Win32/Win64 Programming

For those who are used to programming in ISO standard C or C++, writing software for Windows represents a considerable leap. Within the context of a program written within the standards, there is a clear structure that dictates the flow:

- The `main()` function describes the behaviour and execution of the process.
- Usually, there is reading of data from an input stream: `stdin` or some data file.
- The data is processed in some way.
- Optionally, the user can choose some execution paths depending on options entered from the keyboard.
- The processed data is written to an output stream: `stdout` or another data file.

There is a great deal of software – some of it very good software – written according to this model, but it relies on the limited input and output capabilities of the standard. The reason the standard is so limited in this is that the language definitions are intended to be as general as possible, leaving the more complex user interfaces to the specific implementers of various operating systems: there is no point in writing user interface software for use on, say, the computers on the Mars Explorer robots.

Windows, of course, is a highly interactive operating system that depends on the user interaction with keyboards, mice, game controllers, touch sensitive screens, network cards among other devices, so it is up to Microsoft to define software libraries that allow developers to use these devices. It is what is known as an **event-driven environment**, so the software must be written with the flexibility to handle user interaction. The simple input -> processing -> output model does not work because the well written application must make sense of an astronomical variety of possible user interaction.

The Windows Application Programmer Interface

When Windows was first developed in the 1980s, it ran on Intel machines with a 16 bit architecture (Intel 8088, 8086, 186, 286) and the software development tools included a series of libraries and tools called the **Windows Software Development Kit** (or Win SDK). The libraries were bundled into a programmer interface called **Windows 16 bit Application Programmer Interface** (or Win16 API). At the time, Windows ran not as an operating system, but a user interface layer on top of DOS and Windows used DOS' memory management (such as it was). The API allowed programmers to create windows, send messages between application components, make use of the computer's resources like the mouse, but this was all happening on top of DOS which could be easily modified and mangled to suit the purposes of the developer.

The Win 16 API – a library now reserved for the seventh level of programmer hell – was a blend of C and Pascal. Both languages had a certain amount of popularity within the programming community at the time, though the former's popularity was on the rise and the latter's was falling. C++ as a widespread software tool was in its infancy, but because of its close relationship with C, it could be used with the Win16 API (with some severe limitations).

Starting in the early 1990s, Microsoft took a different direction with the development of the Windows NT line of operating systems. These were designed for computers running in 32 bit mode without DOS. A new set of libraries, very much based on the Win16 API was developed and named the Win32 API. Many of the basic concepts of the earlier version were extended to the new SDK, though the underlying structure had changed considerable with the removal of DOS as a component and the much greater flexibility in memory usage. Some fundamental changes in the API did occur, but one of Microsoft's goal was to ease the transition of developers from the older library to the newer. Another result of this transition was that C++ became a more useful language when used with the Win32 API.

With the development of 64 bit processors in the 2000s, new libraries have been released to support these platforms, but the transition is much smoother from Win32 to Win64 than it was from Win16 to Win32. As well, the API has evolved to meet the needs of new hardware and support for obsolete peripherals and components have been dropped. Libraries such as DirectX fit easily into the API which relieves the programmer of many of the labourious tasks that graphics programming can have, while maintaining a high level of control in the hands of the developer.

Even though Microsoft has devoted much effort into the .NET framework over the last ten years, much of the code written for it still has its roots in the Win32/Win64 API. This is true for Visual Basic and C# which use .NET, so many of the concepts learned in studying the API apply directly to .NET code.

Memory Management

One aspect of Win16 program whose effects continue to Win64 programming is that of addressing a process' objects in memory. In Win16, DOS was effectively restricted to 640 KB of memory, although there were a number of tricks in the architecture that allowed for the use of what was called extended or expanded memory. A Win16 process had unguarded access to the memory including memory being used by other processes. Windows could rearrange that memory at will. Using a pointer to access an object in memory almost guaranteed a program crash. In order to find objects in memory, Win16 kept hash maps that stored the address as the value and kept an unsigned integer as the key. A key was referred to as a `HANDLE`.

Most API calls use a `HANDLE` where most standard C++ programs would use a pointer. This continues into the Win64 API. There are handles to icons (`HICON`), handles to windows (`HWND`) handles to process instances (`HINSTANCE`) among other `typedef`s that use this concept.

Structure of a Win32/Win64 Program.

A program written in ISO standard C generally follows the basic structure:

```
int main()
{
      //Input data
      //Process data
      //Output data
      return 0;
}
```

In standard C++, the `main()` function is sometimes referred to as the **entry point**. This kind of process has no notion of a window or any other graphical component because there is no requirement through the standard that it be run in any graphical environment. Neither is this program designed to handle events from the user or operating system.

