# 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 ensures efficient utilization of existing code structures.


The earlier mention of **HELLOWIN** said it shows the text string in the center of its window.

Actually, the text is centered in the program's **"client area"** — this is the main white space inside the window, below the title bar and inside the borders, like shown in the figure below.


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.

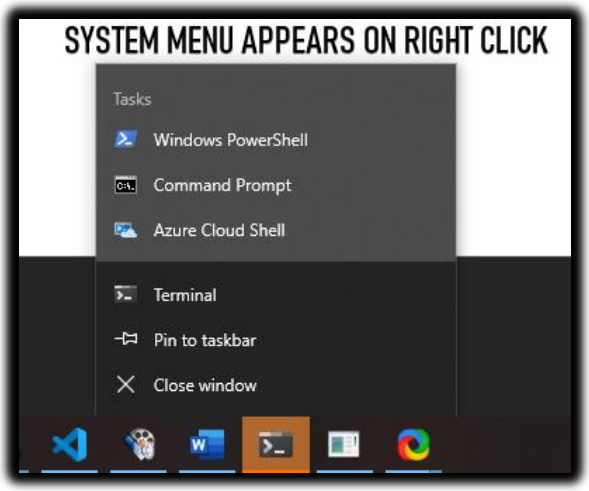
Notably, the program **dynamically adjusts the position of the text string** to the center of its client area when the window size changes.

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.

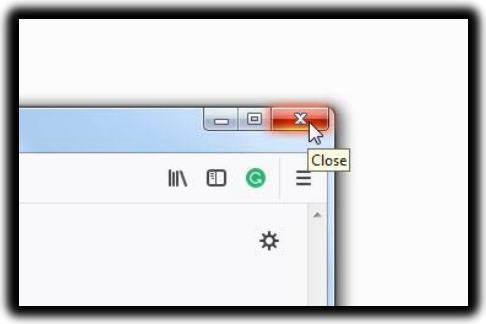
The program's versatility extends to **various interaction modes**, such as **closing** the window to terminate the program.

## You can close a Windows program in three main ways:

1. Choose "Close" from the system menu (click the window icon top-left).




2. Click the “X” button on the top-right of the window.




3. Hover over the window you want to close, double-click the close window icon at the top-left corner.

Later chapters will look closely at HELLOWIN.C. It has a **WinMain function** like the earlier examples, but also a second function called WndProc (or "win prock").

You won't see **WndProc** called directly in the code, but WinMain sets it up. We'll learn why WndProc is important and what it does in the next sections.

# 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

In HELLOWIN.C, you'll see uppercase names like **WM_CREATE** or **IDOK.** These come from Windows *header files*. Many of these names start with two or three letters followed by an underscore (like **WM_** or **ID_**).

The letters at the start show the **group** or **category** the constant belongs to, like messages, commands, or controls.

### 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 in Windows

These are the raw numbers you'd use in code — like **1**, **0x00000001**, or **100**.

```
MessageBox(NULL, "Hello!", "Title", 1); // 1 is a numeric constant (but what does it mean?)
```

**The problem?** That 1 could mean anything — unless you memorize what it stands for (and there are hundreds like this in the Windows API).

You almost **never need to memorize** or look up the actual numbers. Windows takes care of that by giving you all these predefined names in the headers called **symbolic constants** or **uppercase identifiers**, discussed next.

This system **improves readability, maintainability, and reduces errors** — especially when working with things like window styles, message boxes, class styles, and other system-level configurations.

## Using Uppercase Identifiers/Symbolic constants

These are **labels** given to those numeric constants. They make the code more readable and less error-prone. They're usually defined with **#define** in header files.

```
MessageBox(NULL, "Hello!", "Title", MB_OK); // MB_OK is an uppercase identifier
```

### Behind the scenes?

```
#define MB_OK 0x00000000L
```

**MB_OK** is an uppercase identifier, and it represents the numeric constant **0x00000000L**.

So,

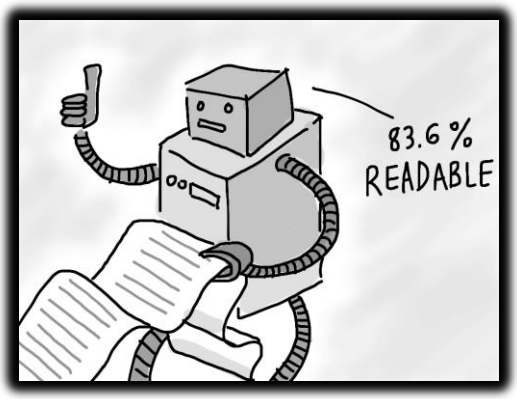
### 🎯 In Summary:

| Term | What It Is | Example |
|---|---|---|
| Numeric Constant | A raw number | `1`, `0xF3`, `100` |
| Uppercase Identifier | A named version of that number | `MB_OK`, `WM_QUIT` |

💡 Think of uppercase identifiers as **nicknames for important numbers** — like speed dial for your code. You don't need to remember the actual digits, just the name.

