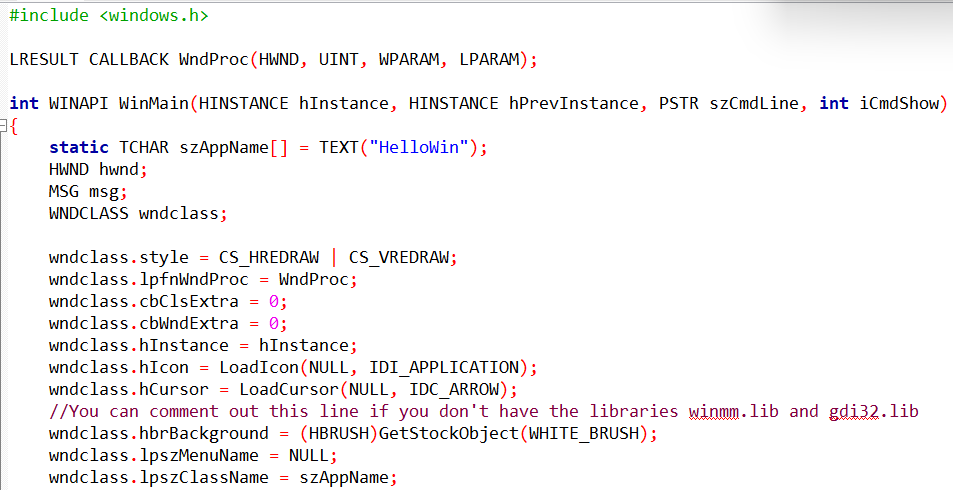
HELLOWIN.C IN DEPTH

**Defining and Initializing the WNDCLASS Structure**

The WNDCLASS structure defines the characteristics of a window class, which is a template for creating windows.

The WNDCLASS structure contains ten fields that specify various aspects of the window's appearance and behavior.

In the provided code, the WNDCLASS structure is declared and initialized as follows:



This initialization sets the following values for the WNDCLASS structure's fields:

**style:** This field specifies the window's style flags. The CS\_HREDRAW and CS\_VREDRAW flags indicate that the window should be completely repainted whenever the horizontal or vertical window size changes, respectively.

**lpfnWndProc:** This field is a pointer to the window procedure function. The window procedure function is responsible for handling all window messages that are sent to windows of this class. In this case, the window procedure function is WndProc.

**cbClsExtra:** This field specifies the size of extra data that is associated with each window of this class. In this case, the value is set to 0, indicating that there is no extra data.

**cbWndExtra:** This field specifies the size of extra data that is associated with each individual window of this class. In this case, the value is set to 0, indicating that there is no extra data.

**hInstance:** This field is the handle of the application instance. The application instance is a unique identifier for the application.

**hIcon:** This field is the handle of the icon that will be used for the application. The icon is a small image that is displayed in the title bar of the window and in the taskbar.

**hCursor:** This field is the handle of the cursor that will be used for the application. The cursor is a small image that follows the mouse pointer.

**hbrBackground:** This field is the handle of the brush that will be used to paint the background of the window. The brush is a tool used to fill in an area with a solid color or pattern.

**lpszMenuName:** This field is the name of the menu resource that will be used for the application. The menu is a list of options that the user can select to control the application. In this case, the value is set to NULL, indicating that no menu will be used.

**lpszClassName:** This field is the name of the window class. The window class is a template for creating windows. The name of the window class is used to identify the class of a window when it is created.

**Registering the Window Class**

After initializing the WNDCLASS structure, it is registered with the Windows operating system using the RegisterClass function. The RegisterClass function takes a pointer to the WNDCLASS structure as its argument. Once the window class is registered, it can be used to create windows.

**Window Procedure**

The window procedure is a function that is responsible for handling all window messages that are sent to windows of a particular class. In this case, the window procedure is WndProc. The WndProc function is responsible for responding to the message in a way that is appropriate for the application. For example, if the message is a WM\_CREATE message, the WndProc function would create any child windows that the application needs.

**Setting the Window Procedure**

The second field of the WNDCLASS structure, lpfnWndProc, is set to the address of the window procedure function, WndProc.

This means that all windows created based on this window class will use the WndProc function to process messages.

