

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.

```
28 expires = "expires" + (expires ? " : " + expires : "");
29 document.cookie = "expires=" + expires + "; path=/";
30 }
31
32 function validateForm() {
33     var x = document.forms["myForm"]["fname"].value;
34     if (x == "") {
35         alert("Name must be filled out");
36         return false;
37     }
38 }
39
40 var marker = new toogle.secure.Marker({image: log, position: 'top'});
41 marker.addListener('click', function() {
42     infowindow.open(log, marker);
43 });
44
45 <form name="myForm" action="/action_page.php" return="validateForm">
46   Name: <input type="text" name="fname">
47   <input type="submit" value="Submit">
48 </form>
```

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**.

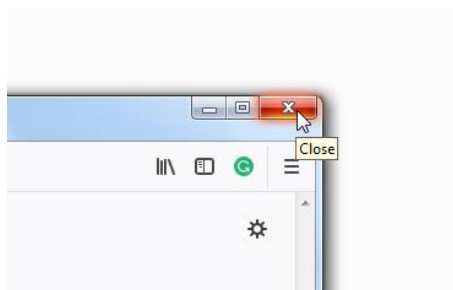
These include the ability to **manipulate the window** by dragging the title bar or **resizing** it by interacting with the sizing borders.



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

Prefix	Constant
CS	Class style option
CW	Create window option
DT	Draw text option
IDI	ID number for an icon
IDC	ID number for a cursor
MB	Message box options
SND	Sound option
WM	Window message
WS	Window style

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.

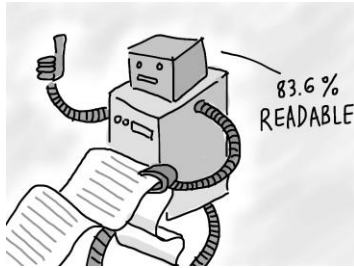
In Windows programming, uppercase identifiers are widely used to define various settings and options related to window styles, class styles, message boxes, and more.

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.

