CHAPTER

1 Getting Started with Windows

Chapter 1 introduces the Microsoft Windows operating system (OS) family and the Windows Application Programming Interface (API) that all family members support. It also briefly describes the 32-bit (Win32) and 64-bit (Win64) API differences and portability issues, and, going forward, we mention Win32 and Win64 only when there is an important distinction.[1] The context will help to distinguish between Windows as an OS and Windows as the API for application development.

The Windows API, like any other OS API, has its own set of conventions and programming techniques, which are driven by the Windows philosophy. A simple file copy example introduces the Windows programming style, and this same style applies to file management, process and memory management, and advanced features such as thread synchronization. In order to contrast Windows with more familiar programming styles, there is a Standard C library version of the first example.

The first step is to review the basic features that any modern OS must provide and, from there, to learn how to use these features in Windows.

Operating System Essentials

Windows makes core OS features available on systems as diverse as cell phones, handheld devices, laptop PCs, and enterprise servers. Considering the most important resources that a modern OS manages helps to explain the Windows API.

- **Memory**. The OS manages a large, flat, virtual memory address space and transparently moves information between physical memory and disk and other secondary storage.

[1] Be aware that Microsoft often uses the term“Win32” generically for unmanaged code; all our code is unmanaged and does not use .NET’s Common Language Runtime (CLR).

- **File systems**. The OS manages a hierarchical, named file space and provides both direct and sequential access as well as directory and file management.
- **Processors**. The OS must efficiently allocate computational tasks to processors, and multiple processors are increasingly common on even the smallest computers.
- **Resource naming and location**. File naming allows for long, descriptive names, and the naming scheme is extended to objects such as devices, synchronization, and interprocess communication objects. The OS also locates and manages access to named objects.
- **Multitasking**. The OS must manage processes, threads, and other units of independent, asynchronous execution. Tasks can be preempted and scheduled according to dynamically calculated priorities.
- **Communication and synchronization**. The OS manages task-to-task communication and synchronization within single computers as well as communication between networked computers and with the Internet.
- **Security and protection**. The OS provides flexible mechanisms to protect resources from unauthorized and accidental access and corruption.

The Microsoft Windows API supports all these OS features and more and makes them available on a range of Windows versions.

Windows Evolution

Several Windows versions support the Windows API. The multiple distinct Windows versions can be confusing, but from the programmer's perspective, they are similar. In particular, they all support subsets of the *identical* Windows API. Programs developed for one system can, with considerable ease, run on another, resulting in source and, in most cases, binary portability.

New Windows versions have added small amounts of new API functionality, although the API has been remarkably stable since the beginning. Major themes in Windows evolution include the following.

- **Scalability**. Newer versions run on a wider range of computers, up to enterprise servers with multiple processors and large memories and storage systems.
- **Performance**. Newer Windows versions contain internal improvements and some new API features that improve performance.

- **Integration**. Each new release integrates additional technology, such as multimedia, wireless networking, Web Services, .NET, and plug-and-play capability. This technology is, in general, out of scope for this book.
- **Ease of use**. Improved graphical desktop appearance and ease of use are readily apparent with each release.
- **Enhanced API**. Important API enhancements have been added over time. The API is the central topic of this book.

Windows Versions

Windows, in an evolving series of versions, has been in use since 1993. The following versions are important to developers at publication time.

- **Windows 7** was released in October 2009, shortly before this book's publication.
- **Windows Vista** is targeted at the individual user. Most commercial PCs sold since 2007, including desktops, laptops, and notebooks, came with an appropriate version of Windows Vista preinstalled.
- **Windows XP** is Vista's predecessor and is still very popular.
- **Windows Server 2008** is targeted at enterprise and server applications, and it was preceded by **Windows Server 2003**. Computers running Windows Server 2008 frequently exploit multicore technology with multiple independent processors. 64-bit applications are common on Windows Server 2008 computers.
- **Windows 2000** is still in use, although Microsoft will retire support in mid-2010.
- **Windows CE** is a specialized Window version targeted at smaller computers, such as phones, palmtops, and embedded processors, and it provides large subsets of Windows features.

Obsolete Previous Windows Versions

Earlier Windows versions are rare and generally not supported, but they are summarized here to give some historical perspective. While there are numerous exceptions, especially in the later chapters, many examples in this book will operate on these systems, although there are no guarantees.

- **Windows NT** 3.1, 3.5, 3.51, and 4.0 date back to 1993. NT was originally targeted at servers and professional users, with Windows 9x (see the next bullet) sold for personal and office use. Windows 2000 was the successor. The NT kernel is the foundation for the current Windows kernel, even though the term "Windows NT" is obsolete.
- **Windows 95**, **Windows 98**, and **Windows Me** (collectively, **Windows 9x**) were primarily desktop and laptop OSs lacking, among other things, the NT security features. Windows XP replaced these Windows versions.

Further back, Windows 3.1, a 16-bit OS, was dominant on personal computers before the Windows 95 introduction, and its graphical user interface (GUI) was a predecessor to the modern Windows GUI. The API, however, did not support many essential OS features, such as true multitasking; memory management of a large, flat address space; and security.

Going further back to the early 1980s, it is possible to identify DOS as the original "IBM PC" OS. DOS had only a simple command line interface, but the Windows command shell still supports DOS commands. In fact, most of the book's examples are command line programs, so you can run them under the command shell; that is, the Windows `cmd` program.

