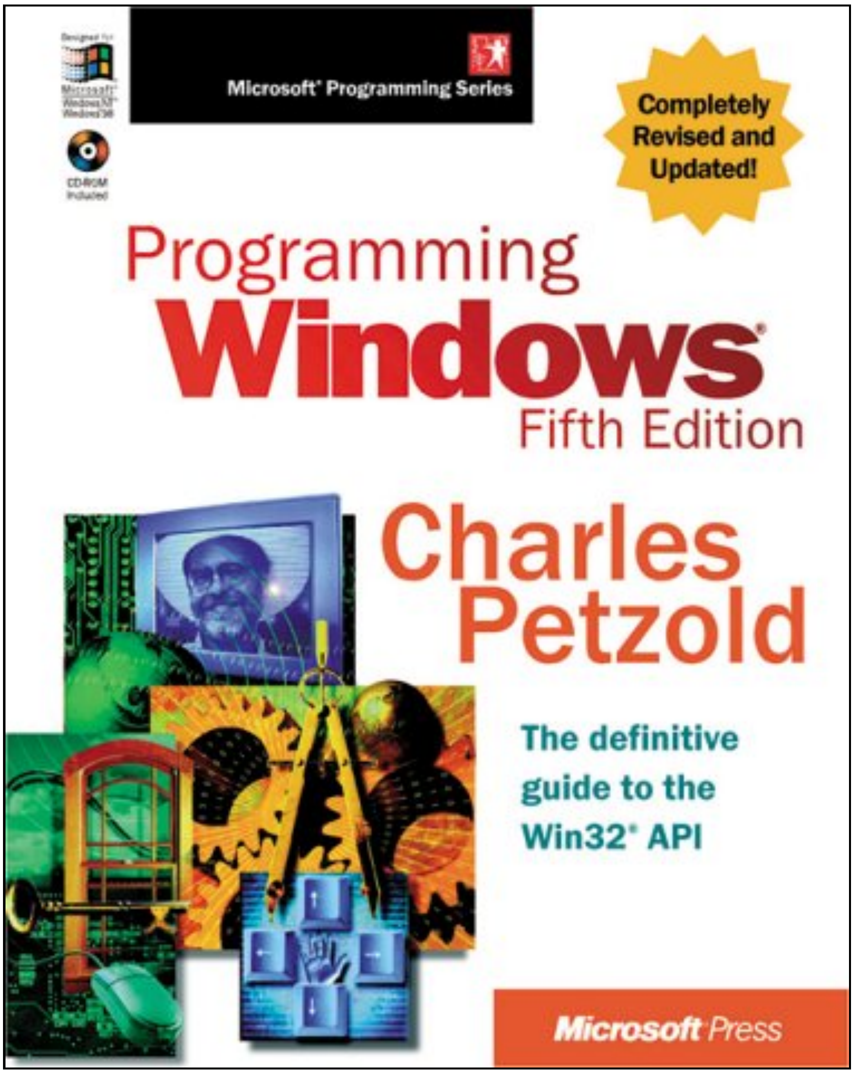

Microsoft® Programming Series
Completely Revised and Updated!
Programming Windows®
Fifth Edition
Charles Petzold
The definitive guide to the Win32® API
Microsoft Press

Author's Note

Visit my web site *www.cpetzold.com* for updated information regarding this book, including possible bug reports and new code listings. You can address mail regarding problems in this book to *charles@cpetzold.com*. Although I'll also try to answer any easy questions you may have, I can't make any promises. I'm usually pretty busy, and my cat refuses to learn the Windows API.

I'd like to thank everyone at Microsoft Press for another great job in putting together this book. I think this "10th Anniversary Edition" of *Programming Windows* is the best edition yet. Many other people at Microsoft (including some of the early developers of Microsoft Windows) also helped out when I was writing the earlier editions, and these fine people are listed in those editions.

Thanks also to my family and friends, and in particular those more recent friends (you know who you are!) whose support has made this book possible. To you this book is dedicated.

Charles Petzold

October 5, 1998

About the Author

Charles Petzold has been writing about personal computer programming since 1984 and has been programming for Microsoft Windows since 1985. He wrote the first magazine article about Windows programming in the December 1986 issue of *Microsoft Systems Journal*. Between 1986 and 1995, he wrote the Environments column for *PC Magazine*, which introduced his readers to many facets of Windows and OS/2 programming.

