HELLOWIN.C

The HELLOWIN.C program, a representative example of Windows programming, is predominantly composed of overhead that is common to virtually every Windows program.

In practice, Windows programmers seldom commit the entirety of this syntax to memory.

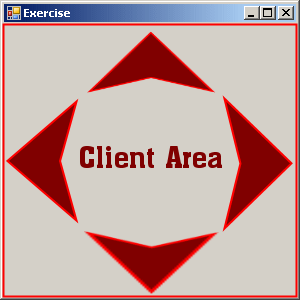
Instead, a common approach is to initiate a new program by duplicating an existing one and subsequently making the necessary modifications.



This flexible method allows for an efficient utilization of existing code structures, a practice explicitly encouraged by the author.

In the earlier mention of HELLOWIN, the assertion that it displays the text string in the center of its window is clarified.

The actual placement is in the center of the program's "client area," delineated in Figure 3-2 as the expansive white space within the title bar and the sizing border.



This distinction is underscored as crucial since the client area represents the canvas within the window where a program can freely draw and present visual output to the user.

Remarkably, despite its relatively concise 80-odd lines of code, HELLOWIN incorporates a myriad of functionalities.

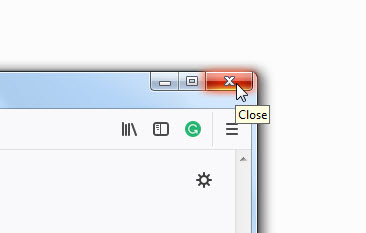
These include the ability to manipulate the window by dragging the title bar or resizing it by interacting with the sizing borders.



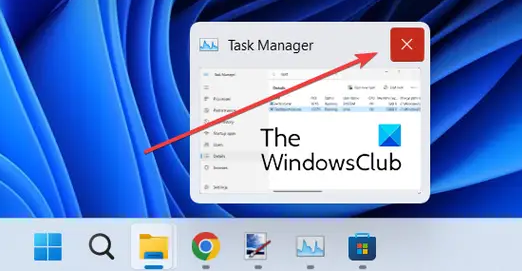
Notably, the program dynamically adjusts the position of the text string to the center of its client area when the window size changes.

Additionally, users can maximize the window to occupy the entire screen, minimize it to remove it from view, and access these options not only through the window buttons but also via the system menu situated at the far left of the title bar.

The program's versatility extends to various interaction modes, such as closing the window to terminate the program.



This termination can be accomplished through multiple avenues, including selecting the Close option from the system menu, clicking the close button at the far right of the title bar, or employing a double-click on the system menu icon.



Looking ahead, the text anticipates an in-depth examination of HELLOWIN.C in the subsequent chapters.

Despite having a WinMain function akin to the sample programs in the initial chapters, it introduces a second function, WndProc, denoted as the window procedure or colloquially referred to as the "win prock" among Windows programmers.

Notably, there is no explicit code within HELLOWIN.C that calls WndProc.

However, its reference in WinMain necessitates its declaration near the program's outset, setting the stage for a comprehensive exploration of its role and significance in the ensuing discussions.

**18 WINDOWS FUNCTIONS THAT HELLOWIN.C CALLS**

LoadIcon

Loads an icon for use by a program. An icon is a small image that is used to identify a program or file. The LoadIcon function loads an icon file into memory and returns a handle to the icon.

LoadCursor

Loads a mouse cursor for use by a program. A mouse cursor is a small image that is displayed on the screen when the mouse is moved. The LoadCursor function loads a cursor file into memory and returns a handle to the cursor.

GetStockObject

Obtains a graphic object, in this case a brush used for painting the window's background. A graphic object is a resource that is used to draw graphics on the screen. The GetStockObject function retrieves a predefined graphic object from the system.

RegisterClass

Registers a window class for the program's window. A window class is a template that defines the characteristics of a window, such as its size, style, and background color. The RegisterClass function registers a window class with the system.

MessageBox

Displays a message box. A message box is a pop-up window that is used to display a message to the user. The MessageBox function creates and displays a message box.

CreateWindow

Creates a window based on a window class. The CreateWindow function creates a window based on a window class that was previously registered with the system.

ShowWindow

Shows the window on the screen. The ShowWindow function makes a window visible on the screen.

UpdateWindow

Directs the window to paint itself. The UpdateWindow function sends a message to a window telling it to repaint itself.

GetMessage

Obtains a message from the message queue. A message is a notification that is sent to a window by the operating system. The GetMessage function retrieves a message from the message queue.

