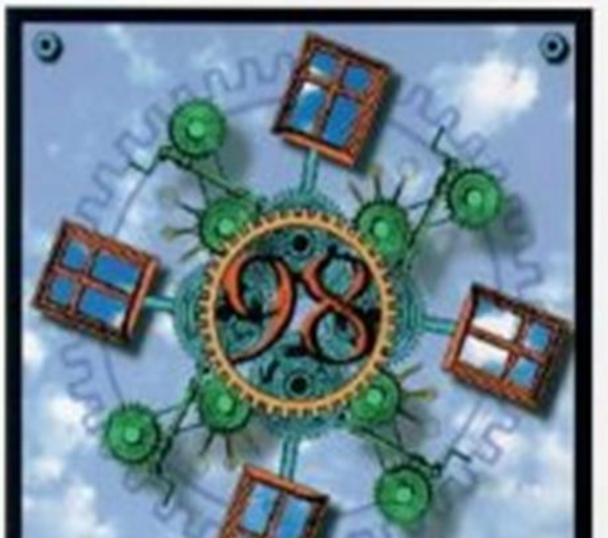
OSBORNE

Open this Book as a Novice and Finish it as a Pro

Windows 98 Programming from the GROUND UP

Take Command of the Windows 98 Programming Environment!



Learn the Basics Through Advanced Topics & Techniques

Supercharge Your Applications for Multiple Monitors, New Controls & Multithreaded Multitasking

Maximize the Power & Versatility of the Win32 API

Designed for Microsoft® Windows®98

FREE CODE ONLINE

BONUS!
www.osborne.com

Herbert Schildt

World's Leading Programming Author with over 2 Million Books Sold

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.

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

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.

The Programming Environment

In this book, I'll be assuming that you're running Microsoft Visual C++ 6.0, which comes in Standard, Professional, and Enterprise editions. The less−expensive Standard edition is fine for doing the programs in this book. Visual C++ is also part of Visual Studio 6.0.

The Microsoft Visual C++ package includes more than the C compiler and other files and tools necessary to compile and link Windows programs. It also includes the Visual C++ Developer Studio, an environment in which you can edit your source code; interactively create resources such as icons and dialog boxes; and edit, compile, run, and debug your programs. If you're running Visual C++ 5.0, you might need to get updated header files and import libraries for Windows 98 and Windows NT 5.0. These are available at Microsoft's web site.

Go to *http://www.microsoft.com/msdn/*, and choose Downloads and then Platform SDK ("software development kit"). You'll be able to download and install the updated files in directories of your choice. To direct the Microsoft Developer Studio to look in these directories, choose Options from the Tools menu and then pick the Directories tab. The *msdn* portion of the Microsoft URL above stands for Microsoft Developer Network. This is a program that provides developers with frequently updated CD−ROMs containing much of what they need to be on the cutting edge of Windows development. You'll probably want to investigate subscribing to MSDN and avoid frequent downloading from Microsoft's web site.

API Documentation

This book is not a substitute for the official formal documentation of the Windows API. That documentation is no longer published in printed form; it is available only via CD−ROM or the Internet. When you install Visual C++ 6.0, you'll get an online help system that includes API documentation. You can get updates to that documentation by subscribing to MSDN or by using Microsoft's Web−based online help system. Start by linking to *http://www.microsoft.com/msdn/*, and select MSDN Library

Online. In Visual C++ 6.0, select the Contents item from the Help menu to invoke the MSDN window. The API documentation is organized in a tree−structured hierarchy. Find the section labeled Platform SDK. All the documentation I'll be citing in this book is from this section. I'll show the location of documentation using the nested levels starting with Platform SDK separated by slashes. (I know the Platform SDK looks like a small obscure part of the total wealth of MSDN knowledge, but I assure you that it's the essential core of Windows programming.) For example, for documentation on how to use the mouse in your Windows programs, you can consult */Platform SDK/User Interface Services/User Input/Mouse*

Input.

I mentioned before that much of Windows is divided into the Kernel, User, and GDI subsystems. The kernel interfaces are in */Platform SDK/Windows Base Services,* the user interface functions are in */Platform SDK/User Interface Services,* and GDI is documented in */Platform SDK/Graphics and Multimedia Services/GDI*.

Your First Windows Program

Now it's time to do some coding. Let's begin by looking at a very short Windows program and, for comparison, a short character−mode program. These will help us get oriented in using the development environment and going through the mechanics of creating and compiling a program.

A Character−Mode Model

A favorite book among programmers is *The C Programming Language* (Prentice Hall, 1978 and 1988) by Brian W. Kernighan and Dennis M. Ritchie, affectionately referred to as K&R. Chapter 1 of this book begins with a C program that displays the words "hello, world." Here's the program as it appeared on page 6 of the first edition of *The C Programming Language*:

```
main ()

{

printf ("hello, world\n") ;

}
```

Yes, once upon a time C programmers used C run−time library functions such as *printf* without declaring them first. But this is the '90s, and we like to give our compilers a fighting chance to flag errors in our code. Here's the revised code from the second edition of K&R:

```
#include <stdio.h>

main ()

{

printf ("hello, world\n") ;

}
```

This program still isn't really as small as it seems. It will certainly compile and run just fine, but many programmers these days would prefer to explicitly indicate the return value of the *main* function, in which case ANSI C dictates that the function actually returns a value:

```
#include <stdio.h>

int main ()

{
```

```
printf ("hello, world\n") ;

return 0 ;

}
```

We could make this even longer by including the arguments to *main*, but let's leave it at that_with an *include* statement, the program entry point, a call to a run−time library function, and a *return* statement.

The Windows Equivalent

The Windows equivalent to the "hello, world" program has exactly the same components as the character−mode version. It has an *include* statement, a program entry point, a function call, and a return statement. Here's the program:

```
/*————————————————————————————————————————————————————————
————————————

HelloMsg.c —— Displays "Hello, Windows 98!" in a message box

(c) Charles Petzold, 1998

————————————————————————————————————————————————————————
————————————*/

#include <windows.h>

9

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,

PSTR szCmdLine, int iCmdShow)

{

MessageBox (NULL, TEXT ("Hello, Windows 98!"), TEXT ("HelloMsg"), 0) ;

return 0 ;

}
```

Before I begin dissecting this program, let's go through the mechanics of creating a program in the

Visual C++ Developer Studio.

To begin, select New from the File menu. In the New dialog box, pick the Projects tab. Select Win32

Application. In the Location field, select a subdirectory. In the Project Name field, type the name of the project, which in this case is HelloMsg. This will be a subdirectory of the directory indicated in the Location field. The Create New Workspace button should be checked. The Platforms section should indicate Win32. Choose OK.

A dialog box labeled Win32 Application

− Step 1 Of 1 will appear. Indicate that you want to create an

Empty Project, and press the Finish button.

Select New from the File menu again. In the New dialog box, pick the Files tab. Select C++ Source File.

The Add to Project box should be checked, and HelloMsg should be indicated. Type HelloMsg.c in the File Name field. Choose OK. Now you can type in the HELLOMSG.C file shown above. Or you can select the Insert menu and the File as Text option to copy the contents of HELLOMSG.C from the file on this book's companion CD−ROM.

Structurally, HELLOMSG.C is identical to the K&R "hello, world" program. The header file STDIO.H has been replaced with WINDOWS.H, the entry point *main* has been replaced with *WinMain*, and the C run−time library function *printf* has been replaced with the Windows API function *MessageBox*.

However, there is much in the program that is new, including several strange−looking uppercase identifiers.

Let's start at the top.

The Header Files

HELLOMSG.C begins with a preprocessor directive that you'll find at the top of virtually every Windows

program written in C:

```
#include <windows.h>
```

WINDOWS.H is a master include file that includes other Windows header files, some of which also include other header files. The most important and most basic of these header files are:

· *WINDEF.H* Basic type definitions.

· *WINNT.H* Type definitions for Unicode support.

· *WINBASE.H* Kernel functions.

· *WINUSER.H* User interface functions.

· *WINGDI.H* Graphics device interface functions.

These header files define all the Windows data types, function calls, data structures, and constant identifiers. They are an important part of Windows documentation. You might find it convenient to use the Find In Files option from the Edit menu in the Visual C++ Developer Studio to search through these header files. You can also open the header files in the Developer Studio and examine them directly.

An Architectural Overview

When programming for Windows, you're really engaged in a type of object−oriented programming. This is most evident in the object you'll be working with most in Windows, the object that gives Windows its name, the object that will soon seem to take on anthropomorphic characteristics, the object that might even show up in your dreams: the object known as the "window." The most obvious windows adorning your desktop are application windows. These windows contain a title bar that shows the program's name, a menu, and perhaps a toolbar and a scroll bar. Another type of window is the dialog box, which may or may not have a title bar.

Less obvious are the various push buttons, radio buttons, check boxes, list boxes, scroll bars, and text−entry fields that adorn the surfaces of dialog boxes. Each of these little visual objects is a window.

More specifically, these are called "child windows" or "control windows" or "child window controls." The user sees these windows as objects on the screen and interacts directly with them using the keyboard or the mouse. Interestingly enough, the programmer's perspective is analogous to the user's perspective. The window receives the user input in the form of "messages" to the window. A window also uses messages to communicate with other windows. Getting a good feel for messages is an important part of learning how to write programs for Windows.

Here's an example of Windows messages: As you know, most Windows programs have sizeable application windows. That is, you can grab the window's border with the mouse and change the window's size. Often the program will respond to this change in size by altering the contents of its window. You might guess (and you would be correct) that Windows itself rather than the application is handling all the messy code involved with letting the user resize the window. Yet the application "knows" that the window has been resized because it can change the format of what it displays.

How does the application know that the user has changed the window's size? For programmers accustomed to only conventional character–mode programming, there is no mechanism for the operating system to convey information of this sort to the user. It turns out that the answer to this question is central to understanding the architecture of Windows. When a user resizes a window,

Windows sends a message to the program indicating the new window size. The program can then adjust the contents of its window to reflect the new size. "Windows sends a message to the program." I hope you didn't read that statement without blinking. What on earth could it mean? We're talking about program code here, not a telegraph system. How can an operating system send a message to a program? When I say that "Windows sends a message to the program" I mean that Windows calls a function within the program_a function that you write and which is an essential part of your program's code. The parameters to this function describe the particular message that is being sent by Windows and received by your program. This function in your program is known as the "window procedure."

You are undoubtedly accustomed to the idea of a program making calls to the operating system. This is how a program opens a disk file, for example. What you may not be accustomed to is the idea of an operating system making calls to a program. Yet this is fundamental to Windows' architecture.

Every window that a program creates has an associated window procedure. This window procedure is a function that could be either in the program itself or in a dynamic–link library. Windows sends a message to a window by calling the window procedure. The window procedure does some processing based on the message and then returns control to Windows. More precisely, a window is always created based on a "window class." The window class identifies the window procedure that processes messages to the window. The use of a window class allows multiple windows to be based on the same window class and hence use the same window procedure. For example, all buttons in all Windows programs are based on the same window class. This window class is associated with a window procedure located in a Windows dynamic−link library that processes messages to all the button windows. In object−oriented programming, an object is a combination of code and data. A window is an object.

The code is the window procedure. The data is information retained by the window procedure and information retained by Windows for each window and window class that exists in the system.

A window procedure processes messages to the window. Very often these messages inform a window of user input from the keyboard or the mouse. For example, this is how a push−button window knows that it's being "clicked." Other messages tell a window when it is being resized or when the surface of the window needs to be redrawn. When a Windows program begins execution, Windows creates a "message queue" for the program.

This message queue stores messages to all the windows a program might create. A Windows application includes a short chunk of code called the "message loop" to retrieve these messages from the queue and dispatch them to the appropriate window procedure. Other messages are sent directly to the window procedure without being placed in the message queue.

If your eyes are beginning to glaze over with this excessively abstract description of the Windows architecture, maybe it will help to see how the window, the window class, the window procedure, the message queue, the message loop, and the window messages all fit together in the context of a real program.

The HELLOWIN Program

Creating a window first requires registering a window class, and that requires a window procedure to process messages to the window. This involves a bit of overhead that appears in almost every Windows program. The HELLOWIN program, shown in Figure 3−1, is a simple program showing mostly that overhead.