Programming Windows was first published by Microsoft Press in 1988 and has since become regarded as the best introductory text on the subject. In May 1994, Petzold was one of only seven people (and the only writer) to be given a Windows Pioneer Award from *Windows Magazine* and Microsoft Corporation for his contribution to the success of Microsoft Windows.

In the fall of 1999, Microsoft Press will publish Charles Petzold's first book for a general audience. Tentatively entitled *Code: The Hidden Language of Computer Hardware and Software*, this book is a unique introduction to the nature of digital information and how computers work with that information

About This Electronic Book

This electronic book was originally created and still may be purchased as a print book. For simplicity, the electronic version of this book has been modified as little as possible from its original form. For instance, there may be occasional references to sample files that come with the book. These files are available with the print version, but are not provided in this electronic edition.

Where do you want to go today?®

Your Information Source

We're the independent publishing division of Microsoft Corporation your source of inside information and unique perspectives about Microsoft products and related technologies. We've been around since 1984, and we offer a complete line of computer books from self−paced tutorials for first−time computer users to advanced technical references for professional programmers.

mspress.microsoft.com

Table of Contents:

Table of Contents:

Table of Contents:

Table of Contents:

Table of Contents:

Table of Contents:

Table of Contents:

Table of Contents:

Table of Contents:

Table of Contents:

Table of Contents:

Table of Contents:

Chapter 1 −− Getting Started

This book shows you how to write programs that run under Microsoft Windows 98, Microsoft Windows NT 4.0, and Windows NT 5.0. These programs are written in the C programming language and use the native Windows application programming interfaces (APIs). As I'll discuss later in this chapter, this is not the only way to write programs that run under Windows. However, it is important to understand the Windows APIs regardless of what you eventually use to write your code.

As you probably know, Windows 98 is the latest incarnation of the graphical operating system that has become the de facto standard for IBM−compatible personal computers built around 32−bit Intel microprocessors such as the 486 and Pentium. Windows NT is the industrial−strength version of Windows that runs on PC compatibles as well as some RISC (reduced instruction set computing) workstations.

There are three prerequisites for using this book. First, you should be familiar with Windows 98 from a user's perspective. You cannot hope to write applications for Windows without understanding its user interface. For this reason, I suggest that you do your program development (as well as other work) on a Windows−based machine using Windows applications.

Second, you should know C. If you don't know C, Windows programming is probably not a good place to start. I recommend that you learn C in a character−mode environment such as that offered under the Windows 98 MS−DOS Command Prompt window. Windows programming sometimes involves aspects of C that don't show up much in character−mode programming; in those cases, I'll devote some discussion to them. But for the most part, you should have a good working familiarity with the language, particularly with C structures and pointers. Some knowledge of the standard C run−time library is helpful but not required.

Third, you should have installed on your machine a 32−bit C compiler and development environment suitable for doing Windows programming. In this book, I'll be assuming that you're using Microsoft Visual C++ 6.0, which can be purchased separately or as a part of the Visual Studio 6.0 package.

That's it. I'm not going to assume that you have any experience at all programming for a graphical user interface such as Windows.

The Windows Environment

Windows hardly needs an introduction. Yet it's easy to forget the sea change that Windows brought to office and home desktop computing. Windows had a bumpy ride in its early years and was hardly destined to conquer the desktop market.

A History of Windows

Soon after the introduction of the IBM PC in the fall of 1981, it became evident that the predominant operating system for the PC (and compatibles) would be MS−DOS, which originally stood for Microsoft Disk Operating System. MS−DOS was a minimal operating system. For the user, MS−DOS provided a command−line interface to commands such as DIR and TYPE and loaded application programs into memory for execution. For the application programmer, MS−DOS offered little more than a set of function calls for doing file input/output (I/O). For other tasks in particular, writing text and sometimes graphics to the video display applications accessed the hardware of the PC directly.

Due to memory and hardware constraints, sophisticated graphical environments were slow in coming to small computers. Apple Computer offered an alternative to character−mode environments when it released its ill−fated Lisa in January 1983, and then set a standard for graphical environments with the Macintosh in January 1984. Despite the Mac's declining market share, it is still considered the standard against which other graphical environments are measured. All graphical environments, including the Macintosh and Windows, are indebted to the pioneering work done at the Xerox Palo Alto Research Center (PARC) beginning in the mid−1970s.