The WndProc function is responsible for handling all messages sent to the window, such as a WM\_CREATE message when the window is created, a WM\_PAINT message when the window needs to be repainted, or a WM\_DESTROY message when the window is destroyed.

**Reserving Extra Space**

The next two fields of the WNDCLASS structure, cbClsExtra and cbWndExtra, are used to reserve extra space in the class structure and the window structure, respectively.

This extra space can be used by the application for its own purposes. The cbClsExtra field specifies the size of the extra space in the class structure, and the cbWndExtra field specifies the size of the extra space in the window structure.

In this case, both fields are set to 0, indicating that no extra space is needed.

**Setting the Instance Handle**

The next field of the WNDCLASS structure, hInstance, is set to the instance handle of the program. The instance handle is a unique identifier for the application instance.

The application instance is the running instance of the application, including all of its data and resources. The hInstance value is passed to the application's WinMain function when the application is started.

**Setting the Icon**

The statement wndclass.hIcon = LoadIcon(NULL, IDI\_APPLICATION); sets the icon for all windows created based on this window class.

The icon is a small bitmap image that represents the program to the user.

When the program is running, the icon appears in the Windows taskbar and at the left side of the program window's title bar.

The LoadIcon function is used to load a predefined icon.

The first argument to the LoadIcon function is set to NULL, indicating that a predefined icon is being loaded.

The second argument to the LoadIcon function is IDI\_APPLICATION, which is the identifier for the predefined icon that is a little picture of a window.

**Loading and Setting the Cursor**

The statement wndclass.hCursor = LoadCursor(NULL, IDC\_ARROW); loads a predefined mouse cursor known as IDC\_ARROW and assigns its handle to the hCursor field of the WNDCLASS structure.

This means that when the mouse cursor appears over the client area of a window created based on this class, the cursor will become a small arrow.

**Setting the Background Color**

The statement wndclass.hbrBackground = GetStockObject(WHITE\_BRUSH); sets the background color of the client area of windows created based on this class.

The GetStockObject function returns a handle to a white brush, which means that the background of the client area of the window will be solid white.

**Specifying the Window Class Menu**

The statement wndclass.lpszMenuName = NULL; indicates that the window class has no application menu. This means that the window will not have a menu bar at the top of the window.

**Setting the Window Class Name**

The statement wndclass.lpszClassName = szAppName; sets the name of the window class to the value stored in the szAppName variable.

This string can be either ASCII or Unicode characters depending on whether the UNICODE identifier has been defined.

**Registering the Window Class**

When all the fields of the WNDCLASS structure have been initialized, the RegisterClass function is called to register the window class.

The only argument to the function is a pointer to the WNDCLASS structure.

There are actually three versions of the RegisterClass function: RegisterClassA, RegisterClassW, and RegisterClassEx.

The specific function used determines whether messages sent to the window will contain ASCII text or Unicode text.

**Handling Unicode Compatibility Issues**

If you compile the program with the UNICODE identifier defined, your program will call RegisterClassW instead of RegisterClassA.

This is fine if you're running the program on Microsoft Windows NT, which has full support for Unicode.

However, if you're running the program on Windows 98, the RegisterClassW function is not fully implemented.

While there is an entry point for the function, it simply returns an error code indicating that the function is not available.

To handle this compatibility issue, the provided code snippet checks the return value of the RegisterClass function. If the function fails, it displays a message box informing the user that the program requires Windows NT and terminates the program.

The MessageBoxW function is used for this purpose because it is one of the few Unicode functions implemented in Windows 98.

The code snippet assumes that RegisterClass is not failing for any other reason, such as an invalid lpfnWndProc field in the WNDCLASS structure.

In such cases, you can use the GetLastError function to determine the exact cause of the error. GetLastError is a general-purpose function in Windows that provides extended error information when a function call fails.

The documentation for individual functions will indicate whether they support error retrieval using GetLastError. In the case of calling RegisterClassW in Windows 98, GetLastError returns 120, which corresponds to the ERROR\_CALL\_NOT\_IMPLEMENTED identifier defined in WINERROR.H.

While some Windows programmers advocate for checking the return value of every function call for errors, this can become tedious and unnecessarily complex.

For instance, checking for errors when allocating memory is crucial, as many Windows functions rely on memory allocation.

