# Main Functions – Console vs GUI

## For Console – main() and wmain()

The main() function is an entry point to a C/C++ program. We all know that!

int main(void);

int main(int argc, char \*\*argv);

int main(int argc, char \*argv[]);

But you might not know that **main()** is used only for console programs and it **receives only ASCII characters (7 bits)**.

If we want a program that could receive **wide characters (16 bits)** from the command line, we will use the **wmain()** function prototypes.

int wmain(void);

int wmain(int argc, wchar\_t \*\*argv);

int wmain(int argc, wchar\_t \*argv[]);

## For GUI – WinMain()

For graphical user interface development, we use one of the **WinMain()** function prototypes.

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

LPSTR lpCmdLine, int nCmdShow);

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

PWSTR pCmdLine, int nCmdShow);

int APIENTRY \_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

LPTSTR lpCmdLine, int nCmdShow);

These three function prototypes are used for entry points for Windows GUI applications.

* The WinMain() function's pCmdLine parameter contains the command-line arguments as ANSI characters (8 bits).
* The wWinMain() function's pCmdLine parameter contains the command-line arguments as Unicode characters (16 bits???).
* The \_tWinMain is a C macro that translates to other two above function prototypes, depending whether the \_UNICODE constant is defined.

# Important Notes Before Starting with Win32

## Suffix

Ex (e.g. RegisterClassEx): Updated version of RegisterClass. More compatible with new versions of Windwos, probably more arguments for more controls of the function.

A (e.g. RegisterClassExA): Use ANSI (not ASCII) strings as input and output

W (e.g. RegisterClassExW): Use Unicode strings (Wide characters) as input and output

## Strings

1. If you want Unicode string, use data type *wchar\_t* (a prefix *w* is added to the old *char\_t*). At the same time, add prefix *L* when initializing the string.

*const wchar\_t g\_szClassName[] = L"myWindowClass";*

Also, make sure you zero-initialize the rest of members of structure WNDCLASSEX that you haven't set: *WNDCLASSEX wc = {0};* OR *ZeroMemory(&wc, sizeof(wc));*

1. If you want to use both ANSI and Unicode string (the OS will automatically choose one depending on your project's Unicode setting), don’t use a suffix (e.g. SomeFunction instead of SomeFunctionA or SomeFunctionW). In a similar way, use macro *\_T* to accept your constant string literals.

*ShellExecute(NULL, \_T("open"), \_T("http://stackoverflow.com"), NULL, NULL, SW\_SHOWNORMAL);*

More details about strings in Win32: <https://docs.microsoft.com/en-us/windows/desktop/learnwin32/working-with-strings>