TranslateMessage

Translates some keyboard messages. The TranslateMessage function converts certain keyboard messages into Windows messages.

DispatchMessage

Sends a message to a window procedure. The DispatchMessage function sends a message to the window procedure for the window that received the message.

PlaySound

Plays a sound file. The PlaySound function plays a sound file.

BeginPaint

Initiates the beginning of window painting. The BeginPaint function prepares a window for painting.

GetClientRect

Obtains the dimensions of the window's client area. The GetClientRect function retrieves the dimensions of the client area of a window.

DrawText

Displays a text string. NThe DrawText function displays a text string on the screen.

EndPaint

Ends window painting. The EndPaint function completes the painting of a window.

PostQuitMessage

Inserts a "quit" message into the message queue. The PostQuitMessage function inserts a "quit" message into the message queue. This message tells the program to terminate.

DefWindowProc

Performs default processing of messages. The DefWindowProc function performs default processing of messages that are not handled by the window procedure.

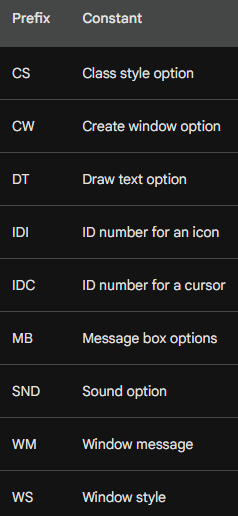
UPPERCASE IDENTIFIERS

Uppercase identifiers are used in HELLOWIN.C because they are defined in the Windows header files.

Several of these identifiers have a two-letter or three-letter prefix followed by an underscore.

The prefix indicates a general category to which the constant belongs, as indicated in the table below.

**Table of Prefixes**

****

**Examples**

**CS\_HREDRAW -** Specifies that the window should be redrawn when its client area is resized.

**DT\_VCENTER -** Specifies that text should be displayed in the center of the rectangle specified by the DrawText function.

**SND\_FILENAME -** Specifies that the PlaySound function should play the sound file specified by the filename parameter.

**CS\_VREDRAW -** Specifies that the window should be redrawn when its vertical scroll bar is moved.

**IDC\_ARROW -** Specifies the standard arrow cursor.

**WM\_CREATE -** Sent when a window is created.

**CW\_USEDEFAULT -** Specifies that the default size and position should be used for the window.

**IDI\_APPLICATION -** Specifies the application's default icon.

**WM\_DESTROY -** Sent when a window is destroyed.

**DT\_CENTER -** Specifies that text should be centered horizontally.

**MB\_ICONERROR -** Specifies that the message box should have an error icon.

**WM\_PAINT -** Sent when a window's client area needs to be painted.

**DT\_SINGLELINE -** Specifies that text should be drawn on a single line.

**SND\_ASYNC -** Specifies that the PlaySound function should play the sound file asynchronously.

**WS\_OVERLAPPEDWINDOW -** Specifies that the window should have a title bar, a minimize button, a maximize button, a system menu, and a sizing border.

**Numeric Constants**

Uppercase identifiers are simply numeric constants.

Uppercase identifiers, also known as symbolic names, are used in programming to represent numeric constants.

They are typically defined in header files and used to improve code readability and maintainability.