Windows NT5 and NT6

Windows 2000, XP, and Server 2003 use Windows NT kernel Version 5, although the minor version (the "x" in 5.x) varies. For example, Windows XP uses kernel Version NT 5.1.2600 ("2600" is the build number). Since the API features depend on the kernel version, it is convenient to use the term "NT5" to refer to these three Windows versions, even though Microsoft no longer uses the term "Windows NT."

The NT6 kernel is the base for Windows 7 (6.1), Vista (6.0), and Server 2008 (6.1 for R2; 6.0 otherwise), and the term "NT6" denotes these three Windows versions.

While many programs will run on earlier versions, in general, we will assume NT5 and NT6, which will allow us to exploit some advanced features. Since some important features are available only in NT6, sample programs test the Windows version number and terminate with an error message if they cannot run on the host computer.

The Microsoft Developer's Network (MSDN) API documentation (www.msdn.microsoft.com) states the version requirements. Check the documentation if there is any doubt about an API function's operation on a particular Windows version. The documentation will name the specific Windows version requirements, such as Windows Vista or Windows Server 2008, whereas we'll frequently state the same requirement as NT6.

Processor Support

Windows can support different underlying processor and computer architectures and has a Hardware Abstraction Layer (HAL) to enable porting to different processor architectures, although this is not a direct concern for the application developer.

Windows runs primarily on the Intel x86 processor family, including the x86-64 (or just x64) 64-bit extension, and compatible Advanced Micro Devices (AMD) processors. Although less common, several Windows server versions run on the Intel Itanium IA-64, a 64-bit architecture radically different from the classic x86 architecture.

The Windows Market Role

Windows is hardly unique in its ability to provide essential functionality on several platforms. After all, numerous proprietary and open OSs have these features, and UNIX[2] and Linux have long been available on a wide range of computers. There are, however, significant advantages, both business and technical, to using Windows and to developing Windows applications.

- Windows dominates the market, especially on the desktop, and has done so for many years with no change in sight.[3] Therefore, Windows applications have a large target market, numbering in the tens of millions and dwarfing other desktop systems, including UNIX, Linux, and Macintosh.
- The market dominance of the Windows OSs means that applications and software development and integration tools are widely and inexpensively available for Windows.
- Windows supports multiprocessor computers. Windows is not confined to the desktop; it can support departmental and enterprise servers and high-performance workstations.[4]

[2] UNIX comments always apply to Linux as well as to any other system that supports the POSIX API.

[3] Linux is occasionally mentioned as a threat to Windows dominance, primarily as a server but also for personal applications. While extremely interesting, speculation regarding future developments, much less the comparative merits of Linux and Windows, is out of scope for this book.

[4] The range of Windows host computers can be appreciated by considering that many programs in this book have been tested on computers spanning from an obsolete 486 computer with 16MB of RAM to a 16-processor, 16GB RAM, 2.4GHz enterprise server.

- Windows applications can use a GUI familiar to tens of millions of users, and many Windows applications are customized or "localized" for the language and user interface requirements of users throughout the world.
- Most OSs, other than UNIX, Linux, and Windows, are proprietary to systems from a single vendor.
- The Windows OSs have many features not available in standard UNIX, although they may be available in some UNIX implementations. Thread pools and Windows Services are two examples.

In summary, Windows provides modern OS functionality and can run applications ranging from word processors and e-mail to enterprise integration systems and large database servers. Furthermore, Windows platforms scale from small devices to the desktop and the enterprise. Decisions to develop Windows applications are driven by both technical features and business requirements.

Windows, Standards, and Open Systems

This book is about developing applications using the Windows API. For a programmer coming from UNIX and open systems, it is natural to ask, "Is Windows open?" "Is Windows an industry standard?" "Is Windows just another proprietary API?" The answers depend very much on the definitions of *open*, *industry standard*, and *proprietary*, as well as on the benefits expected from open systems.

The Windows API is totally different from the POSIX standard API supported by Linux and UNIX. Windows does not conform to the X/Open standard or any other open industry standards formulated by standards bodies or industry consortia.

Windows is controlled by one vendor. Although Microsoft solicits industry input and feedback, it remains the sole arbiter and implementor. This means that the user receives many of the benefits that open standards are intended to provide as well as other advantages.

- Uniform implementations reach the market quickly.
- There are no vendor-specific, nonstandard extensions, although the small differences among the various Windows platforms can be important.
- One vendor has defined and implemented competent OS products with all the required operating system features. Applications developers add value at a higher level.
- The underlying hardware platform is open. Developers can select from numerous platform vendors.

Arguments will continue to rage about whether this situation is beneficial or harmful to users and the computer industry as a whole. This book neither enters nor settles the argument; it is merely intended to help application developers use Windows to solve their problems.

Nonetheless, Windows does support many essential standards. For example, Windows supports the Standard C and C++ libraries and a wide array of open interoperability standards. Thus, Windows Sockets provide a standard networked programming interface for access to TCP/IP and other networking protocols, allowing Internet access and interoperability with non-Windows computers. The same is true with Remote Procedure Calls (RPCs).[5] Diverse computers can communicate with high-level database management system (DBMS) protocols using Structured Query Language (SQL). Finally, Internet support with Web and other servers is part of the total Windows offering. Windows supports the key standards, such as TCP/IP, and many valuable options, including X Windows clients and servers, are available at reasonable cost, or even as open source, in an active market of Windows solution suppliers.

In summary, Windows supports the essential interoperability standards, and while the core API is proprietary, it is available cost-effectively on a wide variety of computers.

Windows Principles

It is helpful to keep in mind some basic Windows principles. The Windows API is different in many ways, both large and small, from other APIs such as the POSIX API. Although Windows is not inherently difficult, it requires its own coding style and technique.