## Why Use Uppercase Identifiers?

✅**Code Readability:** Uppercase identifiers make code more readable, coz they provide meaningful names for numeric constants, which are difficult to remember. *They turn cryptic numbers into clear, meaningful names.* When you read code with **WS_OVERLAPPEDWINDOW**, it's instantly obvious that it has something to do with a window style. No need to guess what **0xCF00** means EG. ***just know what MB_OK means (a message box with an "OK" button).***

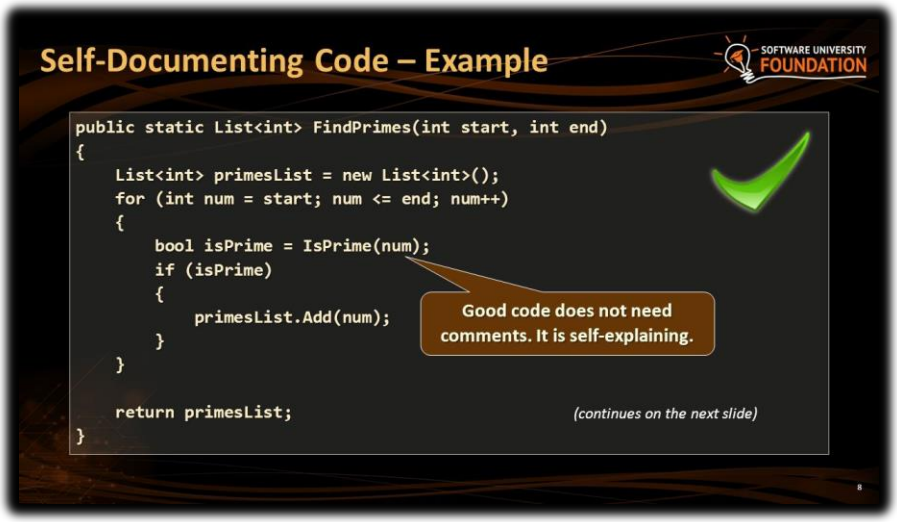


## ✅ Easier Maintenance

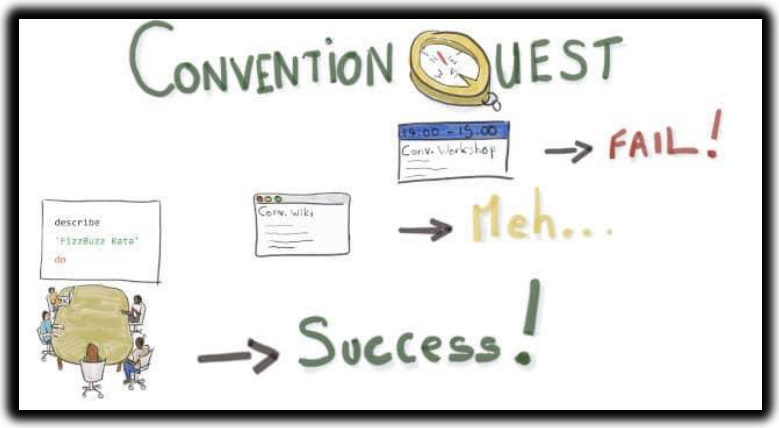
If your code needs updating later, or someone else picks it up, uppercase identifiers make it easier to understand what's going on—no deciphering random numbers.


✅ **Fewer Mistakes:** Let's face it - typing the wrong number is easy. Typing the wrong constant like WM_CLOSE? Less likely. The compiler will even catch it if it's not defined.

✅ **Self-Documenting:** Constants like **MB_YESNO** or **SW_SHOWMAXIMIZED** practically explain themselves. It's like built-in documentation right in the code.




✅ **Consistency with Industry Standards:** ALL CAPS for constants is a long-standing convention, especially in C/C++ and Windows API code. Following it helps your code play nicely with other developers' work and makes collaboration smoother.



# NEW DATA TYPES USED IN HELLOWIN.C

## 🆕 New Data Types in HELLOWIN.C

If you look through the HELLOWIN.C program, you'll notice some weird-looking identifiers — stuff like UINT, WPARAM, and LPARAM. These are custom data types defined in the Windows header files using either typedef or #define.

## 🧠 Why Create These New Types?

Back in the day, **Windows was originally a 16-bit system**, but Microsoft knew they'd eventually move to 32-bit and beyond. So, they introduced these new data types to future-proof code — making it easier to write programs that could transition to newer systems without rewriting everything.

Spoiler: the transition wasn't as smooth as hoped, but the idea was solid.

## 📦 Types of New Data Types

### 1. Shortcut Types (Abbreviations)

Some of these are just easier-to-read aliases:

```
UINT → unsigned int  // Usually a 32-bit value
```

So, when you see UINT, think: "Oh, that's just an unsigned int."

### 2. Special Message Parameters

Others are more Windows-specific:

```
WPARAM → Used for message parameters (used to be a 16-bit WORD)
LPARAM → Another message parameter (used to be a LONG)
```

### In 16-bit Windows:

- **WPARAM** stood for WORD (16-bit unsigned short)
- **LPARAM** stood for LONG (32-bit signed long)

### In modern 32-bit Windows:

- **WPARAM** is now a UINT (32-bit)
- **LPARAM** is still a LONG (also 32-bit)

👉 So the **"W" in WPARAM** doesn't really mean "word" anymore — it's a bit misleading, but it stuck for compatibility.

- **These new types help your code work on both old and new Windows systems.**
- **They also make your code more readable and consistent with Windows API naming conventions.**
- **You don't have to memorize all the types now — just know they're aliases for real C types behind the scenes, and they show up often in Windows programs.**

## 🧱 Data Structures in HELLOWIN.C

In HELLOWIN.C, you'll spot **four key data structures** that come straight from the Windows header files.

