
Chapter 15

Programming Paradigm

A Windows program, like any other interactive program, is for the most part input-driven. However, the input of a Windows program is conveniently predigested by the operating system. When a user presses a key or moves the mouse, Windows intercepts the event, preprocesses it, and dispatches it to the appropriate user program. The program gets all its messages from Windows. It may do something about them or not—in the latter case it lets Windows do the “right” thing (the default processing).

When a Windows program starts, it registers a *window class* with the system (it’s not a C++ class). Through this “class data structure” it gives the system, among other things, a pointer to a callback function called the *window procedure*. Windows will call this procedure whenever it wants to pass a message to the program and to notify it of interesting events. The name “callback” means just this: don’t call Windows, Windows will call you back.

The program also gets a peek at every message in the *message loop* before it gets dispatched to the appropriate window procedure. In most cases, the message loop just forwards the message back to Windows to do the dispatching (see Figure 15.1).

Windows is a multitasking operating system—there may be many programs running at the same time. So how does Windows know which program should get a particular message? Mouse messages, for instance, are usually dispatched to the application that created the window over which the mouse cursor is positioned at a given moment (unless an application “captures” the mouse).

Most Windows programs create one or more *windows* on the screen. At any given time, one of these windows has the *focus* and is considered *active* (its title bar is usually highlighted). Keyboard messages are sent to the window that has the focus.

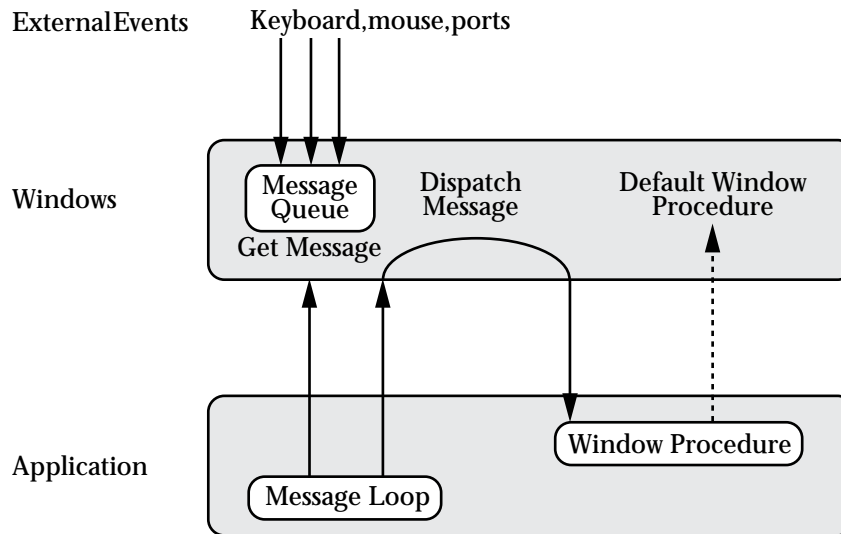


Figure 15.1 Input-driven Windows paradigm

Events such as resizing, maximizing, minimizing, covering, or uncovering a window are handled by Windows, although the concerned program that owns the window also gets a chance to process messages for these events. There are dozens and dozens of types of messages that can be sent to a Windows program. Each program handles the ones that it's interested in and lets Windows deal with the others.

Windows programs use Windows services to output text or graphics to the screen. Windows not only provides a high-level graphical interface, but it separates the program from the actual graphical hardware. In this sense Windows graphics are, to a large extent, device independent.

It is very easy for a Windows program to use standard Windows controls. Menus are easy to create; so are message boxes. Dialog boxes are more general—they can be designed by a programmer using a *dialog editor*; and icons can be created using an *icon editor*. List boxes, edit controls, scroll bars, buttons, radio buttons, check boxes, and so on, are all examples of built-in, ready-to-use controls that make Windows programs so attractive and usable.

All this functionality is available to the programmer through the Windows API. It is a (very large) set of C functions, typedefs, structures, and macros whose declarations are included (directly or indirectly) in `<windows.h>`, and whose code is linked to the program through a set of libraries and DLLs (dynamic-link libraries).

In this chapter we will be using the Code Co-op project named *windows*. Margin notes give the script numbers to unpack.

15.1 Hello Windows!

Start project
windows

Our first Windows program will do nothing more than create a window with the title bar "Hello Windows!" However, it is definitely more complicated than Kernighan and Ritchie's "Hello World!" and our first "Hello!" C++ program. What we are getting here is much more than the simple old-fashioned teletype output. We are creating a window that can be moved around, resized, minimized, maximized, overlapped by other windows, and so on. It also has a standard system menu in the upper left corner. So let's not complain too much!

In Windows, the main procedure is called `WinMain`. It must use the `WINAPI` calling convention and the system calls it with the following parameters:

- `HINSTANCE hInst`—The handle to the current instance of the program.
- `HINSTANCE hPrevInst`—Obsolete in Win32, this is kept for compatibility with Win16.
- `LPSTR cmdParam`—A string with the command-line arguments.
- `int cmdShow`—A flag that says whether to show the main window or not.

Notice the strange type names. You'll have to get used to them—Windows is full of typedefs. In fact, you will rarely see an `int` or a `char` in the description of the Windows API. For now, it's enough to know that `LPSTR` is in fact a typedef for a `char *` (the abbreviation stands for Long Pointer to STRing, where string is a null-terminated array, and "long pointer" is a fossil left over from the times of 16-bit Windows).

In what follows, I will prefix all Windows API functions with a double colon. A double colon simply means that it's a globally defined function (not a member of any class or namespace). It is somehow redundant, but it makes the code more readable. The classes `WinClassMaker`, `WinMaker`, and `Window` will be defined in a moment.

```
int WINAPI WinMain
(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR cmdParam, int cmdShow)
{
    char className [] = "Winnie";
    WinClassMaker winClass (WinProcedure, className, hInst);
    winClass.Register ();
    WinMaker maker (className, hInst);
    Window win = maker.Create ("Hello Windows!");
    win.Display (cmdShow);
}
```

```

// Message loop
MSG msg;
int status;
while ((status = ::GetMessage (& msg, 0, 0, 0)) != 0)
{
    if (status == -1)
        return -1;
    ::DispatchMessage (& msg);
}
return msg.wParam;
}

```

First, the program creates a window class and registers it. Then it creates a window with the caption “Hello Windows!” and displays it (or not, depending on the flag it is given). Finally, it enters the message loop that waits for a message from Windows (the `::GetMessage` API) and then lets the system dispatch it. The message will come back to our program when Windows calls the Window procedure, `WinProcedure`. For the time being we don’t have to worry about the details of the message loop. The program normally terminates when `::GetMessage` returns zero. The `wParam` of the last message contains the return code of our program.

The `::GetMessage` function is an interesting example of three-state logic. It is defined to return the type `BOOL`, which is a typedef for `int`, not `bool` (in fact, there was no `bool` in C++, not to mention C, when Windows was first released). The documentation, however, specifies three types of returns: nonzero, zero, and `-1` (I am not making this up!). Here’s the actual excerpt from the Help file:

If the function retrieves a message other than `WM_QUIT`, the return value is **nonzero**.
 If the function retrieves the `WM_QUIT` message, the return value is **zero**.
 If there is an error, the return value is **-1**.

We introduce the class `WinClassMaker` to encapsulate the `WNDCLASSEX` data structure and the `::RegisterClass` API.

```

class WinClassMaker
{
public:
    WinClassMaker (WNDPROC WinProcedure,
                  char const * className,
                  HINSTANCE hInst);
    void Register ()
    {
        if (::RegisterClassEx (&_class) == 0)
            throw "RegisterClass failed";
    }
}

```

```
private:
    WNDCLASSEX _class;
};
```

In the constructor of `WinClassMaker` we initialize all parameters to some sensible default values. For instance, we set the default for the mouse cursor to an arrow, and the brush (used by Windows to paint the window's background) to the default window color. We will translate most Windows errors into exceptions. For the time being I will use literal strings as exception objects—they can be caught by the following catch clause:

```
catch (char const * msg)
```

Later we'll switch to a more sophisticated mechanism (see page 397).

When registering a class, we have to provide the pointer to the window procedure, the name of the class, and the handle to the instance that owns the class.

```
WinClassMaker: WinClassMaker
(WNDPROC WinProcedure,
 char const * className,
 HINSTANCE hInst)
{
    _class.lpfWndProc = WinProcedure; // Mandatory
    _class.hInstance = hInst;        // Mandatory
    _class.lpszClassName = className; // Mandatory
    _class.cbSize = sizeof (WNDCLASSEX);
    _class.hCursor = ::LoadCursor (0, IDC_ARROW);
    _class.hbrBackground = reinterpret_cast<HBRUSH>
        (COLOR_WINDOW + 1);

    _class.style = 0;
    _class.cbClsExtra = 0;
    _class.cbWndExtra = 0;
    _class.hIcon = 0;
    _class.hIconSm = 0;
    _class.lpszMenuName = 0;
}
```

Notice some of the ugly tricks we have to do. The `hbrBackground` data member of `WNDCLASSEX` is defined as `HBRUSH`. However, you can initialize it using one of the standard color constants defined by Windows, `COLOR_WINDOW` in our case. That requires an explicit type cast. Unfortunately, one of the color constants, `COLOR_SCROLLBAR`, is defined to be zero, so it would be confused by Windows with a null brush. That's why you are required to add the spurious one to the color constant when passing it to `WNDCLASSEX`. Windows is full of such "clever hacks."

The class `WinMaker` initializes and stores all the parameters describing a particular window.

```
class WinMaker
{
```

```

public:
    WinMaker (char const * className, HINSTANCE hInst);
    HWND Create (char const * title);
private:
    HINSTANCE _hInst; // Program instance
    char const *_className; // Name of Window class
    DWORD _style; // Window style
    DWORD _exStyle; // Window extended style
    int _x; // Horizontal position
    int _y; // Vertical position
    int _width; // Window width
    int _height; // Window height
    HWND _hWndParent; // Parent or owner
    HMENU _hMenu; // Menu or child-window ID
    void *_data; // window-creation data
};

```

The constructor of `WinMaker` takes the name of its window class and the handle to program instance. The class name is necessary for Windows to find the window procedure for this window. The rest of the parameters are given some reasonable default values. For instance, we let the system decide the initial position and size of our window. The style, `WS_OVERLAPPEDWINDOW`, is the most common style for top-level windows. It includes a title bar with a system menu on the left and the minimize, maximize, and close buttons on the right. It also provides for a “thick” border that can be dragged with the mouse to resize the window.

```

WinMaker::WinMaker (char const * className,
                    HINSTANCE hInst)
: _style (WS_OVERLAPPEDWINDOW),
  _exStyle (0),
  _className (className),
  _x (CW_USEDEFAULT), // Horizontal position
  _y (0), // Vertical position
  _width (CW_USEDEFAULT), // Window width
  _height (0), // Window height
  _hWndParent (0), // Parent or owner window
  _hMenu (0), // Menu or child-window identifier
  _data (0), // Window-creation data
  _hInst (hInst)
{}

```

All these parameters are passed to the `::CreateWindowEx` API that creates the window (but doesn’t display it yet).

```

HWND WinMaker::Create (char const * title)
{
    HWND hwnd = ::CreateWindowEx (
        _exStyle,
        _className,
        title,
        _style,

```

```

        _x,
        _y,
        _width,
        _height,
        _hWndParent,
        _hMenu,
        _hInst,
        _data);

    if (hwnd == 0)
        throw "Window Creation Failed";
    return hwnd;
}

```

`Create` takes a window title which will appear in the title bar and returns a handle to the successfully created window. We will conveniently encapsulate this handle in a class called `Window`. Other than storing a handle to a particular window, this class will later provide an interface to a multitude of Windows APIs that operate on that window.

```

class Window
{
public:
    Window (HWND h = 0) : _h (h) { }
    void Display (int cmdShow)
    {
        assert (_h != 0);
        ::ShowWindow (_h, cmdShow);
        ::UpdateWindow (_h);
    }
private:
    HWND _h;
};

```

To make the window visible, we have to call `::ShowWindow` with the appropriate parameter, which specifies whether the window should be initially minimized, maximized, or the default size. `::UpdateWindow` causes the contents of the window to be refreshed.

The window procedure must have the following signature, which is hard-coded into Windows:

```

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

```

Notice the calling convention and the types of parameters and the return value. These are all typedefs defined in `windows.h`. `CALLBACK` is a predefined language-independent calling convention (what order the parameters are pushed on the stack, and so on). `LRESULT` is a type of return value. `HWND` is a handle to a window,

UINT is an unsigned integer that identifies the message, and WPARAM and LPARAM are the types of the two parameters that are passed with every message.

WinProcedure is called by Windows every time it wants to pass a message to our program (see Figure 15.2). The window handle identifies the window that is supposed to respond to this message. Remember that the same window procedure may service several instances of the same window class. Each instance will have its own window with a different handle, but they will all go through the same procedure.

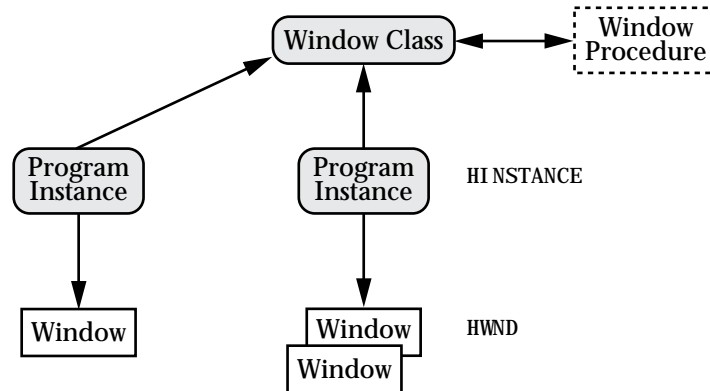


Figure 15.2 The relationship between instances of the same program, their windows, and the program's window class and procedure

The message is just a number. Symbolic names for these numbers are defined in `windows.h`. For instance, the message that tells the window to repaint itself is defined as

```
#define WM_PAINT 0x000F
```

Every message is accompanied by two parameters whose meaning depends on the kind of message. (In the case of `WM_PAINT`, the parameters are meaningless.)

To learn more about window procedures and various messages, study the Help files that come with your compiler.

Here's our minimalist implementation of the window procedure.

```
LRESULT CALLBACK WinProcedure
(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_DESTROY:
        ::PostQuitMessage (0);
        return 0;
    }
}
```

```

    }
    return ::DefWindowProc (hwnd, message, wParam, lParam );
}

```

It doesn't do much in this particular program. It only handles one message, `WM_DESTROY`, that is sent to the window when it is being destroyed. At that point the window has already been closed—all we have to do is terminate `WinMain`. We do it by posting the final *quit* message. We also pass the return code through it—zero in our case. This message will terminate the message loop and control will be returned to `WinMain`.

Figure 15.3 shows the window created by our little application—complete with sizing borders; minimize, maximize, and close buttons; a Windows default icon, and a system menu that can be opened by clicking on the icon.



Figure 15.3 The output of Hello Windows!

Script 1 To make the project more structured, we will distribute the source code between multiple files.

15.2 Encapsulation

A lot of Windows APIs take a large number of arguments. For instance, when you create a window, you have to be able to specify its position, size, style, title, and so on. That's why `CreateWindowEx` takes 12 arguments. `RegisterClassEx` also requires 12 parameters, but they are combined into a single structure, `WNDCLASSEX`. As you can see, there is no consistency in the design of Windows API.

Our approach to encapsulating this type of API is to create a C++ class that combines all the arguments in one place. Since most of these arguments have sensible defaults, the constructor should initialize them appropriately. Those parameters that don't have natural defaults are passed as arguments to the constructor. The idea is that once an object of such a class is constructed, it should be usable without further modification. However, if modifications are desired, they can be done by calling appropriate methods. For instance, if we are to develop the

class `WinMaker` into a more useful form, we should add methods such as `SetPosition` and `SetSize` that override the default settings for `_x`, `_y`, `_width`, and `_height`.

Let's analyze the classes `WinClassMaker` and `WinMaker` in this context. `WinClassMaker` encapsulates the API `RegisterClassEx`. The argument to this API is a structure that we can embed directly into our class. Three of the ten parameters—window procedure, class name, and program instance—cannot use the defaults so they are passed in the constructor. The window background color normally defaults to whatever the current system setting is—that's the `COLOR_WINDOW` constant. The mouse cursor in most cases defaults to whatever the system considers an arrow—that's the `IDC_ARROW` constant. The size of the structure must be set to `sizeof(WNDCLASSEX)`. The rest of the parameters can be safely set to zero. We don't have to do anything else before calling `WinClassMaker::Register`. Of course, in a more sophisticated program we might want to modify some of these settings, and we would most certainly add methods to do that. We'll talk about it later.

In a similar way, the API `CreateWindowEx` is encapsulated in the class `WinMaker`. The nondefaultable parameters are the class name, the program instance, and the title of the window. This time, however, we might want to call `WinMaker::Create` multiple times in order to create more than one window. Most likely these windows would have different titles, so we pass the title as an argument to `Create`.

To summarize, in the main procedure your program creates a window of a particular class and enters the message processing loop. In this loop, the program waits idly for a message from Windows. Once the message arrives, the program dispatches it back to Windows. The system then calls back the program's window procedure with the same message. It is either processed manually or by the default processing. The window procedure must be implemented in your program, and its address is passed to the system during window class registration. After returning from the window procedure, control goes back to the message loop and the whole process repeats itself. Eventually a "quit" message is posted and the loop ends.