Figure 3−1. *The HELLOWIN program.*

HELLOWIN.C

```
/*————————————————————————————————————————————————————————
——————————

HELLOWIN.C —— Displays "Hello, Windows 98!" in client area

(c) Charles Petzold, 1998

——————————————————————————————————————————————————————————
——————————*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM,
LPARAM) ;


int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE
hPrevInstance,

PSTR szCmdLine, int iCmdShow)

{

static TCHAR szAppName[] = TEXT ("HelloWin") ;

HWND hwnd ;

MSG msg ;

WNDCLASS wndclass ;

wndclass.style = CS_HREDRAW | CS_VREDRAW ;

wndclass.lpfnWndProc = WndProc ;

wndclass.cbClsExtra = 0 ;

wndclass.cbWndExtra = 0 ;

wndclass.hInstance = hInstance ;

wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;

wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;

wndclass.hbrBackground = (HBRUSH) GetStockObject
(WHITE_BRUSH) ;
```

```
wndclass.lpszMenuName = NULL ;
wndclass.lpszClassName = szAppName ;
if (!RegisterClass (&wndclass))
{
MessageBox (NULL, TEXT ("This program requires Windows NT!"),
szAppName, MB_ICONERROR) ;
return 0 ;
}
hwnd = CreateWindow (szAppName, // window class name
TEXT ("The Hello Program"), // window caption
WS_OVERLAPPEDWINDOW, // window style
CW_USEDEFAULT, // initial x position
CW_USEDEFAULT, // initial y position
CW_USEDEFAULT, // initial x size
CW_USEDEFAULT, // initial y size
NULL, // parent window handle
NULL, // window menu handle
hInstance, // program instance handle
NULL) ; // creation parameters
ShowWindow (hwnd, iCmdShow) ;
UpdateWindow (hwnd) ;
while (GetMessage (&msg, NULL, 0, 0))
{
TranslateMessage (&msg) ;
DispatchMessage (&msg) ;
}
return msg.wParam ;
```

```
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
HDC hdc ;
PAINTSTRUCT ps ;
RECT rect ;
switch (message)
{
case WM_CREATE:
PlaySound (TEXT ("hellowin.wav"), NULL, SND_FILENAME |
SND_ASYNC) ;
return 0 ;
case WM_PAINT:
hdc = BeginPaint (hwnd, &ps) ;
GetClientRect (hwnd, &rect) ;
DrawText (hdc, TEXT ("Hello, Windows 98!"), −1, &rect,

DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
EndPaint (hwnd, &ps) ;
return 0 ;
case WM_DESTROY:
PostQuitMessage (0) ;
return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

This program creates a normal application window, as shown in Figure 3−2, and displays, "Hello,

Windows 98!" in the center of that window. If you have a sound board installed, you will also hear me

saying the same thing.

Figure 3−2. *The HELLOWIN window.*

A couple of warnings: If you use Microsoft Visual C++ to create a new project for this program, you

need to make an addition to the object libraries the linker uses. Select the Settings option from the

Project menu, and pick the Link tab. Select General from the Category list box, and add WINMM.LIB

("Windows multimedia") to the Object/Library Modules text box. You need to do this because

HELLOWIN makes use of a multimedia function call, and the multimedia object library isn't included in a default project. Otherwise you'll get an error message from the linker indicating that the *PlaySound* function is unresolved.

HELLOWIN accesses a file named HELLOWIN.WAV, which is on the companion CD−ROM in the

HELLOWIN directory. When you execute HELLOWIN.EXE, the default directory must be HELLOWIN.

This is the case when you execute the program within Visual C++, even though the executable will be in the RELEASE or DEBUG subdirectory of HELLOWIN.

Thinking Globally

Most of HELLOWIN.C is overhead found in virtually every Windows program. Nobody really memorizes all the syntax to write this overhead; generally, Windows programmers begin a new program by copying an existing program and making appropriate changes to it. You're free to use the programs on the companion CD−ROM in this manner.

I mentioned above that HELLOWIN displays the text string in the center of its window. That's not precisely true. The text is actually displayed in the center of the program's "client area," which in Figure

3−2 is the large white area within the title bar and the sizing border. This distinction will be important to us; the client area is that area of the window in which a program is free to draw and deliver visual output to the user. When you think about it, this program has an amazing amount of functionality in its 80−odd lines of code. You can grab the title bar with the mouse and move the window around the screen. You can grab the sizing borders and resize the window. When the window changes size, the program automatically repositions the text string in the center of its client area. You can click the maximize button and zoom HELLOWIN to fill the screen. You can click the minimize button and clear it from the screen. You can invoke all these options from the system menu (the small icon at the far left of the title bar). You can also close the window to terminate the program by selecting the Close option from the system menu, by clicking the close button at the far right of the title bar, or by double−clicking the system menu icon. We'll be examining this program in detail for much of the remainder of the chapter. First, however, let's take a more global look.

HELLOWIN.C has a *WinMain* function like the sample programs in the first two chapters, but it also has a second function named *WndProc*. This is the window procedure. (In conversation among Windows programmers, it's called the "win prock.") Notice that there's no code in HELLOWIN.C that calls

WndProc. However, there is a reference to *WndProc* in *WinMain,* which is why the function is declared near the top of the program.

The Windows Function Calls

HELLOWIN makes calls to no fewer than 18 Windows functions. In the order they occur, these functions (with a brief description) are:

· *LoadIcon* Loads an icon for use by a program.

· *LoadCursor* Loads a mouse cursor for use by a program.

GetStockObject Obtains a graphic object, in this case a brush used for painting the window's background.

· *RegisterClass* Registers a window class for the program's window.

· *MessageBox* Displays a message box.

· *CreateWindow* Creates a window based on a window class.

· *ShowWindow* Shows the window on the screen.

· *UpdateWindow* Directs the window to paint itself.

· *GetMessage* Obtains a message from the message queue.

· *TranslateMessage* Translates some keyboard messages.

· *DispatchMessage* Sends a message to a window procedure.

· *PlaySound* Plays a sound file.

· *BeginPaint* Initiates the beginning of window painting.

· *GetClientRect* Obtains the dimensions of the window's client area.

· *DrawText* Displays a text string.

· *EndPaint* Ends window painting.

· PostQuitMessage Inserts a "quit" message into the message queue.

· *DefWindowProc* Performs default processing of messages.

These functions are described in the Platform SDK documentation, and they are declared in various header files, mostly in WINUSER.H.

Creating a Window

Window Classes

A *window class* defines a set of behaviors that several windows might have in common. For example, in a group of buttons, each button has a similar behavior when the user clicks the button. Of course, buttons are not completely identical; each button displays its own text string and has its own screen coordinates. Data that is unique for each window is called *instance data*.

Every window must be associated with a window class, even if your program only ever creates one instance of that class. It is important to understand that a window class is not a "class" in the C++ sense. Rather, it is a data structure used internally by the operating system. Window classes are registered with the system at run time. To register a new window class, start by filling in a **WNDCLASS** structure:

```
// Register the window class.
const wchar_t CLASS_NAME[]  = L"Sample Window Class";

WNDCLASS wc = { };

wc.lpfnWndProc   = WindowProc;
wc.hInstance     = hInstance;
wc.lpszClassName = CLASS_NAME;
```

You must set the following structure members:

- **lpfnWndProc** is a pointer to an application-defined function called the *window procedure* or "window proc." The window procedure defines most of the behavior of the window. We'll examine the window procedure in detail later. For now, just treat this as a forward reference.
- **hInstance** is the handle to the application instance. Get this value from the *hInstance* parameter of **wWinMain**.
- **lpszClassName** is a string that identifies the window class.

Class names are local to the current process, so the name only needs to be unique within the process. However, the standard Windows controls also have classes. If you use any of those controls, you must pick class names that do not conflict with the control class names. For example, the window class for the button control is named "Button".

The **WNDCLASS** structure has other members not shown here. You can set them to zero, as shown in this example, or fill them in. The MSDN documentation describes the structure in detail.

Next, pass the address of the **WNDCLASS** structure to the **RegisterClass** function. This function registers the window class with the operating system.

```
    RegisterClass(&wc);
```

Creating the Window

To create a new instance of a window, call the **CreateWindowEx** function:

```
    HWND hwnd = CreateWindowEx(
        0,                              // Optional window styles.
        CLASS_NAME,                     // Window class
        L"Learn to Program Windows",    // Window text
        WS_OVERLAPPEDWINDOW,            // Window style

        // Size and position
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT,

        NULL,       // Parent window
        NULL,       // Menu
        hInstance,  // Instance handle
        NULL        // Additional application data
        );

    if (hwnd == NULL)
    {
        return 0;
    }
```

You can read detailed parameter descriptions on MSDN, but here is a quick summary:

- The first parameter lets you specify some optional behaviors for the window (for example, transparent windows). Set this parameter to zero for the default behaviors.
- CLASS_NAME is the name of the window class. This defines the type of window you are creating.
- The window text is used in different ways by different types of windows. If the window has a title bar, the text is displayed in the title bar.
- The window style is a set of flags that define some of the look and feel of a window. The constant WS_OVERLAPPEDWINDOW is actually several flags combined with a bitwise **OR**. Together these flags give the window a title bar, a border, a system menu, and **Minimize** and**Maximize** buttons. This set of flags is the most common style for a top-level application window.
- For position and size, the constant CW_USEDEFAULT means to use default values.
- The next parameter sets a parent window or owner window for the new window. Set the parent if you are creating a child window. For a top-level window, set this to NULL.
- For an application window, the next parameter defines the menu for the window. This example does not use a menu, so the value is NULL.
- *hInstance* is the instance handle, described previously. (See WinMain: The Application Entry Point.)
- The last parameter is a pointer to arbitrary data of type **void***. You can use this value to pass a data structure to your window procedure. We'll show one possible way to use this parameter in the section Managing Application State.

CreateWindowEx returns a handle to the new window, or zero if the function fails. To show the window—that is, make the window visible —pass the window handle to the **ShowWindow** function:

```
ShowWindow(hwnd, nCmdShow);
```

The *hwnd* parameter is the window handle returned by **CreateWindowEx**. The *nCmdShow* parameter can be used to minimize or maximize a window. The operating system passes this value to the program through the **wWinMain** function.

Here is the complete code to create the window. Remember that WindowProc is still just a forward declaration of a function.

```
// Register the window class.
const wchar_t CLASS_NAME[]  = L"Sample Window Class";

WNDCLASS wc = { };

wc.lpfnWndProc   = WindowProc;
wc.hInstance     = hInstance;
wc.lpszClassName = CLASS_NAME;

RegisterClass(&wc);

// Create the window.

HWND hwnd = CreateWindowEx(
    0,                              // Optional window styles.
```

```
        CLASS_NAME,                     // Window class
        L"Learn to Program Windows",    // Window text
        WS_OVERLAPPEDWINDOW,            // Window style

        // Size and position
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

        NULL,       // Parent window
        NULL,       // Menu
        hInstance,  // Instance handle
        NULL        // Additional application data
        );

    if (hwnd == NULL)
    {
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);
```

Congratulations, you've created a window! Right now, the window does not contain any content or interact with the user. In a real GUI application, the window would respond to events from the user and the operating system. The next section describes how window messages provide this sort of interactivity.

Window Messages

A GUI application must respond to events from the user and from the operating system.

- **Events from the user** include all of the ways that someone can interact with your program: mouse clicks, key strokes, touch-screen gestures, and so forth.
- **Events from the operating system** include anything "outside" of the program that can affect how the program behaves. For example, the user might plug in a new hardware device, or Windows might enter a lower-power state (sleep or hibernate).

These events can occur at any time while the program is running, in almost any order. How do you structure a program whose flow of execution cannot be predicted in advance?

To solve this problem, Windows uses a message-passing model. The operating system communicates with your application window by passing messages to it. A message is simply a numeric code that designates a particular event. For example, if the user presses the left mouse button, the window receives a message with the following message code.

```
#define WM_LBUTTONDOWN    0x0201
```

Some messages have data associated with them. For example, the **WM_LBUTTONDOWN** message includes the x-coordinate and y-coordinate of the mouse cursor.

To pass a message to a window, the operating system calls the window procedure registered for that window. (And now you know what the window procedure is for.)

The Message Loop

An application will receive thousands of messages while it runs. (Consider that every keystroke and mouse-button click generates a message.) Furthermore, an application can have several windows, each with its own window procedure. How does the program receive all of these messages and deliver them to the right window procedure? The application needs a loop to get the messages and distpatch them to the correct windows.

For each thread that creates a window, the operating system creates a queue for window messages. This queue holds messages for all of the windows that are created on that thread. The queue itself is hidden from your progam. You can't manipulate the queue directly, but you can pull a message from the queue by calling the **GetMessage** function.

```
MSG msg;
GetMessage(&msg, NULL, 0, 0);
```

This function removes the first message from the head of the queue. If the queue is empty, the function blocks until another message is queued. The fact that **GetMessage** blocks will not make your program unresponsive. If there are no messages, there is nothing for the program to do. If you need to perform background processing, you can create additional threads that continue to run while **GetMessage**waits for another message. (See Avoiding Bottlenecks in Your Window Procedure.)

The first parameter of **GetMessage** is the address of a **MSG** structure. If the function succeeds, it fills in the **MSG** structure with information about the message, including the target window and the message code. The other three parameters give you the ability to filter which messages you get from the queue. In almost all cases, you will set these parameters to zero.

Although the **MSG** structure contains information about the message, you will almost never examine this structure directly. Instead, you will pass it directly to two other functions.

```
TranslateMessage(&msg);
DispatchMessage(&msg);
```

The **TranslateMessage** function is related to keyboard input; it translates keystrokes (key down, key up) into characters. You don't really need to know how this function works; just remember to call it right before **DispatchMessage**. The link to the MSDN documentation will give you more information, if you're curious.

The **DispatchMessage** function tells the operating system to call the window procedure of the window that is the target of the message. In other words, the operating system looks up the window handle in its table of windows, finds the function pointer associated with the window, and invokes the function.

For example, suppose the user presses the left mouse button. This causes a chain of events:

1. The operating system places a **WM_LBUTTONDOWN** message on the message queue.
2. Your program calls the **GetMessage** function.
3. **GetMessage** pulls the **WM_LBUTTONDOWN** message from the queue and fills in the **MSG** structure.
4. Your program calls the **TranslateMessage** and **DispatchMessage** functions.
5. Inside **DispatchMessage**, the operating system calls your window procedure.
6. Your window procedure can either respond to the message or ignore it.

When the window procedure returns, it returns back to **DispatchMessage**, which returns to the message loop for the next message. As long as your program is running, messages will continue to arrive on the queue. Therefore, you need a loop that continually pulls messages from the queue and dispatches them. You can think of the loop as doing the following:

```
// WARNING: Don't actually write your loop this way.

while (1)
{
    GetMessage(&msg, NULL, 0,  0);
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

As written, of course, this loop would never end. That's where the return value for the **GetMessage** function comes in.