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

**You'll use these a lot when building Windows programs** — they help organize info the system needs to run your app smoothly.

### Here's a quick breakdown:

| Structure Name | Where It's Used | What It's For |
|---|---|---|
| `MSG` | `WinMain` | Stores messages from the system (like keyboard input, clicks, etc.) |
| `WNDCLASS` | `WinMain` | Holds info about the window class — things like the background color, icon, and window procedure |
| `PAINTSTRUCT` | `WndProc` | Used when drawing stuff in your window (during painting) |
| `RECT` | `WndProc` | Stores rectangle dimensions — used for drawing and layout calculations |

### 🧠 Why They Matter

- 🖌️ **msg and wndclass** are defined in **WinMain**, where you set up and manage your window.
- 🎨 **ps (paint structure) and rect** are defined in **WndProc**, where you handle drawing and window updates.

---

### ✨ So, What's the Big Deal?

Using these structures:

- ✅ **Keeps your code organized** — instead of passing 10 variables, you pass one structure.
- ✅ **Improves readability** — it's easier to understand what's going on at a glance.
- ✅ **Follows the Windows API standards** — so you're writing code the system expects.

---

You don't need to memorize them now — just know that **Windows programming loves using structured data for readability**, and these are your go-to tools for message handling and drawing.

Let's keep moving when you're ready! 🚀

# 🧩 GETTING AND USING HANDLES

## 💡 What Even *Is* a Handle?

In Windows programming, a **handle** is just a fancy name for a special number (usually 32 bits) that **represents an object** managed by the system.

Think of it like:

- *A receipt or ticket stub that points to something real behind the scenes.*
- *You can't see or touch the object directly (like a window, icon, or brush), but with the handle, Windows knows exactly what you're talking about.*

## 🛠️ What Are Handles Used For?

Handles are used for **almost everything** in Windows:

- 🗺️ **HWND** → handle to a **window**
- 🎨 **HBRUSH** → handle to a **brush**
- 🎯 **HCURSOR** → handle to a **cursor**
- 📄 **HICON** → handle to an **icon**
- 🎼 **HMODULE** → handle to a **module (like a DLL or EXE)**
- 💻 **HDC** → Handle to a device context.

## 🤝 Analogy: File Handles in C

If you've used regular C, you've probably worked with FILE * — a pointer (or handle) to a file. Windows handles work the same way:

- You **don't deal with the object itself**, just the handle that represents it.
- You **pass the handle around** to functions that can use or modify that object.
- A **handle** is just a number (an ID) that tells Windows what object you're working with.
- It acts like a **reference tag** for system resources.
- **You'll use handles constantly** when working with windows, GDI objects, icons, menus, and more.

## 🏗️ How Are Handles Obtained?

You don't just make up handles — Windows gives them to you.
Handles are usually returned by Windows API functions when you create or load a system resource.

Here's how it works:

## 📦 Example 1: Creating a Window

When you call:

```
HWND hwnd = CreateWindow(...);
```

👉 Windows creates a new window behind the scenes and gives you back a handle to that window — **stored in hwnd.**

You use this handle later to show, move, resize, or destroy the window.

## 🎨 Example 2: Loading an Icon

```
HICON hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

👉 This loads a predefined icon (in this case, the default application icon), and gives you a handle (hIcon) that represents it.

## 🧠 In Summary

You get handles by calling Windows API functions like CreateWindow, LoadIcon, LoadCursor, etc.

Windows creates or loads the actual object, and gives you a handle so you can work with it.

You never work with the raw object directly — always through the handle.

It's like ordering food at a restaurant — you don't go into the kitchen, you just get a number and wait for your order to be called 😎

## 🛠️ How Are Handles Used?

Once you've got a handle, you can **use it to control or interact with the object it points to.** Think of it like having the **remote control** — the handle is your way of saying,

*"Hey Windows, I want to do something with that thing you gave me earlier."*

## 📌 Example 1: Show a Window