Here are some of the major Windows characteristics, which will become much more familiar as you read through the book.

- Many system resources are represented as a *kernel object* identified and referenced by a *handle*. These handles are somewhat comparable to UNIX file descriptors and process IDs.[6] Several important objects are not kernel objects and will be identified differently.

[5] Windows Sockets and RPCs are not properly part of Windows, but sockets are described in this book because they relate directly to the general subject matter and approach.

[6] These handles are similar to but not the same as the `HWND` and `HDC` handles used in Windows GUI programming. Also, Windows does have a process ID, but it is not used the way a UNIX process ID is used.

- Kernel objects must be manipulated by Windows APIs. There are no "back doors." This arrangement is consistent with the data abstraction principles of object-oriented programming, although Windows is not object oriented.
- Objects include files, processes, threads, pipes for interprocess communication, memory mapping, events, and many more. Objects have security attributes.
- Windows is a rich and flexible interface. First, it contains many functions that perform the same or similar operations; in particular, convenience functions combine common sequences of function calls into one function (`CopyFile` is one such convenience function and is the basis of an example later in this chapter). Second, a given function often has numerous parameters and flags, but you can normally ignore most of them. This book concentrates on the most important functions and options rather than being encyclopedic.
- Windows offers numerous synchronization and communication mechanisms tailored for different requirements.
- The Windows thread is the basic unit of execution. A process can contain one or more threads.
- Windows function names are long and descriptive. The following function names illustrate function name conventions as well as Windows' variety:

  `WaitForSingleObject`

  `WaitForSingleObjectEx`

  `WaitForMultipleObjects`

  `WaitNamedPipe`

  In addition to these features, there are a few conventions for type names.
- The names for predefined data types, required by the API, are in uppercase and are also descriptive. The following typical types occur frequently:

  `BOOL` (defined as a 32-bit object for storing a single logical value)

  `HANDLE` (a handle for a kernel object)

  `DWORD` (the ubiquitous 32-bit unsigned integer)

  `LPTSTR` (a string pointer)

  `LPSECURITY_ATTRIBUTES`

  We'll introduce these and many other data types as required.

- The predefined types avoid the `*` operator and make distinctions such as differentiating `LPTSTR` (defined as `TCHAR *`) from `LPCTSTR` (defined as `const TCHAR *`). *Note:* `TCHAR` may be a normal `char` or a 2-byte `wchar_t`.
- Variable names, at least in function prototypes, also have conventions. For example, `lpszFileName` might be a "long pointer to a zero-terminated string" representing a file name. This is the so-called Hungarian notation, which this book does not generally use for program variables. Similarly, `dwAccess` is a double word (32 bits) containing file access flags; "`dw`" denotes a double word.

Note: It is informative to look at the system include files where the functions, constants, flags, error codes, and so on are defined. Many interesting files, such as the following, are part of the Microsoft Visual Studio C++ environment and are normally installed in an include directory along with Visual Studio:

`windows.h` (this file brings in all the others)

`winnt.h`

`winbase.h`

Finally, even though the original Windows API (Win32) was created from scratch, it was designed to be backward-compatible with the Windows 3.1 Win16 API. This has several lingering and annoying effects, even though backward compatibility ceased to be an issue long ago.

- There are anachronisms in types, such as `LPTSTR` and `LPDWORD`, which refer to the "long pointer" that is simply a 32-bit or 64-bit pointer. There is no need for any other pointer type. At other times, the "long" is omitted, and `LPVOID` and `PVOID` are equivalent.[7]
- "`WIN32`" sometimes appears in macro names, such as `WIN32_FIND_DATA`, even though the macro is also used with Win64.
- The former requirement, no longer relevant, for backward compatibility means that numerous 16-bit functions are never used in this book, even though they might seem important. `OpenFile` is such a function; always use `CreateFile` to open an existing file.

[7] The include files contain types, such as `PVOID`, without the prefix, but the examples conform to the usage in many other books and the Microsoft documentation.

UNIX and Linux programmers will find some interesting differences in Windows. For example, Windows `HANDLE`s are "opaque." They are not integers allocated in sequential order. Thus, the fact that `0`, `1`, and `2` are special file descriptor values, which is important to some UNIX programs, has no analogy in Windows.

Many of the distinctions between, say, UNIX process IDs and file descriptors go away. Windows uses `HANDLE`s to reference both processes and open files, as well as other kernel objects. While Windows does have a process ID, it is used differently than a UNIX process ID. Many important functions treat file, process, event, pipe, and other handles identically.

UNIX programmers familiar with short, lowercase function and parameter names will need to adjust to the more verbose Windows style.

Critical distinctions are made with such familiar concepts as processes. Windows processes do not, for example, have parent-child relationships, although Windows processes can be organized into job objects.

Finally, Windows text files represent the end-of-line sequence with `CR—LF` rather than with `LF` as in UNIX.

32-bit and 64-bit Source Code Portability

Example source code can be built as both 32-bit and 64-bit executable versions (32-bit executables run on 64-bit computers but cannot exploit the larger address spaces). The essential differences between versions are the pointer variable size and the virtual address space size.

Most of the differences, from a programming point of view, concern the size of pointers and careful avoidance of any assumption that a pointer and an integer (`LONG`, `DWORD`, and so on) are of the same length.

Chapter 5 shows additional differences where it is important to use Windows functions that support 64-bit addresses.

With a little care, you will find that it is fairly simple to ensure that your programs will run under either Win32 or Win64. The program examples, both in the book and on the Web site (see the "What You Need to Use This Book" section below), are portable and have been tested on 64-bit computers. There are separate projects for building the 32-bit and 64-bit versions from the same source code.

