CHAPTER

2 Using the Win32 File System and Character I/O

The file system and simple terminal I/O are often the first operating system features that the developer encounters. Early PC operating systems such as MS-DOS did little except manage files and terminal (or *console*) I/O, and these resources are also central features of nearly every operating system.

Files are essential for the long-term storage of data and programs and are the simplest form of program-to-program communication. Furthermore, many aspects of the file system model apply to interprocess and network communication.

The file copy programs in Chapter 1 introduced the four essential sequential file processing functions:

```
CreateFile      WriteFile
ReadFile        CloseHandle
```

This chapter will explain these and other related functions and will also describe character processing and console I/O functions in detail. First, it is necessary to say a few words about the various file systems available and their principal characteristics. In the process, we'll show how to use Unicode wide characters for internationalization. The chapter concludes with an introduction to Win32 file and directory management.

The Win32 File Systems

There are three file systems to deal with, but only the first two will be important throughout this book.

1. The *File Allocation Table* (FAT) file system descends from the original MS-DOS and Windows 3.1 file systems. FAT has been extensively updated to support long file names (DOS and Windows 3.1 have an 11-character limit). This updated system is called the Virtual FAT (VFAT), but the distinction is omitted from now on. The FAT file system is the only one available on diskettes and Windows 9x discs (other than CD-ROMs). FAT32 is a variation that manages disc space more efficiently.
2. The *NT* file system (NTFS) is unique to 2000/NT. In addition to supporting long file names, it provides security, recoverability, compression, extended attributes, and support for very large files and volumes.
3. The *CD-ROM* file system (CDFS), as the name implies, is for accessing information provided on CD-ROMs. Windows NT and Windows 95 both support the CDFS.

All the file systems are accessed in the same way but with limitations. For example, only the NTFS supports security. This book will point out features unique to NTFS as appropriate.

Windows 2000/NT also allows the development of custom file systems, and NT 3.51 supports the so-called *High Performance* file system used with OS/2.

The format of a file system, as a disc volume or partition, is determined when a disc is partitioned. A disc can be partitioned with any combination of the first two file systems.

File Naming

Win32 supports hierarchical file naming, but there are a few subtle distinctions for the UNIX user and basic rules for everyone.

- The full pathname of a disc file starts with a drive name, such as `A:` or `C:`. The `A:` and `B:` drives are normally diskette drives, and `C:`, `D:`, and so on are hard discs and CD-ROMs. Network drives are usually designated by letters that fall later in the alphabet, such as `H:` and `K:`.
- Alternatively, a full pathname, or Universal Naming Code" (UNC), can start with a double backslash, indicating the global root, followed by a server name and a *share* name to indicate a path on a network file server. The first part of the pathname, then, is `\\servername\sharename`.
- The pathname *separator* is the backslash (`\`), although the forward slash (`/`) can be used in API parameters, which is more convenient in C.

- Directory and file names cannot contain any of the ASCII characters with values in the range 1–31 or any of these characters:

  ```
  <  >  :  "  |
  ```

  Names can contain blanks. However, when using file names with blanks on a command line, be sure to put each file name in quotes so that the name is not interpreted as naming two distinct files.

- Directory and file names are *case-insensitive*, but they are also *case-retaining*, so that if the creation name is `MyFile`, the file name will show up as it was created, but the file can also be accessed with the name `myFILE`.

- File and directory names can be as many as 255 characters long, and pathnames are limited to `MAX_PATH` characters (currently 260).

- A period (`.`) separates a file's name from its extension, and extensions quite often indicate the file's type. Thus, `atou.EXE` would be an executable file, and `atou.C` would be a C language source file.

- `.` and `..`, as directory names, indicate the current directory and its parent.

With this introduction, it is now time to learn more about the Win32 functions introduced in Chapter 1.

Opening, Reading, Writing, and Closing Files

The first Win32 function described in detail is `CreateFile`. It is used for opening existing files and creating new ones. This and other functions will be described first by showing the function prototype and then by describing the parameters and operation.

Creating and Opening Files

This is the first Win32 function, so it is described in some detail; later descriptions will frequently be much more streamlined. Nonetheless, `CreateFile` has numerous options not described here; this additional detail can always be found in the on-line help.

The simplest use of `CreateFile` is illustrated in Chapter 1's introductory Win32 program (Program 1–2), in which there are two calls, both of which rely on default values for `fdwShareMode`, `lpsa`, and `hTemplateFile`. `fdwAccess` is either `GENERIC_READ` or `GENERIC_WRITE`.

```
HANDLE CreateFile (
   LPCTSTR lpszName,
   DWORD fdwAccess,
   DWORD fdwShareMode,
   LPSECURITY_ATTRIBUTES lpsa,
   DWORD fdwCreate,
   DWORD fdwAttrsAndFlags,
   HANDLE hTemplateFile)
```

Return: A `HANDLE` to an open file object, or `INVALID_HANDLE_VALUE` in case of failure.

Parameters

The parameter names illustrate some Win32 conventions. The prefix `fdw` is used when a `DWORD` (32 bits, unsigned) contains flags, and `lpsz` ("long pointer to a zero-terminated string") is for pathnames and other strings, although the Microsoft documentation is not entirely consistent. At times, you need to use common sense or read the documentation carefully to determine the correct data types.