Normally,**GetMessage** returns a non-zero value. Whenever you want to quit the application and break out of the message loop, simply call the**PostQuitMessage** function.

```
        PostQuitMessage(0);
```

The **PostQuitMessage** function puts a **WM_QUIT** message on the message queue. **WM_QUIT** is a special message: It causes **GetMessage** to return zero, signaling the end of the message loop. Here is the revised message loop.

```
// Correct.

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

As long as **GetMessage** returns a non-zero value, the expression in the **while** loop evaluates to true. After you call **PostQuitMessage**, the expression becomes false and the program breaks out of the loop. (One interesting consequence of this behavior is that your window procedure never receives a **WM_QUIT** message, so do not need a case statement for this message in your window procedure.)

The next obvious question is: When should you call **PostQuitMessage**? We'll return to this question in the topic Closing the Window, but first we need to write our window procedure.

Posted Messages versus Sent Messages

The previous section talked about messages going onto a queue. In some situations, the operating system will call a window procedure directly, bypassing the queue.

The terminology for this distinction can be confusing:

- *Posting* a message means the message goes on the message queue, and is dispatched through the message loop (**GetMessage** and**DispatchMessage**).
- *Sending* a message means the message skips the queue, and the operating system calls the window procedure directly.

For now, the distinction is not very important. The window procedure handles all messages, but some messages bypass the queue and go directly to your window procedure. However, it can make a difference if your application communicates between windows. You can find a more thorough discussion of this issue in the topic About Messages and Message Queues.

Writing the Window Procedure

The **DispatchMessage** function calls the window procedure of the window that is the target of the message. The window procedure has the following signature.

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

There are four parameters:

- *hwnd* is a handle to the window.
- *uMsg* is the message code; for example, the **WM_SIZE** message indicates the window was resized.
- *wParam* and *lParam* contain additional data that pertains to the message. The exact meaning depends on the message code.

LRESULT is an integer value that your program returns to Windows. It contains your program's response to a particular message. The meaning of this value depends on the message code. **CALLBACK** is the calling convention for the function.

A typical window procedure is simply a large switch statement that switches on the message code. Add cases for each message that you want to handle.

```
switch (uMsg)

{

case WM_SIZE: // Handle window resizing


// etc


}
```

Additional data for the message is contained in the *lParam* and *wParam* parameters. Both parameters are integer values the size of a pointer width (32 bits or 64 bits). The meaning of each depends on the message code (*uMsg*). For each message, you will need to look up the message code on MSDN and cast the parameters to the correct data type. Usually the data is either a numeric value or a pointer to a structure. Some messages do not have any data.

For example, the documentation for the **WM_SIZE** message states that:

- *wParam* is a flag that indicates whether the window was minimized, maximized, or resized.
- *lParam* contains the new width and height of the window as 16-bit values packed into one 32- or 64-bit number. You will need to perform some bit-shifting to get these values. Fortunately, the header file WinDef.h includes helper macros that do this.

A typical window procedure handles dozens of messages, so it can grow quite long. One way to make your code more modular is to put the logic for handling each message in a separate function. In the window procedure, cast the *wParam* and *lParam* parameters to the correct data type, and pass those values to the function. For example, to handle the **WM_SIZE** message, the window procedure would look like this:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg,
WPARAM wParam, LPARAM lParam)
```

```
{
    switch (uMsg)
    {
    case WM_SIZE:
        {
            int width = LOWORD(lParam);  // Macro to get the low-order
word.
            int height = HIWORD(lParam); // Macro to get the high-order
word.

            // Respond to the message:
            OnSize(hwnd, (UINT)wParam, width, height);
        }
        break;

    }
}

void OnSize(HWND hwnd, UINT flag, int width, int height);
{
    // Handle resizing
}
```

The **LOWORD** and **HIWORD** macros get the 16-bit width and height values from *lParam*. (You can look up these kinds of details in the MSDN documentation for each message code.) The window procedure extracts the width and height, and then passes these values to the OnSize function.

Default Message Handling

If you don't handle a particular message in your window procedure, pass the message parameters directly to the **DefWindowProc** function. This function performs the default action for the message, which varies by message type.

```
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
```

Avoiding Bottlenecks in Your Window Procedure

While your window procedure executes, it blocks any other messages for windows created on the same thread. Therefore, avoid lengthy processing inside your window procedure. For example, suppose your program opens a TCP connection and waits indefinitely for the server to respond. If you do that inside the window procedure, your UI will not respond until the request completes. During that time, the window cannot process mouse or keyboard input, repaint itself, or even close.

Instead, you should move the work to another thread, using one of the multitasking facilities that are built into Windows:

- Create a new thread.
- Use a thread pool.
- Use asynchronous I/O calls.
- Use asynchronous procedure calls.

Next

Painting the Window

Send comments about this topic to Microsoft

Build date: 10/5/2010

Community Additions

ADD

office

nothing

cyber_z

10/11/2015

(snicker)

Look carefully.... The parameter has two parts... LOW and HIGH words which are split for use as width/height. ;)

DezertRat

4/12/2013

error, I think in "writing a windows procedure"

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg,
WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_SIZE:
        {
            int width = LOWORD(lParam);  // Macro to get the low-order
word.
            int height = HIWORD(lParam); // Macro to get the high-order
word.

            // Respond to the message:
            OnSize(hwnd, (UINT)wParam, width, height);
        }
        break;

    }
}
```

Painting the Window

You've created your window. Now you want to show something inside it. In Windows terminology, this is called painting the window. To mix metaphors, a window is a blank canvas, waiting for you to fill it.

Sometimes your program will initiate painting to update the appearance of the window. At other times, the operating system will notify you that you must repaint a portion of the window. When this occurs, the operating system sends the window a **WM_PAINT** message. The portion of the window that must be painted is called the *update region*.

The first time a window is shown, the entire client area of the window must be painted. Therefore, you will always receive at least one**WM_PAINT** message when you show a window.

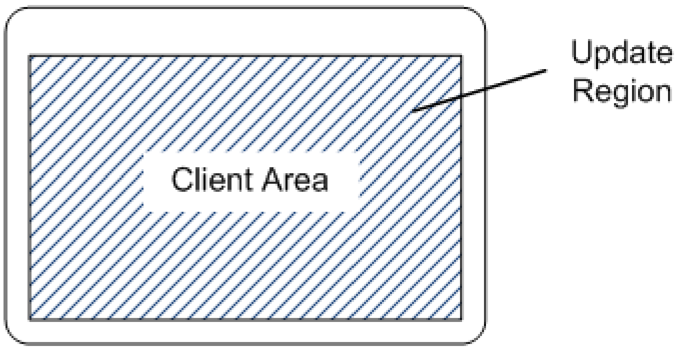


Illustration showing the update region of a window

You are only responsible for painting the client area. The surrounding frame, including the title bar, is automatically painted by the operating system. After you finish painting the client area, you clear the update region, which tells the operating system that it does not need to send another **WM_PAINT** message until something changes.

Now suppose the user moves another window so that it obscures a portion of your window. When the obscured portion becomes visible again, that portion is added to the update region, and your window receives another **WM_PAINT** message.

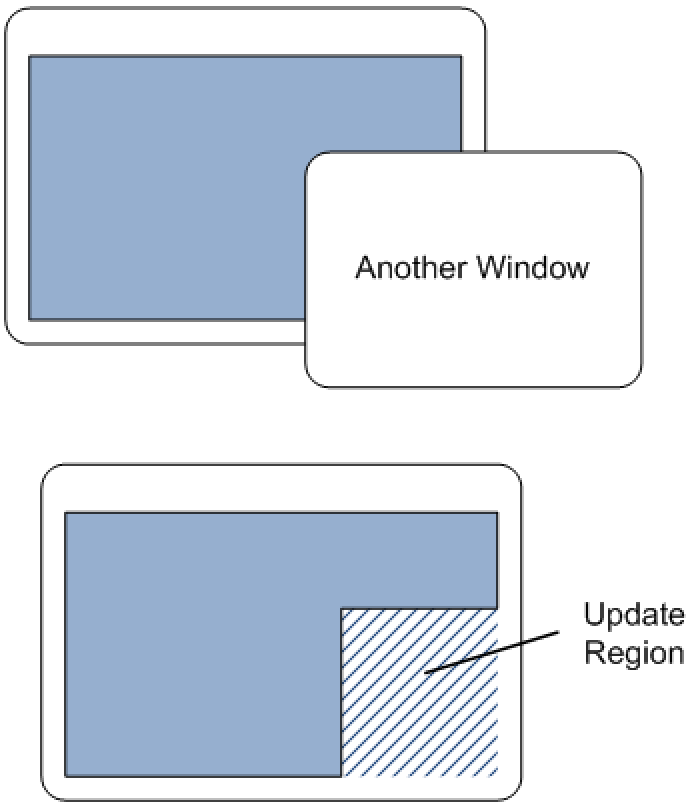


Illustration showing how the update region changes when two windows overlap

The update region also changes if the user stretches the window. In the following diagram, the user stretches the window to the right. The newly exposed area on the right side of the window is added to the update region:

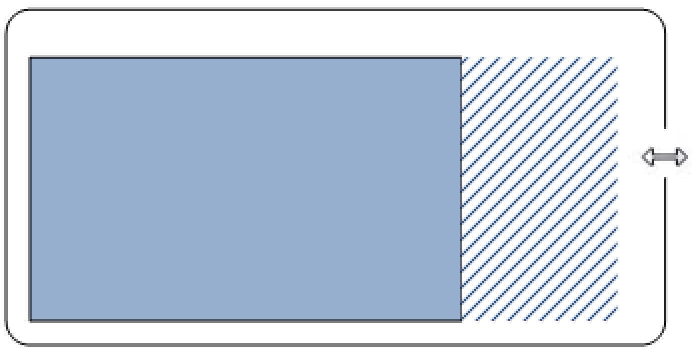


Illustration showing how the update region changes when a window is resized

In our first example program, the painting routine is very simple. It just fills the entire client area with a solid color. Still, this example is enough to demonstrate some of the important concepts.

```
switch (uMsg)
{

case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hwnd, &ps);

        // All painting occurs here, between BeginPaint and EndPaint.

        FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW+1));

        EndPaint(hwnd, &ps);
    }
    return 0;

}
```

Start the painting operation by calling the **BeginPaint** function. This function fills in the **PAINTSTRUCT** structure with information on the repaint request. The current update region is given in the **rcPaint** member of **PAINTSTRUCT**. This update region is defined relative to the client area:

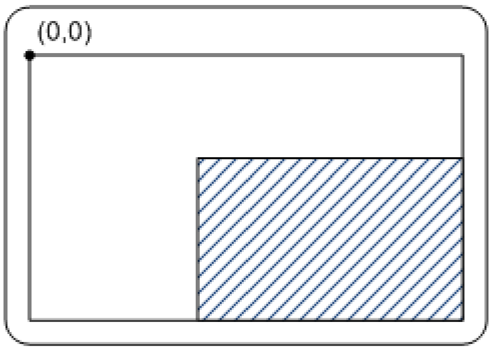


Illustration showing the origin of the client area

In your painting code, you have two basic options:

- Paint the entire client area, regardless of the size of the update region. Anything that falls outside of the update region is clipped. That is, the operating system ignores it.
- Optimize by painting just the portion of the window inside the update region.

If you always paint the entire client area, the code will be simpler. If you have complicated painting logic, however, it can be more efficient to skip the areas outside of the update region.

The following line of code fills the update region with a single color, using the system-defined window background color (COLOR_WINDOW). The actual color indicated by COLOR_WINDOW depends on the user's current color scheme.

```
FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW+1));
```

The details of **FillRect** are not important for this example, but the second parameter gives the coordinates of the rectangle to fill. In this case, we pass in the entire update region (the **rcPaint** member of **PAINTSTRUCT**). On the first **WM_PAINT** message, the entire client area needs to be painted, so **rcPaint** will contain the entire client area. On subsequent **WM_PAINT** messages, **rcPaint** might contain a smaller rectangle.

The **FillRect** function is part of the Graphics Device Interface (GDI), which has powered Windows graphics for a very long time. In Windows 7, Microsoft introduced a new graphics engine, named Direct2D, which supports high-performance graphics operations, such as hardware acceleration. Direct2D is also available for Windows Vista through the Platform Update for Windows Vista and for Windows Server 2008 through the Platform Update for Windows Server 2008. (GDI is still fully supported.)

After you are done painting, call the **EndPaint** function. This function clears the update region, which signals to Windows that the window has completed painting itself.

Closing the Window

When the user closes a window, that action triggers a sequence of window messages.

The user can close an application window by clicking the **Close** button, or by using a keyboard shortcut such as ALT+F4. Any of these actions causes the window to receive a **WM_CLOSE** message.
The **WM_CLOSE** message gives you an opportunity to prompt the user before closing the window. If you really do want to close the window, call the **DestroyWindow** function. Otherwise, simply return zero from the **WM_CLOSE**message, and the operating system will ignore the message and not destroy the window.

Here is an example of how a program might handle **WM_CLOSE**.