```
ShowWindow(hwnd, SW_SHOW);
```

- **hwnd** is a handle to the window (from CreateWindow).
- **SW_SHOW** tells Windows to make the window visible.

You're using the handle to target that specific window and tell it what to do.

## 🎯 Final Thoughts on Handles in Windows Programming

### ✅ Advantages of Using Handles

- **Simplicity Without the Details**
  Handles let you work with complex Windows objects **without needing to know how they're built internally**.
  You just get a reference (the handle), and let Windows handle (pun intended) the rest.
- **Object Sharing Made Easy**
  Different parts of your program — or even different programs — can **share access to the same object** using the handle.
  Think of it like sharing a link instead of copying the whole file.

---

### ⚠️ Disadvantages of Using Handles

- **You Have to Track Them**
  Every handle you get is your responsibility. **If you don't clean them up (close/delete), they'll stay in memory**, leading to resource leaks.
- **Handle Leaks = Resource Problems**
  Just like forgetting to close a file in regular C, forgetting to release a handle can cause your app (or even the system) to slow down or act weird.

---

### 🧠 Conclusion: Handles Are Essential

- **Handles are your gateway** to working with all sorts of system resources — windows, icons, files, devices, and more.
- They **keep your code simple** by abstracting away complex internal logic.
- But like all powerful tools, they come with responsibility: **track them, use them carefully, and always release them when done.**

That's it for handles. Simple! 😉

# 🏷️ HUNGARIAN NOTATION — WHAT'S THAT?

Hungarian Notation is a naming style for variables that came out of Microsoft back in the day, thanks to a programmer named **Charles Simonyi**. It's basically a system where you **add a short prefix to your variable names** to show what type of data they hold — kind of like labeling your jars so you don't accidentally put sugar in your spaghetti.

## 🤔 Why Use It?

The main goal is to make code **easier to read** and **harder to mess up.** If you look at a variable and instantly know what kind of data it holds (like an int, a string, or a pointer), you're less likely to use it wrong or break things accidentally.

## 🧩 The Prefixes (aka the "Hungarian" Part)

Here's a breakdown of the most common Hungarian prefixes, what they mean, and a quick example of how they're used:

| Prefix | Stands For | Data Type | Example Name | Meaning |
|---|---|---|---|---|
| `c` | Count | `int` | `cItems` | Number of items in a list |
| `n` | Number | `short` / small integer | `nScore` | Just a general number |
| `i` | Integer (index) | `int` | `iIndex` | Used for loops or indexing |
| `x`, `y` | Coordinates | `int` | `xPos`, `yPos` | X or Y position |
| `cx`, `cy` | Count X/Y | `int` | `cxWidth`, `cyHeight` | Width or height |
| `b`, `f` | Boolean/Flag | `BOOL` (`int`) | `bIsActive`, `fReady` | True/false values |
| `w` | Word | `WORD` (`unsigned short`) | `wVersion` | 16-bit unsigned value |
| `l` | Long | `LONG` | `lDistance` | 32-bit signed value |
| `dw` | Double Word | `DWORD` (`unsigned long`) | `dwSize` | 32-bit unsigned value |

## 🛠️ Other Common Prefixes:

| Prefix | Stands For | Used With | Example | Meaning |
|---|---|---|---|---|
| `fn` | **Function** | Function name | `fnInitApp` | This is a function |
| `s` | **String** | Non-null-terminated string | `sTitle` | A string, but not C-style |
| `sz` | **String-Zero** | Null-terminated string | `szName` | A C-style string (`\0` at end) |
| `h` | **Handle** | Windows handle types | `hWnd`, `hBrush` | A Windows resource handle |
| `p` | **Pointer** | Pointer to anything | `pBuffer` | Pointer to some data |

## 💡 Mini Example:

```
int cItems = 10;        // 'c' = count → this variable stores how many items
BOOL bIsReady = TRUE;   // 'b' = bool/flag → true or false
char szName[20];        // 'sz' = zero-terminated string → a C-style string
HWND hWnd;              // 'h' = handle → this is a handle to a window
```

## 🧠 Should You Still Use Hungarian Notation?

That's a good question.

Hungarian Notation was **super helpful in the early days of Windows development** when IDEs weren't great at showing variable info.

These days, with smart code editors and modern practices (like meaningful variable names), it's used less often — **except in legacy WinAPI code** or low-level system programming, where it's still pretty common.

So, while you don't *have* to use it everywhere, **you should definitely understand it** if you're diving into Windows programming, especially with older code or APIs.

## ✅ Benefits of Hungarian Notation

Hungarian Notation isn't just some old-school coding style — it was created with good intentions. Here's why it can be helpful:

### 🧩 1. Improves Code Readability

Prefixes like **i, b, sz,** etc, instantly tell you what kind of data a variable holds. It's like labeling your storage bins — less guessing, more clarity.

```
int iScore;      // Oh hey, that's clearly an integer!
BOOL bVisible;   // And this one's a true/false flag.
```

### 🔧 2. Makes Code Easier to Maintain

When you're reading someone else's code (or your own code from six months ago 👀), Hungarian Notation can give quick context. You don't have to scroll around just to figure out what type a variable is.

### 🛡️ 3. Helps Prevent Mistakes

If you try to mix incompatible types (like adding a string to an int), Hungarian-style names can raise a red flag. It makes the wrong stuff *look* wrong before it becomes a bug.

### ❌ Drawbacks of Hungarian Notation

But like everything in tech, it's not all sunshine and semicolons. Hungarian Notation has its downsides too.

**Use it when it adds clarity. Skip it when it adds clutter e.g.** if you're working with low-level Windows apps, device drivers, or old-school GUI code, you'll meet Hungarian notation.

### 📏 1. Can Make Code Verbose

Adding prefixes to every variable makes names longer — sometimes unnecessarily so. That can clutter up your code, especially when modern IDEs already tell you variable types.