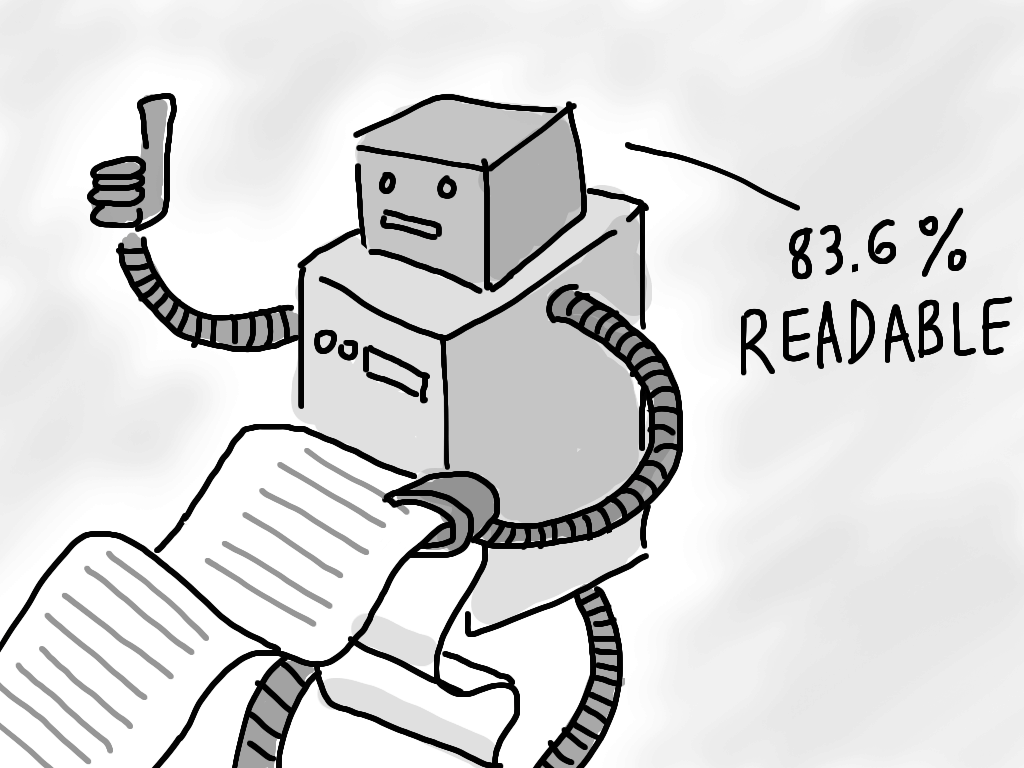
In Windows programming, uppercase identifiers are widely used to define various settings and options related to window styles, class styles, message boxes, and more.

You almost never need to remember numeric constants when programming for Windows because virtually every numeric constant has an identifier defined in the header files.

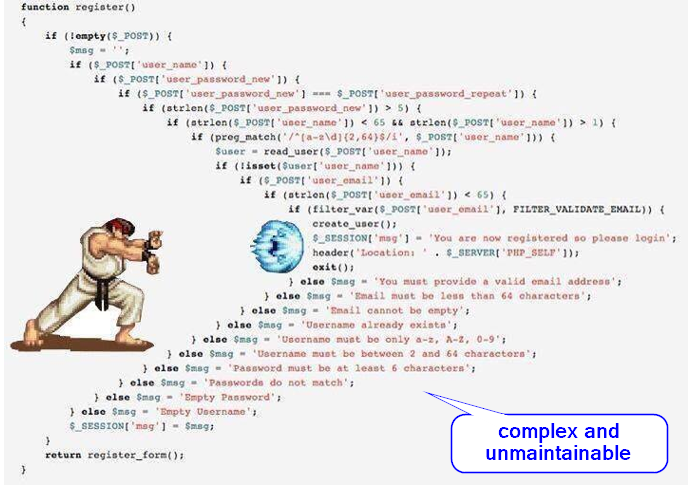


**Benefits of Using Uppercase Identifiers**

**Code Readability:** Uppercase identifiers make code more readable and understandable. This is because they provide meaningful names for numeric constants, which can be difficult to remember and understand.



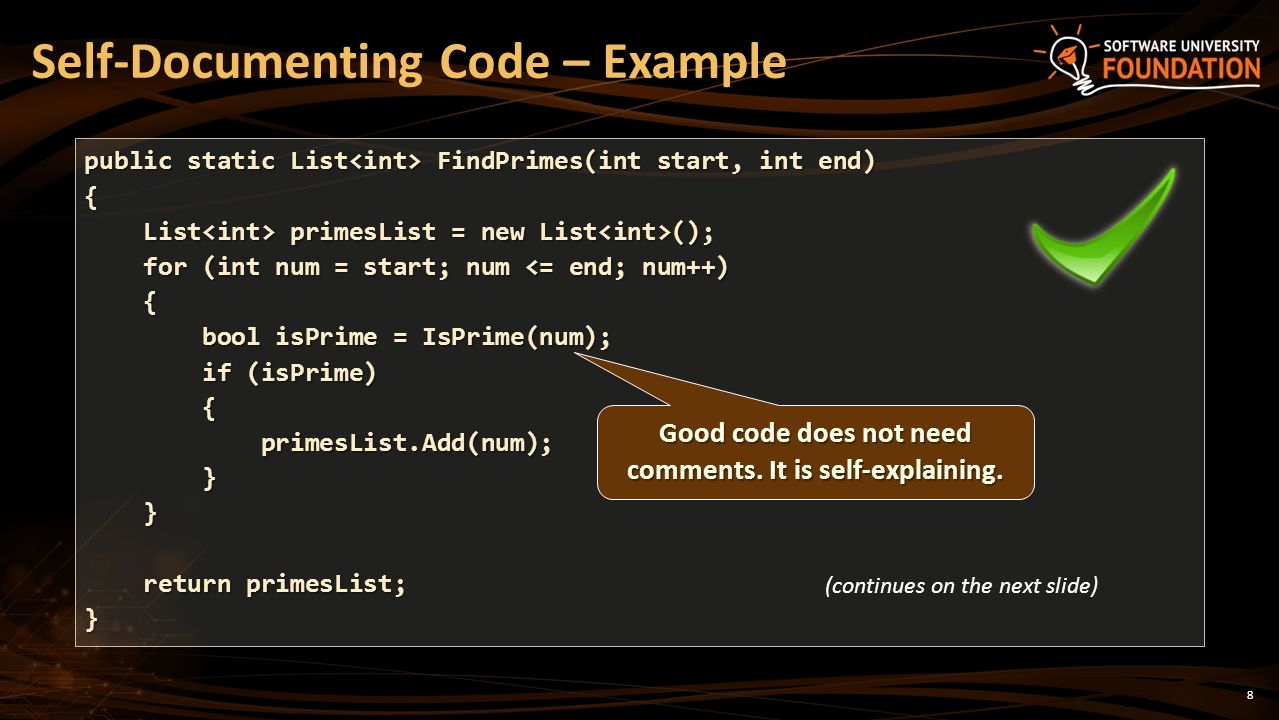
**Code Maintainability:** Uppercase identifiers make code more maintainable. This is because they make it easier to understand and change code, especially for programmers who are not familiar with the code.