A Win32/Win64 program (almost always) has a window as well as icons, mouse cursors and a host of other graphical elements. These graphical components are part of the program. As well, a Windows process responds to events passed to it from the operating system, some of which come from user interaction with the computer, some being events that the operating uses to maintain control over the process and others being sent by other processes

The entry point for a Win32/Win64 program is not defined by a main() function, but by the function:

```
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
```

(Note: When using the Visual Studio Project Wizard, this function will be named `_tWinMain(…)`. This adornment to the function name is used for compatibility of character sets.)

This function signature looks a bit strange.

- The function returns an `int`, which is the same as a standard C or C++ program.

- The added directive `APIENTRY` tells the compiler to generate Pascal type calling for this function. This has to do with the order in which function arguments are pushed onto the processor stack. In C and C++, arguments are passed from right to left, but in Pascal style calling they are passed from left to right. For various technical reasons, this style of calling is faster than the C++ method.
- The first parameter is a HANDLE to the instance of the process.
- The second parameter used to be a HANDLE to a previous instance of a process based on the same program. This parameter is obsolete and not used.
- The third parameter is a "Long Pointer to a String" (long pointers are another hangover from Win16). It is the command line that is used to invoke the program which could be typed in from the command line or stored in a program link.
- The last parameter controls how the main window will be displayed: minimized, maximized, restored, etc.

The next step is to register a class of window. This is not a class in the C++ sense, but to describe a particular type of window to the operating system:

```
WNDCLASSEX wcex;
```

This is a `struct` which holds the information that will be used to register the window class. These fields set the size of the `struct` instance that will be passed to Windows, flags that specify when the operating system will send the window a message to repaint itself and two rarely used extra fields.

```
wcex.cbSize = sizeof(WNDCLASSEX);
wcex.style = CS_HREDRAW | CS_VREDRAW;
wcex.cbClsExtra = 0;
wcex.cbWndExtra = 0;
```

Windows is passed the `HINSTANCE` of the process. This value was passed in as an argument to `WinMain(…)`:

```
wcex.hInstance = hInstance;
```

Then a set of resources is passed to the operating system: the icons, mouse cursors,menu and background brush (or colour) for the window. These resources are stored as part of the source code. They can be edited by the graphical tools in Visual Studio.

```
wcex.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_WIN32DEMO));
wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
wcex.lpszMenuName = MAKEINTRESOURCE(IDC_WIN32DEMO);
wcex.hIconSm = LoadIcon(wcex.hInstance,
                        MAKEINTRESOURCE(IDI_SMALL));
```

A Window class is identified by the name it is passed

```
wcex.lpszClassName = ”MyWindowClass”;
```

Finally, and most importantly, the callback is specified for the window. The callback is the function that the operating system will call to send the window messages. It is never called directly in the code, but responds to the messages that Windows sends the process. These messages are, most often, the result of events. The field in the `struct` is a pointer to the callback.

```
wcex.lpfnWndProc = WndProc;
```

The `WNDCLASSEX struct` is passed to the operating system.

```
RegisterClassEx(&wcex);
```

Next, an instance of the window is constructed and a HANDLE to the new window is returned:

```
HWND hWnd;

hWnd = CreateWindow("MyWindowClass", "My Window Caption",
                    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
                    CW_USEDEFAULT, 0, NULL, NULL,
                    hInstance, NULL);
```

The parameters to the API call are:

- The name of the window class as specified when the window class was registered.
- The window caption that appears in the title bar.
- The style of the window. In this case, the window can overlap and be overlapped by other windows.
- The left screen co-ordinate , the top screen co-ordinate the width and the height of the window. Because the constant `CW_USEDEFAULT` is used, the position and size of the window is left up to the operating system.
- The `HWND` to a parent window. `NULL` indicates that it has no parent window in this process.
- An `HMENU` value. If the window class has a menu handle, then NULL can be passed.
- An HINSTANCE, which can be ignored if the value was passed as part of the window class.
- A pointer to a structure that can carry more detailed information about the window. This is passed to the callback during the creation of the window.