# Window

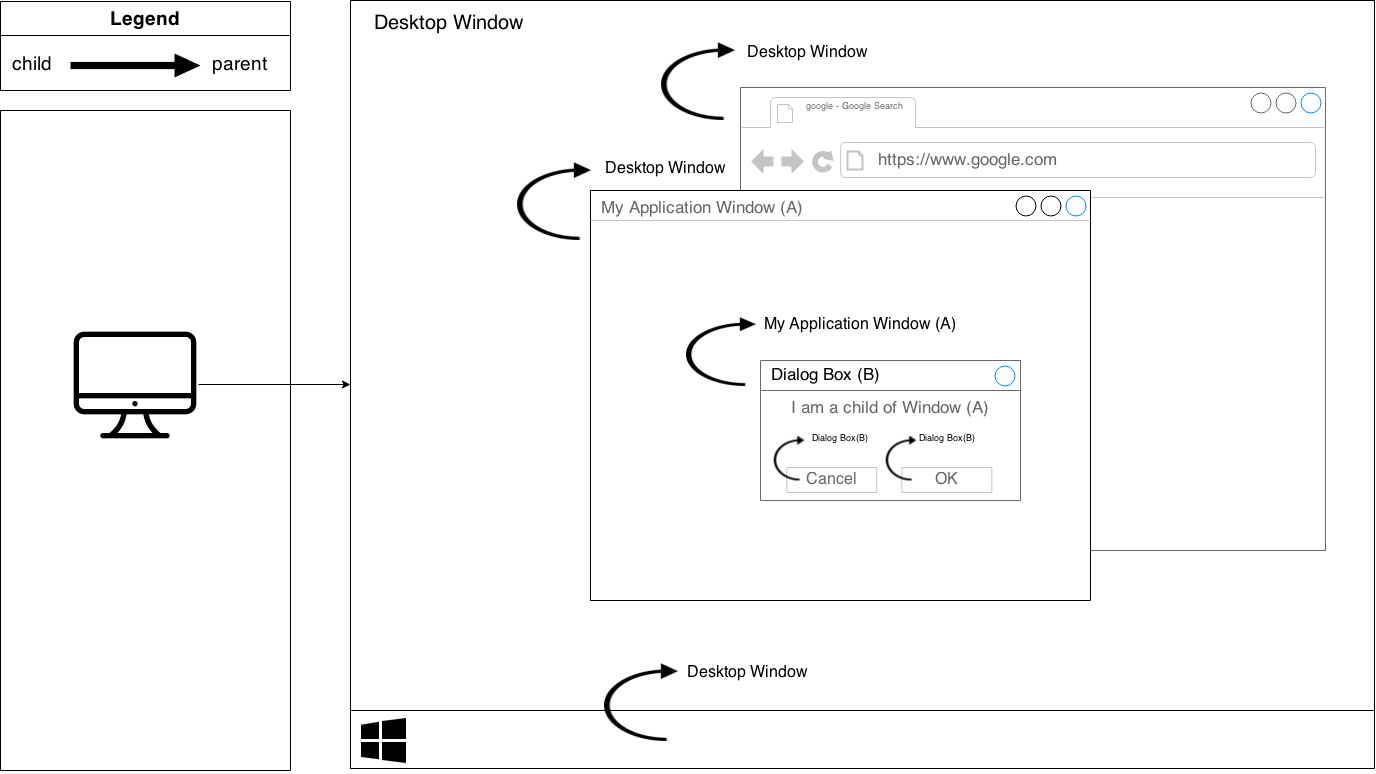
## What Is A Window?

A window is an area of the screen where the application displays output and receives input from the user. **Everything is a window in Windows OS**. Even a button, a static text, an icon; all are windows.

Even the desktop area is part of a window (it is precisely known as the desktop window). Every time you run an application that displays one or more windows, all of its windows become children of the desktop window.

For features of windows such as window types, states, size, and position, check:

<https://docs.microsoft.com/en-us/windows/desktop/winmsg/window-features>



## How to Create a Simple Window?

### 1. WinMain() Function

Every Windows UI application must have at least two functions: the WinMain function and the window procedure.

The [WinMain](https://docs.microsoft.com/en-us/windows/desktop/learnwin32/winmain--the-application-entry-point) function is the **entry point to a Windows UI application (runs first)**. It initializes the application, shows the application window on the screen, and enters the main loop.

In our examples, we use the wWinMain() function, which is used for creating Unicode UI programs.

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

PWSTR pCmdLine, int nCmdShow);

* hInstance is a handle of an instance. It is a 32-bit number identifying the instance of our program within the OS environment. This number is given by Windows when the program starts executing. hInstance is used for things like loading resources and any other task which is performed on a per-module basis.
* hPrevInstance is the handle to the previously run instance of the program, which is a legacy from Win16. In Win32, this no longer applies, so you ignore this parameter by **always setting it NULL**.
* Windows programs can also be started from the command line. The parameters given are stored in pCmdLine parameter. Can we set it to NULL???
* nCmdShow value specifies how the window will be displayed: minimized (nCmdShow=2), maximized (nCmdShow=3), or hidden (nCmdShow=?).

The wWinMain() function terminates when it receives the WM\_QUIT message.

### 2. Registering a Window Class

Before we can create a window, we must register its class within the Windows.In other words, a window is created from a specific window class. This window class defines a set of behaviors that several windows have in common.

To register a [window class](https://docs.microsoft.com/vi-vn/windows/desktop/winmsg/about-window-classes), we must create a WNDCLASS structure. Then call the [RegisterClassExW()](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-registerclassexw).