Windows was announced by Microsoft Corporation in November 1983 (post−Lisa but pre−Macintosh) and was released two years later in November 1985. Over the next two years, Microsoft Windows 1.0 was followed by several updates to support the international market and to provide drivers for additional video displays and printers.

Windows 2.0 was released in November 1987. This version incorporated several changes to the user interface. The most significant of these changes involved the use of overlapping windows rather than the "tiled" windows found in Windows 1.0. Windows 2.0 also included enhancements to the keyboard and mouse interface, particularly for menus and dialog boxes.

Up until this time, Windows required only an Intel 8086 or 8088 microprocessor running in "real mode" to access 1 megabyte (MB) of memory. Windows/386 (released shortly after Windows 2.0) used the "virtual 86" mode of the Intel 386 microprocessor to window and multitask many DOS programs that directly accessed hardware. For symmetry, Windows 2.1 was renamed Windows/286.

Windows 3.0 was introduced on May 22, 1990. The earlier Windows/286 and Windows/386 versions were merged into one product with this release. The big change in Windows 3.0 was the support of the 16−bit protected−mode operation of Intel's 286, 386, and 486 microprocessors. This gave Windows and Windows applications access to up to 16 megabytes of memory. The Windows "shell" programs for running programs and maintaining files were completely revamped. Windows 3.0 was the first version of Windows to gain a foothold in the home and the office.

Any history of Windows must also include a mention of OS/2, an alternative to DOS and Windows that was originally developed by Microsoft in collaboration with IBM. OS/2 1.0 (character−mode only) ran on the Intel 286 (or later) microprocessors and was released in late 1987. The graphical Presentation Manager (PM) came about with OS/2 1.1 in October 1988. PM was originally supposed to be a protected−mode version of Windows, but the graphical API was changed to such a degree that it proved difficult for software manufacturers to support both platforms.

By September 1990, conflicts between IBM and Microsoft reached a peak and required that the two companies go their separate ways. IBM took over OS/2 and Microsoft made it clear that Windows was the center of their strategy for operating systems. While OS/2 still has some fervent admirers, it has not nearly approached the popularity of Windows.

Microsoft Windows version 3.1 was released in April 1992. Several significant features included the TrueType font technology (which brought scaleable outline fonts to Windows), multimedia (sound and music), Object Linking and Embedding (OLE), and standardized common dialog boxes. Windows 3.1 ran *only* in protected mode and required a 286 or 386 processor with at least 1 MB of memory.

Windows NT, introduced in July 1993, was the first version of Windows to support the 32−bit mode of the Intel 386, 486, and Pentium microprocessors. Programs that run under Windows NT have access to a 32−bit flat address space and use a 32−bit instruction set. (I'll have more to say about address spaces a little later in this chapter.) Windows NT was also designed to be portable to non−Intel processors, and it runs on several RISC−based workstations.

Windows 95 was introduced in August 1995. Like Windows NT, Windows 95 also supported the 32−bit programming mode of the Intel 386 and later microprocessors. Although it lacked some of the features of Windows NT, such as high security and portability to RISC machines, Windows 95 had the advantage of requiring fewer hardware resources.

Windows 98 was released in June 1998 and has a number of enhancements, including performance improvements, better hardware support, and a closer integration with the Internet and the World Wide Web.

Aspects of Windows

Both Windows 98 and Windows NT are 32−bit preemptive multitasking and multithreading graphical operating systems. Windows possesses a graphical user interface (GUI), sometimes also called a "visual interface" or "graphical windowing environment." The concepts behind the GUI date from the mid−1970s with the work done at the Xerox PARC for machines such as the Alto and the Star and for environments such as SmallTalk. This work was later brought into the mainstream and popularized by Apple Computer and Microsoft. Although somewhat controversial for a while, it is now quite obvious that the GUI is (in the words of Microsoft's Charles Simonyi) the single most important "grand consensus" of the personal−computer industry.

All GUIs make use of graphics on a bitmapped video display. Graphics provides better utilization of screen real estate, a visually rich environment for conveying information, and the possibility of a WYSIWYG (what you see is what you get) video display of graphics and formatted text prepared for a printed document.