`lpszName` is a pointer to the null-terminated string that names the file, pipe, or other named object to open or create. The pathname is normally limited to `MAX_PATH` (260) characters, but Windows 2000/NT can circumvent this restriction if the pathname is prefixed with `\\?\` to allow for very long pathnames (as long as 32K). The prefix is not part of the name. The `LPCTSTR` data type will be explained in an upcoming section; just regard it as a string data type for now.

`fdwAccess` specifies the read and write access, using `GENERIC_READ` and `GENERIC_WRITE`. Flag values such as `READ` and `WRITE` do not exist. The `GENERIC_` prefix may seem redundant, but it is required. Numerous other constant names may seem longer than necessary.

These values can be combined with a bit-wise "or" (`|`), so to open a file for read and write access, use the following:

```
GENERIC_READ | GENERIC_WRITE
```

`fdwShareMode` is a bit-wise "or" combination of the following:

- 0—The file cannot be shared. Furthermore, not even this process can open a second handle on this file. (Uppercase HANDLE is used only when it is important to emphasize the data type.)
- FILE_SHARE_READ—Other processes, including the one making this call, can open this file for concurrent read access.
- FILE_SHARE_WRITE—This allows concurrent writing to the file.

By using locks or other mechanisms, the programmer must take care to prevent concurrent updates to the same file location. There will be much more about this in Chapter 3.

lpsa points to a SECURITY_ATTRIBUTES structure. Use NULL values for now; security is treated in Chapter 5.

fdwCreate specifies whether to create a new file, whether to overwrite an existing file, and so on. The individual values can be combined with the C bitwise "or" operator.

- CREATE_NEW—Fail if the specified file already exists; otherwise, create a new file.
- CREATE_ALWAYS—An existing file will be overwritten.
- OPEN_EXISTING—Fail if the file does not exist.
- OPEN_ALWAYS—Open the file, creating it if it does not exist.
- TRUNCATE_EXISTING—The file length will be set to zero. fdwCreate must specify at least GENERIC_WRITE access.

fdwAttrsAndFlags specifies file attributes and flags. There are 16 flags and attributes. Attributes are characteristics of the file, as opposed to the open HANDLE, and are ignored when an existing file is opened. Here are some of the more important ones:

- FILE_ATTRIBUTE_NORMAL—This attribute can be used only when no other attributes are set (flags can be set, however).
- FILE_ATTRIBUTE_READONLY—Applications can neither write to nor delete the file.
- FILE_FLAG_DELETE_ON_CLOSE—This is useful for temporary files. The file is deleted when the last open HANDLE is closed.
- FILE_FLAG_OVERLAPPED—This attribute flag is important for asynchronous I/O, which is described in Chapter 14. It must be NULL for Windows 9x except for serial I/O devices.

Several additional flags also specify how a file is processed and help the Win32 implementation optimize performance and file integrity.

- `FILE_FLAG_WRITE_THROUGH`—Intermediate caches are written through directly to the file on disc.
- `FILE_FLAG_NO_BUFFERING`—There is no intermediate buffering or caching in user space, and data transfers occur directly to and from the program's buffers. Accordingly, the buffers are required to be on sector boundaries, and complete sectors must be transferred. Use the `GetDiskFreeSpace` function to determine the sector size when using this flag.
- `FILE_FLAG_RANDOM_ACCESS`—The file is intended for random access, and Windows will attempt to optimize file caching.
- `FILE_FLAG_SEQUENTIAL_SCAN`—The file is for sequential access, and Windows will optimize caching accordingly. These last two access modes are not enforced.

`hTemplateFile` is the handle of an open `GENERIC_READ` file that specifies extended attributes to apply to a newly created file, ignoring `fdwAttrsAndFlags`. Normally, this parameter is `NULL`. `hTemplateFile` is ignored when an existing file is opened. This parameter can be used to set the attributes of a new file to be the same as those of an existing file.

The two `CreateFile` instances in Program 1–2 use default values extensively and are as simple as possible but still appropriate for the task. It could be beneficial to use `FILE_FLAG_SEQUENTIAL_SCAN` in both cases. (Exercise 2–3 explores this option, and Appendix C shows the performance results.)

Notice that if the file share attributes and security permit it, there can be numerous open handles on a given file. The open handles can be owned by the same process or by different processes.

Closing Files

One all-purpose function closes and invalidates handles and releases system resources for nearly all objects. Exceptions will be noted. Closing a handle also decrements the object's handle reference count so that nonpersistent objects such as temporary files and events can be deleted. The system will close all open handles on exit, but it is still good practice for programs to close their handles before terminating.

Closing an invalid handle or closing the same handle twice will cause an exception (Chapter 4 discusses exceptions and exception handling). It is not neces-

sary or appropriate to close standard device handles, which are discussed in the section entitled Standard Devices and Console I/O.

```
BOOL CloseHandle (HANDLE hObject)
```

Return: `TRUE` if the function succeeds; `FALSE` otherwise.

The comparable UNIX functions are different in a number of ways. The UNIX `open` function returns an integer file descriptor rather than a handle, and it specifies access, sharing, create options, and the attributes and flags in the single-integer `oflag` parameter. The options overlap, with Win32 providing a richer set.

There is no UNIX equivalent to `fdwShareMode`. UNIX files are always shareable.

Both systems use security information when creating a new file. In UNIX, the `mode` argument specifies the familiar user, group, and other file permissions.

`close` is comparable to `CloseHandle`, but it is not general purpose.

The C library `<stdio.h>` functions use `FILE` objects, which are comparable to handles (for disc files, terminals, tapes, and other devices) connected to streams. The `fopen` mode parameter specifies whether the file data is to be treated as binary or text. There is a set of options for read-only, update, append at the end, and so on. `freopen` allows `FILE` reuse without closing it first. Security permissions cannot be set.

`fclose` closes a `FILE`. Most `stdio` `FILE`-related functions have the `f` prefix.

Reading Files

```
BOOL ReadFile (
   HANDLE hFile,
   LPVOID lpBuffer,
   DWORD nNumberOfBytesToRead,
   LPDWORD lpNumberOfBytesRead,
   LPOVERLAPPED lpOverlapped)
```

Return: `TRUE` if the read succeeds (even if no bytes were read due to an attempt to read past the end of file).

Assume, until Chapter 14, that the file handle does *not* have the FILE_FLAG_OVERLAPPED option set in fdwAttrsAndFlags. For disc files, this assumption will always hold with Windows 9x and CE. ReadFile, then, starts at the current file position (for the handle) and advances the position by the number of bytes transferred.

The function fails, returning FALSE, if the handle is, or other parameters are, invalid. The function does not fail if the file handle is positioned at the end of file.

Parameters

Because of the long variable names and the natural arrangement of the parameters, they are largely self-explanatory. Nonetheless, here are some brief explanations.

hFile is a file handle with GENERIC_READ access. lpBuffer points to the memory buffer to receive the input data. nNumberOfBytesToRead is the number of bytes to read from the file.

lpNumberOfBytesRead points to the actual number of bytes read by the ReadFile call. This value can be zero if the handle is positioned at the end of file, and message-mode named pipes (Chapter 11) can have a zero-length message.

lpOverlapped points to an OVERLAPPED structure (Chapter 3 and Chapter 14). Use NULL for now.

Writing Files

```
BOOL WriteFile (
   HANDLE hFile,
   CONST VOID *lpBuffer,
   DWORD nNumberOfBytesToWrite,
   LPDWORD lpNumberOfBytesWritten,
   LPOVERLAPPED lpOverlapped)
```

Return: TRUE if the function succeeds; FALSE otherwise.

The parameters are familiar by now. Notice that a successful write does not ensure that the data actually is written through to the disc unless FILE_FLAG_WRITE_THROUGH is specified with CreateFile. If the handle is positioned at the file end, Win32 will extend the length of an existing file.

`ReadFileGather` and `WriteFileGather` allow you to read and write using a collection of buffers of different sizes.

UNIX `read` and `write` are the comparable functions, and the programmer supplies a file descriptor, buffer, and byte count. The functions return the number of bytes actually transferred. `0` on `read` indicates the end of file; `-1` indicates an error. Win32, by contrast, requires a separate transfer count and returns Boolean values to indicate success or failure.

The functions in both systems are general purpose and can read from files, terminals, tapes, pipes, and so on.

The C standard I/O library `fread` and `fwrite` binary I/O functions use object size and object count rather than a single byte count as in UNIX and Win32. A short transfer could be caused by either an end of file or an error; test explicitly with `ferror` or `feof`. The library provides a full set of text-oriented functions, such as `fgetc` and `fputc`, that do not exist outside the C library in either OS.

Interlude: Unicode and Generic Characters

Before proceeding, it is necessary to explain how Windows processes characters and differentiates between 8- and 16-bit characters and generic characters.

Win32 supports standard 8-bit characters (type `char` or `CHAR`) and, in Windows 2000/NT, "wide" 16-bit characters (`WCHAR`, which is defined to be the C `wchar_t` type). The Microsoft documentation refers to the 8-bit character set as "ASCII," but it is actually the "Latin-1" character set; "ASCII" will be used for convenience. These types are capable of representing symbols and letters in all major languages, including English, French, Spanish, German, Japanese, and Chinese, using the Unicode representation.

Here are the steps to follow in order to write a generic Win32 application that can be built to use either Unicode or 8-bit characters:

1. Define all characters and strings using the generic types `TCHAR`, `LPTSTR`, and `LPCTSTR`.
2. Include the definitions `#define UNICODE` and `#define _UNICODE` in all source modules to get Unicode wide characters (ANSI C `wchar_t`); otherwise, with `UNICODE` and `_UNICODE` undefined, `TCHAR` will be equivalent to `CHAR` (ANSI C `char`). The definition must precede the `#include <windows.h>` statement. The first preprocessor variable controls the Win32 function definitions, and the second one controls the C library.
3. Character buffer lengths—as used, for example, in `ReadFile`—must be calculated using `sizeof (TCHAR)`.

4. Use the collection of generic C library string and character I/O functions in <tchar.h>. Representative functions that are available are _fgettc, _itot (for itoa), _stprintf (for sprintf), _tstcpy (for strcpy), _ttoi, _totupper, _totlower, and _tprintf.[1] See the on-line help for a complete and extensive list. All these definitions depend on _UNICODE. This collection is not complete. memchr is an example of a function without a wide character implementation. New versions are provided as required.

5. Constant strings should be in one of three forms. Use these conventions for single characters as well. The first two forms are ANSI C; the third—the _T macro (equivalently, TEXT and _TEXT)—is supplied with the Microsoft C compiler.

```
"This string uses 8-bit characters"

L"This string uses 16-bit characters"

_T ("This string uses generic characters")
```

6. Include <tchar.h> after <windows.h> to get required definitions for text macros and generic C library functions.

Windows 2000/NT uses Unicode throughout, and NTFS file names and pathnames are represented in Unicode. If UNICODE is undefined, 8-bit strings will be converted to wide characters as required by calls to system functions. If the program is to run under Windows 95 or 98, which are not Unicode systems, *do not* define UNICODE and _UNICODE. Under NT, the definition is optional unless the executable is to run under both systems.

All future programs will use TCHAR instead of the normal char for characters and character strings unless there is a clear reason to deal with individual 8-bit characters. Similarly, the type LPTSTR indicates a pointer to a generic string, and LPCTSTR indicates, in addition, a constant string. At times, this choice will add some clutter to the programs, but it is the only choice that allows the flexibility necessary to develop the applications in either Unicode form or as 8-bit character programs that can be easily converted to Unicode at a later date. Furthermore, this choice is consistent with common, if not universal, industry practice.

It is worthwhile to examine the system include files to see how TCHAR and the system function interfaces are defined and how they depend on whether or not UNICODE and _UNICODE are defined. A typical entry is of the following form:

[1] The underscore character will indicate that a function or keyword is provided by Microsoft C. Other development systems provide similar capability but may use different names or keywords.

```
#ifdef UNICODE
#define TCHAR WCHAR
#else
#define TCHAR CHAR
#endif
```

Alternative Generic String Processing Functions

String comparisons can use `lstrcmp` and `lstrcmpi` rather than the generic `_tcscmp` and `_tcscmpi` to account for the specific language and region, or *locale*, at run time and also to perform *word* rather than *string* comparisons.[2] String comparisons simply compare the numerical values of the characters, whereas word comparisons consider locale-specific word order. The two methods can give opposite results for string pairs such as *coop/co-op* and *were/we're*.

There is also a group of Win32 functions for dealing with Unicode characters and strings. These functions handle local characteristics transparently. Typical functions are `CharUpper`, which can operate on strings as well as individual characters, and `IsCharAlphaNumeric`. Other string functions include `CompareString` (which is locale-specific) and `MultiByteToWideChar`. Multibyte characters in Windows 3.1 and 9x extend the 8-bit character set to allow double bytes to represent character sets for languages of the Far East. The generic library functions (`_tprintf` and the like) and the Win32 functions (`CharUpper` and the like) will be used in later examples to demonstrate their use. Examples in later chapters will rely mostly on the generic C library.

The Generic Main Function

The C `main` function, with its argument list (`argv []`), should be replaced by the macro `_tmain`. The macro expands to either `main` or `wmain` depending on the `_UNICODE` definition. `_tmain` is defined in `<tchar.h>`, which must be included after `<windows.h>`. A typical main program heading, then, would look like this:

```
#include <windows.h>
#include <tchar.h>
int _tmain (int argc, LPTSTR argv [])
{
    ...
}
```

[2] Historically, the "`l`" prefix was used to indicate a long pointer to the character string parameters.

The Microsoft C `_tmain` function also supports a third parameter for environment strings. This nonstandard extension is also common in UNIX.

Function Definitions

A function such as `CreateFile` is defined through a preprocessor macro as `CreateFileA` when `UNICODE` is not defined and as `CreateFileW` when `UNICODE` is defined. The definitions also describe the string parameters as 8-bit or wide character strings. Consequently, compilers will report a source code error, such as an illegal parameter to `CreateFile`, as an error in the use of `CreateFileA` or `CreateFileW`.

Unicode Strategies

A programmer who is starting a Windows project, either to develop new code or to port existing code, can select from four strategies, based on project requirements.

1. **8-Bit Only**. Ignore Unicode and continue to use the `char` (or `CHAR`) data type and the Standard C library for functions such as `printf`, `atoi`, and `strcmp`.
2. **8-Bit but Unicode Enabled**. Follow the earlier guidelines for a generic application, but do not define the two Unicode preprocessor variables.
3. **Unicode Only**. Follow the generic guidelines, but define the two preprocessor variables. Alternatively, use wide characters and the wide character functions exclusively. The resulting programs will not run properly under Windows 9x.
4. **Unicode and 8-Bit**. The program includes both Unicode and ASCII code and decides at run time which code to execute, based on a run-time switch or other factors.

As mentioned previously, writing generic code, while requiring extra effort, allows the programmer to maintain flexibility and build separate Windows 9x and NT versions. The examples in this book are generic, using strategies 2 and 3, and have generally been tested both ways. Although all these strategies are in common use, strategies 2 and 3 are becoming increasingly popular.

The locale can be set at run time. Program 2–2 shows how the language for error messages is specified.

The POSIX XPG4 internationalization standard, provided by many UNIX vendors, is considerably different from Unicode. Among other things, characters can be represented by four bytes, two bytes, or one byte, depending on the context, locale, and so on.

Microsoft C implements the Standard C library functions, and there are generic versions. Thus, there is a `_tsetlocale` function in `<wchar.h>`. Windows NT uses `UNICODE` characters, and Windows 9x uses the same multibyte characters (a mix of 8- and 16-bit characters) used by Windows 3.1.

Standard Devices and Console I/O

Like UNIX, Win32 has three standard devices for input, output, and error reporting. UNIX uses well-known values for the file descriptors (0, 1, and 2), but Win32 requires handles and provides a function to obtain them for the standard devices.

```
HANDLE GetStdHandle (DWORD nStdHandle)
```

Return: A valid handle if the function succeeds;
`INVALID_HANDLE_VALUE` otherwise.

`GetStdHandle` *Parameters*

`nStdHandle` must have one of these values:

- `STD_INPUT_HANDLE`
- `STD_OUTPUT_HANDLE`
- `STD_ERROR_HANDLE`

The standard device assignments are normally the console and the keyboard. Standard I/O can be redirected.

`GetStdHandle` does not create a new or duplicate handle on a standard device. Successive calls with the same device argument return the same handle value. Closing a standard device handle makes the device unavailable for future use. For this reason, the examples often obtain a standard device handle but do not close it.

```
BOOL SetStdHandle (
   DWORD nStdHandle,
   HANDLE hHandle)
```

Return: `TRUE` or `FALSE` indicating success or failure.

`SetStdHandle` *Parameters*

`nStdHandle` has the same possible values as in `GetStdHandle`. `hHandle` specifies an open file that is to be the standard device.

The normal method, then, for redirecting standard I/O within a process is to use `SetStdHandle` followed by `GetStdHandle`. The resulting handle is used in subsequent I/O operations.

There are two reserved pathnames for console input (the keyboard) and console output: "`CONIN$`" and "`CONOUT$`". Initially, standard input, output, and error are assigned to the console. It is possible to use the console regardless of any redirection to these standard devices; just open handles to "`CONIN$`" or "`CONOUT$`" using `CreateFile`.

UNIX standard I/O redirection can be done in one of three ways (see Stevens, pp. 61–64).

The first method is indirect and relies on the fact that the `dup` function returns the lowest numbered available file descriptor. Suppose you wish to reassign standard input (file descriptor 0) to an open file description, `fd_redirect`. It is possible to write this code:

```
close (STDIN_FILENO);
dup (fd_redirect);
```

The second method uses `dup2`, and the third uses the `F_DUPFD` on the cryptic and overloaded `fcntl` function.

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 hConsole,
   DWORD fdevMode)
```

Return: TRUE if and only if the function succeeds.

SetConsoleMode *Parameters*

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

fdevMode specifies how characters are processed. Each flag name indicates whether the flag applies to console input or output. On creation, all flags except ENABLE_WINDOW_INPUT are set.

- ENABLE_LINE_INPUT—A read function (ReadConsole) returns when a carriage return character is encountered.
- ENABLE_ECHO_INPUT—Characters are echoed to the screen as they are read.
- ENABLE_PROCESSED_INPUT—This flag causes the system to process backspace, carriage return, and line feed characters.
- ENABLE_PROCESSED_OUTPUT—This flag causes the system to process backspace, tab, bell, carriage return, and line feed characters.
- ENABLE_WRAP_AT_EOL_OUTPUT—Line wrap is enabled for both normal and echoed output.

If SetConsoleMode fails, the mode is unchanged and the function returns FALSE. GetLastError will, as is always the case, return the error code number.

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

```
BOOL ReadConsole (HANDLE hConsoleInput,
   LPVOID lpvBuffer,
   DWORD cchToRead,
   LPDWORD lpcchRead,
   LPVOID lpvReserved)
```

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 `lpvReserved` must be `NULL`. Never use any of the reserved fields that occur in some functions. `WriteConsole` is now self-explanatory. The next example 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 7 shows how a process can be created 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. Additional functions will be easy for you to learn from the reference material after you see the examples.

For historical reasons, Windows is not terminal- and console-oriented in the way that UNIX is, and not all the UNIX terminal functionality is replicated by Win32. Stevens dedicates a chapter to UNIX terminal I/O (Chapter 11) and one to pseudo terminals (Chapter 19).

Serious Win32 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

Function `ConsolePrompt`, which is used in Program 2–1, 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. *Note*: The code on the disc included with this book is thoroughly commented and documented. Within the book, most of the comments are omitted for brevity and to concentrate on Win32 usage.

This example also introduces an include file developed for the programs in the book. The file, "`Envirmnt.h`" (listed in Appendix A), contains the `UNICODE` and `_UNICODE` definitions and related preprocessor variables to specify the environment.

Program 2–1 `PrintMsg`: Console Prompt and Print Utility Functions

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

#include "Envirmnt.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))
         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 MaxTchar, BOOL Echo)

/*  Prompt the user at the console and get a response. */
{
   HANDLE hStdIn, hStdOut;
   DWORD TcharIn, EchoFlag;
   BOOL Success;
   hStdIn = CreateFile (_T ("CONIN$"),
          GENERIC_READ | GENERIC_WRITE, 0,
          NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
   hStdOut = CreateFile (_T ("CONOUT$"), GENERIC_WRITE, 0,
          NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
   EchoFlag = Echo ? ENABLE_ECHO_INPUT : 0;
   Success =
          SetConsoleMode (hStdIn, ENABLE_LINE_INPUT |
             EchoFlag | ENABLE_PROCESSED_INPUT)
          && SetConsoleMode (hStdOut,
             ENABLE_WRAP_AT_EOL_OUTPUT | ENABLE_PROCESSED_OUTPUT)
          && PrintStrings (hStdOut, pPromptMsg, NULL)
          && ReadConsole (hStdIn, pResponse,
             MaxTchar, &TcharIn, NULL);
   if (Success) pResponse [TcharIn - 2] = '\0';
   CloseHandle (hStdIn);
   CloseHandle (hStdOut);
   return Success;
}
```

Notice that `ConsolePrompt` returns a Boolean success indicator, exploiting ANSI C's guaranteed left-to-right evaluation of logical "and" (`&`) where evaluation stops on encountering the first `FALSE`. This coding style may appear compact, but it has the advantage of presenting the system calls in a clear, sequential order without the clutter of numerous conditional statements. Furthermore, `GetLastError` will return the error from the function that failed. Win32's Boolean return values (for many functions) encourage the technique.

The function does not report an error; the calling program can do this if necessary.

The code exploits the documented fact that `WriteConsole` fails if the handle is redirected to something other than a console handle. Therefore, it is not necessary to interrogate the handle properties. The function will take advantage of the console mode when the handle is attached to a console.

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.

Example: Error Processing

Program 1–2 showed some rudimentary error processing, obtaining the `DWORD` error number with the `GetLastError` function. A function call, rather than a global error number, such as the UNIX `errno`, ensures that system errors can be unique to the threads (Chapter 8) that share data storage.

The function `FormatMessage` turns the message number into a meaningful message, in English or one of many other languages, returning the message length.

Program 2–2 shows a useful general-purpose error-processing function, `ReportError`, which is similar to the C library `perror` and to `err_sys`, `err_ret`, and other functions in Stevens (pp. 682ff). `ReportError` prints a message specified in the first argument and will terminate with an exit code or return, depending on the value of the second argument. The third argument determines whether the system error message should be displayed.

Notice the arguments to `FormatMessage`. The value returned by `GetLastError` is used as one parameter, and a flag indicates that the message is to be generated by the system. The generated message is stored in a buffer allocated by the function, and the address is returned in a parameter. There are several other parameters with default values. The language for the message can be set at either compile time or run time. `FormatMessage` will not be used again in this book, so there is no further explanation in the text.

`ReportError` can simplify error processing and will be used in nearly all subsequent examples. Chapter 4 modifies this function to generate exceptions.

Program 2–2 introduces the include file `EvryThng.h`. As the name implies, this file includes `<windows.h>`, `Envirmnt.h`, and the other include files explicitly shown in Program 2–1. It also defines commonly used functions, such as `PrintMsg`, `PrintStrings`, and `ReportError` itself. All subsequent examples will use this single include file, which is listed in Appendix A.

Notice the call to the function `HeapFree` near the end of the program. This function will be explained in Chapter 6.

Program 2–2 `ReportError` for Reporting System Call Errors

```
#include "EvryThng.h"
VOID ReportError (LPCTSTR UserMessage, DWORD ExitCode,
       BOOL PrintErrorMsg)

/*  General-purpose function for reporting system errors. */
{
    DWORD eMsgLen, LastErr = GetLastError ();
    LPTSTR lpvSysMsg;
    HANDLE hStdErr = GetStdHandle (STD_ERROR_HANDLE);
    PrintMsg (hStdErr, UserMessage);
    if (PrintErrorMsg) {
        eMsgLen = FormatMessage
               (FORMAT_MESSAGE_ALLOCATE_BUFFER |
               FORMAT_MESSAGE_FROM_SYSTEM, NULL, LastErr,
               MAKELANGID (LANG_NEUTRAL, SUBLANG_DEFAULT),
               (LPTSTR) &lpvSysMsg, 0, NULL);
        PrintStrings (hStdErr, _T ("\n"), lpvSysMsg,
               _T ("\n"), NULL);
        /* Free the memory block containing the error message. */
        HeapFree (GetProcessHeap (), 0, lpvSysMsg); /* See Ch 6. */
    }
    if (ExitCode > 0)
        ExitProcess (ExitCode);
    else
        return;
}
```

Example: Copying Multiple Files to Standard Output

Program 2–3 illustrates standard I/O and extensive error checking as well as user interaction. This program is a limited implementation of the UNIX `cat` command, which copies one or more specified files—or standard input if no files are specified—to standard output.

Program 2–3 includes complete error handling. The error checking is omitted or minimized in most other programs, but the disc contains the complete programs with extensive error checking and documentation. Also, notice the `Options` function (listed in Appendix A), which is called at the start of the program. This function, included on the disc and used throughout the book, evaluates command line option flags and returns the `argv` index of the first file name. Use `Options` in much the same way as `getopt` is used in many UNIX programs.

Program 2–3 `cat`: File Concatenation to Standard Output

```
/*  Chapter 2. cat. */
/*  cat [options] [files] Only the -s option, which suppresses error
    reporting if one of the files does not exist. */

#include "EvryThng.h"
#define BUF_SIZE 0x200

static VOID CatFile (HANDLE, HANDLE);
int _tmain (int argc, LPTSTR argv [])
{
    HANDLE hInFile, hStdIn = GetStdHandle (STD_INPUT_HANDLE);
    HANDLE hStdOut = GetStdHandle (STD_OUTPUT_HANDLE);
    BOOL DashS;
    int iArg, iFirstFile;

    /*  DashS will be set only if "-s" is on the command line. */
    /*  iFirstFile is the argv [] index of the first input file. */
    iFirstFile = Options (argc, argv, _T ("s"), &DashS, NULL);
    if (iFirstFile == argc) { /* No input files in arg list. */
                            /* Use standard input. */
        CatFile (hStdIn, hStdOut);
        return 0;
    }
/*  Process each input file. */
    for (iArg = iFirstFile; iArg < argc; iArg++) {
        hInFile = CreateFile (argv [iArg], GENERIC_READ,
                0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hInFile == INVALID_HANDLE_VALUE && !DashS)
            ReportError (_T ("Cat file open Error"), 1, TRUE);
        CatFile (hInFile, hStdOut);
        CloseHandle (hInFile);
    }
    return 0;
}

/*  Function that does the work:
/*  read input data and copy it to standard output. */
static VOID CatFile (HANDLE hInFile, HANDLE hOutFile)
{
    DWORD nIn, nOut;
    BYTE Buffer [BUF_SIZE];
    while (ReadFile (hInFile, Buffer, BUF_SIZE, &nIn, NULL)
           && (nIn != 0)
           && WriteFile (hOutFile, Buffer, nIn, &nOut, NULL));
    return;
}
```

Example: ASCII to Unicode Conversion

Program 2–4 builds on Program 1–3, which used the `CopyFile` convenience function. File copying is familiar by now, so this example also converts a file to Unicode, assuming it is ASCII; there is no test. The program also includes some error reporting and an option to suppress replacement of an existing file, and it replaces the final call to `CopyFile` with a new function that performs the ASCII to Unicode file conversion.

This program is concerned mostly with ensuring that the conversion can take place successfully. The operation is captured in a single function call at the end. This boilerplate, similar to that in the previous program, will be used again in the future but will not be repeated in the text.

Notice the call to `_taccess`, which tests the file's existence. This is a generic version of the `access` function, which is in the UNIX library but is not a part of the Standard C library. It is defined in `<io.h>`. More precisely, `_taccess` tests to see whether the file is accessible according to the mode in the second parameter. A value of 0 tests for existence, 2 for write permission, 4 for read permission, and 6 for read-write permission. The alternative to test for the file's existence would be to open a handle with `CreateFile` and then close the handle after a validity test.

Program 2–4 `atou`: File Conversion with Error Reporting

```
/*  Chapter 2. atou - ASCII to Unicode file copy. */

#include "EvryThng.h"

BOOL Asc2Un (LPCTSTR, LPCTSTR, BOOL);
int _tmain (int argc, LPTSTR argv [])
{
   DWORD LocFileIn, LocFileOut;
   BOOL DashI = FALSE;
   TCHAR YNResp [3] = _T ("y");

/*  Get the command line options and the index of the input file. */
   LocFileIn = Options (argc, argv, _T ("i"), &DashI, NULL);
   LocFileOut = LocFileIn + 1;

   if (DashI) { /* Does output file exist? */
      /* Generic version of access function to test existence. */
      if (_taccess (argv [LocFileOut], 0) == 0) {
         _tprintf (_T ("Overwrite existing file? [y/n]"));
         _tscanf (_T ("%s"), &YNResp);
         if (lstrcmp (CharLower (YNResp), YES) != 0)
            ReportError (_T ("Will not overwrite"), 4, FALSE);
      }
```

```
    }
    /* This function is modeled on CopyFile. */
    Asc2Un (argv [LocFileIn], argv [LocFileOut], FALSE);
    return 0;
}
```

Program 2–5 is the conversion function `Asc2Un` called by Program 2–4.

Program 2–5 `Asc2Un` Function

```
#include "EvryThng.h"
#define BUF_SIZE 256

BOOL Asc2Un (LPCTSTR fIn, LPCTSTR fOut, BOOL bFailIfExists)

/* ASCII to Unicode file copy function.
   Behavior is modeled after CopyFile. */
{
   HANDLE hIn, hOut;
   DWORD fdwOut, nIn, nOut, iCopy;
   CHAR aBuffer [BUF_SIZE];
   WCHAR uBuffer [BUF_SIZE];
   BOOL WriteOK = TRUE;

   hIn = CreateFile (fIn, GENERIC_READ, 0, NULL,
          OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

   /* Determine CreateFile action if output file already exists. */
   fdwOut = bFailIfExists ? CREATE_NEW : CREATE_ALWAYS;
   hOut = CreateFile (fOut, GENERIC_WRITE, 0, NULL,
          fdwOut, FILE_ATTRIBUTE_NORMAL, NULL);
   while (ReadFile (hIn, aBuffer, BUF_SIZE, &nIn, NULL)
          && nIn > 0 && WriteOK) {
      for (iCopy = 0; iCopy < nIn; iCopy++)
          /* Convert each character. */
          uBuffer [iCopy] = (WCHAR) aBuffer [iCopy];
      WriteOK = WriteFile (hOut, uBuffer, 2 * nIn, &nOut, NULL);
   }
   CloseHandle (hIn);
   CloseHandle (hOut);
   return WriteOK;
}
```

Performance

Appendix C shows that the performance of the file conversion program can be improved by using such techniques as providing a larger buffer and by specifying `FILE_FLAG_SEQUENTIAL_SCAN` with `CreateFile`. Appendix C also contrasts performance on FAT and NTFS file systems.

File and Directory Management

This section introduces the basic functions for file and directory management.

File Management

Win32 provides a number of functions, which are generally straightforward, to manage files. The following functions delete, copy, and rename files. There is also a function to create temporary file names.

Delete files by specifying the file names. Recall that all absolute pathnames start with a drive letter or a server name. It is not possible to delete an open file in Windows 2000/NT, but it is possible in Windows 9x and UNIX; attempting to do so in NT will result in an error. This limitation is hardly a loss; it is actually a beneficial feature.

```
BOOL DeleteFile (LPCTSTR lpszFileName)
```

Copy an entire file using a single function.

```
BOOL CopyFile (
   LPCTSTR lpszExistingFile,
   LPCTSTR lpszNewFile,
   BOOL fFailIfExists)
```

`CopyFile` copies the named existing file and assigns the specified new name to the copy. If a file with the new name already exists, it will be replaced only if `fFailIfExists` is `FALSE`.

A pair of functions is available to rename, or "move," a file. These functions also work for directories (`DeleteFile` and `CopyFile` are restricted to files).

Win32 does not support any file linking whereby two file names can indicate the same actual file. Close examination of Microsoft documentation will show a "number of links" member field in the `BY_HANDLE_FILE_INFO` structure. This field is used by the POSIX subsystem—which must implement links—and is not relevant to Win32.

Shortcuts are supported by the Windows shells, which interpret the file contents to locate the actual file, but not by Win32. Shortcuts provide linklike features, but only to shell users.

```
BOOL MoveFile (
   LPCTSTR lpszExisting,
   LPCTSTR lpszNew)

BOOL MoveFileEx (
   LPCTSTR lpszExisting,
   LPCTSTR lpszNew,
   DWORD fdwFlags)
```

`MoveFile` fails if the new file already exists; use `MoveFileEx` for existing files. Also, Windows 9x does not usefully implement `MoveFileEx`; it will just return a `FALSE`, indicating an error.

Parameters

`lpszExisting` specifies the name of the existing file *or* directory.

`lpszNew` specifies the new file or directory name, which cannot exist in the case of `MoveFile`. A new file can be on a different file system or drive, but new directories must be on the same drive. If `NULL`, the existing file is deleted.

`fdwFlags` specifies options as follows:

- `MOVEFILE_REPLACE_EXISTING`—Use this option to replace an existing file.
- `MOVEFILE_WRITETHROUGH`—Use this option to ensure that the function does not return until the copied file is flushed through to the disc.
- `MOVEFILE_COPY_ALLOWED`—When the new file is on a different volume, the move is achieved with a `CopyFile` followed by a `DeleteFile`.

- 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 the system restarts.

There are a couple of important limitations when you're moving (renaming) files.

- Since Windows 9x does not implement MoveFileEx, you must perform a CopyFile followed by a DeleteFile. This means that two copies will exist temporarily, which could be a problem with a nearly full disc or a large file. The effect on file time attributes is different from that of a true move.
- Wildcards are not allowed in file or directory names. Specify the actual name.

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 Win32 file naming.

UNIX does not have a function to copy files directly. Instead, you must write a small program or fork a process 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 directory that is not empty. A new directory can exist if it is empty.

As mentioned previously, Win32 does not support the concept of links.

Directory Management

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

```
BOOL CreateDirectory (
   LPCTSTR lpszPath,
   LPSECURITY_ATTRIBUTES lpsa)

BOOL RemoveDirectory (LPCTSTR lpszPath)
```

lpszPath points to a null-terminated string with the name of the directory that is to be created or deleted. The security attributes should be NULL for the time being. 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. The programmer can both get and set the current directory. The first function sets the directory.

```
BOOL SetCurrentDirectory (LPCTSTR lpszCurDir)
```

`lpszCurDir` 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 buffer provided by the programmer.

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

Return: The string length of the returned pathname, or the required buffer size if the buffer is not large enough; zero if the function fails.

Parameters

`cchCurDir` is the character (not byte) length of the buffer for the directory name. The length must allow for the terminating null character. `lpszCurDir` 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 Win32 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 5 is one of many examples.

An alternative approach, illustrated with the `GetFileSecurity` function in Program 5–4, 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.

Example: Printing the Current Directory

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

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

```
/*  Chapter 2. pwd - Print working directory. */

#include "EvryThng.h"
#define DIRNAME_LEN MAX_PATH + 2

int _tmain (int argc, LPTSTR argv [])
{
   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;
}
```

Summary

Win32 supports a complete set of file processing and character processing functions. In addition, you can write portable, generic applications that will operate under all Windows platforms.

The Win32 functions resemble their UNIX and C library counterparts in many ways, but the differences are also apparent. Appendix B contains a table showing the Win32, 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

Unicode

Developing International Applications for Windows 95 and Windows NT, by Nadine Kano, shows how to use Unicode in practice, with guidelines, international standards, and culture-specific issues.

The Microsoft home page has several helpful articles on Unicode. *Unicode Support in Win32* is the basic paper; a search will turn up others.

UNIX

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

UNIX in a Nutshell, by Daniel Gilly, is a useful quick reference on the UNIX commands.

For information on the POSIX international character set and its use, as well as general internationalization issues, see *Programming for the World*, by Sandra O'Donnell.

NTFS

Inside the Windows NT File System, by Helen Custer, is a short monograph describing the goals and implementation of the NTFS. This information is helpful in both this chapter and the next.

Exercises

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

2–2. Modify the `CatFile` function in Program 2–3 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 Program 2–5 and determine whether there is a performance improvement for large files. Appendix C shows results on several systems. Also try `FILE_FLAG_NO_BUFFERING`.

2–4. Determine whether there are detectable performance differences between the FAT and NTFS file systems when using `atou` to convert large files.

2–5. Run Program 2–4 with and without `UNICODE` defined. What is the effect, if any, under Windows 2000/NT and Windows 9x?

2–6. Compare the information provided by `perror` (in the C library) and `ReportError` for common errors such as opening a nonexisting file.

2–7. Test the `ConsolePrompt` (Program 2–1) function's suppression of keyboard echo by using it to ask the user to enter and confirm a password.

2–8. Determine what happens when performing console output with a mixture of generic C library and Win32 `WriteFile` or `WriteConsole` calls. What is the explanation?

2–9. 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 on-line help are useful.

2–10. Extend the `Options` function implementation so that it will report an error if the command line option string contains any characters not in the list of permitted options in the function's `OptionString` parameter.