```
case WM_CLOSE:

    if (MessageBox(hwnd, L"Really quit?", L"My application", MB_OKCANCEL) == IDOK)

    {

        DestroyWindow(hwnd);

    }

    // Else: User canceled. Do nothing.

    return 0;
```

In this example, the **MessageBox** function shows a modal dialog that contains **OK** and **Cancel** buttons. If the user clicks **OK**, the program calls **DestroyWindow**. Otherwise, if the user clicks **Cancel**, the call to **DestroyWindow** is skipped, and the window remains open. In either case, return zero to indicate that you handled the message.

If you want to close the window without prompting the user, you could simply call **DestroyWindow** without the call to **MessageBox**. However, there is a shortcut in this case. Recall that **DefWindowProc** executes the default action for any window message. In the case of**WM_CLOSE**, **DefWindowProc** automatically calls **DestroyWindow**. That means if you ignore the **WM_CLOSE** message in your **switch**statement, the window is destroyed by default.

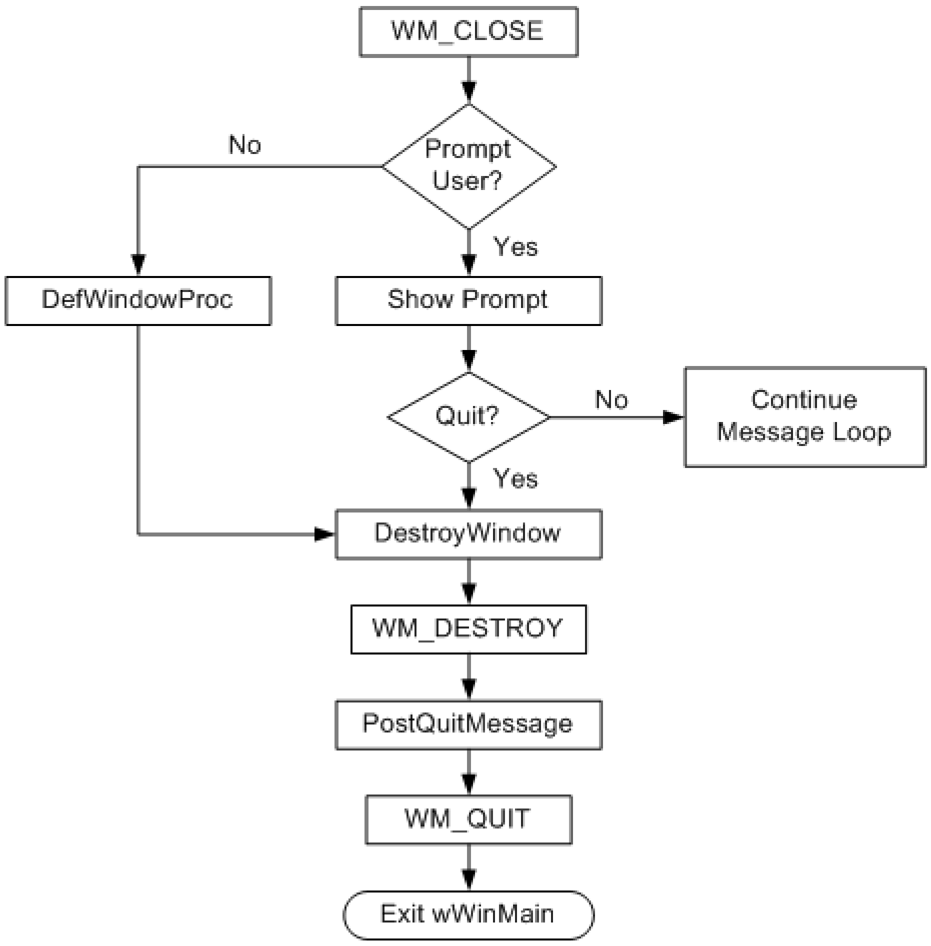
When a window is about to be destroyed, it receives a **WM_DESTROY** message. This message is sent after the window is removed from the screen, but before the destruction occurs (in particular, before any child windows are destroyed).

In your main application window, you will typically respond to **WM_DESTROY** by calling **PostQuitMessage**.

```
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
```

We saw in the Window Messages section that **PostQuitMessage** puts a **WM_QUIT** message on the message queue, causing the message loop to end.

Here is a flow chart showing the typical way to process **WM_CLOSE** and **WM_DESTROY** messages:



Flow chart showing how to handle WM_CLOSE and WM_DESTROY messages

The Windows Function Calls

HELLOWIN makes calls to no fewer than 18 Windows functions. In the order they occur, these

functions (with a brief description) are:

· *LoadIcon* Loads an icon for use by a program.

· *LoadCursor* Loads a mouse cursor for use by a program.

GetStockObject Obtains a graphic object, in this case a brush used for painting the window's

background.

·

· *RegisterClass* Registers a window class for the program's window.

· *MessageBox* Displays a message box.

· *CreateWindow* Creates a window based on a window class.

· *ShowWindow* Shows the window on the screen.

· *UpdateWindow* Directs the window to paint itself.

· *GetMessage* Obtains a message from the message queue.

· *TranslateMessage* Translates some keyboard messages.

· *DispatchMessage* Sends a message to a window procedure.

· *PlaySound* Plays a sound file.

· *BeginPaint* Initiates the beginning of window painting.

· *GetClientRect* Obtains the dimensions of the window's client area.

· *DrawText* Displays a text string.

· *EndPaint* Ends window painting.

· *PostQuitMessage* Inserts a "quit" message into the message queue.

· *DefWindowProc* Performs default processing of messages.

These functions are described in the Platform SDK documentation, and they are declared in various

header files, mostly in WINUSER.H.

Application Creation

The Main Window Class

There are two primary things you must do in order to create even the simplest window: you must create the central point of the program, and you must tell the operating system how to respond when the user does what.

Just like a C++ program always has a **main()** function, a Win32 program needs a central function call **WinMain**. The syntax of that function is:

INT WINAPI WinMain(HINSTANCE *hInstance*, HINSTANCE *hPrevInstance*,

LPSTR *lpCmdLine*, int *nCmdShow*);

Unlike the C++ **main()** function, the arguments of the **WinMain()** function are not optional. Your program will need them to communicate with the operating system.

The first argument, *hInstance*, is a handle to the instance of the program you are writing.

The second argument, *hPrevInstance,* is used if your program had any previous instance. If not, this argument can be ignored, which will always be the case.

The third argument, *lpCmdLine*, is a string that represents all items used on the command line to compile the application.

The last argument, *nCmdShow*, controls how the window you are building will be displayed.

An object that displays on your screen is called a window. Because there can be various types of windows in your programs, your first responsibility is to control them, know where they are, what they are doing, why, and when. The first control you must exercise on these different windows is to host them so that all windows of your program belong to an entity called the main window. This main window is created using an object that can be called a class (strictly, a structure).

The Win32 library provides two classes for creating the main window and you can use any one of them. They are **WNDCLASS** and **WNDCLASSEX**. The second adds only a slight feature to the first. Therefore, we will mostly use the **WNDCLASSEX** structure for our lessons.

The **WNDCLASS** and the **WNDCLASSEX** classes are defined as follows:

```
typedef struct _WNDCLASS {        typedef struct _WNDCLASSEX {
  UINT      style;                  UINT      cbSize;
  WNDPROC   lpfnWndProc;            UINT      style;
  int       cbClsExtra;             WNDPROC   lpfnWndProc;
  int       cbWndExtra;             int       cbClsExtra;
```

```
    HINSTANCE  hInstance;            int       cbWndExtra;
    HICON      hIcon;                HINSTANCE  hInstance;
    HCURSOR    hCursor;              HICON      hIcon;
    HBRUSH     hbrBackground;        HCURSOR    hCursor;
    LPCTSTR    lpszMenuName;         HBRUSH     hbrBackground;
    LPCTSTR    lpszClassName;        LPCTSTR    lpszMenuName;
  } WNDCLASS, *PW                    LPCTSTR    lpszClassName;
                                     HICON      hIconSm;
                                   } WNDCLASSEX,
                                   *PWNDCLASSEX;
```

To create a window, you must "fill out" this class, which means you must provide a value for each of its members so the operating system would know what your program is expected to do.

The first thing you must do in order to create an application is to declare a variable of either**WNDCLASS** or **WNDCLASSEX** type. Here is an example of a **WNDCLASSEX** variable:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX WndClsEx;


    return 0;
}
```

The Size of the Window Class

After declaring a **WNDCLASSEX** variable, you must specify its size. This is done by initializing your variable with the sizeof operator applied to the window class as follows:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
```

```
{
        WNDCLASSEX WndClsEx;

        WndClsEx.cbSize = sizeof(WNDCLASSEX);

   return 0;
}
```

Additional Memory Request

Upon declaring a **WNDCLASSEX** variable, the compiler allocates an amount of memory space for it, as it does for all other variables. If you think you will need more memory than allocated, assign the number of extra bytes to the *cbClsExtra* member variable. Otherwise, the compiler initializes this variable to 0. If you do not need extra memory for your **WNDCLASSEX** variable, initialize this member with 0. Otherwise, you can do it as follows:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow)
{
        WNDCLASSEX WndClsEx;

        WndClsEx.cbSize     = sizeof(WNDCLASSEX);
        WndClsEx.cbClsExtra = 0;

        return 0;
}
```

The Application's Instance

Creating an application is equivalent to creating an instance for it. To communicate to the**WinMain()** function that you want to create an instance for your application, which is, to make it available as a resource, assign the **WinMain()**'s *hInstance* argument to your **WNDCLASS**variable:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
          WNDCLASSEX WndClsEx;

          WndClsEx.cbSize     = sizeof(WNDCLASSEX);
          WndClsEx.cbClsExtra = 0;
          WndClsEx.hInstance  = hInstance;

          return 0;
}
```

Window Extra-Memory

When an application has been launched and is displaying on the screen, which means an instance of the application has been created, the operating system allocates an amount of memory space for that application to use. If you think that your application's instance will need more memory than that, you can request that extra memory bytes be allocated to it. Otherwise, you can let the operating system handle this instance memory issue and initialize the *cbWndExtra* member variable to 0:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
          WNDCLASSEX WndClsEx;

          WndClsEx.cbSize     = sizeof(WNDCLASSEX);
          WndClsEx.cbClsExtra = 0;
          WndClsEx.cbWndExtra = 0;
          WndClsEx.hInstance  = hInstance;

          return 0;
}
```

The Main Window's Style

The *style* member variable specifies the primary operations applied on the window class. The actual available styles are constant values. For example, if a user moves a window or changes its size, you would need the window to be redrawn to get its previous characteristics. To redraw the window horizontally, you would apply the **CS_HREDRAW**. In the same way, to redraw the window vertically, you can apply the **CS_VREDRAW**.

The styles are combined using the bitwise OR (|) operator. The **CS_HREDRAW** and the**CS_VREDRAW** styles can be combined and assigned to the style member variable as follows:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
        WNDCLASSEX WndClsEx;

        WndClsEx.cbSize     = sizeof(WNDCLASSEX);
        WndClsEx.style      = CS_HREDRAW | CS_VREDRAW;
        WndClsEx.cbClsExtra = 0;
        WndClsEx.cbWndExtra = 0;
        WndClsEx.hInstance  = hInstance;

        return 0;
}
```

Message Processing

The name of the window procedure **we reviewed in the previous lesson** must be assigned to the *lpfnWndProc* member variable of the **WNDCLASS** or **WNDCLASSEX** variable. This can be defined as follows:

```
#include <windows.h>
```

```
LRESULT WndProcedure(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
          WNDCLASSEX WndClsEx;

          WndClsEx.cbSize      = sizeof(WNDCLASSEX);
          WndClsEx.style       = CS_HREDRAW | CS_VREDRAW;
          WndClsEx.lpfnWndProc = WndProcedure;
          WndClsEx.cbClsExtra  = 0;
          WndClsEx.cbWndExtra  = 0;
          WndClsEx.hInstance   = hInstance;

          return 0;
}

LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg, WPARAM
wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
```

The Application Main Icon

An icon can be used to represent an application in My Computer or Windows Explorer. To assign this small picture to your application, you can either use an existing icon or design your own. To make your programming a little faster, Microsoft Windows installs a few icons. The icon is assigned to the *hIcon* member variable using the **LoadIcon()** function. For a Win32 application, the syntax of this function is:

HICON LoadIcon(HINSTANCE *hInstance*, LPCTSTR *lpIconName*);

The *hInstance* argument is a handle to the file in which the icon was created. This file is usually stored in a library (DLL) of an executable program. If the icon was created as part of your application, you can use the *hInstance* of your application. If your are using one of the icons below, set this argument to NULL.

The *lpIconName* is the name of the icon to be loaded. This name is added to the resource file when you create the icon resource. It is added automatically if you add the icon as part of your resources; otherwise you can add it manually when creating your resource script. Normally, if you had created and designed an icon and gave it an identifier, you can pass it using the **MAKEINTRESOURCE** macro.

To make your programming a little faster, Microsoft Windows installs a few icons you can use for your application. These icons have identification names that you can pass to the**LoadIcon()** function as the *lpIconName* argument. The icons are:

ID	Picture
IDI_APPLICATION	▭
IDI_INFORMATION	ℹ
IDI_ASTERISK	ℹ
IDI_QUESTION	?

IDI_WARNING	⚠
IDI_EXCLAMATION	⚠
IDI_HAND	❌
IDI_ERROR	❌

If you designed your own icon (you should make sure you design a 32x32 and a 16x16 versions, even for convenience), to use it, specify the *hInstance* argument of the **LoadIcon()**function to the instance of your application. Then use the **MAKEINTRESOURCE** macro to convert its identifier to a null-terminated string. This can be done as follows:

```
WndCls.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_STAPLE));
```

The icon can be specified by its name, which would be a null-terminated string passed as*lpszResourceName*. If you had designed your icon and gave it an ID, you can pass this identifier to the **LoadIcon()** method.

The **LoadIcon()** member function returns an **HICON** object that you can assign to the *hIcon*member variable of your **WNDCLASS** object. Besides the regular (32x32) icon, the WNDCLASSEX structure allows you to specify a small icon (16x16) to use in some circumstances. You can specify both icons as follows:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
        WNDCLASSEX WndClsEx;


        WndClsEx.cbSize      = sizeof(WNDCLASSEX);
        WndClsEx.style       = CS_HREDRAW | CS_VREDRAW;
        WndClsEx.lpfnWndProc = WndProcedure;
        WndClsEx.cbClsExtra  = 0;
        WndClsEx.cbWndExtra  = 0;
        WndClsEx.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
        WndClsEx.hInstance   = hInstance;
        WndClsEx.hIconSm     = LoadIcon(NULL, IDI_APPLICATION);


        return 0;
}
```

Introduction to Cursors

A cursor is used to locate the position of the mouse pointer on a document or the screen. To use a cursor, call the Win32 **LoadCursor()** function. Its syntax is:

HCURSOR LoadCursor(HINSTANCE *hInstance*, LPCTSTR *lpCursorName*);

The *hInstance* argument is a handle to the file in which the cursor was created. This file is usually stored in a library (DLL) of an executable program. If the cursor was created as part of your application, you can use the *hInstance* of your application. If your are using one of the below cursors, set this argument to NULL.

When Microsoft Windows installs, it also installs various standard cursors you can use in your program. Each one of these cursors is recognized by an ID which is simply a constant integers. The available cursors are:

ID	**Picture**	**Description**
IDC_APPSTARTING	↖⌛	Used to show that something undetermined is going on or the application is not stable
IDC_ARROW	↖	This standard arrow is the most commonly used cursor
IDC_CROSS	+	The crosshair cursor is used in various circumstances such as drawing
IDC_HAND	☝	The Hand is standard only in Windows 2000. If you are using a previous operating system and need this cursor, you may have to create your own.
IDC_HELP	↖?	The combined arrow and question mark cursor is used when providing help on a specific item on a window object
IDC_IBEAM	I	The I-beam cursor is used on text-based object to show the position of the caret

IDC_ICON		This cursor is not used anymore
IDC_NO	⊘	This cursor can be used to indicate an unstable situation
IDC_SIZE		This cursor is not used anymore
IDC_SIZEALL	✥	The four arrow cursor pointing north, south, east, and west is highly used to indicate that an object is selected or that it is ready to be moved
IDC_SIZENESW	⤢	The northeast and southwest arrow cursor can be used when resizing an object on both the length and the height
IDC_SIZENS	↕	The north - south arrow pointing cursor can be used when shrinking or heightening an object
IDC_SIZENWSE	⤡	The northwest - southeast arrow pointing cursor can be used when resizing an object on both the length and the height
IDC_SIZEWE	↔	The west - east arrow pointing cursor can be used when narrowing or enlarging an object
IDC_UPARROW	↑	The vertical arrow cursor can be used to indicate the presence of the mouse or the caret
IDC_WAIT	⌛	The Hourglass cursor is usually used to indicate that a window or the application is not ready.

The **LoadCursor()** member function returns an **HCURSOR** value. You can assign it to the*hCursor* member variable of your **WNDCLASS** object. Here is an example:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
        WNDCLASSEX WndClsEx;


        WndClsEx.cbSize      = sizeof(WNDCLASSEX);
```

```
	WndClsEx.style       = CS_HREDRAW | CS_VREDRAW;
	WndClsEx.lpfnWndProc = WndProcedure;
	WndClsEx.cbClsExtra  = 0;
	WndClsEx.cbWndExtra  = 0;
	WndClsEx.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
	WndClsEx.hCursor     = LoadCursor(NULL, IDC_ARROW);
	WndClsEx.hInstance   = hInstance;
	WndClsEx.hIconSm     = LoadIcon(NULL, IDI_APPLICATION);

	return 0;
}
```

The Window's Background Color

To paint the work area of the window, you must specify what color will be used to fill it. T
valid **HBRUSH** or you can cast a known color to **HBRUSH**. The Win32 library defines a
the **BLACK_BRUSH** constant to the**GetStockObject()** function, cast it to **HBRUSH** anc

In addition to the stock objects, the Microsoft Windows provides a series of colors for its
from their
name) **COLOR_ACTIVEBORDER**, **COLOR_ACTIVECAPTION**,**COLOR_APPWO**
LIGHT, **COLOR_HIGHLIGHTTEXT**, **COLOR_INACTIVEBORDER**,**COLOR_IN**
can use any of these colors to paint the background of your window. First cast it to **HBRU**

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
		LPSTR lpCmdLine, int nCmdShow)
{
	WNDCLASSEX WndClsEx;

	WndClsEx.cbSize        = sizeof(WNDCLASSEX);
	WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
	WndClsEx.lpfnWndProc   = WndProcedure;
	WndClsEx.cbClsExtra    = 0;
	WndClsEx.cbWndExtra    = 0;
	WndClsEx.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
	WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
	WndClsEx.hbrBackground = GetStockObject(WHITE_BRUSH);
	WndClsEx.hInstance     = hInstance;
	WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

	return 0;
}
```

<u>The Application's Main Menu</u>

If you want the window to display a menu, first create or design the resource menu (we will eventually learn how to do this). After creating the menu, assign its name to the*lpszMenuName* name to your **WNDCLASS** or **WNDCLASSEX** variable. Otherwise, pass this argument as NULL. Here is an example:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
	WNDCLASSEX WndClsEx;

	WndClsEx.cbSize        = sizeof(WNDCLASSEX);
	WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
	WndClsEx.lpfnWndProc   = WndProcedure;
	WndClsEx.cbClsExtra    = 0;
	WndClsEx.cbWndExtra    = 0;
	WndClsEx.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
	WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
	WndClsEx.hbrBackground = GetStockObject(WHITE_BRUSH);
	WndClsEx.lpszMenuName  = NULL;
	WndClsEx.hInstance     = hInstance;
	WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);


	return 0;
}
```

The Window's Class Name

To create a window, you must provide its name as everything else in the computer has a name. The class name of your main window must be provided to the *lpszClassName* member variable of your **WNDCLASS** or **WNDCLASSEX** variable. You can provide the name to the variable or declare a global null-terminated string. Here is an example:

```
LPCTSTR ClsName = L"BasicApp";


INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
	WNDCLASSEX WndClsEx;


	WndClsEx.cbSize        = sizeof(WNDCLASSEX);
	WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
	WndClsEx.lpfnWndProc   = WndProcedure;
	WndClsEx.cbClsExtra    = 0;
	WndClsEx.cbWndExtra    = 0;
	WndClsEx.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
	WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
	WndClsEx.hbrBackground = GetStockObject(WHITE_BRUSH);
```

```
        WndClsEx.lpszMenuName  = NULL;
        WndClsEx.lpszClassName = ClsName;
        WndClsEx.hInstance     = hInstance;
        WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

        return 0;
}
```

Finalizing an Application

Window Registration

After initializing the window class, you must make it available to the other controls that will be part of your application. This process is referred to as registration. To register the window class, call the **RegisterClass()** for a **WNDCLASS** variable. If you created your window class using the **WNDCLASSEX** structure, call the **RegisterClassEx()** function. Their syntaxes are:

```
ATOM RegisterClass(CONST WNDCLASS *lpWndClass);

ATOM RegisterClassEx(CONST WNDCLASSEX *lpwcx);
```

The function simply takes as argument a pointer to a **WNDCLASS** or **WNDCLASSEX**. This call can be done as follows:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow)
{
        WNDCLASSEX WndClsEx;


        . . .
```

```
    RegisterClassEx(&WndClsEx);


    return 0;
}
```

Window Creation

The **WNDLCLASS** and the **WNDCLASSEX** classes are used to initialize the application window class. To display a window, that is, to give the user an object to work with, you must create a window object. This window is the object the user uses to interact with the computer.

To create a window, you can call either the **CreateWindow()** or the **CreateWindowEx()**function. We will come back to these functions.

You can simply call this function and specify its arguments after you have registered the window class. Here is an example:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                   LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX   WndCls;


    . . .


    RegisterClassEx(&WndClsEx);


    CreateWindow(. . .);
}
```

If you are planning to use the window further in your application, you should retrieve the result of the **CreateWindow()** or the **CreateWindowEx()** function, which is a handle to the window that is being created. To do this, you can declare an **HWND** variable and initialize it with the create function. This can be done as follows:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    HWND        hWnd;
    WNDCLASSEX  WndClsEx;

    . . .

    RegisterClassEx(&WndClsEx);

    hWnd = CreateWindow(. . .);
}
```

We will come back to the other characteristics of a window in future lessons.

Window's Message Decoding

Once a window has been created, the user can use it. This is done by the user clicking things with the mouse or pressing keys on the keyboard. A message that a window sends is received by the application. This application must analyze, translate, and decode the message to know what object sent the message and what the message consists of. To do this, the application uses the **GetMessage()** function. Its syntax is:

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

The *lpMsg* argument is a pointer to the **MSG** structure. The MSG structure is defined as follows:

```
typedef struct tagMSG {
  HWND   hwnd;
  UINT   message;
  WPARAM wParam;
  LPARAM lParam;
```

```
    DWORD  time;
    POINT  pt;
} MSG, *PMSG;
```

The *hWnd* argument identifies which window sent the message. If you want the messages of all windows to be processed, pass this argument as NULL.

The *wMsgFilterMin* and the *wMsgFilterMax* arguments specify the message values that will be treated. If you want all messages to be considered, pass each of them as 0.

Once a message has been sent, the application analyzes it using the **TranslateMessage()**function. Its syntax is:

BOOL TranslateMessage(CONST MSG **lpMsg*);

This function takes as argument the **MSG** object that was passed to the **GetMessage()**function and analyzes it. If this function successfully translates the message, it returns TRUE. If it cannot identify and translate the message, it returns FALSE.

Once a message has been decoded, the application must send it to the window procedure. This is done using the **DispatchMessage()** function. Its syntax is:

LRESULT DispatchMessage(CONST MSG **lpMsg*);

This function also takes as argument the **MSG** object that was passed to **GetMessage()** and analyzed by **TranslateMessage()**.
This **DispatchMessage()** function sends the *lpMsg*message to the window procedure. The window procedure processes it and sends back the result, which becomes the return value of this function. Normally, when the window procedure receives the message, it establishes a relationship with the control that sent the message and starts treating it. By the time the window procedure finishes with the message, the issue is resolved (or aborted). This means that, by the time the window procedure returns its result, the message is not an issue anymore. For this reason you will usually, if ever, not need to retrieve the result of the **DispatchMessage()** function.

This translating and dispatching of messages is an on-going process that goes on as long as your application is running and as long as somebody is using it. For this reason, the application uses a **while** loop to continuously check new messages. This behavior can be implemented as follows:

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

If the **WinMain()** function successfully creates the application and the window, it returns the*wParam* value of the MSG used on the application.

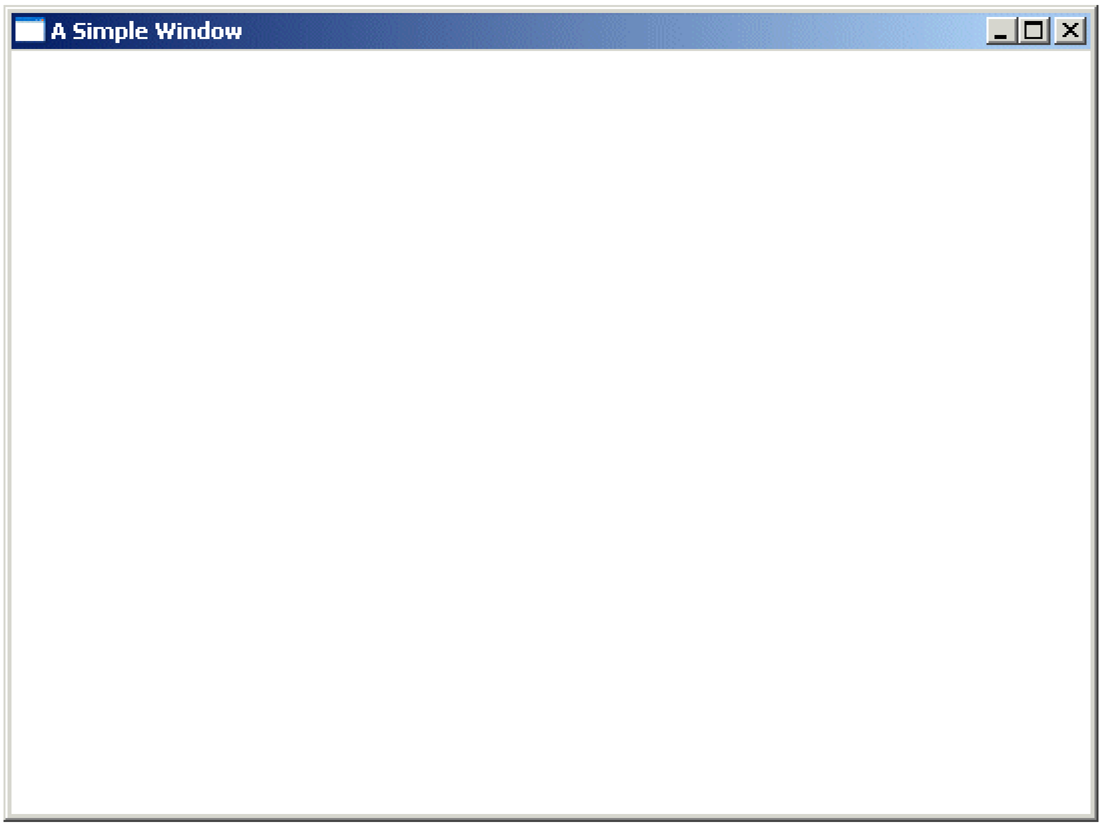
Practical Learning: Creating a Sample Application

1. Replace the file with the following (the file in Borland C++ Builder contains some lines with #pragma; you don't need to delete these files because their presence or absence will not have a negative impact on the compilation of the program):

```
#include <windows.h>


LPCTSTR ClsName = L"BasicApp";
LPCTSTR WndName = L"A Simple Window";


LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                              WPARAM wParam, LPARAM lParam);


INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
          LPSTR lpCmdLine, int nCmdShow)
{
     MSG        Msg;
     HWND       hWnd;
     WNDCLASSEX WndClsEx;
```

```
// Create the application window
WndClsEx.cbSize        = sizeof(WNDCLASSEX);
WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
WndClsEx.lpfnWndProc   = WndProcedure;
WndClsEx.cbClsExtra    = 0;
WndClsEx.cbWndExtra    = 0;
WndClsEx.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH)
WndClsEx.lpszMenuName  = NULL;
WndClsEx.lpszClassName = ClsName;
WndClsEx.hInstance     = hInstance;
WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

// Register the application
RegisterClassEx(&WndClsEx);

// Create the window object
hWnd = CreateWindow(ClsName,
              WndName,
              WS_OVERLAPPEDWINDOW,
              CW_USEDEFAULT,
              CW_USEDEFAULT,
              CW_USEDEFAULT,
              CW_USEDEFAULT,
              NULL,
              NULL,
                        hInstance,
                        NULL);

        // Find out if the window was created
        if( !hWnd ) // If the window was not created,
            return 0; // stop the application

        // Display the window to the user
        ShowWindow(hWnd, SW_SHOWNORMAL);
        UpdateWindow(hWnd);

        // Decode and treat the messages
        // as long as the application is running
        while( GetMessage(&Msg, NULL, 0, 0) )
        {
          TranslateMessage(&Msg);
          DispatchMessage(&Msg);
        }

        return Msg.wParam;
}

LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                        WPARAM wParam, LPARAM lParam)
{
   switch(Msg)
   {
    // If the user wants to close the application
    case WM_DESTROY:
        // then close it
        PostQuitMessage(WM_QUIT);
        break;
    default:
        // Process the left-over messages
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    // If something was not done, let it go
    return 0;
}
```

2. To execute the program, if you are using Borland C++ Builder, press F9
   If you are using Microsoft Visual C++, press Ctrl + F5 and click Yes



3. To close the window, click its system Close button ☒ and return to your programming environment.

Introduction to Resources

Resources Fundamentals

Introduction

A resource is an object that cannot be defined in C++ terms but that is needed to complete a program. In the strict sense, it is text that contains a series of terms or words that the program can interpret through code. Examples of resources are menus, icons, cursors, dialog boxes, sounds, etc.

There are various means of creating a resource and the approach you use depends on the resource. For example, some resources are completely text-based, such is the case for the String Table or the Accelerator Table. Some other resources must be designed, such is the case for icons and cursors. Some other resources can be imported from another, more elaborate application, such is the case for high graphic pictures. Yet some resources can be a combination of different resources.

Resource Creation

As mentioned already, resources are not a C++ concept but a Microsoft Windows theory o
application. Therefore, the programming environment you use may or may not provide yo
creating certain resources. Some environments like Borland C++ Builder or Visual C++ (6
complete with (almost) anything you need to create (almost) any type of resources. Some o
may appear incomplete, allowing you to create only some resources, the other resources m
an external application not provided; such is the case for C++BuilderX.

Upon creating a resource, you must save it. Some resources are created as their own file, s
pictures, icons, cursors, sound, etc. Each of these resources has a particular extension depe
resource. After creating the resources, you must add them to a file that has the extension .r
listed in this file using a certain syntax. That's the case for icons, cursors, pictures, sounds,
resources must be created directly in this file because these resources are text-based; that's
strings, accelerators, version numbers, etc.

After creating the resource file, you must compile it. Again, some environments, such as M
do this automatically when you execute the application. Some other environments may rec
compile the resource. That's the case for Borland C++ Builder and C++BuilderX. (The fac
environments require that you compile the resource is not an anomaly. For example, if you
application that is form-based in C++ Builder 6 or Delphi, you can easily add the resource
automatically compiled and added to the application. If you decide to create a Win32 appl
believes that you want to completely control your application; so, it lets you decide when a
resource. This means that it simply gives you more control).

Practical Learning: Introducing Windows Resources

1. If you are using Borland C++ Builder, create a new Win32 application using Conso
   Lesson 1
   a. Save the project as **Resources1** in a new folder called **Resources1**
   b. Save the unit as **Exercise.cpp**
2. If you are using Microsoft Visual C++,
   a. Create a new Win32 Application as done the previous time. Save the project a
   b. Create a new C++ Source file named **Exercise.cpp**
3. Implement the source file as follows(for Borland C++ Builder, only add the parts th
   code):

```
//---------------------------------------------------------------------------

#include <windows.h>

#pragma hdrstop


//---------------------------------------------------------------------------


#pragma argsused

//---------------------------------------------------------------------------

LPCTSTR ClsName = L"FundApp";

LPCTSTR WndName = L"Resources Fundamentals";

LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,

                              WPARAM wParam, LPARAM lParam);
```

```
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
        MSG        Msg;
        HWND       hWnd;
        WNDCLASSEX WndClsEx;

        // Create the application window
        WndClsEx.cbSize        = sizeof(WNDCLASSEX);
        WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
        WndClsEx.lpfnWndProc   = WndProcedure;
        WndClsEx.cbClsExtra    = 0;
        WndClsEx.cbWndExtra    = 0;
        WndClsEx.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
        WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
        WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
        WndClsEx.lpszMenuName  = NULL;
        WndClsEx.lpszClassName = ClsName;
        WndClsEx.hInstance     = hInstance;
        WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

        // Register the application
        RegisterClassEx(&WndClsEx);

        // Create the window object
        hWnd = CreateWindowEx(0,
                ClsName,
                WndName,
                                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,
                                CW_USEDEFAULT,
                CW_USEDEFAULT,
                                CW_USEDEFAULT,
                NULL,
                NULL,
                hInstance,
                NULL);

// Find out if the window was created
if( !hWnd ) // If the window was not created,
          return FALSE; // stop the application

// Display the window to the user
ShowWindow(hWnd, nCmdShow);// SW_SHOWNORMAL);
UpdateWindow(hWnd);

// Decode and treat the messages
// as long as the application is running
while( GetMessage(&Msg, NULL, 0, 0) )
{
  TranslateMessage(&Msg);
  DispatchMessage(&Msg);
}
```

```
        return Msg.wParam;
}
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM lParam)
{
   switch(Msg)
   {
   case WM_DESTROY:
      PostQuitMessage(WM_QUIT);
      break;
   default:
      // Process the left-over messages
      return DefWindowProc(hWnd, Msg, wParam, lParam);
   }
   // If something was not done, let it go
   return 0;
}
//---------------------------------------------------------------------------
```

4. Execute the application to test it

Introduction to Resources: Cursors

Introduction

A cursor is an object used to show or represent the current position of the mouse on the sc
Windows ships with many cursors as follows:

IDC_APPSTARTING	↖⌛	IDC_ARROW	↖	IDC_CROSS	+	IDC_HAN

IDC_HELP	↖?	IDC_IBEAM	I	IDC_ICON	*Not Used(*)*	IDC_NO
IDC_SIZE	*Not Used(*)*	IDC_SIZEALL	✥	IDC_SIZENESW	⤢	IDC_SIZE
IDC_SIZENWSE	⤡	IDC_SIZEWE	↔	IDC_UPARROW	↑	IDC_WAI

(*) The **IDC_ICON** and the **IDC_SIZE** cursors should not be used anymore.

To use a cursor, the user points the mouse on a specific item. Because a cursor can be larg
areas of a window can act upon receiving the mouse, the cursor has a special area called a
matter how big the cursor is, the mouse can apply its behavior only on what the hot spot to
cursor, which is the most widely used cursor, the hot spot is located on the tip of the arrow
left corner. When you create your own cursor, you can position the hot spot anywhere on t
but you must always know where the hot spot of your cursor is and you should always mal
user.

Cursor Creation

To use any of the above cursors, simply pass its name as the second argument to the**Load**
Here is an example:

LoadCursor(NULL, IDC_HELP);

Creating your own cursor or a few cursors for your application involves a few steps. These
different depending on the programming environment you are using; but the process is the

You should first create a cursor. You can copy one of the existing cursors that ship with yo
environment. Both Borland C++ Builder and Microsoft Visual C++ install many cursors.
your own cursor from scratch. For Borland C++ Builder, this can be done using the Image
separate application that ships with the compiler. For Microsoft Visual C++, you can do th
image editor. The cursor is its own file with the extension .cur.

After creating and possibly testing the cursor, you must create a header file, usually called
you define the cursor. In this file, each cursor must be defined with the formula:

#define *Identifier Constant*

The **#define** preprocessor is required. The identifier must be a non-quoted string that will
cursor. The *Constant* must an integer with a unique value in the file.

After creating the header file, you must create the main resource file. This file has an exte
usually has the same name as the project but this is only a habit. In the resource file, you s
header file you just created. In the file, each resource must be represented. For a cursor, yo

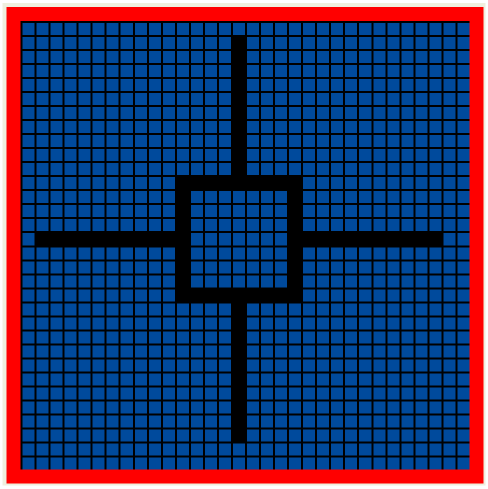
Identifier **CURSOR** *FileName*

The *Identifier* must be the same you defined in the Resource header file. The keyword CU
to let the compiler know that this particular resource is a cursor, making it distinct from ot
icons or bitmaps, etc. The *FileName* must be the cursor with extension .cur that you had c

To use the resource (.rc) file in your project, you must explicitly Add it to your project. Ne
Builder nor Microsoft Visual C++ will add this rc file for you. The main difference is that,
Builder, you can add this rc file before or after compiling it, as long as the compile is awa
the whole project. With Microsoft Visual C++, you should first (although you don't have t
your project. In fact, this makes it easy for Microsoft Visual C++ to compile the rc file for

Using Borland C++ Builder

1. To create the cursor file, on the main menu, click Tools -> Image Editor
2. When the Image Editor displays, on the main menu, click File -> New -> Cursor File (.cur). Press Ctrl + I three times to zoom in.
3. Design the cursor as follows:



4. To define the hot spot, on the main menu of Image Editor, click Cursor -> Set Hot Spot... Set the Horizontal (X) and the Vertical (Y) values to 15 each and click OK
5. To save the cursor, on the main menu of Image Editor, click File -> Save
6. Locate the folder where the current exercise is located and display in the Save In combo box

7. Replace the contents of the File Name edit box with **Target**
   The right extension (.cur) will be added to the file

8. Click Save and return to C++ Builder

9. To create the resource header file, on the main menu, click File -> New... or File -> New -> Other...

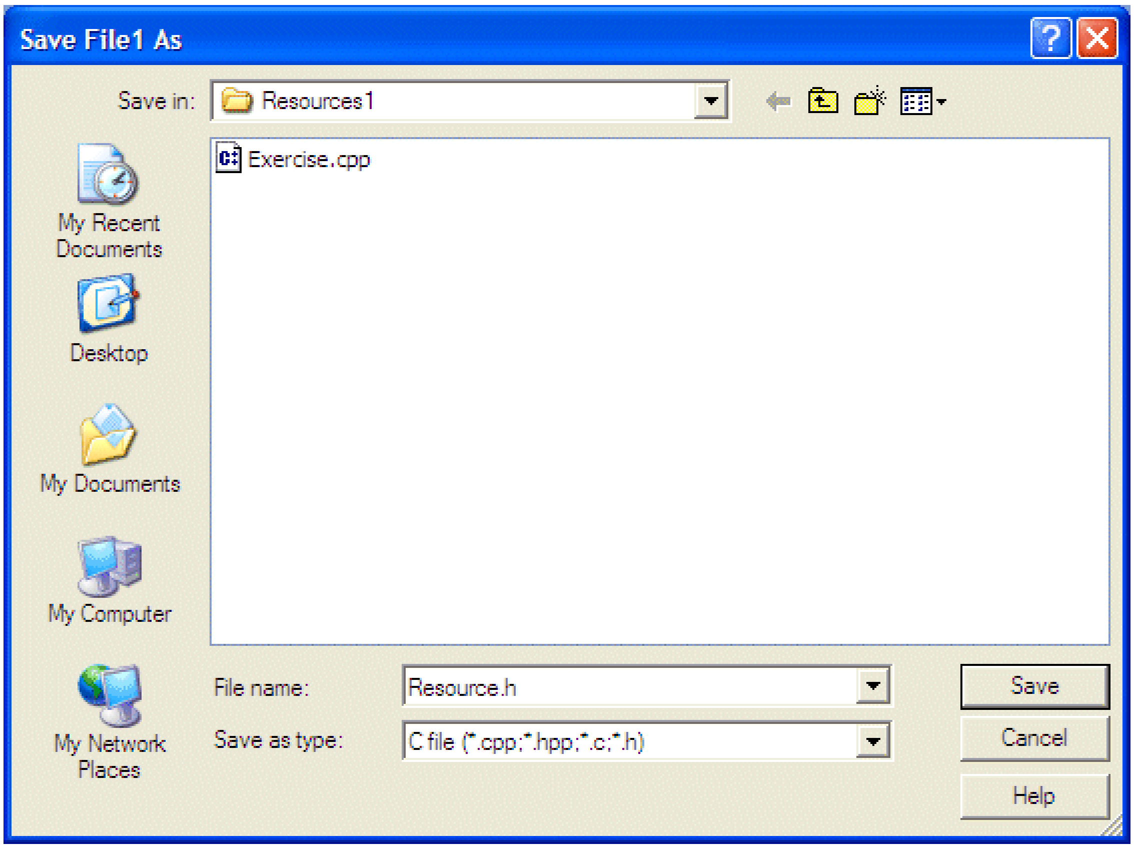
10. In the New Items dialog box, click Header File and click OK

11. In the header file, type:

```
#define IDC_TARGET  1000
```

12. To save the header file, on the Standard toolbar, click the Save All button

13. Type Resource.h and make sure you add the extension



File name: Resource.h

Save as type: C file (*.cpp;*.hpp;*.c;*.h)

14. Click Save

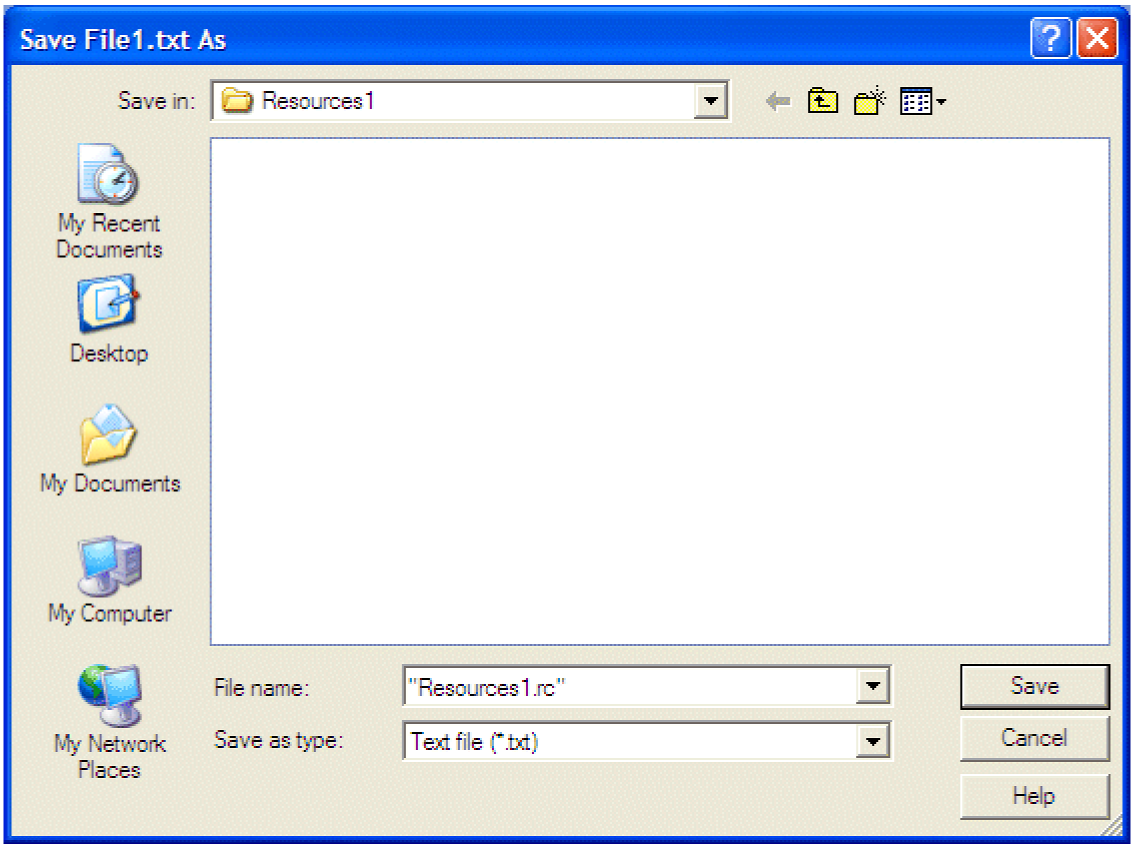
15. To create the actual resource file, on the main menu, click File -> New... or File -> New -> Other...

16. In the New Items dialog box, scroll down, click the Text icon, and click OK.

17. In the empty file, type:

```
#include "Resource.h"

IDC_TARGET  CURSOR  "Target.cur"
```

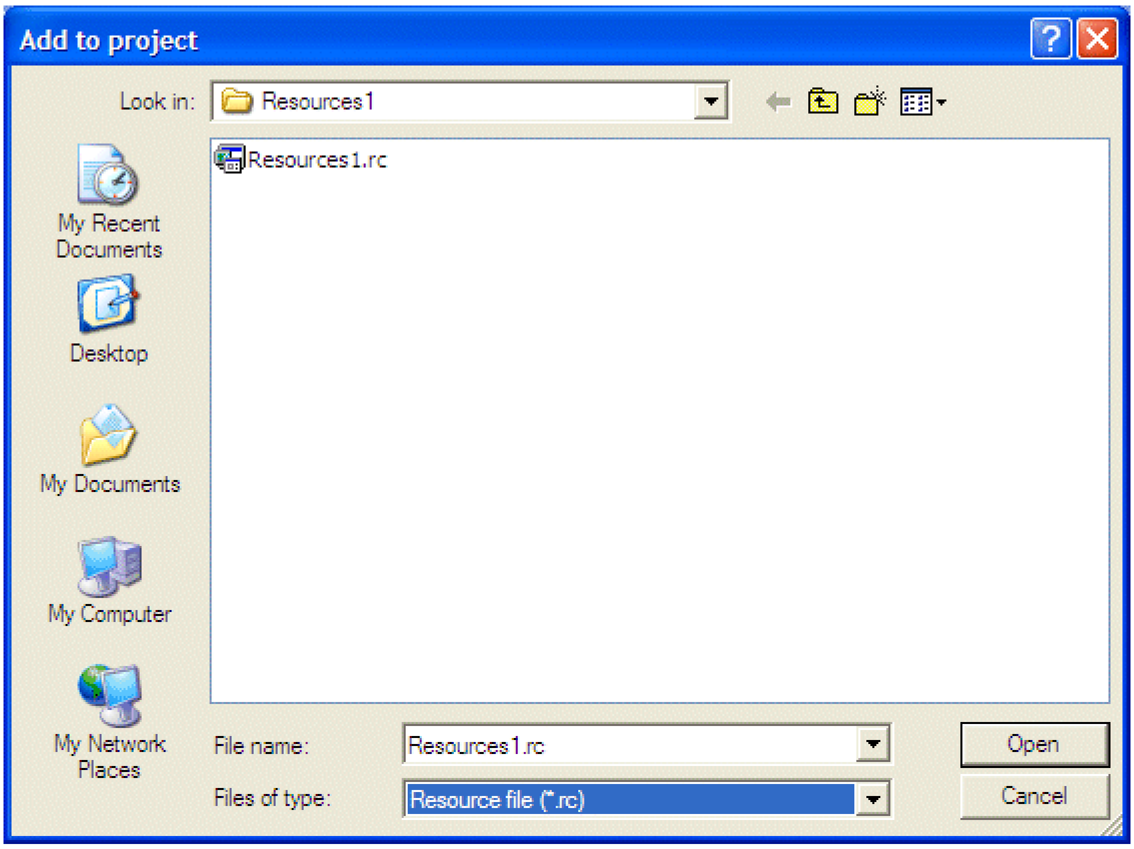
18. To save the resource file, on the Standard toolbar, click the Save All button.
19. Type "Resources1.rc"
    The reason for the double-quotes is to make sure that not only the file is not saved with a txt extension but also it is actually saved with the rc extension



Save in: Resources1
File name: "Resources1.rc"
Save as type: Text file (*.txt)

20. Click Save
21. To add the resource file to the current project, on the main menu, click Project -> Add to Project...

22.In the Files of Type combo box, select Resource File (*.rc)



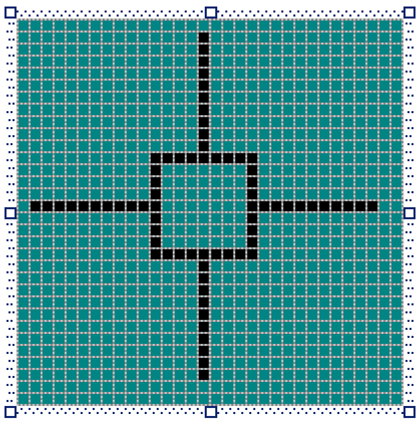
File name: Resources1.rc

Files of type: Resource file (*.rc)

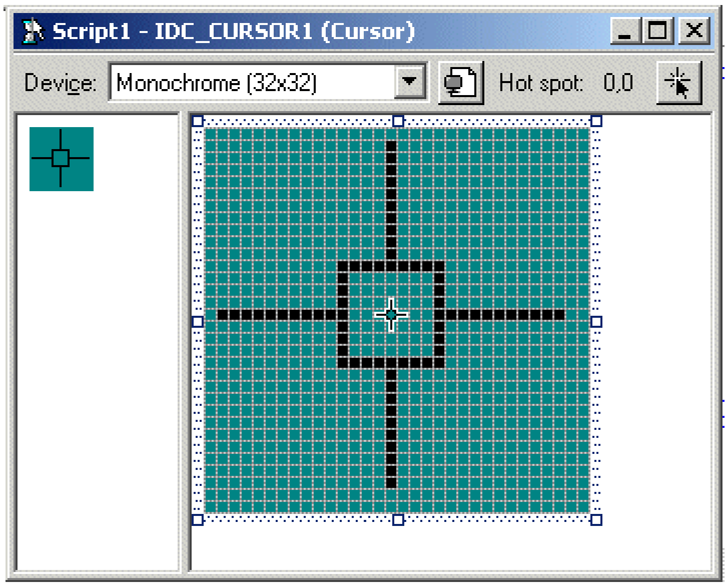
23.In the list box, click Resources.rc.rc and click Open

Using Microsoft Visual C++

1. To create the cursor, on the main menu, click Insert -> Resource... In the Insert Resource dialog box, click Cursor and click New
2. If you are using MSVC 6, on the main menu, click View -> Properties...
   In the Properties window, change the ID to **IDC_TARGET** and press Tab
   Make sure the file name is changed to **target.cur** and press Enter
3. Design the cursor as follows:



4. To define the hot spot, on the toolbar of the editor, click the Set Hot Spot button. Click the middle of the square:



5. To save and close the resource file, click its system Close button (the system Close button of the child window) of the cursor and close the child window of the script (the window with Script1 or Script2)

6. When asked to save the script, click Yes

7. Locate the folder where the current exercise is located and display in the Save In combo box. Change the name of the file to **Resources1.rc**
The right extension (.rc) will be added to the file

8.

9. Click Save

10. To add the resource file to the current project, on the main menu, click Project -> Add To Project -> Files...
11. In the list box, click Win32B.rc and click OK
12. In the Workspace, click the ResourceView tab.
    Expand the Win32B resource node by clicking its + button. Expand the Cursor node.
    Make sure the IDC_TARGET cursor is present

Custom Cursors

After creating the resource file, you must compile it. After the rc file has been compiled, it creates a file with extension .res. In Borland C++ Builder, you must explicitly compile the resource file, which is easy but you must remember to do it. Microsoft Visual C++ transparently compiles the rc file for you but you must have added it to your project.

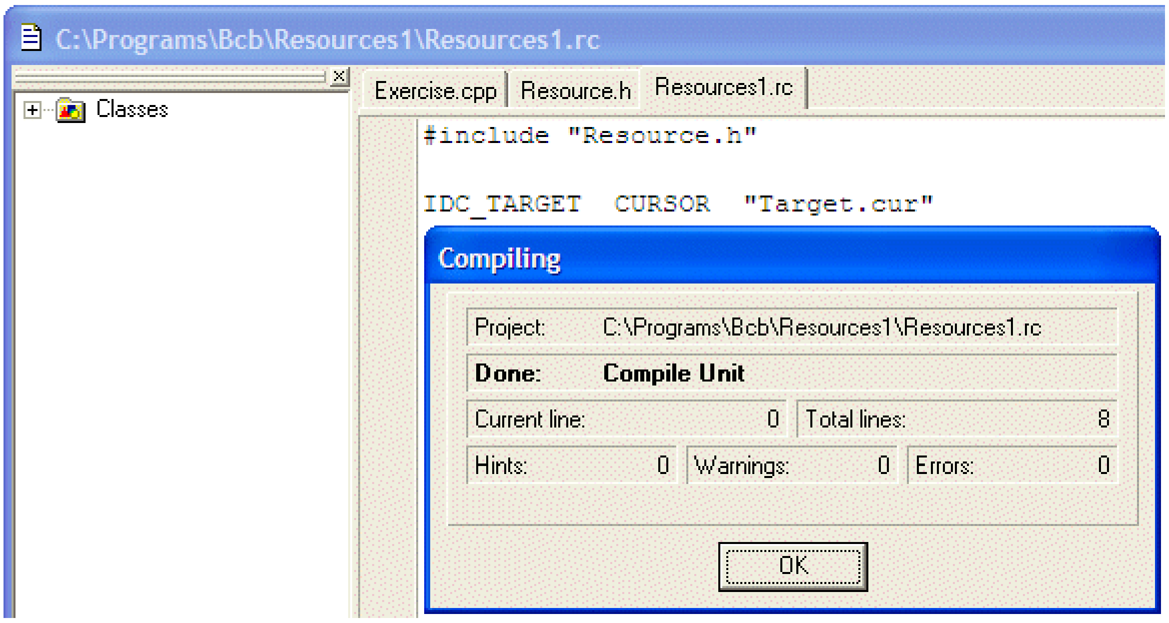
After creating the resource file, because the Resource header file holds the identifiers of the resource, remember to include it in the file where you want to use the resources. This is done with a simple:

```
#include "Resource.h"
```

To use your own cursor, assign the result of the **LoadCursor()** function to the *hCursor*member variable of the **WNDCLASS** or the **WNDCLASSEX** class. The first argument of the function should be the instance of the application you are using. For the second argument, use the **MAKEINTRESOURCE** macro, passing it the identifier of the cursor.

Practical Learning: Using a Cursor

1. If you are using Borland C++ Builder, click the Resources1.rc tab in the Code Editor
2. Then, on the main menu, click Project -> Compile Unit. After the resource file has been compiled, click OK



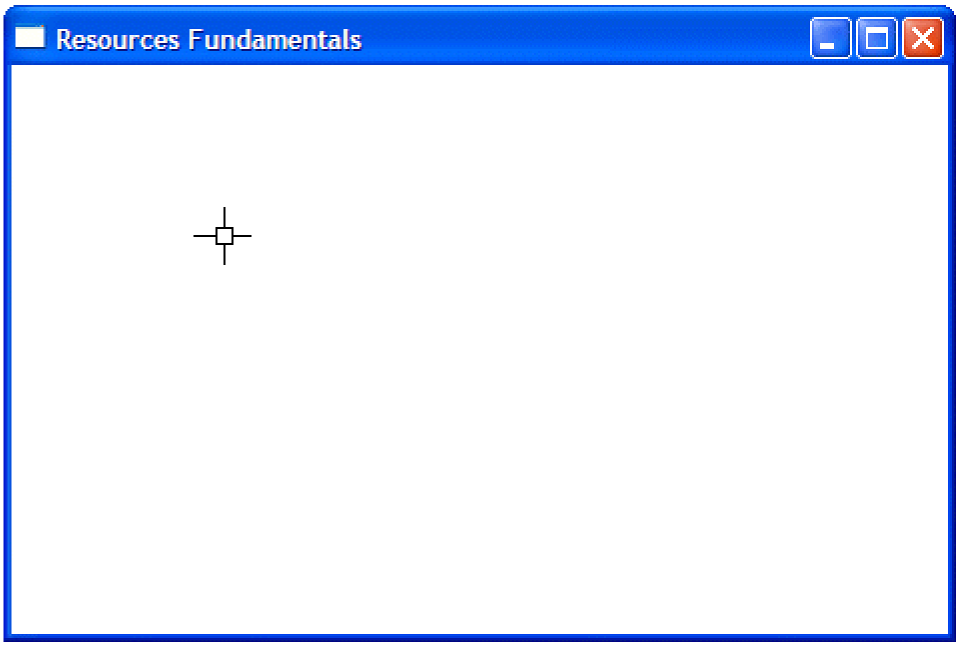
```
#include "Resource.h"

IDC_TARGET   CURSOR   "Target.cur"
```

```
Project:      C:\Programs\Bcb\Resources1\Resources1.rc
Done:         Compile Unit
Current line:          0   Total lines:         8
Hints:       0   Warnings:       0   Errors:       0
```

3. In both compilers, change the Main.cpp file as follows:

```
//---------------------------------------------------------------------------
#include <windows.h>
#include "resource.h"
//---------------------------------------------------------------------------
HWND hWnd;
LPCTSTR ClsName = L"SimpleWindow";
LPCTSTR WindowCaption = L"A Simple Window";
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg,
WPARAM wParam, LPARAM lParam);
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
    MSG        Msg;
    WNDCLASSEX  WndClsEx;
    WndClsEx.cbSize        = sizeof(WNDCLASSEX);
    WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc   = WndProc;
    WndClsEx.cbClsExtra    = NULL;
    WndClsEx.cbWndExtra    = NULL;
    WndClsEx.hInstance     = hInstance;
    WndClsEx.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    WndClsEx.hCursor       = LoadCursor(hInstance,
                              MAKEINTRESOURCE(IDC_TARGET));
    WndClsEx.hbrBackground =
(HBRUSH)GetStockObject(WHITE_BRUSH);


    . . . No Change


    return Msg.wParam;
}
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg,
WPARAM wParam, LPARAM lParam)
{
    . . . No Change
}
//---------------------------------------------------------------------------
```

4. To execute your program, in Borland C++ Builder, on the main menu, click Project -> Build Resources1. When it is ready, press F9
   In Microsoft Visual C++, press Ctrl + F5 and Enter



5. Close the window and return to your programming environment

List-Based Resources

Menus

Introduction

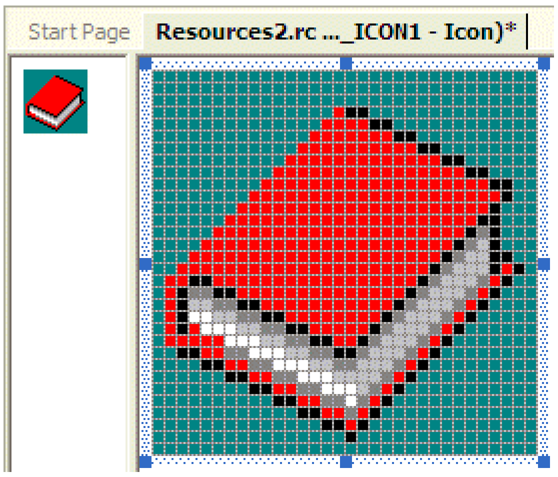
A menu is a list of commands that allow the user to interact with an application. To use one of the commands, the user accesses the list and clicks the desired item. There are two main types of menus. On most applications, a menu is represented in the top section with a series of words such as File, Edit, Help. Each of these words represents a category of items. To use this type of menu, the use can display one of the categories (using the mouse or the keyboard). A list would display and the user can select one of the items. The second type of menu is called context sensitive. To use this type of menu, the user typically right-clicks a certain area of the application, a list comes up and the user can select one of the items from the list.

Practical Learning: Introducing List-Based Resources

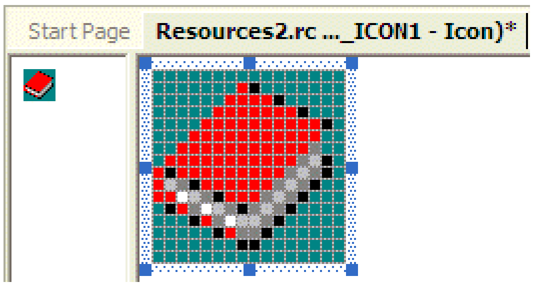
1. Create a new Win32 Project named **Resources2** and create it as an empty project

2. On the main menu, click Project -> Add Resource...
3. Double-click Icon and design it as follows (make sure you add the 16x16 version)

32 x 32



16 x 16



4. Change the ID of the icon to **IDI_RESFUND2** and its File Name to resfund2.ico
5. Create a source file and name it **Exercise**
6. From what we have learned so far, type the following code in the file:

```
//---------------------------------------------------------------------------
#include <windows.h>
#include "resource.h"


//---------------------------------------------------------------------------
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                              WPARAM wParam, LPARAM lParam);
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)
{
        MSG         Msg;
        HWND        hWnd;
```

```
  WNDCLASSEX WndClsEx;
LPCTSTR ClsName = L"ResFund";
  LPCTSTR WndName = L"Resources Fundamentals";

  // Create the application window
  WndClsEx.cbSize        = sizeof(WNDCLASSEX);
  WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
  WndClsEx.lpfnWndProc   = WndProcedure;
  WndClsEx.cbClsExtra    = 0;
  WndClsEx.cbWndExtra    = 0;
  WndClsEx.hIcon         = LoadIcon(hInstance,
                                    MAKEINTRESOURCE(IDI_RESFUND2));
  WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
  WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
  WndClsEx.lpszMenuName  = NULL;
  WndClsEx.lpszClassName = ClsName;
  WndClsEx.hInstance     = hInstance;
  WndClsEx.hIconSm       = LoadIcon(hInstance,
                                    MAKEINTRESOURCE(IDI_RESFUND2));

  // Register the application
  RegisterClassEx(&WndClsEx);

  // Create the window object
  hWnd = CreateWindowEx(0,
              ClsName,
              WndName,
                        WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,
                        CW_USEDEFAULT,
                CW_USEDEFAULT,
                        CW_USEDEFAULT,
                NULL,
                NULL,
                hInstance,
                NULL);

        // Find out if the window was created
        if( !hWnd ) // If the window was not created,
                return FALSE; // stop the application

        // Display the window to the user
        ShowWindow(hWnd, nCmdShow);// SW_SHOWNORMAL);
        UpdateWindow(hWnd);

        // Decode and treat the messages
        // as long as the application is running
        while( GetMessage(&Msg, NULL, 0, 0) )
        {
          TranslateMessage(&Msg);
          DispatchMessage(&Msg);
        }

//      return Msg.wParam;
```

```
    return 0;
}
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        // Process the left-over messages
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    // If something was not done, let it go
    return 0;
}
//---------------------------------------------------------------------------
```

7. Execute the application to test it

Menu Creation

A menu is one of the text-based resources. It is created directly in the rc file. As with other resources, the process of creating a menu depends on the environment you are using. If you are using Borland C++ Builder, you can open your rc file and manually create your menu.

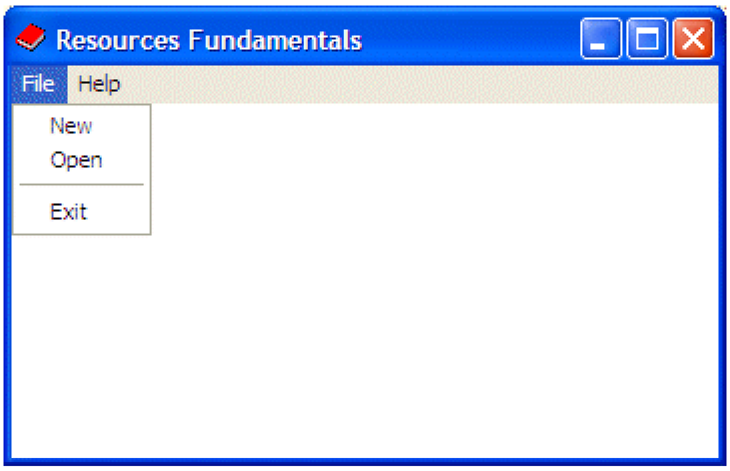
If you are using Microsoft Visual C++, you can use the built-in menu editor. In this case, the actual text that defines and describes the menu would be automatically added to the rc file.

Practical Learning: Creating a Menu

1. On the main menu, click Project -> Add Resource...
2. In the Insert Resource dialog box, click Menu and click New
3. While the first menu item is selected, type **&File**
4. Click the next empty item under File. Type **&New**
5. In the Properties window, click the ID edit box, type **IDM_FILE_NEW** and press Enter
6. Click the next empty item under New and type **&Open**
7. In the Properties window, click the ID edit box, type **IDM_FILE_OPEN** and press Tab
8. Click the next empty item under Open and type -
9. Click the next empty item under the new separator and type **E&xit**
10. In the Properties window, click the ID edit box, type **IDM_FILE_EXIT** and press Tab. In the Caption edit box, press Enter
11. Click the next empty item on the right side of File. Type **&Help**
12. Click the next empty item under Help and type **&About**
13. In the Properties window, click the ID edit box, type **IDM_HELP_ABOUT** and press Tab. In the Caption edit box, and press Enter
14. In the ResourceView tab of the Workspace, under the Menu node, click IDR_MENU1. In the Menu Properties window, change the ID to **IDR_MAINFRAME**
15. Open the Exercise.cpp source file and change the lpszMenuName member of the WndClsEx variable as follows:

```
WndClsEx.lpszMenuName  =
MAKEINTRESOURCE(IDR_MAINFRAME);
```

16.To test the application, press Ctrl + F5 and press Enter



17.Return to your programming environment