```
int cchStringLength;  // vs just int length;
```

## REGISTERING A WINDOW CLASS (WITH REGISTERCLASS)

Before you can pop a window on the screen in a WinAPI app, you need to **define what kind of window it is**.

That's where **registering a window class comes in** — it's like giving Windows a blueprint so it knows how to build and style your custom window.

Think of it like this: if you're **building houses (windows)**, a ***window class*** is the architectural plan.

It **defines things like** the window's behavior, appearance, and what happens when someone clicks the close button.

## What Does a Window Class Include?

When you register a window class, you're basically filling out a form (the WNDCLASS or WNDCLASSW struct) with all the window's key info:

- **What function should handle messages (like clicks, keystrokes, etc)?**
- **What background color should the window have?**
- **Should it use a custom cursor or icon?**
- **What style(s) should the window use?**
- **What menu (if any) does it come with?**

Here's the struct in action:

```
WNDCLASS wc = {0};

wc.lpfnWndProc   = MyWindowProc;        // Pointer to your window procedure
wc.hInstance     = hInstance;           // Your app's instance handle
wc.lpszClassName = TEXT("MyWindowClass"); // Name of the class
wc.hCursor       = LoadCursor(NULL, IDC_ARROW); // Cursor icon
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);    // Background color
```

If you're using Unicode (which you should be in modern code), use **WNDCLASSW.**

For ANSI/legacy apps, it's **WNDCLASSA.** Most of the time, just use the generic *WNDCLASS macro* — Windows will pick the right one.

---

## Registering the Class with Windows

Once the struct is filled out, call **RegisterClass** to let the operating system know about it:

```
RegisterClass(&wc);
```

Now Windows knows what a **"MyWindowClass"** window looks like and how it behaves.

## Using the Class to Create a Window

With the class registered, you can now create actual windows using it:

```
HWND hWnd = CreateWindow(
    TEXT("MyWindowClass"), // The class name you registered
    TEXT("My First Window"), // Window title
    WS_OVERLAPPEDWINDOW,     // Window style
    CW_USEDEFAULT, CW_USEDEFAULT, 800, 600, // Position and size
    NULL, NULL, hInstance, NULL
);
```

This creates a new window using the **"MyWindowClass" blueprint** — it'll automatically get the cursor, background, icon, and behavior you defined earlier. Unless you override something, those settings stay in place.

---

## TLDR

- Registering a window class = defining a reusable template for creating windows.
- You use a WNDCLASS or WNDCLASSW struct to set up the window's behavior and appearance.
- Once it's registered with RegisterClass(), you can make windows with CreateWindow() or CreateWindowEx().

# 🧩 WINDOW CLASSES & WHY THEY MATTER

In Windows programming, a **window class isn't a "class" in the C++ sense** — it's more like a **template** or **blueprint** for creating windows. You define it once, and then you can use it to create multiple windows that all share the same core behavior and style.

---

## 🏛️ What's in a Window Class?

A window class defines key traits like:

- **How the window looks (background color, icon, cursor)**
- **How it behaves (what happens when it gets a message — like mouse clicks, keystrokes, etc.)**
- **What code handles its events (your custom WindowProc function)**

You define these traits in a structure like **WNDCLASS** or **WNDCLASSEX**, and register that class with the system using RegisterClass() (or RegisterClassEx() for the extended version).

---

## 👀 Why Use a Window Class?

Let's say you want to create multiple windows in your app — like a main window, a few dialog boxes, maybe even some custom tool windows. Instead of redefining their behavior every time, you just:

- ❖ **Define a window class once.**
- ❖ **Use CreateWindow() to spin up new windows based on that class.**

All those windows will **inherit the same behavior** — unless you choose to override something.

This is super helpful when:

- *You're making multiple windows that should act the same.*
- *You want to keep your code clean and reusable.*
- *You need to standardize UI elements (especially in desktop apps or system tools).*

---

## 🧠 TLDR

- A **window class** is a reusable config for creating windows.
- It **defines how a window looks**, behaves, and interacts with the OS.
- You **register it once**, then use it to create windows as needed.
- It **saves time**, keeps your code clean, and makes multi-window apps easier to manage.

## 🔍 Key Fields of the WNDCLASS Structure

The **WNDCLASS struct** is where you define the personality and behavior of your custom window.

When you register a window class using **RegisterClass()**, you pass in this struct to tell Windows how your window should behave, look, and interact.

Let's break it down field by field:

### 🧱 style

Specifies special options for the window class. You can mix different style flags to control behavior like redraws and ownership.

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

- 🔷 **CS_HREDRAW: Redraw window on width changes**
- 🔷 **CS_VREDRAW: Redraw on height changes**
- 🔷 **There are others (like CS_OWNDC for OpenGL apps)**

### 🧠 lpfnWndProc

A **pointer to your window procedure** — the function that handles all the messages Windows sends (like mouse clicks, keypresses, paint requests, etc.).

```
wc.lpfnWndProc = MyWindowProc;
```

💡 This is where the magic happens. You'll define MyWindowProc() to handle messages like WM_PAINT, WM_DESTROY, etc.

### ➕ cbClsExtra

Extra bytes to reserve for the **class as a whole** (rarely used these days, but available if you need to stash extra data per class).

```
wc.cbClsExtra = 0;
```

### ➕ cbWndExtra

Extra bytes to reserve for **each window instance** created from this class.

```
wc.cbWndExtra = 0;
```

👀 You might use this for custom data storage in some advanced cases (or with SetWindowLongPtr()).

### 📦 hInstance

Handle to the application instance (passed into WinMain()).

```
wc.hInstance = hInstance;
```

This tells Windows **which app** owns this class, and is needed for loading icons, menus, and other resources.

### hIcon

The **icon shown in the title bar** and taskbar for windows of this class.

```
wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

You can use LoadIcon() to load a built-in or custom icon.

### hCursor

The cursor that shows when the mouse is over the window.

```
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
```

Common choices: IDC_ARROW, IDC_HAND, IDC_WAIT, etc.

### hbrBackground

Brush used to **paint the background** of the window when it first draws.

```
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
```

COLOR_WINDOW+1 is a default white background. You can also use custom brushes here.

### lpszMenuName

Name of a **menu resource** to attach to windows of this class.

```
wc.lpszMenuName = NULL;  // No menu
```

If your window needs a top menu bar, this is where you link it. The name must match a menu in your resource file.

## 🏷️ lpszClassName

The **name of this window class** — used later when you call CreateWindow() or CreateWindowEx().

```
wc.lpszClassName = TEXT("MyWindowClass");
```

## ✅ Why It Matters

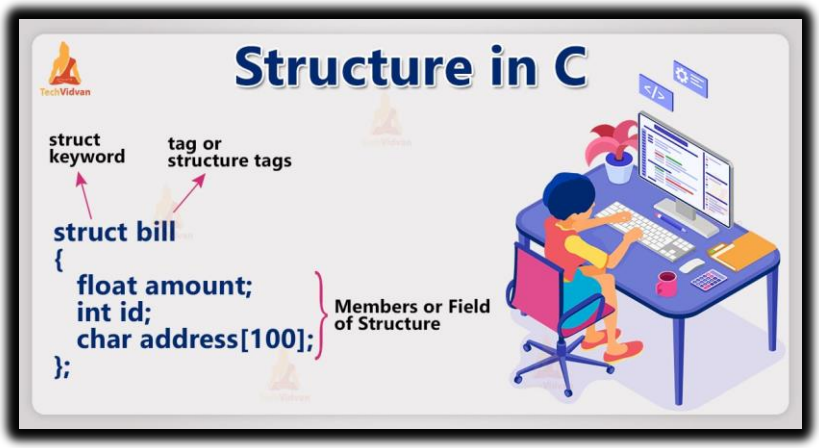
Registering a window class with WNDCLASS is the **first step to creating a real window**. It lets you define a reusable template, so you don't need to repeat yourself for every window you create. It keeps your code modular, consistent, and easier to maintain.

---

## 🧠 TLDR

**WNDCLASS** = your window's DNA

- You **fill it out** with things like icon, cursor, background, and message handler
- Then you **register it** with RegisterClass() before creating windows with CreateWindow()



```c
#include <windows.h>

// Window procedure – basic skeleton
LRESULT CALLBACK MyWindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    switch (uMsg) {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}

// Main function
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    WNDCLASS wc = {0}; // Initialize all fields to 0

    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = MyWindowProc;
    wc.hInstance     = hInstance;
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszClassName = TEXT("MyWindowClass");

    if (!RegisterClass(&wc)) {
        MessageBox(NULL, TEXT("Window class registration failed!"), TEXT("Error"), MB_ICONERROR);
        return 0;
    }

    // At this point, you can call CreateWindow() using "MyWindowClass"
    return 1;
}
```

🧠The code defines a WNDCLASSA if you're using the ASCII (ANSI) version of the Windows API. Otherwise, it's **WNDCLASSW.** The generic WNDCLASS macro switches between the two based on your project settings.

## 🧠 What's Going on in This Code?

### 🎨 style = CS_HREDRAW | CS_VREDRAW

These flags tell Windows:

- **Redraw the window when its width changes** (CS_HREDRAW)
- **Redraw when its height changes** (CS_VREDRAW)

This helps make sure the window updates correctly when resized.

### 🧠 lpfnWndProc = MyWindowProc

This is a **pointer to your window procedure** — a function you write that will handle all the messages Windows sends to this window (like keyboard input, painting, closing, etc).

It looks like this:

```
LRESULT CALLBACK MyWindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    // message handling goes here
}
```

### 📦 hInstance = GetModuleHandle(NULL)

This gets a **handle to your application instance**, which is needed when you're registering the class or loading resources (like icons or menus).

GetModuleHandle(NULL) returns a handle to the executable (.exe) that's running — basically saying "give me a handle to myself."

---

### 🖥️ hIcon = LoadIcon(NULL, IDI_APPLICATION)

Loads a default Windows application icon.

You can replace IDI_APPLICATION with your own icon resource if you have one.

---

### 🖱️ hCursor = LoadCursor(NULL, IDC_ARROW)

Loads a standard arrow cursor — the one we all know and love.

---

### 🎨 hbrBackground = (HBRUSH)(COLOR_WINDOW + 1)

Sets the background color of the window using a predefined system color.

COLOR_WINDOW + 1 is required because Windows treats these color constants as indices, and you must cast it to a HBRUSH.

---

### 📋 lpszMenuName = NULL

This means **no menu** will be attached to this window.

If you want a menu, you'd put the **resource name** of the menu here.

---

### 🏷️ lpszClassName = "MyWindowClass"

This sets the **name** of the window class you're registering. You'll use this exact name later when calling CreateWindow().

---

### 📌 RegisterClassA(&wndclass)

This registers the class with Windows so you can create windows based on it.

- The A suffix means you're explicitly using the ANSI (ASCII) version.
- If the function fails, you can display an error and bail out.

---

### 🚨 Error Handling

If RegisterClassA() fails (returns 0), something went wrong — usually a field is missing or misconfigured.

You might see code like this:

```
if (!RegisterClassA(&wndclass)) {
    MessageBox(NULL, "Registration failed!", "Error", MB_ICONERROR);
    return 0;
}
```

### OR

## 🚨 Handling Errors with RegisterClassA()

Once you've set up your WNDCLASS structure, you call RegisterClassA() to register it. But — what if something goes wrong?

That's where this if statement comes in:

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

### 🔄 !RegisterClassA(&wndclass)

- The **! (NOT) operator** checks if RegisterClassA() failed.
- If it **returns 0**, that means **registration failed**, and the code inside the if block runs.

---

### 📜 MessageBoxA(...)

If something goes wrong, we pop up a **message box** to let the user know.

```
MessageBoxA(
    NULL,                          // No parent window
    "Failed to register window class!", // Message text
    "Error",                       // Title bar text
    MB_ICONEXCLAMATION | MB_OK     // Show a warning icon and an OK button
);
```

- **MB_ICONEXCLAMATION:** Displays a yellow warning icon ❗
- **MB_OK:** Just shows an OK button to dismiss

You could also use MessageBoxW() for Unicode strings, or just MessageBox() and let the macro decide based on your build settings.

### ⛔ return 1;

After showing the error message, we return 1 to signal that the program hit a problem and is exiting early.

- **return 0** usually means success
- **return 1** (or any non-zero value) usually means **"something went wrong"**

### 🧠 Why This Matters

This is basic but essential **error handling**:

- It helps **catch problems early** (like a typo in the class name or a bad resource)
- It makes your program more **robust and debuggable**
- It also gives users a heads-up when something breaks

---

### ✅ TLDR

If the class fails to register:

- A message box explains what went wrong.
- The app exits with an error code.
- This keeps things clean and prevents weird bugs later.

## 📚 Hungarian Notation in WNDCLASS

We're circling back to Hungarian Notation, but now with a **focused lens on how it shows up in the WNDCLASS structure**.

Hungarian Notation **uses prefixes in variable names** to hint at their **type** and sometimes their **purpose**.

**It's not enforced by the compiler**, but it helps developers understand what kind of data they're dealing with — just by looking at the name.

Here's how it applies specifically to the members of the WNDCLASS structure.

### 📋 Hungarian Prefix Table for `WNDCLASS`

| Prefix | Meaning | Example(s) |
|---|---|---|
| `lpfn` | Long pointer to a function | `lpfnWndProc` – pointer to the window procedure |
| `cb` | Count of bytes | `cbClsExtra`, `cbWndExtra` |
| `h` | Handle | `hInstance`, `hIcon`, `hCursor`, `hbrBackground` |
| `lpsz` | Long pointer to a null-terminated string | `lpszMenuName`, `lpszClassName` |

🔷 **"Long pointer"** is a ***legacy term*** from 16-bit Windows. In modern Win32, all pointers are flat (32 or 64-bit), but the naming stuck around.

---

### 🏷️ lpszClassName and lpszMenuName: Why the Prefix Matters

These two fields:

- **lpszClassName**
- **lpszMenuName**

Both use the prefix **lpsz** to tell you:

- **It's a long pointer (now just "pointer") to a**
- **String (sz = string, zero-terminated)**

These strings are:

- The **name of your window class**
- The **name of your menu resource**, if you're using one

---

### 🌐 Unicode Version: WNDCLASSW

There are two versions of the WNDCLASS structure:

- **WNDCLASSA** – ASCII (ANSI strings)
- **WNDCLASSW** – Wide-character (Unicode strings)

In WNDCLASSW, both lpszMenuName and lpszClassName point to **wide-character (wchar_t) strings**, allowing support for international characters and languages.

✅ **Example:**

```
WNDCLASSW wc = {0};
wc.lpszClassName = L"MyUnicodeWindowClass";  // L = wide string literal
RegisterClassW(&wc);
```

💡 Unicode is the default in modern Windows applications — it's more future-proof and international-friendly.

- Hungarian prefixes like lpfn, cb, h, and lpsz give you a quick hint at a **variable's type and role.**
- In WNDCLASS, lpszClassName and lpszMenuName are **pointers to null-terminated strings**, used to name the class and its menu.
- Unicode versions (WNDCLASSW) are the modern standard — **they use wide-character strings** instead of ASCII.

# UNDERSTANDING WINUSER.H AND TYPER ALIASES

### 🗀 What's Up with winuser.h and Type Aliases?

Alright, time to peek under the hood a bit.

The Windows API is massive, and a lot of it is organized into header files. One of the MVPs of this whole system is:

```
#include <winuser.h>
```

This header gives you **tons of core definitions** — including the stuff we've been working with like:

- **WNDCLASSA (for ASCII)**
- **WNDCLASSW (for Unicode)**
- **RegisterClass, CreateWindow, MessageBox, and many others**

## ⚙️ ASCII vs Unicode: Why Two Versions?

In Windows programming, you'll notice a lot of functions and types have an **A** or **W** suffix:

| Type / Function | Meaning |
| --- | --- |
| `WNDCLASSA` | ASCII version (old school) |
| `WNDCLASSW` | Wide-char (Unicode) version |
| `RegisterClassA` | ASCII-only register |
| `RegisterClassW` | Unicode-friendly register |

That's because Windows tries to support both:

- **Legacy ASCII/ANSI programs.**
- **Modern Unicode-aware apps.**

---

## 🧩 Clean Abstraction: Type Aliases

The cool part is that you **don't need to write two versions of your code.** The Windows headers help out with this using **type aliases** and **macros**.

Thanks to this little trick in winuser.h:

```
#ifdef UNICODE
  typedef WNDCLASSW WNDCLASS;
#else
  typedef WNDCLASSA WNDCLASS;
#endif
```

So, when you write:

```
WNDCLASS wc;
RegisterClass(&wc);
```

The compiler automatically swaps in the **W** or **A** version **based on whether Unicode is enabled** in your build.

## What's "Conditional Compilation"?

This is just a fancy way of saying:

*"Only include this code if a certain flag is set."*

In our case:

- If **UNICODE** is defined (either in your build settings or via #define), you get the Unicode (W) versions
- If not, you get the **ASCII** (A) versions

These are controlled by **preprocessor directives** like:

```
#ifdef UNICODE
// Use wide-character (Unicode) versions
#else
// Use ANSI versions
#endif
```

## How Do You Know Which You're Using?

It depends on your project settings (especially in Visual Studio):

- **If UNICODE is defined** → you're using WNDCLASSW, RegisterClassW, etc.
- **If not** → you're on WNDCLASSA, RegisterClassA, and old-school ANSI strings.

---

## TLDR

- winuser.h defines a lot of core Windows stuff.
- It gives you both ASCII (A) and Unicode (W) versions of types like WNDCLASS.
- **You write generic names** (WNDCLASS, RegisterClass), and the compiler picks the right one based on the UNICODE flag.
- This is done using **conditional compilation** with preprocessor logic.

## Type Aliases in WinAPI: Giving Things Nicknames

In programming, sometimes you deal with long, messy types — especially in the Windows API. So, instead of repeating complex type names all over the place, we can **give them nicknames** using something called a **type alias**.

In C, this is done using typedef.

---

## Simple Example

```
typedef HWND WindowHandle;
```

Now anywhere in your code where you'd normally use **HWND**, you can just write **WindowHandle**. Same thing — just easier to read.

It's like calling your friend Christopher... "Chris." 😎

---

## In the Windows API (winuser.h)

The Windows headers do this a *lot*. For example:

```
#ifdef UNICODE
typedef WNDCLASSW WNDCLASS;
#else
typedef WNDCLASSA WNDCLASS;
#endif
```

So, when you write:

```
WNDCLASS wc;
```

...it actually becomes:

- **WNDCLASSW wc;** if UNICODE is defined
- **WNDCLASSA wc;** otherwise

This is all handled **behind the scenes**, so you can just use WNDCLASS and not worry about what kind of characters you're dealing with.

---

## 🧠 What About Pointer Aliases?

Windows also gives you **pointer versions** of these structures:

| Alias | Meaning |
| --- | --- |
| `PWNDCLASS` | Pointer to a `WNDCLASS` |
| `LPWNDCLASS` | Long pointer (same thing in modern C) |
| `NPWNDCLASS` | Near pointer (legacy, ignore this) |

These aliases all map to WNDCLASSA or WNDCLASSW depending on the Unicode setting.

---

## 🎨 Teen-Friendly Analogy for Aliases

Let's say your friend's full name is **"Alexander Jonathan McSomething III"**... but you call him **"AJ."**

That's what typedef is in programming.
Instead of writing out **struct WithALongName** every time, you just say AJ.

Same energy.

---

## Expert-Level Explanation

A **type alias** (via typedef) lets you create a new, user-friendly name for an existing type. It:

- *Improves readability.*
- *Simplifies code maintenance.*
- *Allows abstraction from implementation details.*

In Windows headers, aliases like WNDCLASS, LPWNDCLASS, etc., are **automatically adjusted based on compile-time flags** like UNICODE, using conditional compilation (#ifdef / #endif).

This lets developers write platform-agnostic code without worrying about character encoding differences — a key part of writing **international-ready Windows apps.**

## Code Sample: Using a Type Alias for HWND

Let's clean up and fix your example:

```c
#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;
}
```

## 📜 Longhand vs Shorthand Struct Definition (Clean & Clear)

You might see the WNDCLASS structure written in two different ways. Both do the same thing — they define the structure and its aliases — but one is more verbose, and one is cleaner.

### 📦 Longhand (More Verbose)

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

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

### ⚡ Shorthand (Cleaner and Common)

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

**Both give you:**

- **WNDCLASS** — the structure
- **PWNDCLASS** — a pointer to the structure

But the shorthand is easier to read and more compact, so it's more commonly used in real-world WinAPI code.

## 📜 TLDR: WNDCLASS Wrap-Up

- *WNDCLASS is the blueprint for how your window behaves and looks.*
- *You fill it out with info like which function handles messages (lpfnWndProc), what the cursor looks like, what icon to use, and what menu to load.*
- *You register it using RegisterClass (or RegisterClassEx).*
- *Once registered, you can create windows from it using CreateWindow or CreateWindowEx.*
- *The header file winuser.h handles the Unicode/ASCII versions for you, thanks to #ifdef UNICODE logic.*

---

## ✅ Your Next Step?

Now that you're done with:

- ***Hungarian notation.***
- ***WNDCLASS structure.***
- ***Registering a window class.***
- ***Type aliasing and conditional compilation.***

You're totally set to move into **CreateWindow() and WndProc + the message loop** — the heart of any WinAPI app. 💥🫱❤️‍🩹