Finally, the window is shown and updated:

```
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
```

All that is left is to create a `MSG struct` , which holds the messages passed from the operating system, and start the event loop. The message translated and dispatched to the

window as described in the window class and instantiated in the `CreateWindow(…)` call:

```
MSG msg;

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

After a window gets its last message, `WinMain(…)` exits passing an integer value stored in its `wParam` field:

```
   return (int) msg.wParam;
```

In comparison to standard C++, this is much more complicated, but all of this is necessary to set up the relationship between a process, its window and the operating system.

The Callback

Here is a small callback:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
                 default:
                     return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;
        case WM_PAINT:
             hdc = BeginPaint(hWnd, &ps);
             // TODO: Add any drawing code here...
             EndPaint(hWnd, &ps);
             break;
         case WM_DESTROY:
             PostQuitMessage(0);
             break;
         default:
             return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

The callback returns a `long int` (`typedef`ed into an `LRESULT`). As with the `WinMain(…),` the parameters are passed onto the processor stack using Pascal style calling in order to speed up the execution.

Parameters

An invocation of the callback function has four parameters:

- A HANDLE to the window that is receiving the message,
- A command value.
- A "word" parameter (`wParam`). In Win16 this was a 16 bit unsigned integer. In Win32/64 this is 32 bits long.
- A "long" parameter (`lParam`). This is a 32 bit unsigned integer.

The callback contains several switch statements that identifies the type of message being sent and handles them appropriately. The Win32/Win64 API defines hundreds of different messages. The constants all begin with the prefix `WM_`, which stands for "Windows Message". For each kind of `message`, various parameters are stored in the `wParam` and `lParam` parameters. The meaning of these values depends on the kind of message sent. The example shows how three types of message, `WM_COMMAND`, `WM_PAINT` and `WM_DESTROY` are handled in the callback.

WM_COMMAND

This message is sent when the user chooses a command from the menu, a keyboard short cut or a control like a button sends a message to its parent. Each command has a 16 bit unsigned integer value that identifies it. It is stored in the least significant half of the `wParam`. This is pulled out of the `wParam` with the use of the `LOWORD` macro:

```
#define LOWORD(value) (0x0000FFFF & value)
```

The `HIWORD` macro pulls out the most significant half and shifts it 16 bits to the right:

```
#define HIWORD(value) ((0xFFFF0000 & value) >> 16)
```

These two macros are used extensively because of the memory limitations in the Win16 environment and was not changed for Win32/Win64.

In this example, the function looks for the `IDM_EXIT` value which is passed when the "Exit" option is chosen from the menu. If the message is not handled, it is passed back to the operating system with the call:

```
    return DefWindowProc(hWnd, message, wParam, lParam);
```

WM_PAINT

This message is sent to a window by the operating system when it detects that a window needs to repaint itself. It might seem odd that the operating system needs to tell a window to do this, but a window does not know if other windows are being moved to uncover it or whether the user has restored or maximized the window.

The code that handles the repainting starts with a call to:

```
HDC hdc = BeginPaint(hWnd, &ps);
```

This function takes a handle to the window to be repainted (`HWND`) and a pointer to a PAINTSTRUCT, which holds various painting parameters. It returns a handle to a device context (`HDC`) which holds many parameters that describe the device to which the painting will occur. All painting functions take an `HDC` as a parameter to identify the display.

After any painting is done, the application calls:

```
EndPaint(hWnd, &ps);
```

This returns the device context back to Windows.

WM_DESTROY

This message is sent just before a window is destroyed. The standard procedure for the this message is to call:

```
PostQuitMessage(0);
```

This function causes the event loop in the `WinMain(…)` function to exit and the value passed is returned in the last line of `WinMain(…)`.

Custom Messages

Developers are not restricted to the set of standard defined messages. Depending on what is needed, a custom message can be set up as long as a unique value for the message parameter. The SDK defines the value WM_USER as the first unique value that can be used. If defining custom messages, the code will look similar to this:

```
#define CUSTOM_MESSAGE1 (WM_USER + 0)
#define CUSTOM_MESSAGE2 (WM_USER + 1)
#define CUSTOM_MESSAGE3 (WM_USER + 2)
#define CUSTOM_MESSAGE4 (WM_USER + 3)
```

After defining custom messages, a developer can specify the meanings of the `lParam` and `wParam` values.

Return Values From a Callback

If a callback function handles a message, it is important to tell Windows this by returning 0. This tells the operating system that the message was completely handled and no further action is required. If the message was not handled in the callback, it should call:

```
return DefWindowProc(hWnd, message, wParam, lParam);
```

This tells the operating system that the message was not handled and that it should take care of it.

"Calling" a Callback

In Windows, a program never explicitly calls a callback function. This is reserved for the operating system. It is a very good policy to follow in your own programming.

There will be times, though, where you will want to call a callback function for a window, especially if you are sending a custom message. In this case, a program requests that Windows call the function on your process' behalf using the functions:

```
PostMessage(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
```

or

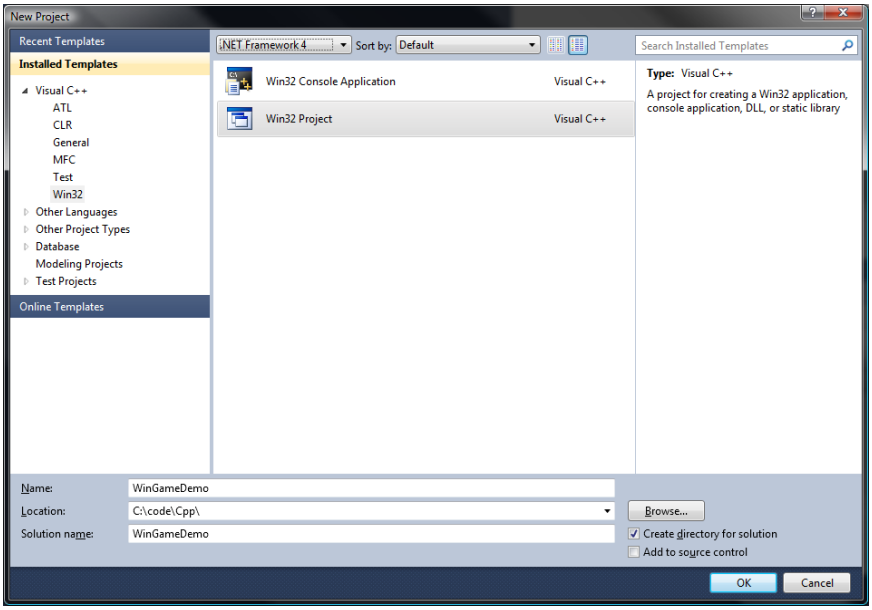
```
SendMessage(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
```

The difference between the two demonstrates a basic structure of Windows. The operating system maintains a message queue for each window of each process. When a developer uses `PostMessage(…)`, the operating system places the message inside the window's queue and it is processed in order. If a message is sent using `SendMessage(…)`, the queue is bypassed and the function is called immediately before any other of the waiting messages are sent. Usually, this will not make a difference in the execution of a process, but this distinction will occasionally make a very big difference.

Creating a Win32/64 Project

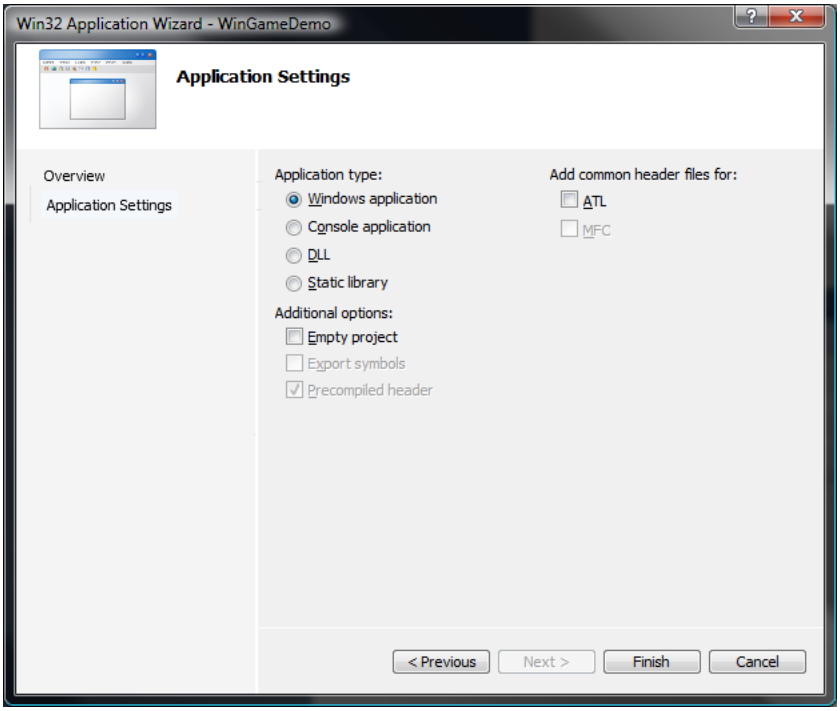
Win32/64 projects contain a lot of “boiler-plate” code that is easily filled out by the New Project Wizard in Visual Studio 2010. These are the steps to create a new project:

1. In Visual Studio, choose New -> Project… and the New Project Wizard appears.



2. Under Visual C++, choose Win32 and Win32 Project. Fill out the controls that specify the name and location of your new project.

3. Click OK and the Win32 Application Wizard is shown. Click on Next:



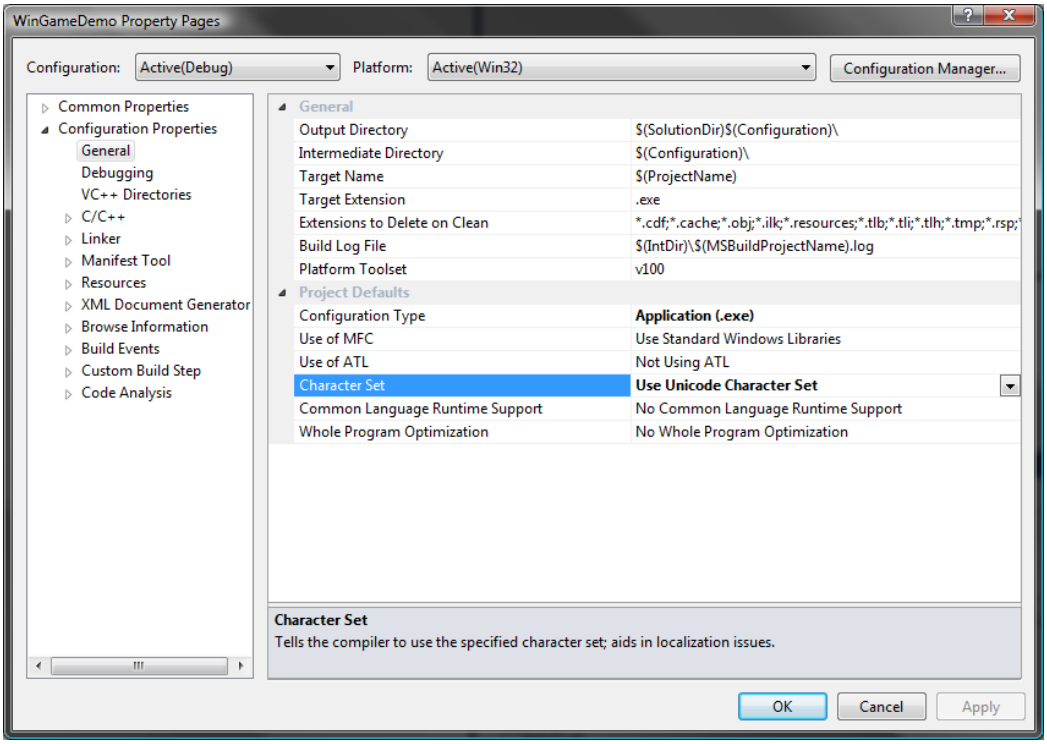
4. Choose Windows Application from Application Type and Click Finish.

You now have a basic Win32/Win64 application project that has a simple window. There are some added features like a menu, keyboard accelerators and an “About…” dialog box. You will notice that dialog boxes have a callback function, just as main windows do.

Text Settings

When a project is set up in the wizard, there are many project settings required in order for the compilation to be successful. For the most part, you should leave these settings alone unless you know precisely what a setting is done. One setting that you may want to adjust at this stage is the character set.

In the Solution Explorer, highlight the project and from the context menu, choose "Properties…". Select "General" and "Character Set":



You can select between Unicode and Multi-Byte characters. This difference needs to be considered in your code. The multi-byte set is a carry-over from the times where ASCII characters were sufficient to represent strings. Literals are encoded in double quotation marks:

```
"This is a message in MBCS";
```

Unicode literals require a prepended L before the quoted string:

```
L"This is a Unicode message.";
```

This table shows the basic data types functions and classes used with both character sets:

	Multi-Byte Characters	**Unicode**
Basic C data type	`char*`	`wchar_t*`

C Functions	`strlen(…),`
	`strcpy(…),`
	`strcmp(…), etc`
		`wcslen(…)`
		`wcscpy(…)`
		`wcscmp(…)`

C++ Class	`std::string`	`std::wstring`
Literals	`"ASCII literal";`	`L"UNICODE Literal"`

Note: the Windows SDK has `typedef`ed `WCHAR` from wchar_t, so the two are interchangable.

You have two choices:

- "Simplify" your code by changing the project setting to MBCS
- Using UNICODE including the literal notation for wide characters.

While the first option will work for school and demonstration code, most development shops will insist on using UNICODE. Localization is a crucial issue in modern games and this is much easier to achieve using UNICODE than MBCS. If you know your game is only going to be played by English speaking players and those of other cultures that play your game will not mind this, then you can feel free to just use MBCS, but this is becoming less and less common.

Win32 -- An Example