However, if a function like RegisterClass fails due to memory allocation issues, the system is likely already in a critical state.

For the sample programs in this book, the author minimizes error checking to focus on illustrating the main concepts and avoid distracting from the core learning objectives.

This doesn't imply that error checking is unimportant; it's an essential practice in real-world software development.

**Historical Context of hPrevInstance Check**

In some older Windows sample programs, you might encounter code that checks the value of hPrevInstance before initializing the window class and registering it.

This practice is rooted in the behavior of 16-bit versions of Windows.

In 16-bit Windows, when you launched a new instance of a program that was already running, the hPrevInstance parameter passed to the WinMain function would contain the instance handle of the existing instance.

This allowed multiple instances of the same program to share the same window class, which was a memory-saving technique.

Therefore, the window class was only registered if hPrevInstance was NULL, indicating that no other instances of the program were running.

With the advent of 32-bit Windows, the behavior of hPrevInstance changed.

In 32-bit versions of Windows, hPrevInstance is always NULL, regardless of whether another instance of the program is running or not.

This means that the code snippet mentioned earlier, which checks for hPrevInstance to be NULL before registering the window class, is no longer necessary.

While the code snippet will still work properly in 32-bit Windows, it's an outdated practice that doesn't reflect the current behavior of the hPrevInstance parameter.

It's recommended to avoid this unnecessary check and directly register the window class without checking for hPrevInstance.

**Distinction between Window Class and Window**

A window class defines the general characteristics of a type of window, such as its default behavior, appearance, and functionality.

It serves as a template for creating multiple windows with similar attributes.

When you create a specific window using the CreateWindow function, you provide more detailed information about the window's placement, size, and behavior.

The distinction between the window class and the window lies in the level of abstraction.

The window class represents a general category of windows, while the window represents a specific instance of that class.

This separation allows for efficient memory management and code reuse.

For example, all push buttons in Windows are created based on the same window class, which encapsulates the common behavior and appearance of push buttons.

This window class handles keyboard and mouse input, defines the button's appearance, and ensures that all push buttons behave consistently.

However, individual push buttons can have different sizes, locations, and text labels, which are specified when the button is created.

**CreateWindow Function and Its Arguments**

The CreateWindow function creates a new window based on a specified window class and additional parameters.

It takes several arguments, each providing specific information about the window:

**szAppName:** The name of the window class, which determines the window's general characteristics.

**TEXT("The Hello Program"):** The text that appears in the title bar of the window.

**WS\_OVERLAPPEDWINDOW:** The window style, which defines the window's appearance and behavior, such as its border, title bar, and minimize/maximize buttons.

**CW\_USEDEFAULT:** The initial x-position of the window. CW\_USEDEFAULT indicates that Windows should automatically position the window on the screen.

**CW\_USEDEFAULT:** The initial y-position of the window. CW\_USEDEFAULT indicates that Windows should automatically position the window on the screen.

**CW\_USEDEFAULT:** The initial x-size of the window. CW\_USEDEFAULT indicates that Windows should use the default width for the window class.

**CW\_USEDEFAULT:** The initial y-size of the window. CW\_USEDEFAULT indicates that Windows should use the default height for the window class.

**NULL:** The handle of the parent window. If not specified, the window has no parent window.

**NULL:** The handle of the window menu. If not specified, the window has no menu.

**hInstance:** The instance handle of the program. This is the same handle passed to the WinMain function when the program starts.

**NULL:** Creation parameters. These parameters are typically used for advanced window creation options.

**Code Explanation**

The CreateWindow call in HELLOWIN.C creates a new window with the following characteristics:

* The window class is named szAppName.
* The window caption is "The Hello Program".
* The window style is WS\_OVERLAPPEDWINDOW, which means it is an overlapping window with a title bar, border, minimize/maximize buttons, and a system menu.
* The initial x-position and y-position are set to CW\_USEDEFAULT, indicating that Windows should automatically position the window on the screen.
* The initial x-size and y-size are also set to CW\_USEDEFAULT, indicating that Windows should use the default width and height for the window class.
* The parent window handle is set to NULL, indicating that the window has no parent window.
* The window menu handle is set to NULL, indicating that the window has no menu.
* The program instance handle is set to hInstance, which is the same handle passed to the WinMain function.
* There are no creation parameters specified.