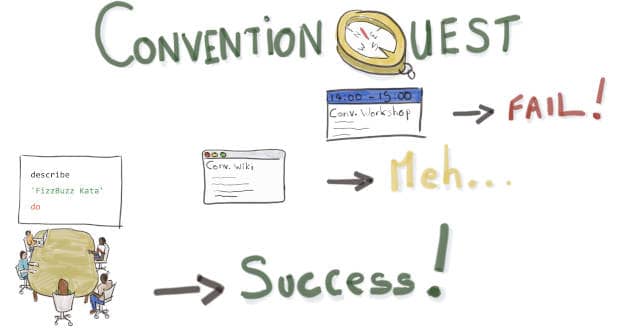
**Error Reduction:** Uppercase identifiers can help to reduce errors in code. This is because they make it more likely that programmers will use the correct constant for a particular situation.



**Self-Documenting Code:** Uppercase identifiers serve as a form of self-documenting code. By providing descriptive names for constants, developers are essentially embedding documentation within the code itself, making it easier for others to understand the purpose of different variables and values.



**Consistency with Programming Conventions:** The use of uppercase identifiers is a well-established convention in many programming languages, particularly in Windows programming. Following these conventions improves code consistency and makes it easier for developers to collaborate and understand each other's work.



NEW DATA TYPES USED IN HELLOWIN.C

Some of the identifiers used in HELLOWIN.C are new data types, which are also defined in the Windows header files using either typedef or #define statements.

These new data types were originally introduced to ease the transition of Windows programs from the original 16-bit system to future operating systems that would be based on 32-bit technology.

While this transition didn't quite work as smoothly and transparently as everyone thought at the time, the concept of using new data types was fundamentally sound.

**Types of the new datatypes**

Some of the new data types are simply convenient abbreviations. For example, the UINT data type used for the second parameter to WndProc is simply an unsigned int, which in Windows 98 is a 32-bit value.

Other new data types are less obvious. For example, the third and fourth parameters to WndProc are defined as WPARAM and LPARAM, respectively. These names have their origins in the history of Windows.

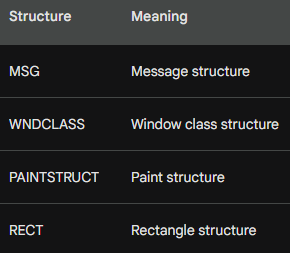
When Windows was a 16-bit system, the third parameter to WndProc was defined as a WORD, which was a 16-bit unsigned short integer, and the fourth parameter was defined as a LONG, which was a 32-bit signed long integer. This is why these parameters have the "W" and "L" prefixes.

In the 32-bit versions of Windows, however, WPARAM is defined as a UINT and LPARAM is defined as a LONG (which is still the C long data type), so both parameters to the window procedure are 32-bit values.

This can be a bit confusing because the WORD data type is still defined as a 16-bit unsigned short integer in Windows 98, so the "W" prefix to "PARAM" creates somewhat of a misnomer.

**Data Structures**

HELLOWIN.C also uses four data structures (which will be discussed later in the chapter) that are defined in the Windows header files. These data structures are shown in the table below.