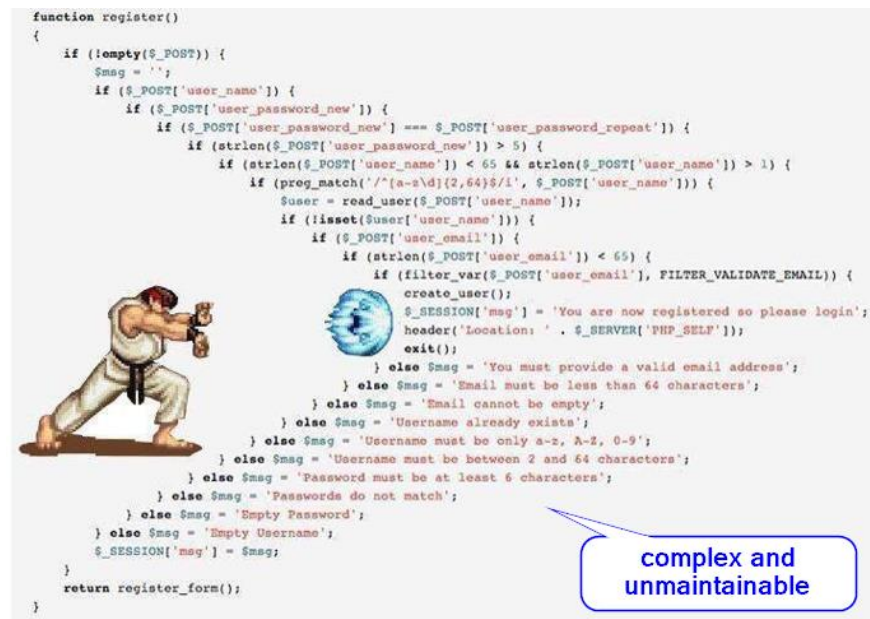


Benefits of Using Uppercase Identifiers

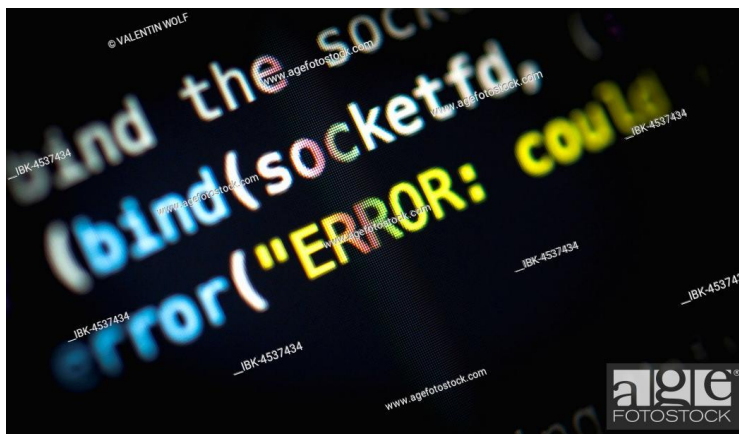
Code Readability: Uppercase identifiers make code more readable and understandable. This is because they provide meaningful names for numeric constants, which can be difficult to remember and understand.



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.

Self-Documenting Code – Example



```
public static List<int> FindPrimes(int start, int end)
{
    List<int> primesList = new List<int>();
    for (int num = start; num <= end; num++)
    {
        bool isPrime = IsPrime(num);
        if (isPrime)
        {
            primesList.Add(num);
        }
    }

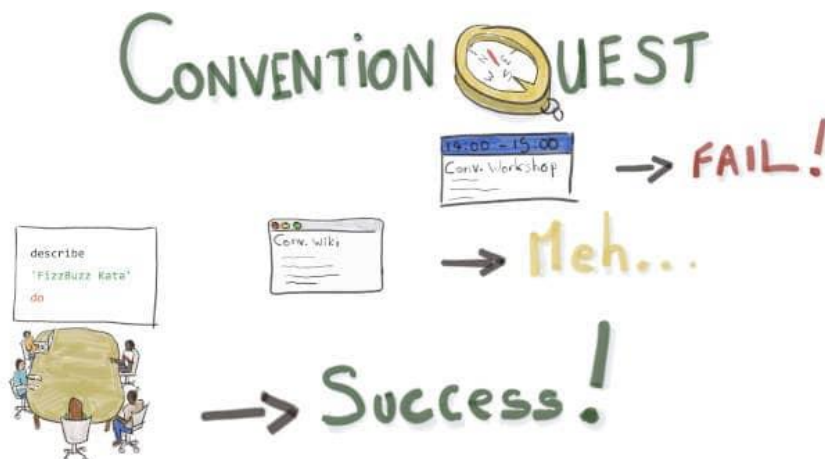
    return primesList;
}
```

Good code does not need comments. It is self-explaining.

(continues on the next slide)

8

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.

Structure	Meaning
MSG	Message structure
WNDCLASS	Window class structure
PAINTSTRUCT	Paint structure
RECT	Rectangle structure

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.

```
#include<iostream>

void greet() {
    // code
}

int main() {
    ... ..
    greet();
    ... ..
}
```