The Standard C Library: When to Use It for File Processing

Despite the unique Windows features, it is still possible to achieve most file processing (the subject of Chapters 2 and 3) by using the familiar C programming language and its ANSI Standard C library, which are layered on the Windows API.

The C library (the adjectives ANSI and Standard are often omitted) also contains numerous indispensable functions that do not correspond to Windows system calls, such as functions defined in `string.h`, `stdlib.h`, `signal.h`, formatted I/O functions, and character I/O functions. Other functions, however, correspond closely to system calls, such as the `fopen` and `fread` functions in `stdio.h`.

When is the C library adequate, and when is it necessary to use native Windows file management system calls? This same question could be asked about using C++ I/O streams or the system I/O provided within .NET. There is no easy answer, but portability to non-Windows platforms is a consideration in favor of non-Windows functions if an application needs only file processing and not, for example, process management. However, many programmers have formulated guidelines as to when the C library is or is not adequate, and these same guidelines should apply to Windows. In addition, given the increased power, performance potential, and flexibility provided by Windows, it is often convenient or even necessary to go beyond the C library, as we will see starting as early as Chapter 2. Windows file processing features not available with the C library include file locking, memory mapping (required for memory sharing and performance), asynchronous I/O, random access to very long files (more than 4GB in length), and interprocess communication.

The C library file management functions are often adequate for simple programs. With the C library, it is possible to write portable applications without learning Windows, but options are limited. For example, Chapter 5 exploits memory-mapped files for performance and programming convenience, and this functionality is not included in the C library.

What You Need to Use This Book

Here is what you need to build and run the examples in this chapter and the rest of the book.

First, of course, it is helpful to bring your knowledge of applications development; knowledge of C programming is assumed.

Why Use C? Why Not C++?

The examples all use the C language, and, as necessary, use Microsoft extensions. The API is defined in C syntax, and C++ programmers will have no difficulty using the API or extending the C examples. Furthermore, for a variety of reasons, large amounts of legacy and some new code is written in C. Using C also makes the examples accessible to novice as well as intermediate and advanced programmers, all of whom will find portions of the book to be useful.

At times, this choice results in code that is more awkward than one might wish, and the code may strike some readers as a bit backward. For example, variables declarations occur at the start of program blocks rather than at the point of first use, and comments use the `/* . . . */` syntax.

Using the Examples

Before you use the examples, however, you will need some basic hardware and software.

- A computer running Windows.[8]
- A C/C++ compiler and development system, such as Microsoft Visual Studio 2005 or 2008.[9] Other vendors also supply development systems, and although none have been tested with the examples, several readers have mentioned using other development systems successfully with only minor adjustments. *Note:* We concentrate on developing Windows console applications and will not truly exploit Microsoft Visual Studio's full powers.
- Enough RAM and disk space for program development. Nearly any commercially available computer will have more than enough memory, disk space, and processing power to run all the example programs and the development system, but check the requirements for the development system.[10]
- The on-line Microsoft Developer's Network (MSDN) documentation, such as that provided with Microsoft Visual Studio. It may be helpful to install this documentation on your disk because you will access it frequently, but you can easily access the information on the MSDN Web site.
- Download the *"Examples"* file, `WSP4_Examples.zip`, from the book's Web site (www.jmhartsoftware.com). Unzip the file and read `ReadMe.txt`. *Examples* (the name used from now on) contains source code, Visual Studio projects, executables, and everything else you need to build and run the examples in this book.

[8] I've tested Windows 7, Windows Vista, Windows XP, Windows Server 2003, and Windows Server 2008.

[9] At the time of writing, Visual Studio 2010 is in beta test. I've tested several examples with VS 2010 and experienced no conversion difficulties.

[10] The rapid pace of improvements in cost and performance is illustrated by recalling that in 1997 the first edition of this book specified, without embarrassment or apology, 16MB of RAM and 256MB of disk space. This fourth edition is being written on a laptop costing less than $800, with more than 100 times the RAM (the RAM space exceeds the previously required disk space), 300 times the disk space, and a processor running 50 times as fast as the one used when starting the first edition on a $2,500 PC.

Example: A Simple Sequential File Copy

The following sections show short example programs implementing a simple sequential file copy program in three different ways:

1. Using the Standard C library
2. Using Windows
3. Using a single Windows convenience function, `CopyFile`

In addition to showing contrasting programming models, these examples show the capabilities and limitations of the C library and Windows. Alternative implementations will enhance the program to improve performance and increase flexibility.

Sequential file processing is the simplest, most common, and most essential capability of any file system, and nearly any large program processes at least some files sequentially. Therefore, a simple file processing program is a good way to introduce Windows and its conventions.

File copying, often with updating, and the merging of sorted files are common forms of sequential processing. Compilers and text processing tools are examples of other applications that access files sequentially.

Although sequential file processing is conceptually simple, efficient processing that attains optimal speed can be much more difficult to achieve. It can require overlapped I/O, memory mapping, threads, or other techniques.

Simple file copying is not very interesting by itself, but comparing programs gives us a quick way to contrast different systems and to introduce Windows. The following examples implement a limited version of the UNIX `cp` command, copying one file to another, where the file names are specified on the command line. Error checking is minimal, and existing files are simply overwritten. Subsequent Windows implementations of this and other programs will address these and other shortcomings.

File Copying with the Standard C Library

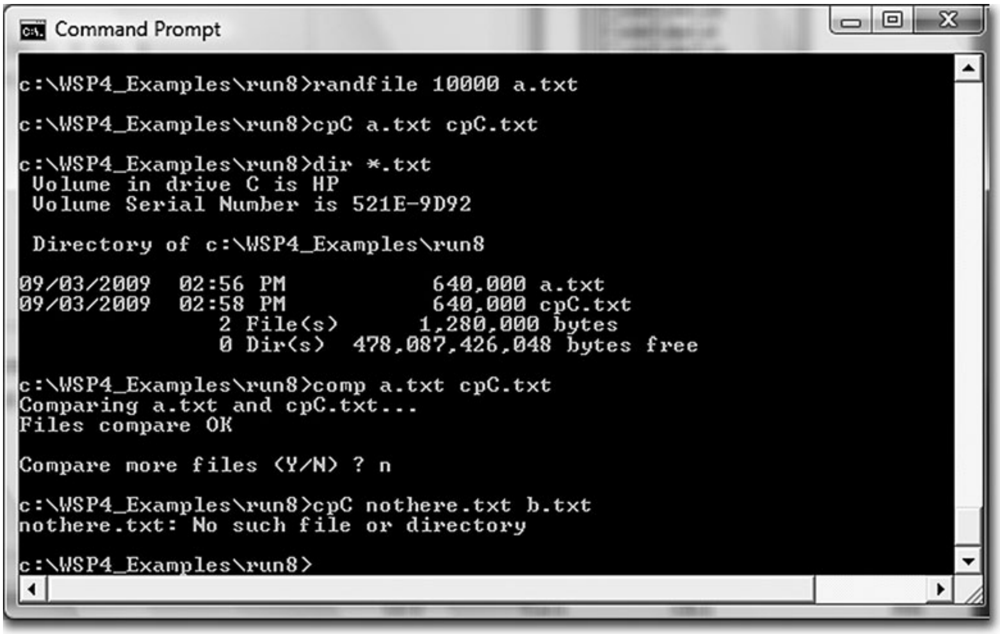
As illustrated in `cpC` (Program 1–1), the Standard C library supports stream `FILE` I/O objects that are similar to, although not as general as, the Windows `HANDLE` objects shown in `cpW` (Program 1–2). This program *does not* use the Windows API directly, but Microsoft's C Library implementation does use the API directly.

Program 1–1 `cpC`: File Copying with the C Library

```
/* Chapter 1. cpC. Basic file copy program.
   C library Implementation. */
```

```
/* cpC file1 file2: Copy file1 to file2. */

#include <stdio.h>
#include <errno.h>
#define BUF_SIZE 256

int main(int argc, char *argv[])
{
   FILE *inFile, *outFile;
   char rec[BUF_SIZE];
   size_t bytesIn, bytesOut;

   if (argc != 3) {
      printf("Usage: cp file1 file2\n");
      return 1;
   }

   inFile = fopen(argv[1], "rb");
   if (inFile == NULL) {
      perror(argv[1]);
      return 2;
   }

   outFile = fopen(argv[2], "wb");
   if (outFile == NULL) {
      perror(argv[2]);
      return 3;
   }

   /* Process the input file a record at a time. */
   while ((bytesIn = fread(rec, 1, BUF_SIZE, inFile)) > 0) {
      bytesOut = fwrite(rec, 1, bytesIn, outFile);
      if (bytesOut != bytesIn) {
         perror("Fatal write error.");
         return 4;
      }
   }

   fclose(inFile);
   fclose(outFile);
   return 0;
}
```

Run 1–1 is a screenshot of `cpC` execution with a short test.



```
c:\WSP4_Examples\run8>randfile 10000 a.txt

c:\WSP4_Examples\run8>cpC a.txt cpC.txt

c:\WSP4_Examples\run8>dir *.txt
 Volume in drive C is HP
 Volume Serial Number is 521E-9D92

 Directory of c:\WSP4_Examples\run8

09/03/2009  02:56 PM           640,000 a.txt
09/03/2009  02:58 PM           640,000 cpC.txt
               2 File(s)      1,280,000 bytes
               0 Dir(s)  478,087,426,048 bytes free

c:\WSP4_Examples\run8>comp a.txt cpC.txt
Comparing a.txt and cpC.txt...
Files compare OK

Compare more files (Y/N) ? n

c:\WSP4_Examples\run8>cpC nothere.txt b.txt
nothere.txt: No such file or directory

c:\WSP4_Examples\run8>
```

Run 1–1 `cpC`: Execution and Test

- The working directory is set to the directory `run8` in the *Examples* directory (see the "Using the Examples" section above). This directory contains the 32-bit programs built with Visual Studio 2008, and we use this directory for nearly all the example program screen shots.
- We need a text file for the test, and the `randfile` program generates a text file with 64-byte records with some random content. In this case, the output file is `a.txt` with 10,000 records. We use `randfile` frequently, and it's available in the *Examples* if you're curious about its operation.
- The second line in the screenshot shows `cpC` execution.
- The next commands show all the text files and compares them to be sure that the copy was correct. Note that the time stamps are different on the two files.
- The final line shows the error message if you try to copy a file that does not exist.

This simple example clearly illustrates some common programming assumptions and conventions that do not always apply with Windows.