The first two data structures are used in WinMain to define two structures named msg and wndclass.

The second two are used in WndProc to define two structures named ps and rect.

The use of new data types and data structures is an important part of Windows programming.

These data types and structures help to make code more readable, maintainable, and efficient.

GETTING AND USING HANDLES

**What are Handles?**

Handles are a fundamental concept in Windows programming. They are simply numbers (usually 32 bits in size) that refer to objects.

Windows uses handles to manage resources such as windows, icons, cursors, brushes, and more.

Handles are similar to file handles used in conventional C programming.

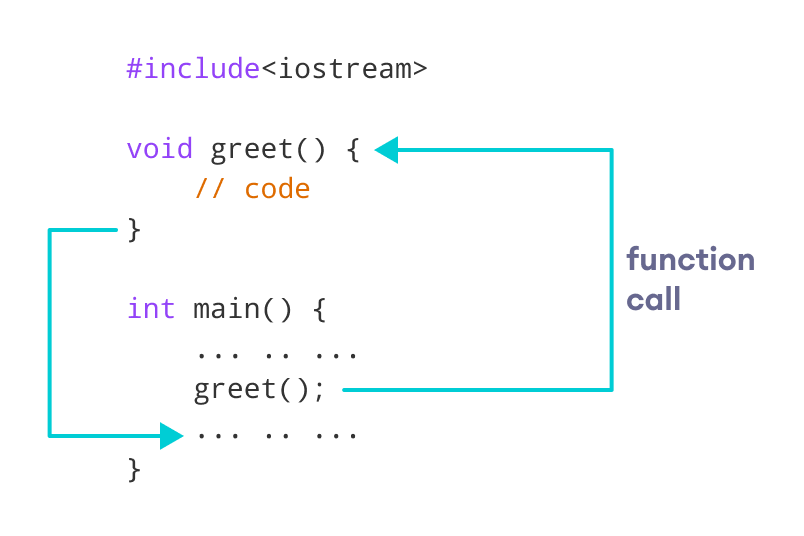


How are Handles Obtained?

Programs typically obtain handles by calling Windows functions.

For example, the CreateWindow function is used to create a window and returns a handle to the newly created window.

The LoadIcon function is used to load an icon from a file and returns a handle to the icon.

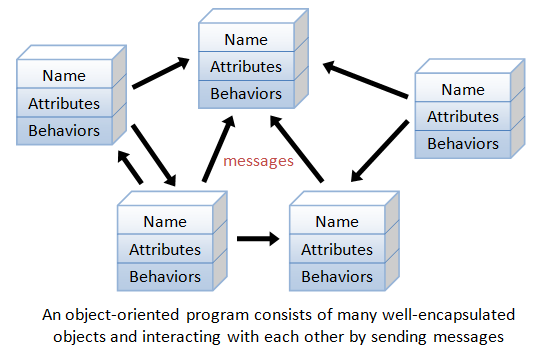


**How are Handles Used?**

Once a program has a handle to an object, it can use the handle in other Windows functions to refer to the object.

For example, the ShowWindow function is used to show a window, and it takes a window handle as a parameter.

The DrawIcon function is used to draw an icon on the screen, and it takes an icon handle as a parameter.



**Advantages of Using Handles**

* Handles provide a way for programs to refer to objects in a way that is independent of the object's internal structure. This makes it possible for programs to use objects without having to know how they are implemented.
* Handles also make it possible for programs to share objects with each other.

**Disadvantages of Using Handles**

* Handles can be difficult to manage.
* Programs must keep track of handles and close them when they are no longer needed.
* If a program fails to close a handle, the object that the handle refers to can be leaked.

**Examples of Handles**

**HINSTANCE -** Handle to an "instance" (the program itself).

**HWND -** Handle to a window.

**HDC -** Handle to a device context.

**HICON -** Handle to an icon.

**HCURSOR -** Handle to a mouse cursor.

**HBRUSH -** Handle to a graphics brush.

**Conclusion**

Handles are an important part of Windows programming.

They provide a way for programs to refer to objects in a way that is independent of the object's internal structure.

Handles also make it possible for programs to share objects with each other.