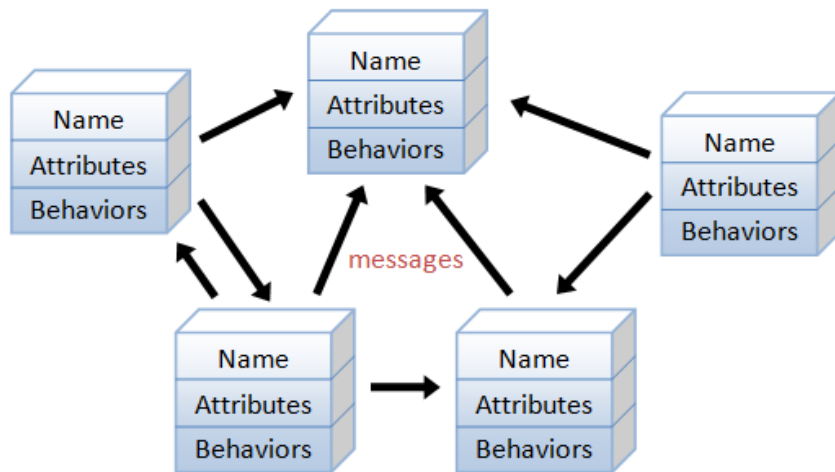
function call

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**.



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

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:

Prefix	Data Type	Example
c	char or WCHAR or TCHAR	cMyCharacter
by	BYTE (unsigned char)	byMyByte
n	short	nMyShort
i	int	iMyInteger
x, y	int used as x-coordinate or y-coordinate	xMyCoordinate , yMyCoordinate
cx, cy	int used as x or y length; c stands for "count"	cxMyLength , cyMyLength
b or f	BOOL (int); f stands for "flag"	bMyFlag , fMyFlag
w	WORD (unsigned short)	wMyWord
l	LONG (long)	lMyLong
dw	DWORD (unsigned long)	dwMyDword

Prefix	Meaning	Example
fn	function	fnMyFunction
s	string	sMyString
sz	string terminated by 0 character	szMyString
h	handle	hMyHandle
p	pointer	pMyPointer

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:

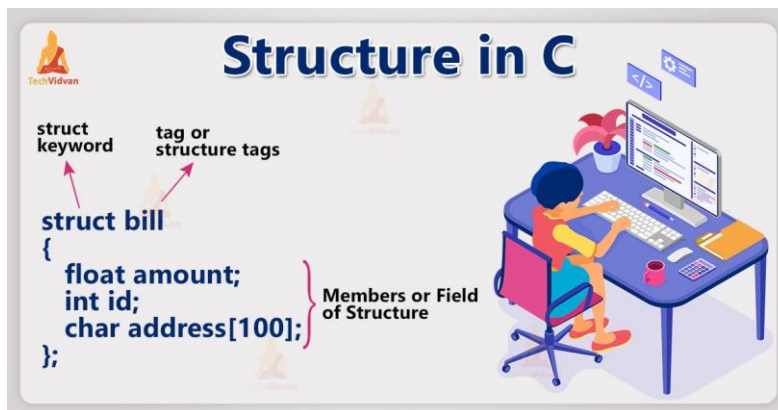
```
#include <windows.h>

WNDCLASSA wndclass;

wndclass.style = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc = WndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.hInstance = GetModuleHandle(NULL);
wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wndclass.lpszMenuName = NULL;
wndclass.lpszClassName = "MyWindowClass";

if (!RegisterClassA(&wndclass)) {
    MessageBox(NULL, "Failed to register window class!", "Error", MB_ICONEXCLAMATION | MB_OK);
    return 1;
}
```

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 **lpstrMenuName** 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:

Prefix	Meaning	Example
lpfn	long pointer to a function	lpfnWndProc
cb	count of bytes	cbClsExtra , cbWndExtra
h	handle	hInstance , hIcon , hCursor , hbrBackground
lpsz	long pointer to a string terminated with a zero	lpszMenuName , lpszClassName

lpfnWndProc

The **lpfnWndProc** member points to the window procedure, which is the function that handles messages sent to windows created based on this class. The lpfn prefix indicates that lpfnWndProc is a long pointer to a function.

cbClsExtra and cbWndExtra

The **cbClsExtra** and **cbWndExtra** members specify the number of extra bytes to allocate for the window class and for each window created based on that class, respectively. The cb prefix indicates that these members are counts of bytes.

hInstance, hIcon, hCursor, and hbrBackground

The **hInstance**, **hIcon**, **hCursor**, and **hbrBackground** members are handles to various resources, such as the application instance handle, the icon, the cursor, and the background brush. The h prefix indicates that these members are handles.

lpszMenuName and lpszClassName

The **lpszMenuName** and **lpszClassName** members are pointers to constant wide-character strings. The lpsz prefix indicates that these members are long pointers to strings terminated with a zero.