1. Open file objects are identified by pointers to `FILE` structures (UNIX uses integer file descriptors). `NULL` indicates an invalid value. The pointers are, in effect, a form of handle to the open file object.
2. The call to `fopen` specifies whether the file is to be treated as a text file or a binary file. Text files contain system-specific character sequences to indicate situations such as an end of line. On many systems, including Windows, I/O operations on a text file convert between the end-of-line character sequence and the null character that C interprets as the end of a string. In the example, both files are opened in binary mode.
3. Errors are diagnosed with `perror`, which, in turn, accesses the global variable `errno` to obtain information about the function call failure. Alternatively, the `ferror` function could be used to return an error code that is associated with the `FILE` rather than the system.
4. The `fread` and `fwrite` functions directly return the number of objects processed rather than return the value in an argument, and this arrangement is essential to the program logic. A successful read is indicated by a non-negative value, and `0` indicates an end of file.
5. The `fclose` function applies only to `FILE` objects (a similar statement applies to UNIX file descriptors).
6. The I/O is synchronous so that the program must wait for the I/O operation to complete before proceeding.
7. The C library `printf` I/O function is useful for error messages and occurs even in the initial Windows example.

The C library implementation has the advantage of portability to UNIX, Windows, and other systems that support ANSI C. Furthermore, as shown in Appendix C, C library performance for sequential I/O is competitive with alternative implementations. Nonetheless, programs are still constrained to synchronous I/O operations, although this constraint will be lifted somewhat when using Windows threads (starting in Chapter 7).

C library file processing programs, like their UNIX equivalents, are able to perform random access file operations (using `fseek` or, in the case of text files, `fsetpos` and `fgetpos`), but that is the limit of sophistication of Standard C library file I/O. *Note:* Microsoft C++ does provide nonstandard extensions that support, for example, file locking. Finally, the C library cannot control file security.

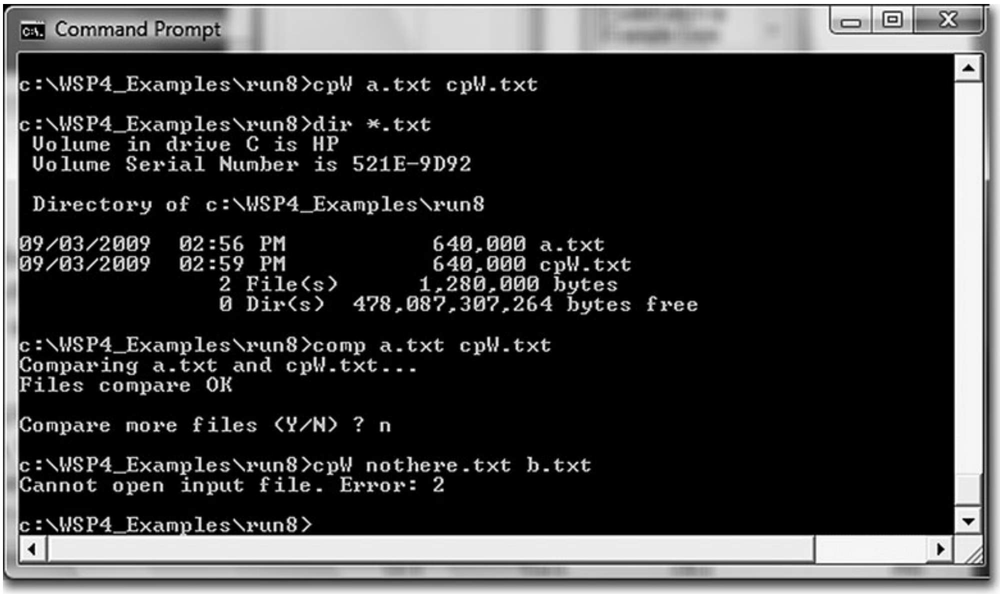
In summary, if simple synchronous file or console I/O is all that is needed, then use the C library to write portable programs that will run under Windows.

File Copying with Windows

`cpW` (Program 1–2) shows the same program using the Windows API, and the same basic techniques, style, and conventions are used throughout this book.

Program 1–2 `cpW`**:** File Copying with Windows, First Implementation

```
/* Chapter 1. cpW. Basic file copy program. Windows Implementation. */
/* cpW file1 file2: Copy file1 to file2. */

#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 256
int main(int argc, LPTSTR argv[])
{
   HANDLE hIn, hOut;
   DWORD nIn, nOut;
   CHAR buffer[BUF_SIZE];

   if (argc != 3) {
      printf("Usage: cp file1 file2\n");
      return 1;
   }

   hIn = CreateFile(argv[1], GENERIC_READ, FILE_SHARE_READ, NULL,
         OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
   if (hIn == INVALID_HANDLE_VALUE) {
      printf("Cannot open input file. Error: %x\n", GetLastError());
      return 2;
   }

   hOut = CreateFile(argv[2], GENERIC_WRITE, 0, NULL,
         CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
   if (hOut == INVALID_HANDLE_VALUE) {
      printf("Cannot open output file. Error: %x\n",
            GetLastError());
      return 3;
   }

   while (ReadFile(hIn, buffer, BUF_SIZE, &nIn, NULL) && nIn > 0) {
      WriteFile(hOut, buffer, nIn, &nOut, NULL);
      if (nIn != nOut) {
         printf("Fatal write error: %x\n", GetLastError());
         return 4;
      }
   }
   CloseHandle(hIn); CloseHandle(hOut);
   return 0;
}
```



```
c:\WSP4_Examples\run8>cpW a.txt cpW.txt

c:\WSP4_Examples\run8>dir *.txt
 Volume in drive C is HP
 Volume Serial Number is 521E-9D92

 Directory of c:\WSP4_Examples\run8

09/03/2009  02:56 PM           640,000 a.txt
09/03/2009  02:59 PM           640,000 cpW.txt
               2 File(s)      1,280,000 bytes
               0 Dir(s)  478,087,307,264 bytes free

c:\WSP4_Examples\run8>comp a.txt cpW.txt
Comparing a.txt and cpW.txt...
Files compare OK

Compare more files (Y/N) ? n

c:\WSP4_Examples\run8>cpW nothere.txt b.txt
Cannot open input file. Error: 2

c:\WSP4_Examples\run8>
```

Run 1–2 `cpW`: Execution and Test

Run 1–2 shows `cpW` execution, showing the same information as Run 1–1. All text files other than `a.txt` were removed before the run.

This simple example illustrates some Windows programming features that Chapter 2 will start to explain in detail.

1. `windows.h` is always necessary and contains all Windows function definitions and data types.
2. Although there are some important exceptions, most Windows objects in this book are identified by variables of type `HANDLE`, and a single generic `CloseHandle` function applies to most objects.
3. Close all open handles when they are no longer required so as to free resources. However, the handles will be closed automatically by Windows when a process exits, and Windows will destroy an object and free its resources, as appropriate, if there are no remaining handles referring to the object. (*Note:* Closing the handle does not destroy the file.)
4. Windows defines numerous symbolic constants and flags. Their names are usually quite long and often describe their purposes. `INVALID_HANDLE_VALUE` and `GENERIC_READ` are typical.

5. Functions such as `ReadFile` and `WriteFile` return `BOOL` values, which you can use in logical expressions, rather than byte counts, which are arguments. This alters the loop logic slightly.[11] The end of file is detected by a zero byte count and is not a failure.
6. System error codes, as `DWORD`s, can be obtained immediately after a failed system call through `GetLastError`. Program 2–1 shows how to obtain Windows-generated textual error messages.
7. Windows has a powerful security system, described in Chapter 15. The output file in this example is owned by the user and will be secured with the user's default settings.
8. Functions such as `CreateFile` have a rich set of options, and the example uses default values.

File Copying with a Windows Convenience Function

Windows has a number of convenience functions that combine several functions to perform a common task. These convenience functions can also improve performance in some cases (see Appendix C). `CopyFile`, for example, greatly simplifies the file copy program, `cpCF` (Program 1–3). Among other things, there is no need to be concerned with the appropriate buffer size, which was arbitrarily 256 in the two preceding programs. Furthermore, `CopyFile` copies file metadata (such as time stamps) that will not be preserved by the other two programs.

Program 1–3 `cpCF`: File Copying with a Windows Convenience Function

```
/* Chapter 1. cpCF. Basic file copy program. Windows implementation
   using CopyFile for convenience and improved performance. */
/* cpCF file1 file2: Copy file1 to file2. */

#include <windows.h>
#include <stdio.h>

int main(int argc, LPTSTR argv[])
{
   if (argc != 3) {
      printf("Usage: cpCF file1 file2\n");
      return 1;
   }
```

[11] Notice that the loop logic depends on ANSI C's left-to-right evaluation of logical "and" (`&&`) and logical "or" (`||`) operations.

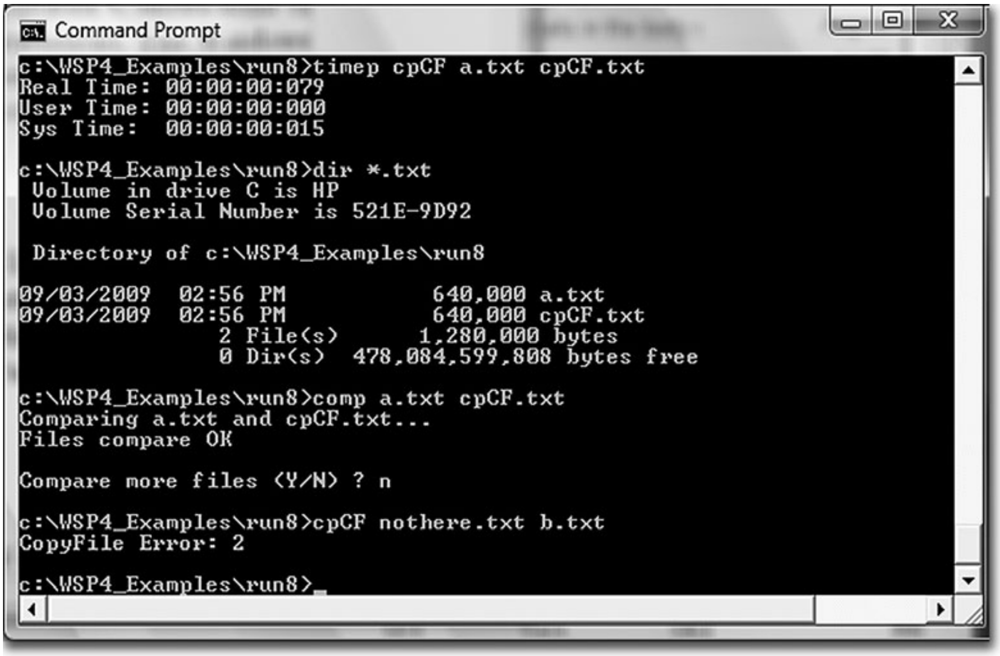
```
    if (!CopyFile(argv[1], argv[2], FALSE)) {
        printf("CopyFile Error: %x\n", GetLastError());
        return 2;
    }
    return 0;
}
```

Run 1–3 shows the `cpCF` test; notice that `CopyFile` preserves the file time and other attributes of the original file. The previous two copy programs changed the file time.

Also notice the `timep` program, which shows the execution time for a program; `timep` implementation is described in Chapter 6, but it's helpful to use it now. In this example, `a.txt` is small, and the execution time is minimal and not measured precisely. However, you can easily create larger files with `randfile`.

Summary

The introductory examples, three simple file copy programs, illustrate many differences between C library and Windows programs. Appendix C shows some of the performance differences among the various implementations. The Windows exam-



```
c:\WSP4_Examples\run8>timep cpCF a.txt cpCF.txt
Real Time: 00:00:00:079
User Time: 00:00:00:000
Sys Time:  00:00:00:015

c:\WSP4_Examples\run8>dir *.txt
 Volume in drive C is HP
 Volume Serial Number is 521E-9D92

 Directory of c:\WSP4_Examples\run8

09/03/2009  02:56 PM           640,000 a.txt
09/03/2009  02:56 PM           640,000 cpCF.txt
               2 File(s)      1,280,000 bytes
               0 Dir(s)  478,084,599,808 bytes free

c:\WSP4_Examples\run8>comp a.txt cpCF.txt
Comparing a.txt and cpCF.txt...
Files compare OK

Compare more files (Y/N) ? n

c:\WSP4_Examples\run8>cpCF nothere.txt b.txt
CopyFile Error: 2

c:\WSP4_Examples\run8>_
```

Run 1–3 `cpCF:` Execution and Test, with Timing ples clearly illustrate Windows programming style and conventions but only hint at the functionality available to Windows programmers.

Looking Ahead

Chapters 2 and 3 take a much more extensive look at I/O and the file system. Topics include console I/O, ASCII and Unicode character processing, file and directory management, file attributes, and advanced options, as well as registry programming. These two chapters develop the basic techniques and lay the groundwork for the rest of the book.

Additional Reading

Publication information about the following books is listed in the bibliography.

Windows API

Windows via C/C+ by Jeffrey Richter and Christophe Nasarre, covers Windows programming with significant overlap with this book.

The hypertext on-line MSDN help available with Microsoft Visual C++ documents every function, and the same information is available from the Microsoft home page, www.msdn.microsoft.com, which also contains numerous technical papers covering different Windows subjects. Start with MSDN and search for any topic of interest. You'll find a variety of function descriptions, coding examples, white papers, and other useful information.

Windows History

See Raymond Chen's *The Old New Thing: Practical Development Throughout the Evolution of Windows* for a fascinating insider's look at Windows development with explanations of why many Windows features were designed as they are.

Windows NT Architecture

Windows Internals: Including Windows Server 2008 and Windows Vista, by Mark Russinovich, David Solomon, and Alex Ionescu, is for the reader who wants to know more about Windows design objectives or who wants to understand the underlying architecture and implementation. The book discusses objects, processes, threads, virtual memory, the kernel, and I/O subsystems. You may want to refer to *Windows Internals* as you read this book. Also note the earlier books by these authors and Helen Custer that preceded this book and provide important historical insight into Windows evolution.

UNIX

Advanced Programming in the UNIX Environment, by W. Richard Stevens and Stephen A. Rago, discusses UNIX in much the same terms in which this book discusses Windows. This remains the standard reference on UNIX features and offers a convenient working definition of what UNIX, as well as Linux, provides. This book also contrasts C library file I/O with UNIX I/O, and this discussion is relevant to Windows.

If you are interested in OS comparisons and an in-depth UNIX discussion, *The Art of UNIX Programming*, by Eric S. Raymond, is fascinating reading, although many Windows users may find the discussion slightly biased.

Windows GUI Programming

Windows user interfaces are not covered here. See Brent Rector and Joseph M. Newcomer, *Win32 Programming*, and Charles Petzold, *Programming Windows, Fifth Edition*.

Operating Systems Theory

There are many good texts on general OS theory. *Modern Operating Systems,* by Andrew S. Tanenbaum, is one of the more popular.

The ANSI Standard C Library

The Standard C Library, by P. J. Plauger, is a comprehensive guide. For a quick overview, *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, lists and explains the complete library, and this book remains the classic book on C. These books can be used to help decide whether the C library is adequate for your file processing requirements.

Windows CE

SAMS Teach Yourself Windows CE Programming in 24 Hours, by Jason P. Nottingham, Steven Makofsky, and Andrew Tucker, is recommended for those who wish to apply the material in this book to Windows CE.

Exercises

1–1. Compile, build, and execute the three file copy programs. Other possibilities include using UNIX compatibility libraries, including the Microsoft Visual C++ library (a program using this library is included in *Examples*). *Note:* All source code is in the *Examples* file, along with documentation to describe how to build and run the programs using Microsoft Visual Studio.

1–2. Become familiar with a development environment, such as Microsoft Visual Studio 2005 or 2008. In particular, learn how to build console applications. Also experiment with the debugger on the programs in this chapter. *Examples* will get you started, and you will find extensive information on the Microsoft MSDN site and with the development environment's documentation.

1–3. Windows uses the carriage return–line feed (`CR—LF`) sequence to denote an end of line. Determine the effect on Program 1–1 if the input file is opened in binary mode and the output file in text mode, and conversely. What is the effect under UNIX or some other system?

1–4. Time the file copy programs using large files. Use `timep` to time program execution and use `randfile`, or any other technique, to generate large files. Obtain data for as many of the combinations as possible and compare the results. Needless to say, performance depends on numerous factors, but by keeping other system parameters the same, it is possible to get helpful comparisons between the implementations. *Suggestion:* Tabulate the results in a spreadsheet to facilitate analysis. Chapter 6 contains a program, `timep`, for timing program execution, and the executable, `timep.exe`, is in the *Examples* file *run* directories. Appendix C gives some experimental results.