ATOM RegisterClassExW(const WNDCLASSEXW \* lpWndClassExW);

Exception: Many controls have their window classes already registered. For example, **when we create a button or a static text, we DON’T need to register a window class for them**.

A window class stores information about:

**typedef** **struct** **tagWNDCLASSEXW** {

UINT cbSize; // The size in bytes of WNDCLASSEXW

// Usually set it to sizeof(WNDCLASSEXW)

UINT style; // The style of the class window - [Class Styles](https://docs.microsoft.com/vi-vn/windows/desktop/winmsg/about-window-classes) (CS\_\*)

// Not to be confused with Window Styles (WS\_\*).

// Defines how to update the window after moving or

// resizing it,

// how to process double-clicks of the mouse,

// how to allocate space for the device context,

// and other aspects of the window.

WNDPROC lpfnWndProc; // Pointer to the window procedure for this window class

**int** cbClsExtra; // Amount of extra data allocated for this class in memory

// Usually 0.

**int** cbWndExtra; // Amount of extra data allocated in memory per window

// of this type. Usually 0.

HINSTANCE hInstance; // Handle to the current application instance

// (we got it in the first parameter of wWinMain()).

HICON hIcon; // Handle to large (usually 32x32) icon shown

// when the user presses Alt+Tab.

HCURSOR hCursor; // Handle to cursor displayed over the window

HBRUSH hbrBackground; // Can be a handle to the brush used for painting

// the client area of the window

// Or it can be a color value

LPCWSTR lpszMenuName; // The menu name (can directly assign a string here)

// Or pointer to the string storing the menu resource

// used for the windows with this class.

LPCWSTR lpszClassName; // This time, the window class name.

HICON hIconSm; // Handle to small (usually 16x16) icon shown

// in the taskbar and top left corner of the window

} WNDCLASSEXW, \*PWNDCLASSEXW, NEAR \*NPWNDCLASSEXW, FAR \*LPWNDCLASSEXW;

**These attributes are shared by all windows which use this kind of class**. It is important to note that **a window class must be unique**. This way, you can register a class once and create as many windows as you want from it, without having to specify all those attributes over and over. Most of the attributes you set in the window class can be changed on a per-window basis if desired.

Note: A window class has NOTHING to do with C++ classes.

Now let’s assign specific values to the window class of our first program:

const wchar\_t g\_lpszClassName[] = L"myWindowClass";

The variable above stores the name of our window class, we will use it shortly to register our window class with the system.

// Step 1: Registering the Window Class

WNDCLASSEXW wc = {0};

wc.cbSize = sizeof(WNDCLASSEXW);

wc.style = CS\_HREDRAW | CS\_VREDRAW; // redraw if size changes

wc.lpfnWndProc = WndProc;

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

wc.hInstance = hInstance;

// The icon is retrieved from system resources with the LoadIcon() function

// IDI\_APPLICATION is a value for a default application icon

wc.hIcon = [LoadIcon](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-loadicona)(NULL, IDI\_APPLICATION);

wc.hCursor = [LoadCursor](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-loadcursora)(NULL, IDC\_ARROW);

// Default color value for a window. [More values](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/ns-winuser-tagwndclassexa) for other types of windows

wc.hbrBackground = (HBRUSH)(COLOR\_WINDOW+1);

wc.lpszMenuName = NULL; // no menu for this program

wc.lpszClassName = g\_lpszClassName; // must be unique

wc.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

// If the class register fails

if(!RegisterClassExW(&wc))

return 0;

### 3. Creating a Window

The window is created by calling the [CreateWindowExW()](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-createwindowexw) function.

HWND CreateWindowExW(DWORD dwExStyle, LPCWSTR lpClassName,

LPCWSTR lpWindowName, DWORD dwStyle,

int x, int y, int nWidth, int nHeight,

HWND hWndParent, HMENU hMenu, HINSTANCE hInstance, LPVOID lpParam);

Params:

* dwExStyle is the [extended window style](https://msdn.microsoft.com/5830B16E-CD52-4a1a-A1BD-3AFE66BA5FDD) of the window being created. For a list of possible values.
* lpClassName uniquely identifies the window. It is lpszClassName of WNDCLASSEXW struct under which we registered the window.
* lpWindowName is the window name. Its effect depends on the context – it **can be title of the window in parent windows** or **a label in child windows like button or static text**.
* dwStyle defines the [window style](https://docs.microsoft.com/vi-vn/windows/desktop/winmsg/window-styles) or several styles (when they’re combined using | symbol).
* x and y specify the initial horizontal and vertical position of the window. For an overlapped or pop-up window, x and y are relative to the screen coordinates. For a child window, x and y are relative to the parent window's client area.
* nWidth and nHeight specify the window width and height.
* hWndParent is a handle to the parent window. For windows that do not have parents, we use NULL.
* For a parent window, the hMenu is an optional handle to the menu (set NULL to disable the menu). For a child window, it is a [control identifier](https://docs.microsoft.com/en-us/windows/desktop/winauto/uiauto-controltype-ids) (a unique number defined by user) to notify its parent about events.
* hInstance is a handle to the program instance. (MS: A handle to the instance of the module to be associated with the window.
* lpParam is an optional value passed to the window through the [CREATESTRUCT](https://msdn.microsoft.com/en-us/library/ms632603(v=VS.85).aspx) structure (*lpCreateParams* member) pointed to by the *lParam* param of the [WM\_CREATE](https://msdn.microsoft.com/en-us/library/ms632619(v=VS.85).aspx) message.

Return value: The CreateWindowExW() function **returns a handle to the newly created window**. **If a window parent is specified before that, the newly created window will become a child of the parent**.

After we've created the window and checked to make sure we have a valid handle, it’s time to show the window using [ShowWindow](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-showwindow)(hwnd, nCmdShow) and then update it to ensure that it has properly redrawn itself on the screen with [UpdateWindow](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-updatewindow)(hwnd).

Now let’s create a simple window for our first program:

// Step 2: Creating the window

HWND hwnd;

hwnd = CreateWindowExW(

WS\_EX\_CLIENTEDGE, // The window has a border with a sunken edge

g\_lpszClassName,

L"The title of my window",

WS\_OVERLAPPEDWINDOW, // top-level window with a title bar, border, client area

// (meant to serve as an application's main window)

400, 200, 600, 400,

NULL, NULL, // No owner window, no menu

hInstance,

NULL); // no window-creation data

if(!hwnd)

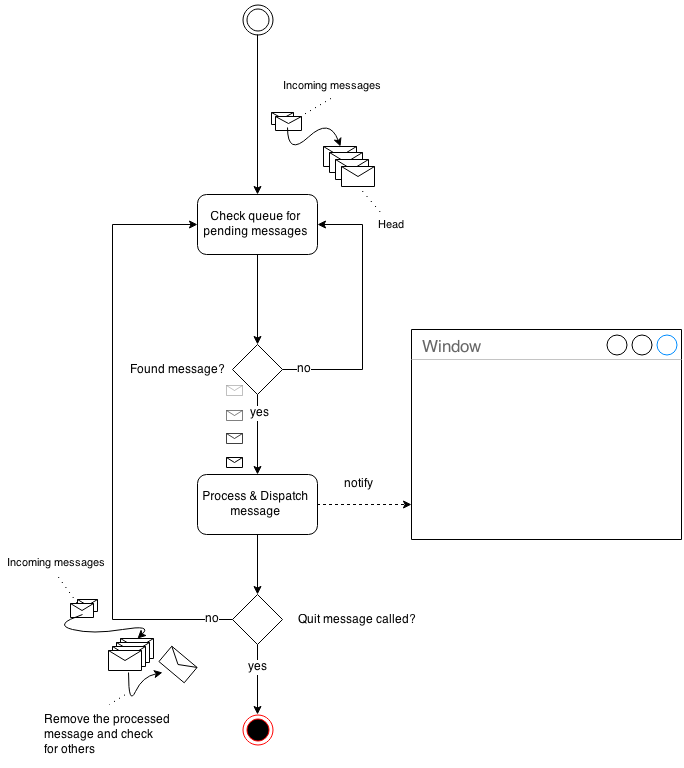
return 0;

ShowWindow(hwnd, nCmdShow);

UpdateWindow(hwnd);

### 4. The Message Loop

Windows OS sends and receives events in an [event-driven architecture](https://en.wikipedia.org/wiki/Event-driven_architecture). The message loop is the heart of the whole program, pretty much everything that your program does passes through this point of control.



[GetMessageW()](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-getmessagew) gets a message from your application's message queue. Any time the user moves the mouse, types on the keyboard, clicks on your window's menu, etc. messages are generated by the system and entered into your program's message queue. By calling GetMessageW() you are requesting the next available message to be removed from the queue and returned to you for processing. If there is no message, GetMessageW() will wait until there is a message, and then returns it to you.

BOOL GetMessageW(LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMax);

Params:

* lpMsg is a pointer to an [MSG](https://msdn.microsoft.com/en-us/library/ms644958(v=VS.85).aspx) structure that receives message information from the thread's message queue.
* hWnd is a handle to the window whose messages are to be retrieved. The window must belong to the current thread.
* **If hWnd is NULL, GetMessageW retrieves messages for any window that belongs to the current thread**, and any messages on the current thread's message queue whose hWnd value is NULL (see the MSG structure). Therefore, if hWnd is NULL, both window messages and thread messages are processed.
* If hWnd is -1, GetMessageW retrieves only messages on the current thread's message queue whose hWnd value is NULL, that is, thread messages as posted by [PostMessage](https://msdn.microsoft.com/en-us/library/ms644944(v=VS.85).aspx) (when hWnd is **NULL**) or [PostThreadMessage](https://msdn.microsoft.com/en-us/library/ms644946(v=VS.85).aspx).
* wMsgFilterMin and wMsgFilterMax is respectively the lowest and highest message value to be retrieved. **If they are both 0, GetMessageW returns all available messages** (that is, no range filtering is performed).

Now move back to our first program:

// Step 3: The message loop

MSG Msg;

while(GetMessageW(&Msg, NULL, 0, 0))

{

TranslateMessage(&Msg);

DispatchMessageW(&Msg);

}

[TranslateMessage()](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-translatemessage) does some additional processing on keyboard events like generating WM\_CHAR messages to go along with WM\_KEYDOWN messages.

[DispatchMessageW()](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-dispatchmessagew) sends the message out to the window that the message was sent to. This could be our main window, or a control, and in some cases a window created behind the scenes by the system or another program. This **isn't something you need to worry** about because all we are concerned with is that we get the message and send it out, the system takes care of the rest and make sure it gets to the proper window.

At the end of the application, the exit code is returned to the system:

return Msg.wParam;

### 5. The Window Procedure

**If the message loop is the heart of the program, the window procedure is the brain**. This is where all the messages that are sent to our window get processed. Every window has an associated window procedure. **The window procedure is called for each message**.

LRESULT CALLBACK WindowProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);

Params:

* hwnd is the handle of the current window being processed – the one that the message applies to. This is important because you might have two or more windows of the same class and they will use the same window procedure. The difference is that hwnd will be different depending on which window it is. For example, when we get the WM\_CLOSE message, we destroy the window. Because this window has a unique hwnd, any other windows will not be affected.
* message is a [system-defined message](https://msdn.microsoft.com/en-us/library/windows/desktop/ms644927(v=vs.85).aspx#system_defined). This is used to determine the nature of the event, like keyboard input, mouse input, window focus acquisition/loss, etc.
* wParam and lParam contain additional message information. The contents of these parameters depend on the value of the uMsg parameter.

Return value: The result of the message processing, depending on the message sent.

CALLBACK is a preprocessor macro for \_\_stdcall – a calling convention. When WindowProc is associated with CALLBACK, it becomes a function which will be called by some other function on occurrence of an event.

Now let’s define two basic window procedures for our first program:

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch(msg)

{

case WM\_CLOSE:

DestroyWindow(hwnd);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hwnd, msg, wParam, lParam);

}

return 0;

}

**Close a window:**

WM\_CLOSE is sent when the user presses the Close button [X] or types Alt-F4. This will cause the window to be destroyed by default, but we often want to handle it explicitly because this is the perfect spot to do cleanup checks, or ask the user to save files, etc. before exiting the program.

BOOL DestroyWindow(HWND hWnd);

When we call [DestroyWindow()](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-destroywindow), the system sends the WM\_DESTROY message to the window getting destroyed and then destroys any remaining child windows before finally removing our window from the system.

**Terminate the message loop:**

void PostQuitMessage(int nExitCode);

When we call [PostQuitMessage()](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-postquitmessage), the system posts the WM\_QUIT message to the message loop. **We never receive this message because it causes GetMessageW() to return FALSE (we called it in our message loop)**. When that happens, we stop processing messages and return the application exit code – this value is used as the wParam parameter of the WM\_QUIT message.

For a list of most common messages a Win32 application usually handles, check the ‘Messages’ chapter.

# Messages

If I or someone else refers to handling a message, they mean to add it into the WndProc() of your window class.

## Mouse Clicks

WM\_LBUTTONDOWN handler for left clicks

WM\_ RBUTTONDOWN handler for right clicks

WM\_ MBUTTONDOWN handler for middle clicks

## WM\_CREATE

### Center a Window

The SetWindowPos() function changes the position, size, and [Z order](https://en.wikipedia.org/wiki/Z-order) (overlapping order) of a child, pop-up, or top-level window.

BOOL SetWindowPos(HWND hWnd, HWND hWndInsertAfter,

int X, int Y, int cx, int cy,

UINT uFlags);

Params:

* hWnd is handle to the window.
* hWndInsertAfter is handle to the window to precede the positioned window in the Z order. This parameter must be a window handle or one of the [following values](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-setwindowpos).
* X and Y is the new position of the left side and top (respectively) of the window, in client coordinates.
* cx and cy is the new width and height (respectively) of the window, in pixels.
* uFlags is the window sizing and positioning flags. This parameter can be a combination of the [following values](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-setwindowpos).

Also, we need two other function called [GetWindowRect()](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-getwindowrect) and [GetSystemMetrics()](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-getsystemmetrics) to get the dimensions of the window and of the screen.

Example:

// Position the application window in the center of the screen

void centerWindow(HWND hwnd)

{

// Step 1: Get the dimensions of the window and of the screen.

RECT rc = {0};

// Retrieve the dimensions of the bounding rectangle of the specified window

GetWindowRect(hwnd, &rc);

// Compute the window width and height

int winWidth = rc.right - rc.left;

int winHeight = rc.bottom - rc.top;

// Determine the screen width and height

int screenWidth = GetSystemMetrics(SM\_CXSCREEN);

int screenHeight = GetSystemMetrics(SM\_CYSCREEN);

// Step 2: Changes the position of the top-level window

SetWindowPos(hwnd,

HWND\_TOP, // places the window at the top of the Z order

(screenWidth - winWidth)/2, // new position of the left side of the window

(screenHeight - winHeight)/2, // new position of the top of the window

0, 0, // new width and height of the window (cx and cy params)

SWP\_NOSIZE); // retains the current size (ignores cx and cy params)

}

In the switch case of WndProc(), we add:

case WM\_CREATE:

CenterWindow(hwnd);

break;

### Create Multiple Windows

**Each window is created from a specific window class**. A window class defines a set of behaviors that several windows have in common. Some classes are already predefined in the system. A custom window class must be registered. After that, we can create windows of this new window class. A window is created using the CreateWindowW() function.

**Each window has a window procedure**. It is a function that is called by the OS when users interact with the window. In the following example, we create three windows: one parent window and two child windows.

The main window creates the child windows in response to the WM\_CREATE message.

// Window procedure for Panel

LRESULT CALLBACK WndProc\_Panel(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch(msg)

{

// If left click on the background of the two child windows

case WM\_LBUTTONUP:

[MessageBeep](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-messagebeep)(MB\_OK); // hear the Windows default beep sound

break;

}

return DefWindowProcW(hwnd, msg, wParam, lParam);

}

// Register the Window Class for RedPanel window

void RegisterClass\_RedPanel(void)

{

WNDCLASSW wc = {0};

wc.lpszClassName = g\_lpszClassName\_RedPanel;

wc.hbrBackground = CreateSolidBrush(RGB(255, 0, 0)); // red background

wc.lpfnWndProc = WndProc\_Panel;

wc.hCursor = LoadCursor(0, IDC\_ARROW);

RegisterClassW(&wc);

}

// Register the Window Class for BluePanel window

void RegisterClass\_BluePanel(void)

{

WNDCLASSW wc = {0};

wc.lpszClassName = g\_lpszClassName\_BluePanel;

wc.hbrBackground = CreateSolidBrush(RGB(0, 0, 255)); // blue background

wc.lpfnWndProc = WndProc\_Panel;

wc.hCursor = LoadCursor(0, IDC\_ARROW);

RegisterClassW(&wc);

}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch(msg)

{

case WM\_CLOSE:

DestroyWindow(hwnd);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

case WM\_CREATE:

RegisterClass\_RedPanel();

HWND hWnd\_RedPanel = CreateWindowW(g\_lpszClassName\_RedPanel, NULL,

WS\_CHILD | WS\_VISIBLE,

20, 20, 80, 80, // (20, 20) is relative to parent window's client area

hwnd, (HMENU)1, NULL, NULL);

RegisterClass\_BluePanel();

HWND hWnd\_BluePanel = CreateWindowW(g\_lpszClassName\_BluePanel, NULL,

WS\_CHILD | WS\_VISIBLE,

120, 20, 80, 80,

hwnd, (HMENU)2, NULL, NULL);

break;

default:

return DefWindowProc(hwnd, msg, wParam, lParam);

}

return 0;

}

## WM\_SIZE

### Resize All Child Windows When Their Parent Is Resized

// This application-defined callback function receives each child window handle

// We need this function to resize our child functions when the parent window is resized

BOOL CALLBACK EnumChildProc(HWND hwndChild, LPARAM lParam)

{

LPRECT rectParent = (LPRECT)lParam;

if(hwndChild == hEditingArea)

{

// Change size and position of the EditingArea child window

MoveWindow(hwndChild,

10,

10,

rectParent->right / 3,

rectParent->bottom - 70,

TRUE);

}

else if(hwndChild == hDrawingArea)

{

// Change size and position the DrawingArea child window

MoveWindow(hwndChild,

// place DrawingArea at the right side of EditingArea with 20 pixels in distance

(rectParent->right / 3) + 20,

10,

// DrawingArea is 30 pixels in distance from the right side of the parent window

(rectParent->right) - (rectParent->right / 3) - 30,

rectParent->bottom - 70,

TRUE);

}

ShowWindow(hwndChild, SW\_SHOW);

return TRUE;

}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch(msg)

{

case WM\_CLOSE:

DestroyWindow(hwnd);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

case WM\_CREATE:

// Create EditingArea window

RegisterClass\_EditingArea();

hEditingArea = CreateWindowW(g\_lpszClassName\_EditingArea, NULL,

WS\_CHILD | WS\_VISIBLE | WS\_BORDER | WS\_VSCROLL | WS\_HSCROLL

| ES\_LEFT | ES\_MULTILINE | ES\_AUTOHSCROLL | ES\_AUTOVSCROLL

| ES\_WANTRETURN, // Edit Control Styles

20, 20, 300, 700,

hwnd, (HMENU)ID\_EDITING\_AREA, NULL, NULL);

ShowWindow(hEditingArea, SW\_SHOW);

// Create DrawingArea window

RegisterClass\_DrawingArea();

hDrawingArea = CreateWindowW(g\_lpszClassName\_DrawingArea, NULL,

WS\_CHILD | WS\_VISIBLE | WS\_BORDER,

400, 20, 400, 700,

hwnd, (HMENU)ID\_DRAWING\_AREA, NULL, NULL);

ShowWindow(hDrawingArea, SW\_SHOW);

break;

// When main window changes size

case WM\_SIZE:

RECT rectClient;

// Retrieve the coordinates of a window's client area

GetClientRect(hwnd, &rectClient);

// Enumerate child windows that belong to the specified parent window

// by passing the handle to each child window

EnumChildWindows(hwnd, EnumChildProc, (LPARAM)&rectClient);

break;

default:

return DefWindowProc(hwnd, msg, wParam, lParam);

}

return 0;

}

[MoveWindow](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-movewindow)

[GetClientRect](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-getclientrect)

[EnumChildWindows](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-enumchildwindows)

## WM\_COMMAND

### Button

Button is a simple control with a text label. It is used to trigger an action. When we click on a button, it sends a WM\_COMMAND message to its parent window. The low-order word of the wParam parameter contains the control identifier.

<http://zetcode.com/gui/winapi/controls/>

<https://docs.microsoft.com/en-us/windows/desktop/controls/button-styles>

### Dialog Box for Opening File

<http://zetcode.com/gui/winapi/dialogs/>

<https://docs.microsoft.com/en-us/windows/desktop/dlgbox/open-and-save-as-dialog-boxes>

## WM\_PAIN

### BeginPaint/EndPaint vs GetDC/ReleaseDC

Which is better, to use [BeginPaint](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-beginpaint)/[EndPaint](https://docs.microsoft.com/vi-vn/windows/desktop/api/winuser/nf-winuser-endpaint), or to use [GetDC](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-getdc)/[ReleaseDC](https://docs.microsoft.com/vi-vn/windows/desktop/api/winuser/nf-winuser-releasedc)?

It depends! If you are handling WM\_PAINT, you should use BeginPaint/EndPaint. Otherwise, you should use GetDC/ReleaseDC.

**Windows sends WM\_PAINT to your message queue as soon as a new area of the window’s client area becomes invalidated.**

If Windows finds an invalidated area, it sets a flag in the message pump indicating that a new WM\_PAINT is waiting for processing. If no messages are waiting in the queue, it sends the WM\_PAINT to the window procedure.

An area of the client area of the window becomes invalidated in many ways. For example, when a portion of the window is covered by another window, Windows combines the area covered by the other window with the currently invalidated area of the window. In addition, you can [invalidate an area of the window](https://docs.microsoft.com/en-us/windows/desktop/gdi/invalidating-the-client-area) using functions like [**InvalidateRect**](https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-invalidaterect)(to invalidate a rectangular area.) Those functions add the area specified to the currently invalidated area (i.e. combine the new area with the currently invalidated area of the window.)

Remember that, Windows continues sending WM\_PAINT messages to your message queue as long as there’s an invalidated area. So, you should validate the client area before leaving the WM\_PAINT handler block. That’s why it is recommended using BeginPaint/EndPaint in WM\_PAINT message handler because EndPaint does validate the entire client area of the window.

The following is a pseudo-code for EndPaint:

BOOL EndPaint(...)

{

. . .

validate client area

e.g. call ValidateRect()

release the DC

do the necessary finalization

. . .

}

Therefore, using GetDC/ReleaseDC in WM\_PAINT would clog the message pump with a sequence of WM\_PAINT messages that would divert your application from continuing its work, unless you validate the client area before jumping out of WM\_PAINT handler.

On the other hand, using BeginPaint/EndPaint outside the WM\_PAINT handler would validate the client area each time you call EndPaint. And that would prevent WM\_PAINT from arriving to your message queue.

Another interesting point to consider is the following block of code inside the window procedure:

switch (uMsg)

{

. . .

case WM\_PAINT:

. . .

return 0;

. . .

}

Why is the previous code considered wrong? Yes, you are right. It leaves the WM\_PAINT with neither validating the client area nor passing the message to the default window procedure.

The default window procedure actually did nothing interesting inside the WM\_PAINT. However, it is required to pass the WM\_PAINT to the default window procedure if you are not going to handle WM\_PAINT or you're not validating the client area inside the WM\_PAINT handler. That’s because Windows simply calls BeginPaint and EndPaint in pair. Thus, validates the client area.

case WM\_PAINT:

BeginPaint(hWnd, &ps);

EndPaint(hWnd, &ps);

return 0;

**Thus, you should use BeginPaint/EndPaint in WM\_PAINT only, and GetDC/ReleaseDC in all other places in your code.**

MUST read to understand more: <https://stackoverflow.com/questions/5841299/difference-between-getdc-and-beginpaint>

<https://docs.microsoft.com/en-us/windows/desktop/gdi/using-the-getdc-function>

<http://www.functionx.com/win32/Lesson12.htm>