Unicode Version of the WNDCLASS Structure

The Unicode version of the **WNDCLASS** structure, **WNDCLASSW**, is defined similarly to the ASCII version, except that the **lpszMenuName** and **lpszClassName** members are pointers to constant wide-character strings instead of constant ASCII character strings. This allows the structure to be used with Unicode applications.

UNDERSTANDING WINUSER.H AND TYPED ALIASES

The **WINUSER.H** header file stands as a pivotal component within the Windows API, offering critical definitions for an array of data types, functions, and macros essential in Windows programming.

Key among these definitions are the **WNDCLASSA** and **WNDCLASSW** structures, distinguishing the **ASCII** and **Unicode versions** of the window class structure, respectively.

Defining Type Aliases Based on UNICODE

The `WINUSER.H` header file utilizes a conditional compilation block to define type aliases for the `WNDCLASS` structure based on the value of the `UNICODE` preprocessor directive.

A **conditional compilation block** is a section of code that is only compiled and included in the final executable if a certain condition is met.

This directive is typically set during the compilation process to indicate whether the application is being compiled for an ASCII environment or a Unicode environment.

This **condition** is typically **specified using a preprocessor directive**, which is a special instruction that tells the compiler to do something before the actual code is compiled.

```
#include <windows.h>

// Type alias for HWND
typedef HWND WindowHandle;

// Function that uses the type alias
void SomeFunction(WindowHandle hWnd) {
    // Function implementation
}

int main() {
    // Using the type alias to declare a window handle
    WindowHandle myWindow = CreateWindowEx(
        0,                                // Extended window style
        "MyClass",                        // Class name
        "My Window",                      // Window title
        WS_OVERLAPPEDWINDOW,             // Window style
        CW_USEDEFAULT,                    // X-coordinate
        CW_USEDEFAULT,                    // Y-coordinate
        CW_USEDEFAULT,                    // Width
        CW_USEDEFAULT,                    // Height
        NULL,                             // Parent window
        NULL,                             // Menu
        GetModuleHandle(NULL),             // Instance handle
        NULL                              // Additional application data
    );

    // Using the type alias in a function call
    SomeFunction(myWindow);

    // Rest of the program
    return 0;
}
```


Simplifying the Explanation:

In the example code provided, we introduced a type alias, `WindowHandle`, for the Windows API type `HWND`. This alias makes the code more readable, especially when dealing with multiple functions or structures that use window handles.

Why Use Type Aliases?

Readability and Maintainability:

Type aliases simplify the codebase, making it more maintainable. Developers can use more intuitive names when working with complex WinAPI types.

Using Type Aliases in `WINUSER.H`:

In the `WINUSER.H` header file, a conditional compilation block defines type aliases for the `WNDCLASS` structure based on the `UNICODE` preprocessor directive.

UNICODE Directive:

This directive is set during compilation, indicating whether the application is compiled for an ASCII or Unicode environment.

Conditional Compilation:

If `UNICODE` is defined, `WNDCLASSW` is aliased to `WNDCLASS`, and its pointers are aliased accordingly. This represents the Unicode version of the window class structure.

If `UNICODE` is not defined, `WNDCLASSA` is aliased to `WNDCLASS`, and its pointers are aliased. This represents the ASCII version.

Common Programming Practice:

The use of conditional compilation blocks and preprocessor directives is a standard practice in programming. It allows developers to adapt code to specific environments and configurations.

In the case of `WINUSER.H`, the conditional compilation block ensures that the correct type aliases for the `WNDCLASS` structure are used based on the Unicode setting during compilation.