In earlier days, the video display was used solely to echo text that the user typed using the keyboard. In a graphical user interface, the video display itself becomes a source of user input. The video display shows various graphical objects in the form of icons and input devices such as buttons and scroll bars. Using the keyboard (or, more directly, a pointing device such as a mouse), the user can directly manipulate these objects on the screen. Graphics objects can be dragged, buttons can be pushed, and scroll bars can be scrolled.

The interaction between the user and a program thus becomes more intimate. Rather than the one−way cycle of information from the keyboard to the program to the video display, the user directly interacts with the objects on the display.

Users no longer expect to spend long periods of time learning how to use the computer or mastering a new program. Windows helps because all applications have the same fundamental look and feel. The program occupies a window usually a rectangular area on the screen. Each window is identified by a caption bar. Most program functions are initiated through the program's menus. A user can view the display of information too large to fit on a single screen by using scroll bars. Some menu items invoke dialog boxes, into which the user enters additional information. One dialog box in particular, that used to open a file, can be found in almost every large Windows program. This dialog box looks the same (or nearly the same) in all of these Windows programs, and it is almost always invoked from the same menu option.

Once you know how to use one Windows program, you're in a good position to easily learn another. The menus and dialog boxes allow a user to experiment with a new program and explore its features. Most Windows programs have both a keyboard interface and a mouse interface. Although most functions of Windows programs can be controlled through the keyboard, using the mouse is often easier for many chores.

From the programmer's perspective, the consistent user interface results from using the routines built into Windows for constructing menus and dialog boxes. All menus have the same keyboard and mouse interface because Windows rather than the application program handles this job.

To facilitate the use of multiple programs, and the exchange of information among them, Windows supports multitasking. Several Windows programs can be displayed and running at the same time. Each program occupies a window on the screen. The user can move the windows around on the screen, change their sizes, switch between different programs, and transfer data from one program to another. Because these windows look something like papers on a desktop (in the days before the desk became dominated by the computer itself, of course), Windows is sometimes said to use a "desktop metaphor" for the display of multiple programs.

Earlier versions of Windows used a system of multitasking called "nonpreemptive." This meant that Windows did not use the system timer to slice processing time between the various programs running under the system. The programs themselves had to voluntarily give up control so that other programs could run. Under Windows NT and Windows 98, multitasking is preemptive and programs themselves can split into multiple threads of execution that seem to run concurrently.

An operating system cannot implement multitasking without doing something about memory management. As new programs are started up and old ones terminate, memory can become fragmented. The system must be able to consolidate free memory space. This requires the system to move blocks of code and data in memory.

Even Windows 1.0, running on an 8088 microprocessor, was able to perform this type of memory management. Under real−mode restrictions, this ability can only be regarded as an astonishing feat of software engineering. In Windows 1.0, the 640−kilobyte (KB) memory limit of the PC's architecture was effectively stretched without requiring any additional memory. But Microsoft didn't stop there: Windows 2.0 gave the Windows applications access to expanded memory (EMS), and Windows 3.0 ran in protected mode to give Windows applications access to up to 16 MB of extended memory. Windows NT and Windows 98 blow away these old limits by being full−fledged 32−bit operating systems with flat memory space.

Programs running in Windows can share routines that are located in other files called "dynamic−link libraries." Windows includes a mechanism to link the program with the routines in the dynamic−link libraries at run time. Windows itself is basically a set of dynamic−link libraries.

Windows is a graphical interface, and Windows programs can make full use of graphics and formatted text on both the video display and the printer. A graphical interface not only is more attractive in appearance but also can impart a high level of information to the user.

Programs written for Windows do not directly access the hardware of graphics display devices such as the screen and printer. Instead, Windows includes a graphics programming language (called the Graphics Device Interface, or GDI) that allows the easy display of graphics and formatted text. Windows virtualizes display hardware. A program written for Windows will run with any video board or any printer for which a Windows device driver is available. The program does not need to determine what type of device is attached to the system.

Putting a device−independent graphics interface on the IBM PC was not an easy job for the developers of Windows. The PC design was based on the principle of open architecture. Third−party hardware manufacturers were encouraged to develop peripherals for the PC and have done so in great number. Although several standards have emerged, conventional MS−DOS programs for the PC had to individually support many different hardware configurations. It was fairly common for an MS−DOS word−processing program to be sold with one or two disks of small files, each one supporting a particular printer. Windows programs do not require these drivers because the support is part of Windows.