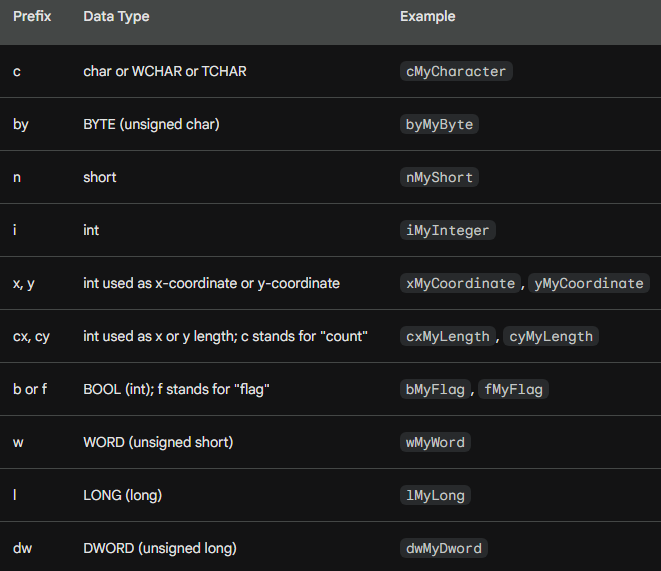
HUNGARIAN NOTATION

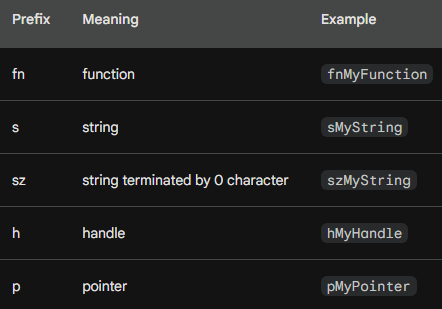
Hungarian Notation is a variable-naming convention that was developed by Charles Simonyi, a legendary Microsoft programmer.

The convention uses prefixes to indicate the data type of a variable, as well as other information about the variable's purpose.

This can help to make code more readable and maintainable, and can also help to prevent errors.

The prefixes used in Hungarian Notation are as follows:





**Here are some examples of how Hungarian Notation is used:**

**szCmdLine:** String terminated by 0 character, passed as a parameter to WinMain.

**hInstance:** Handle to an "instance" (the program itself).

**iCmdShow:** Integer indicating how the window should be shown when it is created.

**msg:** Structure of the MSG type, used to store information about messages sent to the window procedure.

**wndclass:** Structure of the WNDCLASS type, used to register a window class.

**ps:** PAINTSTRUCT structure, used to store information about the current paint operation.

**rect:** RECT structure, used to represent a rectangle.

**Benefits of Hungarian Notation**

**Improved code readability:** The prefixes used in Hungarian Notation can make code more readable and easier to understand.

**Enhanced code maintainability:** Hungarian Notation can make code more maintainable by making it easier to understand the purpose of variables.

**Reduced error proneness:** Hungarian Notation can help to prevent errors by making it more difficult to use variables with the wrong data type.

**Drawbacks of Hungarian Notation**

**Increased verbosity:** Hungarian Notation can make code more verbose, which can make it more difficult to read and write.

**Inconsistent application:** Hungarian Notation is not always applied consistently, which can make it difficult to understand code written by others.

**Limited expressiveness:** Hungarian Notation is not expressive enough to convey all of the information that can be conveyed by more descriptive variable names.

**Conclusion**

Hungarian Notation is a variable-naming convention that can be helpful for improving code readability, maintainability, and error reduction. However, it is important to weigh the benefits of Hungarian Notation against its drawbacks before deciding whether or not to use it.

REGISTERING A WINDOW CLASS USING THE REGISTERCLASS FUNCTION

In the context of Windows programming, registering a window class refers to the process of defining and storing the characteristics of a window template with the Windows operating system.

This template serves as the blueprint for creating windows, allowing you to reuse the predefined properties of the class rather than having to define them individually for each window.

When you register a window class, you provide information about the window's appearance, behavior, and how it interacts with the operating system.

This information is stored in a data structure called a WNDCLASS or WNDCLASSW structure, depending on whether you are using the ASCII or Unicode version of the Windows API.

Once a window class is registered, you can create windows based on that class by calling the CreateWindow or CreateWindowEx function.

These functions will create a new window and associate it with the specified window class.

The window will inherit the properties defined in the window class, unless you explicitly override them when creating the window.

**Window Classes and Their Significance**

In the realm of Windows programming, a window class serves as a blueprint for creating windows.

It encapsulates the fundamental characteristics of a window, including its behavior, appearance, and how it interacts with the operating system.

When you create a window, you specify the window class upon which it should be based.

This allows you to reuse the predefined properties of the class rather than having to define them individually for each window.

**Registering a Window Class with RegisterClass**

To register a window class, you utilize the RegisterClass function.