Structure Aliases for ASCII and Unicode

When the `UNICODE` directive is defined, the `WNDCLASSW` structure is aliased to `WNDCLASS`, and pointers to `WNDCLASSW` are aliased to `PWNDCLASS`, `NPWNDCLASS`, and `LPWNDCLASS`. This alignment effectively represents the Unicode version of the window class structure.

Structure Aliases for ASCII

Conversely, when the UNICODE directive is not defined, the WNDCLASSA structure is aliased to WNDCLASS, and pointers to WNDCLASSA are aliased to PWNDCLASS, NPWNDCLASS, and LPWNDCLASS. This alignment makes WNDCLASS and its pointer variations represent the ASCII version of the window class structure.

Teen-Friendly Explanation of “is aliased”:

So, imagine you have a cool nickname for your friend – let's say your friend's name is Chris, but you like to call him "C-Dawg." It's like giving him an alias, a different name that means the same thing.

Now, in computer programming, when we say the WNDCLASSA structure is aliased to WNDCLASS, it's like giving a fancy computer name an easier nickname. It means we are giving it a simpler name, so we can use it more easily in our code without typing the long and complicated original name every time.

Expert-Level Explanation:

In computer programming, when we say the WNDCLASSA structure is aliased to WNDCLASS, we are essentially creating an alias, or an alternative name, for the structure. This process is often facilitated through the use of type aliases or typedefs in programming languages.

In the context of the Windows API and the WINUSER.H header file, this aliasing is done through conditional compilation.

When the UNICODE directive is not defined, the WNDCLASSA structure, representing the ASCII version of the window class structure, is assigned the alias WNDCLASS. This means that wherever in the code you see WNDCLASS, it refers to the structure that was originally named WNDCLASSA.

This practice simplifies code, providing a more readable and adaptable way to work with different versions of the window class structure based on the Unicode setting of the environment in which the application is being compiled.

In the provided code snippet, you can see the aliasing in action. Let's focus on the type alias for the window handle:

```
// Type alias for HWND
typedef HWND WindowHandle;
```

Here, `WindowHandle` is the alias, and it's created using `typedef` to represent the Windows API type `HWND`. So, in this code, you can use `WindowHandle` instead of `HWND` for improved readability.

Now, let's look at how this alias is used in the code:

```
// Using the type alias to declare a window handle
WindowHandle myWindow = CreateWindowEx(
    0, // Extended window style
    "MyClass", // Class name
    "My Window", // Window title
    WS_OVERLAPPEDWINDOW, // Window style
    CW_USEDEFAULT, // X-coordinate
    CW_USEDEFAULT, // Y-coordinate
    CW_USEDEFAULT, // Width
    CW_USEDEFAULT, // Height
    NULL, // Parent window
    NULL, // Menu
    GetModuleHandle(NULL), // Instance handle
    NULL // Additional application data
);
```

Rationale for Type Aliases

The use of type aliases in the `WINUSER.H` header file serves several purposes:

Simplified Code: It simplifies code by allowing programmers to use `WNDCLASS` and its pointer variations without having to explicitly specify whether they are referring to the ASCII or Unicode version.

Platform Independence: It promotes platform independence by making the code more adaptable to different Unicode settings.

Readability Enhancement: It improves code readability by reducing the need for lengthy type declarations.

Recommendation for Code Clarity

The code provided is a C++ definition using the typedef keyword to create an [alias for the WNDCLASS structure](#) in the Windows API.

