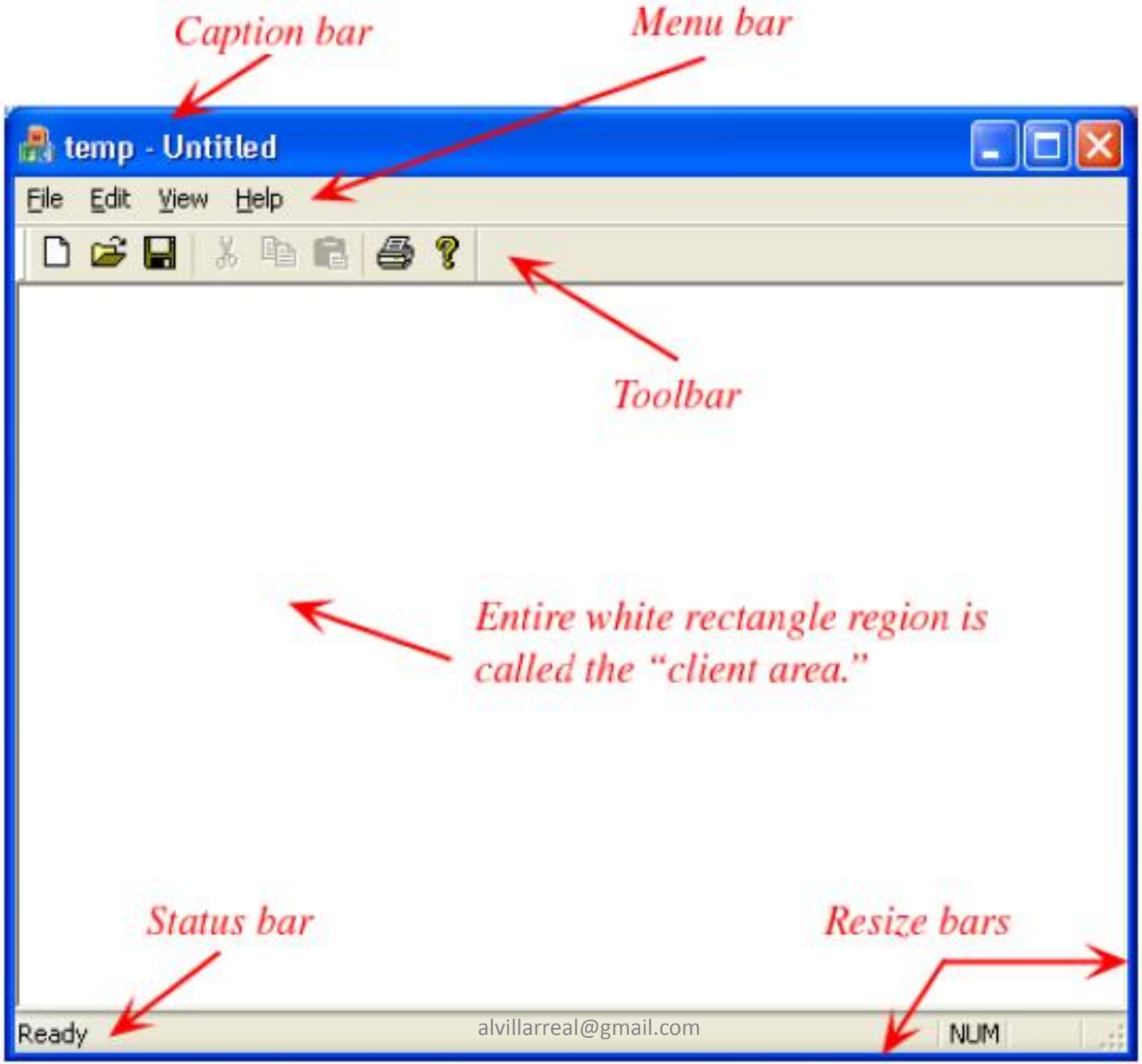
Win32 API (Application Programming Interface).

Programación 2

MGTI Alejandro Villarreal Mojica



Basic window creation

- Unfortunatley, creating a basic window using the Win32 API is not as simple as just executing a single function, due to the flexibility in how we can create a window.

Basic window creation

- The main steps involved are:

1. Define a window class
2. Register a window class
3. Create the window
4. Show the window
5. Setup a callback procedure

Basic window creation

- The window class defines the overall 'look and feel' of the window that we want to create. For example we can set the background color, the style of mouse pointer, menus that we might be using, etc..

