Switching Mechanism

The program handles menu switching within the WM_COMMAND message handler:

```
switch (wParam)
{
case IDM_MAIN:
    SetMenu(hwnd, hMenuMain);
    return

0;
case IDM_FILE:
    SetMenu(hwnd, hMenuFile);
    return

0;
case IDM_EDIT:
    SetMenu(hwnd, hMenuEdit);
    return 0;
}
```

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. This mechanism allows for dynamic menu switching based on user interaction.

Menu Destruction

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

```
SetMenu(hwnd, hMenuMain);
DestroyMenu(hMenuFile);
DestroyMenu(hMenuEdit);
```

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

The provided code demonstrates the NOPOPUPS program's approach to menu management. It utilizes a combination of separate menu resources, dynamic menu switching, and explicit menu destruction to create a unique menu system that deviates from the traditional nested menu structure.