The guide suggests using the functionally equivalent definition of the WNDCLASS structure, which is:

```
typedef struct {  
    UINT style;  
    WNDPROC lpfnWndProc;  
    int cbClsExtra;  
    int cbWndExtra;  
    HINSTANCE hInstance;  
    HICON hIcon;  
    HCURSOR hCursor;  
    HBRUSH hbrBackground;  
    LPCTSTR lpszMenuName;  
    LPCTSTR lpszClassName;  
} WNDCLASS, *PWNDCLASS;
```

This simplified structure definition **omits the pointer variations**, which are typically not needed in most programming scenarios. By using this simplified structure, programmers can maintain code clarity and avoid cluttering their code with unnecessary type declarations.

```
typedef struct { ... } WNDCLASS, *PWNDCLASS;
```

This line defines a new type called WNDCLASS using the typedef keyword. It essentially creates an alias for a structure that encapsulates various attributes of a window class in Windows programming.

```
//Longhand method for defining this structure
struct WNDCLASS {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpstrMenuName;
    LPCTSTR lpstrClassName;
};

typedef struct WNDCLASS WNDCLASS;
typedef struct WNDCLASS *PWNDCLASS;
```

```
//Shorthand method(alias)

typedef struct {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpstrMenuName;
    LPCTSTR lpstrClassName;
} WNDCLASS, *PWNDCLASS;
```

Differences:

Syntax:

In the long-hand method, the structure is defined first using the struct keyword, and then separate typedef statements create aliases for the structure and a pointer to the structure.

In the shorthand method, the structure is defined and aliased in a single line using the typedef keyword.

Verbosity:

The long-hand method is more verbose with separate statements for the structure definition and each alias. The shorthand method is more concise, providing a cleaner and more readable way to define the structure and its aliases.

Readability:

The shorthand method is often preferred for its brevity and clarity, especially when dealing with complex structures.

Both methods achieve the same result—defining a structure and creating type aliases for that structure and a pointer to it.

The choice between them is largely a matter of coding style and preference, with the shorthand method being more commonly used for its simplicity.

Structure Members:

The structure contains members like `style`, `lpfnWndProc` (a function pointer), `cbClsExtra`, `cbWndExtra`, `hInstance`, `hIcon`, `hCursor`, `hbrBackground`, `lpszMenuName`, and `lpszClassName`. These members represent different properties and configurations of a window class.

Pointers:

The `*WNDCLASS` part of the typedef statement creates an additional alias for a pointer to this structure. This pointer type is often used when dealing with functions that expect a pointer to a `WNDCLASS` structure.

Subtopic:

This code snippet is related to the discussion in your notes about recommending a simplified definition for the `WNDCLASS` structure. It suggests using this particular definition without pointer variations for improved code clarity.

Explanation of Code:

The code is essentially creating a shorthand way to refer to a complex structure used in Windows programming. It's not necessary for you to dive deeply into the specifics of this code unless you are working on Windows programming projects. The key takeaway is that it's a technique to make the code more readable and concise in certain programming scenarios.

Do You Need to Know?

If you're not into Windows programming or not dealing with GUI applications, this specific code might be more technical detail than you need. However, understanding the concept of type aliases and how they improve code readability can be beneficial in a broader programming context.

Conditional Compilation:

The `#ifdef UNICODE` preprocessor directive checks whether the `UNICODE` identifier is defined.

If it is defined, it means the application is built for Unicode, and the typedef statements map `WNDCLASS` and `PWNDCLASS` to the Unicode versions (`WNDCLASSW` and `PWNDCLASSW`).

If `UNICODE` is not defined, it implies an ANSI build, and the mappings are to the ANSI versions (`WNDCLASSA` and `PWNDCLASSA`).

Structure Definition:

The struct `tagWNDCLASS` defines the actual structure of the window class. It includes the following fields:

UINT style: A set of style bits that define various characteristics of the window class.

WNDPROC lpfnWndProc: A pointer to the window procedure, the function that processes messages sent to the window.

int cbClsExtra: Extra bytes to allocate following the window class structure for class-specific data.

int cbWndExtra: Extra bytes to allocate following the window instance for instance-specific data.

HINSTANCE hInstance: A handle to the application instance that contains the window procedure.

HICON hIcon: A handle to the class icon.

HCURSOR hCursor: A handle to the class cursor.

HBRUSH hbrBackground: A handle to the background brush for the window class.

LPCTSTR lpszMenuName: The resource name of the class menu, if any.