1. To create a window class an fill out its attributes is done like so:

```
WNDCLASSEX wClass;
ZeroMemory(&wClass,sizeof(WNDCLASSEX));

wClass.cbClsExtra=NULL;
wClass.cbSize=sizeof(WNDCLASSEX);
wClass.cbWndExtra=NULL;
wClass.hbrBackground=(HBRUSH)COLOR_WINDOW;
wClass.hCursor=LoadCursor(NULL,IDC_ARROW);
wClass.hIcon=NULL;
wClass.hIconSm=NULL;
wClass.hInstance=hInst;
wClass.lpfnWndProc=(WNDPROC)WinProc;
wClass.lpszClassName="Window Class";
wClass.lpszMenuName=NULL;
wClass.style=CS_HREDRAW|CS_VREDRAW;
```

Quite a few of the parameters did not even require setting up. But, I make use of the intellisense feature bundled with Visual Studio to make sure that I haven't forgotten about anything.

cbClsExtra	Additional parameters
cbSize	Specifies the size of the window class
cbWndExtra	Additional parameters
hbrBackground	Sets background color for the window
hCursor	The cursor that will appear in the window
hIcon	Icon for the window
hIconSm	Small icon for the window
hInstance	Handle to the application instance
lpfnWndProc	The callback procedure (more on that later)
lpszClassName	The unique name of the window class
lpszMenuName	Used for menus
style	The look and feel of the window

One thing that may be handy to know at this stage is the different cursor types that are available, specified in the 'hCursor' parameter. You can choose what type of cursor is displayed when the mouse hovers over your window.

2. Register the window class.

```
// Register
RegisterClassEx(&wClass);

// Or Register and handle error
if(!RegisterClassEx(&wClass))
{
     int nResult=GetLastError();
     MessageBox(NULL,
          "Window class creation failed",
          "Window Class Failed",
          MB_ICONERROR);
}
```

3. Create the window

```
HWND hWnd=CreateWindowEx(NULL,
          "Window Class",
          "Windows application",
          WS_OVERLAPPEDWINDOW,
          200,
          200,
          640,
          480,
          NULL,
          NULL,
          hInst,
          NULL);
```

alvillarreal@gmail.com

Create the window

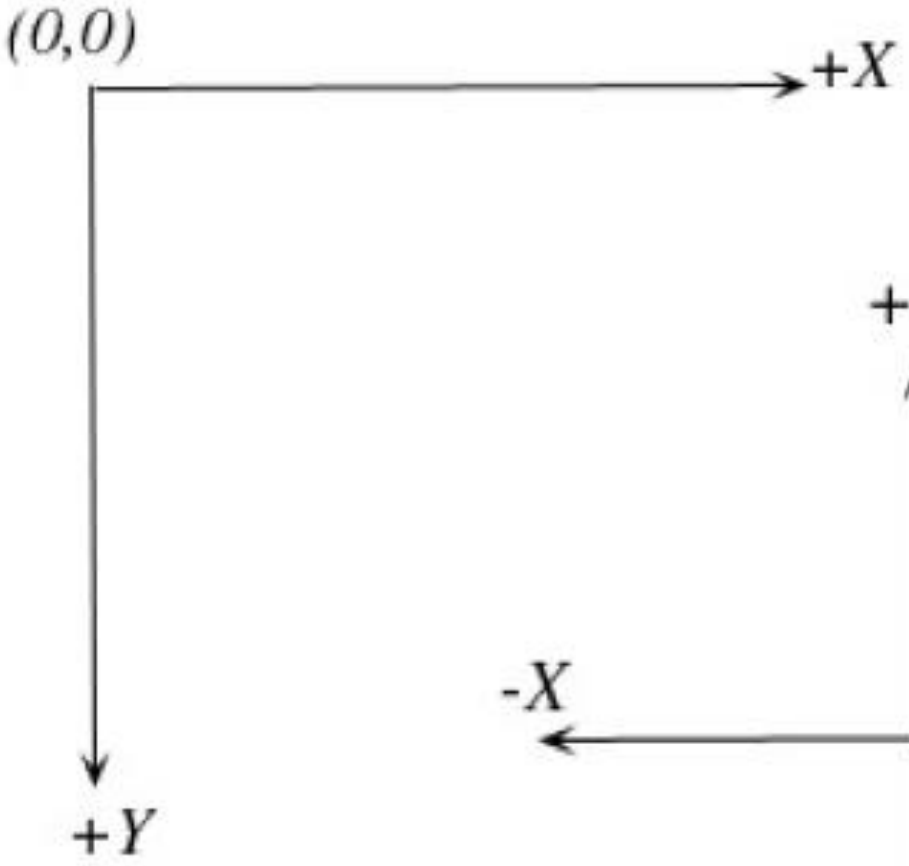
```
HWND CreateWindow(
    LPCTSTR lpClassName,
    LPCTSTR lpWindowName,
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    LPVOID lpParam
);
```