This function accepts a single argument, a pointer to a WNDCLASS structure, which contains the essential information about the window class.

The WNDCLASS structure is defined in two different ways in the WINUSER.H header file:

**WNDCLASSA:** This is the ASCII version of the structure, intended for use with ANSI applications.

**WNDCLASSW:** This is the Unicode version of the structure, intended for use with Unicode applications.

**Key Fields of the WNDCLASS Structure**

The WNDCLASS structure encompasses several crucial fields that determine the characteristics of the window class:

**style:** This field specifies the style of the window, including its appearance (border, caption, scroll bars, etc.) and behavior (how it responds to user interactions).

**lpfnWndProc:** This field is a pointer to the window procedure, the function that will handle messages sent to the window. This function is responsible for processing user actions and updating the window's appearance accordingly.

**cbClsExtra:** This field specifies the number of extra bytes to allocate for the window class. This memory can be used to store additional information about the window class that is not directly related to its functionality.

**cbWndExtra:** This field specifies the number of extra bytes to allocate for each window created based on this class. This memory can be used to store additional information about individual windows.

**hInstance:** This field is a handle to the instance of the application that is registering the window class. The instance handle is used to identify the application to the operating system.

**hIcon:** This field is a handle to the icon that will be displayed in the title bar and minimized window of windows created based on this class.

**hCursor:** This field is a handle to the cursor that will be used when the mouse is over windows created based on this class.

**hbrBackground:** This field is a handle to the brush that will be used to fill the background of windows created based on this class.

**lpszMenuName:** This field is a pointer to a character string that specifies the name of the menu resource that will be associated with windows created based on this class.

**lpszClassName:** This field is a pointer to a character string that specifies the name of the window class. This name must be unique within the application.

Registering window classes is an essential step in creating windows in Windows programming. It allows you to define and reuse window templates, making your code more efficient and maintainable.

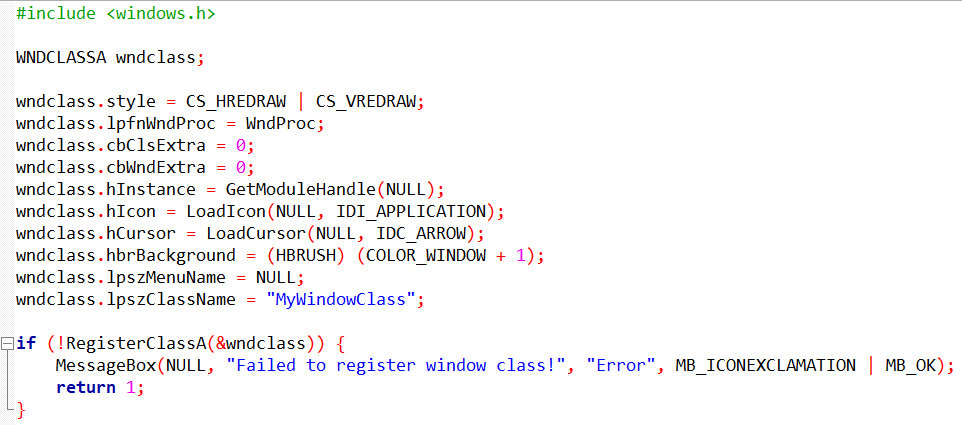
**Here's a simplified explanation of the process:**

**Define a WNDCLASS structure:** This structure holds the information about the window class, such as its style, window procedure, icon, cursor, and background brush.