Dynamic Linking

Central to the workings of Windows is a concept known as "dynamic linking." Windows provides a wealth of function calls that an application can take advantage of, mostly to implement its user interface and display text and graphics on the video display. These functions are implemented in dynamic−link libraries, or DLLs. These are files with the extension .DLL or sometimes .EXE, and they are mostly located in the \WINDOWS\SYSTEM subdirectory under Windows 98 and the \WINNT\SYSTEM and \WINNT\SYSTEM32 subdirectories under Windows NT.

In the early days, the great bulk of Windows was implemented in just three dynamic−link libraries. These represented the three main subsystems of Windows, which were referred to as Kernel, User, and GDI. While the number of subsystems has proliferated in recent versions of Windows, most function calls that a typical Windows program makes will still fall in one of these three modules. Kernel (which is currently implemented by the 16−bit KRNL386.EXE and the 32−bit KERNEL32.DLL) handles all the stuff that an operating system kernel traditionally handles memory management, file I/O, and tasking. User (implemented in the 16−bit USER.EXE and the 32−bit USER32.DLL) refers to the user interface, and implements all the windowing logic. GDI (implemented in the 16−bit GDI.EXE and the 32−bit GDI32.DLL) is the Graphics Device Interface, which allows a program to display text and graphics on the screen and printer.

Windows 98 supports several thousand function calls that applications can use. Each function has a descriptive name, such as *CreateWindow*. This function (as you might guess) creates a window for your program. All the Windows functions that an application may use are declared in header files.

In your Windows program, you use the Windows function calls in generally the same way you use C library functions such as *strlen*. The primary difference is that the machine code for C library functions is linked into your program code, whereas the code for Windows functions is located outside of your program in the DLLs.

When you run a Windows program, it interfaces to Windows through a process called "dynamic linking." A Windows .EXE file contains references to the various dynamic−link libraries it uses and the functions therein. When a Windows program is loaded into memory, the calls in the program are resolved to point to the entries of the DLL functions, which are also loaded into memory if not already there.

When you link a Windows program to produce an executable file, you must link with special "import libraries" provided with your programming environment. These import libraries contain the dynamic−link library names and reference information for all the Windows function calls. The linker uses this information to construct the table in the .EXE file that Windows uses to resolve calls to Windows functions when loading the program.

Windows Programming Options

To illustrate the various techniques of Windows programming, this book has lots of sample programs. These programs are written in C and use the native Windows APIs. I think of this approach as "classical" Windows programming. It is how we wrote programs for Windows 1.0 in 1985, and it remains a valid way of programming for Windows today.

APIs and Memory Models

To a programmer, an operating system is defined by its API. An API encompasses all the function calls that an application program can make of an operating system, as well as definitions of associated data types and structures. In Windows, the API also implies a particular program architecture that we'll explore in the chapters ahead.

Generally, the Windows API has remained quite consistent since Windows 1.0. A Windows programmer with experience in Windows 98 would find the source code for a Windows 1.0 program very familiar. One way the API has changed has been in enhancements. Windows 1.0 supported fewer than 450 function calls; today there are thousands.

The biggest change in the Windows API and its syntax came about during the switch from a 16−bit architecture to a 32−bit architecture. Versions 1.0 through 3.1 of Windows used the so−called segmented memory mode of the 16−bit Intel 8086, 8088, and 286 microprocessors, a mode that was also supported for compatibility purposes in the 32−bit Intel microprocessors beginning with the 386. The microprocessor register size in this mode was 16 bits, and hence the C *int* data type was also 16 bits wide. In the segmented memory model, memory addresses were formed from two components a 16−bit *segment* pointer and a 16−bit *offset* pointer. From the programmer's perspective, this was quite messy and involved differentiating between *long,* or *far,* pointers (which involved both a segment address and an offset address) and *short,* or *near,* pointers (which involved an offset address with an assumed segment address).

Beginning in Windows NT and Windows 95, Windows supported a 32−bit flat memory model using the 32−bit modes of the Intel 386, 486, and Pentium processors. The C *int* data type was promoted to a 32−bit value. Programs written for 32−bit versions of Windows use simple 32−bit pointer values that address a flat linear address space.

The API for the 16−bit versions of Windows (Windows 1.0 through Windows 3.1) is now known as Win16. The API for the 32−bit versions of Windows (Windows 95, Windows 98, and all versions of Windows NT) is now known as Win32. Many function calls remained the same in the transition from Win16 to Win32, but some needed to be enhanced. For example, graphics coordinate points changed from 16−bit values in Win16 to 32−bit values in Win32. Also, some Win16 function calls returned a two−dimensional coordinate point packed in a 32−bit integer. This was not possible in Win32, so new function calls were added that worked in a different way.

All 32−bit versions of Windows support both the Win16 API to ensure compatibility with old applications and the Win32 API to run new applications. Interestingly enough, this works differently in Windows NT than in Windows 95 and Windows 98. In Windows NT, Win16 function calls go through a translation layer and are converted to Win32 function calls that are then processed by the operating system. In Windows 95 and Windows 98, the process is opposite that: Win32 function calls go through a translation layer and are converted to Win16 function calls to be processed by the operating system.

At one time, there were two other Windows API sets (at least in name). Win32s ("s" for "subset") was an API that allowed programmers to write 32−bit applications that ran under Windows 3.1. This API supported only 32−bit versions of functions already supported by Win16. Also, the Windows 95 API was once called Win32c ("c" for "compatibility"), but this term has been abandoned.

At this time, Windows NT and Windows 98 are both considered to support the Win32 API. However, each operating system supports some features not supported by the other. Still, because the overlap is considerable, it's possible to write programs that run under both systems. Also, it's widely assumed that the two products will be merged at some time in the future.

Language Options

Using C and the native APIs is not the only way to write programs for Windows 98. However, this approach offers you the best performance, the most power, and the greatest versatility in exploiting the features of Windows. Executables are relatively small and don't require external libraries to run (except for the Windows DLLs themselves, of course). Most importantly, becoming familiar with the API provides you with a deeper understanding of Windows internals, regardless of how you eventually write applications for Windows.

Although I think that learning classical Windows programming is important for any Windows programmer, I don't necessarily recommend using C and the API for every Windows application. Many programmers particularly those doing in−house corporate programming or those who do recreational programming at home enjoy the ease of development environments such as Microsoft Visual Basic or Borland Delphi (which incorporates an object−oriented dialect of Pascal). These environments allow a programmer to focus on the user interface of an application and associate code with user interface objects. To learn Visual Basic, you might want to consult some other Microsoft Press books, such as *Learn Visual Basic Now* (1996), by Michael Halvorson.

Among professional programmers particularly those who write commercial applications Microsoft Visual C++ with the Microsoft Foundation Class Library (MFC) has been a popular alternative in recent years. MFC encapsulates many of the messier aspects of Windows programming in a collection of C++ classes. Jeff Prosise's *Programming Windows with MFC, Second Edition* (Microsoft Press, 1999) provides tutorials on MFC.

Most recently, the popularity of the Internet and the World Wide Web has given a big boost to Sun Microsystems' Java, the processor−independent language inspired by C++ and incorporating a toolkit for writing graphical applications that will run on several operating system platforms. A good Microsoft Press book on Microsoft J++, Microsoft's Java development tool, is *Programming Visual J++ 6.0* (1998), by Stephen R. Davis.

Obviously, there's hardly any one right way to write applications for Windows. More than anything else, the nature of the application itself should probably dictate the tools. But learning the Windows API gives you vital insights into the workings of Windows that are essential regardless of what you end up using to actually do the coding. Windows is a complex system; putting a programming layer on top of the API doesn't eliminate the complexity it merely hides it. Sooner or later that complexity is going to jump out and bite you in the leg. Knowing the API gives you a better chance at recovery.

Any software layer on top of the native Windows API necessarily restricts you to a subset of full functionality. You might find, for example, that Visual Basic is ideal for your application except that it doesn't allow you to do one or two essential chores. In that case, you'll have to use native API calls. The API defines the universe in which we as Windows programmers exist. No approach can be more powerful or versatile than using this API directly.

MFC is particularly problematic. While it simplifies some jobs immensely (such as OLE), I often find myself wrestling with other features (such as the Document/View architecture) to get them to work as I want. MFC has not been the Windows programming panacea that many hoped for, and few people would characterize it as a model of good object−oriented design. MFC programmers benefit greatly from understanding what's going on in class definitions they use, and find themselves frequently consulting MFC source code. Understanding that source code is one of the benefits of learning the Windows API.