- lpClassName:
  - The name of the WNDCLASS instance to use in order to create the window (i.e., the name we specified for wc.lpszClassName).
- lpWindowName:
  - A unique string name to give the window we are creating. This is the name that will appear in the window's title/caption bar.
- dwStyle:
  - A combination of style flags specifying how the window should look. Typically, this is set to WS_OVERLAPPEDWINDOW, which is a combination of styles WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and S_MAXIMIZEBOX. See the Win32 API documentation for complete details on window styles.
- x:
  - The x-coordinate position of the upper-left corner of the window, relative to the screen, and measured in pixels.
- y:
  - The y-coordinate position of the upper-left corner of the window, relative to the screen, and measured in pixels.

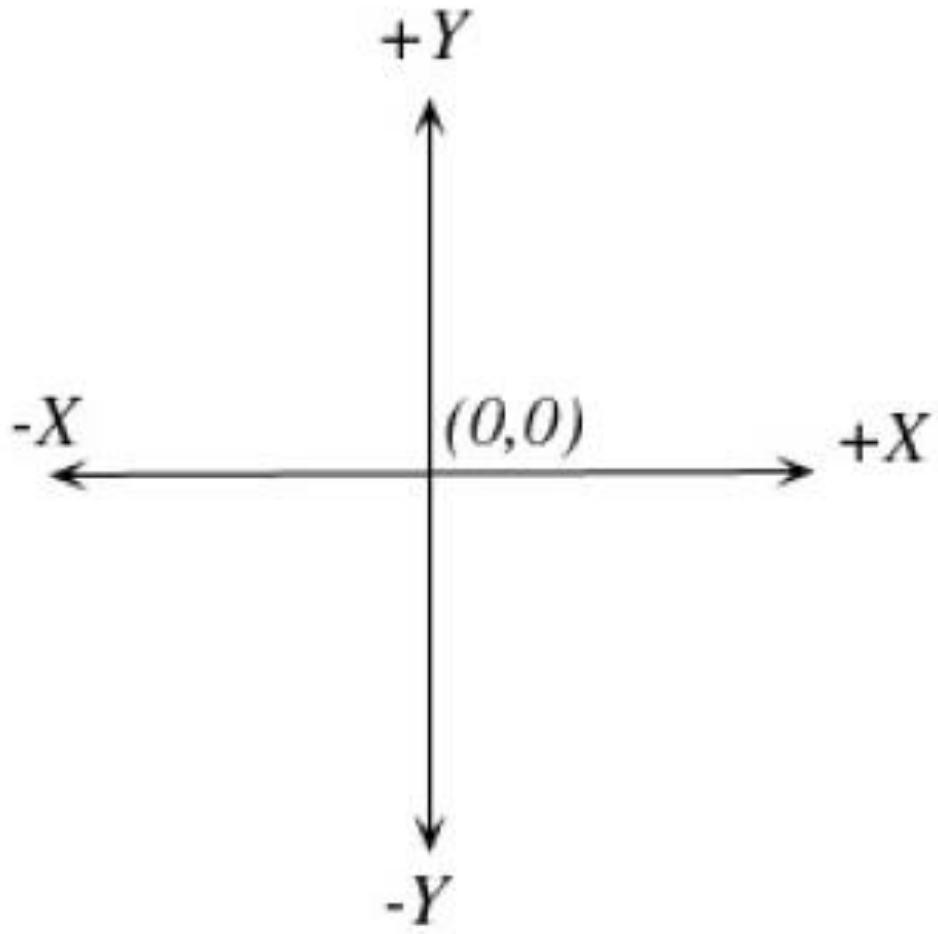
- nWidth:
  - The width of the window, measured in pixels.
- nHeight:
  - The height of the window, measured in pixels.
- hWndParent:
  - Handle to a parent window. Windows can be arranged in a hierarchical fashion.
  - For example, controls such as buttons are child windows, and the window they lie on is the parent window.
  - If you wish to create a window with no parent (e.g., the main application window) then specify null for this value.