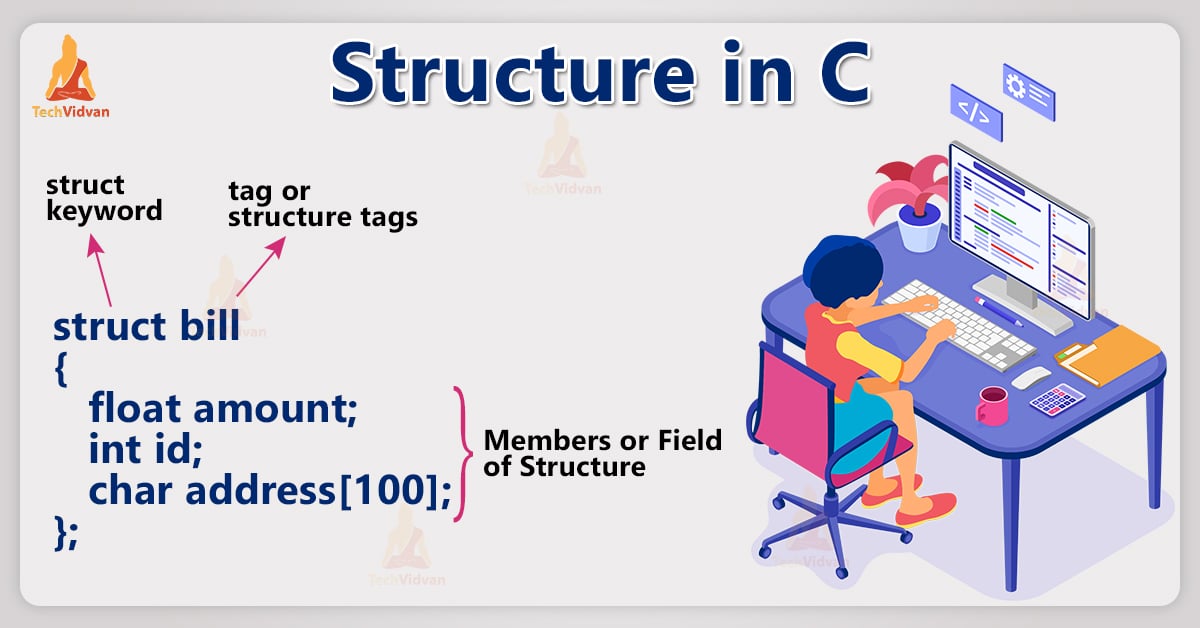
**Register the WNDCLASS structure:** Use the RegisterClass or RegisterClassW function to pass the WNDCLASS structure to the operating system. This makes the window class available for creating windows.

**Create windows based on the registered class:** Use the CreateWindow or CreateWindowEx function to create new windows. These windows will inherit the properties defined in the registered window class.

The **code for registering a window class** using the RegisterClass function involves defining and initializing a WNDCLASS structure and then passing it to the RegisterClass function. Here's an example of how to register a window class using the ASCII version of the WNDCLASS structure:



This code defines a WNDCLASSA structure named wndclass and initializes its members with the desired values.



The CS\_HREDRAW and CS\_VREDRAW styles ensure that the window will be redrawn whenever its size changes.

The WndProc function pointer specifies the function that will handle messages sent to windows created based on this class.

The GetModuleHandle(NULL) function retrieves the instance handle of the application.

The LoadIcon and LoadCursor functions load the icon and cursor resources, respectively.

The COLOR\_WINDOW + 1 value is a predefined color that represents the default window background color.

The NULL value for lpszMenuName indicates that the windows created based on this class will not have a menu.

Finally, the RegisterClassA function is called to register the window class. If the registration fails, an error message is displayed, and the application exits.

Here, RegisterClassA(&wndclass) is a function call to RegisterClassA, which is a Windows API function responsible for registering a window class.

This function informs the Windows operating system about the characteristics and behavior of a particular type of window that the application intends to create.



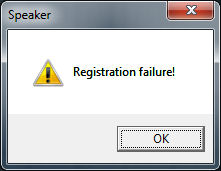
**Now, let's analyze the if statement:**

**The ! operator** is a logical NOT, so !RegisterClassA(&wndclass) checks if the RegisterClassA function call returns a false value, indicating that the registration of the window class has failed.

**If the registration fails**, the code inside the curly braces is executed. In this case, a message box is displayed using the MessageBox function. This message box notifies the user that there was a failure in registering the window class. The parameters for the MessageBox function include:

**NULL:** This parameter specifies that the message box is not associated with any window.

**"Failed to register window class!":** This is the message displayed in the message box, indicating the nature of the error.



**"Error":** The title of the message box.

**MB\_ICONEXCLAMATION | MB\_OK:** These are flags that determine the appearance and behavior of the message box. MB\_ICONEXCLAMATION specifies an exclamation point icon, indicating a warning. MB\_OK provides an OK button for the user to acknowledge the message.

After displaying the error message, the function returns 1, which is a common convention in C to indicate an error condition or abnormal termination of a program.

In essence, this code is a defensive mechanism. If the registration of the window class fails, it alerts the user through a message box, allowing them to be aware of the issue, and the program terminates with an error code (1).

This kind of error handling is essential for robust applications, ensuring that users are informed about issues and developers can diagnose and address problems during development or debugging.

**An in-depth explanation of the prefixes used in Hungarian Notation for the WNDCLASS structure:**

**Prefixes and Their Meanings**

Hungarian Notation uses prefixes to indicate the data type and purpose of variables. In the context of the WNDCLASS structure, the following prefixes are used: