- MOVEFILE_REPLACE_EXISTING—Use this option to replace an existing file.
- MOVEFILE_WRITE_THROUGH—Use this option to ensure that the function does not return until the copied file is flushed through to the disk.
- MOVEFILE_COPY_ALLOWED—When the new file is on a different volume, the move is achieved with a CopyFile followed by a DeleteFile. You cannot move a file to a different volume without using this flag, and moving a file to the same volume just involves renaming without copying the file data, which is fast compared to a full copy.
- MOVEFILE_DELAY_UNTIL_REBOOT—This flag, which cannot be used in conjunction with MOVEFILE_COPY_ALLOWED, is restricted to administrators and ensures that the file move does not take effect until Windows restarts. Also, if the new file name is null, the existing file will be deleted when Windows restarts.

UNIX pathnames do not include a drive or server name; the slash indicates the system root. The Microsoft C library file functions also support drive names as required by the underlying Windows file naming.

UNIX does not have a function to copy files directly. Instead, you must write a small program or call system() to execute the cp command.

unlink is the UNIX equivalent of DeleteFile except that unlink can also delete directories.

rename and remove are in the C library, and rename will fail when attempting to move a file to an existing file name or a directory to a non-empty directory.

Directory Management

Creating or deleting a directory involves a pair of simple functions.

```
BOOL CreateDirectory (
   LPCTSTR lpPathName,
   LPSECURITY_ATTRIBUTES lpSecurityAttributes)

BOOL RemoveDirectory (LPCTSTR lpPathName)
```

lpPathName points to a null-terminated string with the name of the directory that is to be created or deleted. The security attributes, as with other functions, should be NULL for the time being; Chapter 15 describes file and object security. Only an empty directory can be removed.

A process has a current, or working, directory, just as in UNIX. Furthermore, each individual drive keeps a working directory. Programs can both get and set the current directory. The first function sets the directory.

```
BOOL SetCurrentDirectory (LPCTSTR lpPathName)
```

`lpPathName` is the path to the new current directory. It can be a relative path or a fully qualified path starting with either a drive letter and colon, such as `D:`, or a UNC name (such as `\\ACCTG_SERVER\PUBLIC`).

If the directory path is simply a drive name (such as `A:` or `C:`), the working directory becomes the working directory on the specified drive. For example, if the working directories are set in the sequence

```
C:\MSDEV
INCLUDE
A:\MEMOS\TODO
C:
```

then the resulting working directory will be

```
C:\MSDEV\INCLUDE
```

The next function returns the fully qualified pathname into a specified buffer.

```
DWORD GetCurrentDirectory (DWORD cchCurDir,
   LPTSTR lpCurDir)

Return: The string length of the pathname, or the required buffer
    size (in characters including the terminating character) if the
    buffer is not large enough; zero if the function fails.
```

`cchCurDir` is the character (not byte; the `ccb` prefix denotes byte length) length of the buffer for the directory name. The length must allow for the terminating null character. `lpCurDir` points to the buffer to receive the pathname string.

Notice that if the buffer is too small for the pathname, the return value tells how large the buffer should be. Therefore, the test for function failure should test both for zero and for the result being larger than the `cchCurDir` argument.

This method of returning strings and their lengths is common in Windows and must be handled carefully. Program 2–6 illustrates a typical code fragment that performs the logic. Similar logic occurs in other examples. The method is not always consistent, however. Some functions return a Boolean, and the length parameter is used twice; it is set with the length of the buffer before the call, and the function changes the value. `LookupAccountName` in Chapter 15 is one of more complex functions in terms of returning results.

An alternative approach, illustrated with the `GetFileSecurity` function in Program 15–4, is to make two function calls with a buffer memory allocation in between. The first call gets the string length, which is used in the memory allocation. The second call gets the actual string. The simplest approach in this case is to allocate a string holding `MAX_PATH` characters.

Examples Using File and Directory Management Functions

`pwd` (Program 2–6) uses `GetCurrentDirectory`. Example programs in Chapter 3 and elsewhere use other file and directory management functions.

Console I/O

Console I/O can be performed with `ReadFile` and `WriteFile`, but it is simpler to use the specific console I/O functions, `ReadConsole` and `WriteConsole`. The principal advantages are that these functions process generic characters (`TCHAR`) rather than bytes, and they also process characters according to the console mode, which is set with the `SetConsoleMode` function.

```
BOOL SetConsoleMode (
   HANDLE hConsoleHandle,
   DWORD dwMode)
```

Return: `TRUE` if and only if the function succeeds.

Parameters

`hConsoleHandle` identifies a console input or screen buffer, which must have `GENERIC_WRITE` access even if it is an input-only device.

`dwMode` specifies how characters are processed. Each flag name indicates whether the flag applies to console input or output. Five commonly used flags, listed here, control behavior; they are all enabled by default.

- ENABLE_LINE_INPUT—Specify that ReadConsole returns when it encounters a carriage return character.
- ENABLE_ECHO_INPUT—Echo characters to the screen as they are read.
- ENABLE_PROCESSED_INPUT—Process backspace, carriage return, and line feed characters.
- ENABLE_PROCESSED_OUTPUT—Process backspace, tab, bell, carriage return, and line feed characters.
- ENABLE_WRAP_AT_EOL_OUTPUT—Enable line wrap for both normal and echoed output.

If SetConsoleMode fails, the mode is unchanged and the function returns FALSE. GetLastError returns the error code number.

The ReadConsole and WriteConsole functions are similar to ReadFile and WriteFile.

```
BOOL ReadConsole (HANDLE hConsoleInput,
   LPVOID lpBuffer,
   DWORD cchToRead,
   LPDWORD lpcchRead,
   LPVOID lpReserved)
```

Return: TRUE if and only if the read succeeds.

The parameters are nearly the same as with ReadFile. The two length parameters are in terms of generic characters rather than bytes, and lpReserved must be NULL. Never use any of the reserved fields that occur in this and other functions. WriteConsole is now self-explanatory. The next example (Program 2–5) shows how to use ReadConsole and WriteConsole with generic strings and how to take advantage of the console mode.

A process can have only one console at a time. Applications such as the ones developed so far are normally initialized with a console. In many cases, such as a server or GUI application, however, you may need a console to display status or debugging information. There are two simple parameterless functions for this purpose.

```
BOOL FreeConsole (VOID)
BOOL AllocConsole (VOID)
```

`FreeConsole` detaches a process from its console. Calling `AllocConsole` then creates a new one associated with the process's standard input, output, and error handles. `AllocConsole` will fail if the process already has a console; to avoid this problem, precede the call with `FreeConsole`.

Note: Windows GUI applications do not have a default console and must allocate one before using functions such as `WriteConsole` or `printf` to display on a console. It's also possible that server processes may not have a console. Chapter 6 shows how to create a process without a console.

There are numerous other console I/O functions for specifying cursor position, screen attributes (such as color), and so on. This book's approach is to use only those functions needed to get the examples to work and not to wander further than necessary into user interfaces. It is easy to learn additional functions from the MSDN reference material after you see the examples.

For historical reasons, Windows does not support character-oriented terminals in the way that UNIX does, and not all the UNIX terminal functionality is replicated by Windows. For example, UNIX provides functions for setting baud rates and line control functions. Stevens and Rago dedicate a chapter to UNIX terminal I/O (Chapter 11) and one to pseudo terminals (Chapter 19).

Serious Windows user interfaces are, of course, graphical, with mouse as well as keyboard input. The GUI is outside the scope of this book, but everything we discuss works within a GUI application.

Example: Printing and Prompting

The `ConsolePrompt` function, which appears in `PrintMsg` (Program 2–5), is a useful utility that prompts the user with a specified message and then returns the user's response. There is an option to suppress the response echo. The function uses the console I/O functions and generic characters. `PrintStrings` and `PrintMsg` are the other entries in this module; they can use any handle but are normally used with standard output or error handles. The first function allows a variable-length argument list, whereas the second one allows just one string and is for convenience only. `PrintStrings` uses the `va_start`, `va_arg`, and `va_end` functions in the Standard C library to process the variable-length argument list.

Example programs will use these functions and the generic C library functions as convenient.

See Run 2–6 after Program 2–6 for sample outputs. Chapters 11 and 15 have examples using `ConsolePrompt`.

Program 2–5 `PrintMsg`: Console Prompt and Print Utility Functions

```
/* PrintMsg.c: ConsolePrompt, PrintStrings, PrintMsg */

#include "Environment.h" /* #define or #undef UNICODE here. */
#include <windows.h>
#include <stdarg.h>

BOOL PrintStrings(HANDLE hOut, ...)
/* Write the messages to the output handle. */
{
   DWORD msgLen, count;
   LPCTSTR pMsg;
   va_list pMsgList; /* Current message string. */
   va_start(pMsgList, hOut); /* Start processing messages. */
   while ((pMsg = va_arg(pMsgList, LPCTSTR)) != NULL) {
      msgLen = _tcslen(pMsg);
         /* WriteConsole succeeds only for console handles. */
      if (!WriteConsole(hOut, pMsg, msgLen, &count, NULL)
               /* Call WriteFile only if WriteConsole fails. */
            && !WriteFile(hOut, pMsg, msgLen * sizeof(TCHAR),
               &count, NULL))
         va_end(pMsgList);
         return FALSE;
   }
   va_end(pMsgList);
   return TRUE;
}

BOOL PrintMsg(HANDLE hOut, LPCTSTR pMsg)
/* Single message version of PrintStrings. */
{
   return PrintStrings(hOut, pMsg, NULL);
}

BOOL ConsolePrompt(LPCTSTR pPromptMsg, LPTSTR pResponse,
      DWORD maxChar, BOOL echo)
/* Prompt the user at the console and get a response
   which can be up to maxChar generic characters. */
{
   HANDLE hIn, hOut;
   DWORD charIn, echoFlag;
   BOOL success;

   hIn = CreateFile(_T("CONIN$"), GENERIC_READ | GENERIC_WRITE, 0,
         NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

```
    hOut = CreateFile(_T("CONOUT$"), GENERIC_WRITE, 0,
           NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    echoFlag = echo ? ENABLE_ECHO_INPUT : 0;
    success =
           SetConsoleMode(hIn, ENABLE_LINE_INPUT |
              echoFlag | ENABLE_PROCESSED_INPUT)
           && SetConsoleMode(hOut,
              ENABLE_WRAP_AT_EOL_OUTPUT | ENABLE_PROCESSED_OUTPUT)
           && PrintStrings(hOut, pPromptMsg, NULL)
           && ReadConsole(hIn, pResponse, maxChar - 2, &charIn, NULL);

    /* Replace the CR-LF by the null character. */
    if (success)
       pResponse[charIn - 2] = '\0';
    else
       ReportError(_T("ConsolePrompt failure."), 0, TRUE);

    CloseHandle(hIn);
    CloseHandle(hOut);
    return success;
}
```

Notice that `ConsolePrompt` returns a Boolean success indicator. Furthermore, `GetLastError` will return the error from the function that failed, but it's important to call `ReportError`, and hence `GetLastError`, before the `CloseHandle` calls.

Also, `ReadConsole` returns a carriage return and line feed, so the last step is to insert a null character in the proper location over the carriage return. The calling program must provide the `maxChar` parameter to prevent buffer overflow.

Example: Printing the Current Directory

`pwd` (Program 2–6) implements a version of the UNIX command `pwd`. The `MAX_PATH` value specifies the buffer size, but there is an error test to illustrate `GetCurrentDirectory`.

Program 2–6 `pwd`: Printing the Current Directory

```
/* Chapter 2. pwd. */
/* pwd: Print the current directory. */
/* This program illustrates:
   1. Windows GetCurrentDirectory
   2. Testing the length of a returned string */
```

```
#include "Everything.h"

#define DIRNAME_LEN (MAX_PATH + 2)

int _tmain(int argc, LPTSTR argv[])
{
   /* Buffer to receive current directory allows for the CR,
      LF at the end of the longest possible path. */
   TCHAR pwdBuffer[DIRNAME_LEN];
   DWORD lenCurDir;

   lenCurDir = GetCurrentDirectory(DIRNAME_LEN, pwdBuffer);
   if (lenCurDir == 0)
      ReportError(_T("Failure getting pathname."), 1, TRUE);
   if (lenCurDir > DIRNAME_LEN)
      ReportError(_T("Pathname is too long."), 2, FALSE);

   PrintMsg(GetStdHandle(STD_OUTPUT_HANDLE), pwdBuffer);
   return 0;
}
```

Run 2–6, shows the results, which appear on a single line. The Windows Command Prompt produces the first and last lines, whereas `pwd` produces the middle line.

Run 2–6 `pwd`: Determining the Current Directory

Summary

Windows supports a complete set of functions for processing and managing files and directories, along with character processing functions. In addition, you can write portable, generic applications that can be built for either ASCII or Unicode operation.

The Windows functions resemble their UNIX and C library counterparts in many ways, but the differences are also apparent. Appendix B discusses portable coding techniques. Appendix B also has a table showing the Windows, UNIX, and C library functions, noting how they correspond and pointing out some of the significant differences.

Looking Ahead

The next step, in Chapter 3, is to discuss direct file access and to learn how to deal with file and directory attributes such as file length and time stamps. Chapter 3 also shows how to process directories and ends with a discussion of the registry management API, which is similar to the directory management API.

Additional Reading

NTFS and Windows Storage

Inside Windows Storage, by Dilip Naik, is a comprehensive discussion of the complete range of Windows storage options including directly attached and network attached storage. Recent developments, enhancements, and performance improvements, along with internal implementation details, are all described.

Inside the Windows NT File System, by Helen Custer, and *Windows NT File System Internals*, by Rajeev Nagar, are additional references, as is the previously mentioned *Windows Internals: Including Windows Server 2008 and Windows Vista*.

Unicode

Developing International Software, by Dr. International (that's the name on the book), shows how to use Unicode in practice, with guidelines, international standards, and culture-specific issues.

UNIX

Stevens and Rado cover UNIX files and directories in Chapters 3 and 4 and terminal I/O in Chapter 11.

UNIX in a Nutshell, by Arnold Robbins et al., is a useful quick reference on the UNIX commands.

Exercises

2–1. Write a short program to test the generic versions of `printf` and `scanf`.

2–2. Modify the `CatFile` function in `cat` (Program 2–2) so that it uses `WriteConsole` rather than `WriteFile` when the standard output handle is associated with a console.

2–3. `CreateFile` allows you to specify file access characteristics so as to enhance performance. `FILE_FLAG_SEQUENTIAL_SCAN` is an example. Use this flag in `cci_f` (Program 2–4) and determine whether there is a performance improvement for large files, including files larger than 4GB. Also try `FILE_FLAG_NO_BUFFERING` after reading the MSDN `CreateFile` documentation carefully. Appendix C shows results on several Windows versions and computers.

2–4. Run `cci` (Program 2–3) with and without `UNICODE` defined. What is the effect, if any?

2–5. Compare the information provided by `perror` (in the C library) and `ReportError` for common errors such as opening a nonexistent file.

2–6. Test the `ConsolePrompt` (Program 2–5) function's suppression of keyboard echo by using it to ask the user to enter and confirm a password.

2–7. Determine what happens when performing console output with a mixture of generic C library and Windows `WriteFile` or `WriteConsole` calls. What is the explanation?

2–8. Write a program that sorts an array of Unicode strings. Determine the difference between the word and string sorts by using `lstrcmp` and `_tcscmp`. Does `lstrlen` produce different results from those of `_tcslen`? The remarks under the `CompareString` function entry in the Microsoft online help are useful.

2–9. Appendix C provides performance data for file copying and `cci` conversion using different program implementations. Investigate performance with the test programs on computers available to you. Also, if possible, investigate performance using networked file systems, SANs, and so on, to understand the impact of various storage architectures when performing sequential file access.

CHAPTER

3 Advanced File and Directory Processing, and the Registry

File systems provide more than sequential processing; they must also provide random access, file locking, directory processing, and file attribute management. Starting with random file access, which is required by database, file management, and many other applications, this chapter shows how to access files randomly at any location and shows how to use Windows 64-bit file pointers to access files larger than 4GB.

The next step is to show how to scan directory contents and how to manage and interpret file attributes, such as time stamps, access, and size. Finally, file locking protects files from concurrent modification by more than one process (Chapter 6) or thread (Chapter 7).

The final topic is the Windows registry, a centralized database that contains configuration information for applications and for Windows itself. Registry access functions and program structure are similar to the file and directory management functions, as shown by the final program example, so this short topic is at the chapter's end rather than in a separate chapter.

The 64-Bit File System

The Windows NTFS supports 64-bit file addresses so that files can, in principle, be as long as 2^{64} bytes. The 2^{32}-byte length limit of older 32-bit file systems, such as FAT, constrains file lengths to 4GB (4×10^9 bytes). This limit is a serious constraint for numerous applications, including large database and multimedia systems, so any complete modern OS must support much larger files.

Files larger than 4GB are sometimes called *very large* or *huge* files, although huge files have become so common that we'll simply assume that any file could be huge and program accordingly.

Needless to say, some applications will never need huge files, so, for many programmers, 32-bit file addresses will be adequate. It is, however, a good idea to start working with 64-bit addresses from the beginning of a new development project, given the rapid pace of technical change and disk capacity growth,[1] cost improvements, and application requirements.

Win32, despite the 64-bit file addresses and the support for huge files, is still a 32-bit OS API because of its 32-bit memory addressing. Win32 addressing limitations are not a concern until Chapter 5.

File Pointers

Windows, just like UNIX, the C library, and nearly every other OS, maintains a *file pointer* with each open file handle, indicating the current byte location in the file. The next `WriteFile` or `ReadFile` operation will start transferring data sequentially to or from that location and increment the file pointer by the number of bytes transferred. Opening the file with `CreateFile` sets the pointer to zero, indicating the start of the file, and the handle's pointer is advanced with each successive read or write. The crucial operation required for random file access is the ability to set the file pointer to an arbitrary value, using `SetFilePointer` and `SetFilePointerEx`.

The first function, `SetFilePointer`, is obsolete, as the handling of 64-bit file pointers is clumsy. `SetFilePointerEx`, one of a number of "extended"[2] functions, is the correct choice, as it uses 64-bit pointers naturally. Nonetheless, we describe both functions here because `SetFilePointer` is still common. In the future, if the extended function is supported in NT5 and is actually superior, we mention the nonextended function only in passing.

`SetFilePointer` shows, for the first time, how Windows handles addresses in large files. The techniques are not always pretty, and `SetFilePointer` is easiest to use with small files.

---

[1] Even inexpensive laptop computers contain 80GB or more of disk capacity, so "huge" files larger than 4GB are possible and sometimes necessary, even on small computers.

[2] The extended functions have an "Ex" suffix and, as would be expected, provide additional functionality. There is no consistency among the extended functions in terms of the nature of the new features or parameter usage. For example, `MoveFileEx` (Chapter 2) adds a new flag input parameter, while `SetFilePointerEx` has a `LARGE_INTEGER` input and output parameters. The registry functions (end of this chapter) have additional extended functions.