- hMenu:
  - Handle to a menu which would be attached to the window. We will examine menus in later chapters. For now we set this to null.
- hInstance:
  - Handle to the application instance the window is associated with.
- lpParam:
  - A pointer to optional user-defined data; this is optional and can be set to null.

screen space



Screen Space
System



Typical Coordinate
System

4. Display the Window

```
/* Mostrar la ventana */
ShowWindow(hWnd,nShowCmd);
```

we create a 'main loop' for our program.

- Finally, enter the message loop. After you have created the main window of your application, you are ready to enter the message loop.
- The message loop will constantly check for and handle messages as the message queue gets filled. The message loop will not exit until a quit message (WM_QUIT) is received.

Main loop

```
MSG msg;
ZeroMemory(&msg,sizeof(MSG));

while(GetMessage(&msg,NULL,0,0))
{
     TranslateMessage(&msg);
     DispatchMessage(&msg);
}
```

ZeroMemory

- ZeroMemory
  - is a Win32 function that clears out all the bits of a variable to zero, thereby “zeroing out” the object; the first parameter is a pointer to the object to zero out and the second parameter is the size, in bytes, of the object to zero out.

The MSG Structure

- A message in Windows is represented with the following MSG structure:

```
struct MSG {
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
};
```

The MSG Structure

- hwnd:
  - This member is the handle to the window for which the message is designated.
- message:
  - This member is a predefined unique unsigned integer symbol that denotes the specific type of message.
- wParam:
  - A 32-bit value that contains extra information about the message. The exact information is specific to the particular message.

The MSG Structure

- lParam:
  - Another 32-bit value that contains extra information about the message. The exact information is specific to the particular message.
- time:
  - The time stamp at which time the message was generated.
- pt:
  - The (x, y) coordinates, in screen space, of the mouse cursor at the time the message was generated.

Here are some example message types, which would be placed in message:

- WM_QUIT: This message is sent when the user has indicated their desire to quit the application (by pressing the close ‘X’ button, for example).
- WM_COMMAND: This message is sent to a window when the user selects an item from the window’s menu. Some child windows, such as button controls, also send this message when they are pressed.
- WM_LBUTTONDBLCLK: This message is sent to a window when the user double-clicks with the left mouse button over the window’s client area.
- WM_LBUTTONDOWN: This message is sent to a window when the user presses the left mouse button over the window’s client area.
- WM_KEYDOWN: This message is sent to the window with keyboard focus when the user presses a key. The wParam of the message denotes the specific key that was pressed.
- WM_SIZE: This message is sent to a window when the user resizes the window.

Ciclo de Mensajes-Ventana Cliente

```
while(GetMessage(&msg,NULL,0,0))
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }
```

Ciclo de Mensajes-Diálogo No Modal

```
while(GetMessage(&msg, 0, 0, 0 ))
{
   if(ghDlg == 0||!IsDialogMessage(ghDlg, &msg ))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
}
```

- Here we create a message handler.
- Our instance of MSG gets filled with the GetMessage() call.
- TranslateMessage() translates virtual key messages to character messages.
- And finally DispatchMessage() sends any messages to the callback procedure, where we can then handle the messages that are being received appropriately.

The callback procedure

- The callback procedure handles all of the windows messages that will be thrown at your application. Things like windows resizing, clicking menus and buttons, keyboard input, etc..

- Every time someone puts any sort of input into your application, the callback procedure jumps into action. From there, depending on what input was receive, your program will respond in some way.

The callback procedure

- The callback procedure prototype looks like this;

```
LRESULT CALLBACK WinProc(
      HWND hWnd,
      UINT message,
      WPARAM wParam,
      LPARAM lParam);
```

The callback procedure

```
LRESULT CALLBACK WinProc(HWND hWnd,UINT
message,WPARAM wParam,LPARAM lParam);
```

The first parameter is the handle to the window that we want to monitor, the second is the message that we are currently receiving, the third and fourth parameters are, essentially, additional information for the received message.

For example, it is no good to know that we have just received a key press if we don't know what key was pushed, is it?

The callback procedure

```
// Esta función es llamada por la
// función del API DispatchMessage()
LRESULT CALLBACK WinProc(HWND hWnd,UINT msg,WPARAM
wParam,LPARAM lParam)
{
	switch(msg)
	{
		case WM_DESTROY:
		{	PostQuitMessage(0);
			return 0;
		}
		break;
	}
	return DefWindowProc(hWnd,msg,wParam,lParam);
}
```

alvillarreal@gmail.com