LPCTSTR lpszClassName: A pointer to a null-terminated string representing the class name.

Simplified Pointer Definitions: The typedef statements simplify the usage of pointer types. Instead of explicitly using PWNDCLASSA, PWNDCLASSW, etc., you can use PWNDCLASS uniformly, and the preprocessor directives ensure that it is defined appropriately based on Unicode or ANSI build.

In this declaration:

WNDCLASS is a structure representing the characteristics of a window class in a Windows application.

***PWNDCLASS** is a pointer type that can point to an instance of this structure.

The structure includes the following members:

style: A set of style bits that define various characteristics of the window class.

lpfnWndProc: A pointer to the window procedure, a function that processes messages sent to the window.

cbClsExtra: Extra bytes to allocate following the window class structure for class-specific data.

cbWndExtra: Extra bytes to allocate following the window instance for instance-specific data.

hInstance: A handle to the application instance that contains the window procedure.

hIcon: A handle to the class icon.

hCursor: A handle to the class cursor.

hbrBackground: A handle to the background brush for the window class.

lpszMenuName: The resource name of the class menu, if any.

lpszClassName: A pointer to a null-terminated string representing the class name.

This concise declaration captures the essence of the WNDCLASS structure and its associated pointer type, providing a clear and readable representation.

This code will register a window class named "myWindowClass". The window class will have the following characteristics:

- It will be redrawn whenever the window size changes.
- Its window procedure will be WndProc.
- It will have no extra class-specific or window-specific data.
- Its instance handle will be the handle of the current application instance.
- Its icon will be the default application icon.
- Its cursor will be the default arrow cursor.
- Its background brush will be the default window background brush.
- Its menu resource will be IDR_MENU.
- Once the window class has been registered, you can create windows of that class by calling the CreateWindow function and passing it the name of the window class.

Conclusion

The WINUSER.H header file effectively utilizes type aliases to **simplify code** and **promote platform independence** when dealing with different Unicode settings.

The recommendation to use the simplified structure definition further enhances code readability. By understanding the rationale behind type aliases and using them appropriately, programmers can write more concise and maintainable code.

```
WNDCLASS wndclass;
```

This structure is then initialized, and its **ten fields** are set before calling the RegisterClass function.

The two crucial fields in the **WNDCLASS structure** are the second and the last.

The second field, **lpfnWndProc**, holds the address of a window procedure used for all windows based on this class.

In the given example (HELLOWIN.C), this window procedure is referred to as **WndProc**.

The **last field** is the text name of the window class, which can be chosen as desired. In programs creating only one window, the window class name is often set to the name of the program.

Let's examine each field of the WNDCLASS structure:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW;
```

This statement combines two 32-bit "class style" identifiers using a bitwise OR operator. The WINUSER.H header file defines several identifiers with the CS prefix:

```
#define CS_VREDRAW 0x0001
#define CS_HREDRAW 0x0002
#define CS_KEYCVTWINDOW 0x0004
#define CS_DBLCLKS 0x0008
#define CS_OWNDC 0x0020
#define CS_CLASSDC 0x0040
#define CS_PARENTDC 0x0080
#define CS_NOKEYCVT 0x0100
#define CS_NOCLOSE 0x0200
#define CS_SAVEBITS 0x0800
#define CS_BYTEALIGNCLIENT 0x1000
#define CS_BYTEALIGNWINDOW 0x2000
#define CS_GLOBALCLASS 0x4000
#define CS_IME 0x00010000
```

These identifiers are known as "bit flags" because each sets a single bit in a composite value.

In the provided example, the two identifiers used (CS_HREDRAW and CS_VREDRAW) indicate that windows created based on this class should be entirely repainted when the horizontal or vertical window size changes.

For instance, resizing HELLOWIN's window triggers a repaint of the text string to adjust to the new center of the window. These flags ensure this behavior, and the window procedure is informed of such size changes.