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 starting a Windows project, either to develop new code or to enhance or 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 or Unicode with generic code**. Follow the earlier guidelines for generic code. The example programs generally use this strategy with the Unicode macros undefined to produce 8-bit code.
3. **Unicode only**. Follow the generic guidelines, but define the two preprocessor variables. Alternatively, use wide characters and the wide character functions exclusively.
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 and creating awkward-looking code, allows the programmer to maintain maximum flexibility. However, Unicode only (Strategy 3) is increasingly common, especially with applications requiring a graphical user interface.

`ReportError` (Program 2–1) shows how to specify the language for error messages.

The POSIX XPG4 internationalization standard is considerably different from Unicode. Among other things, characters can be represented by 4 bytes, 2 bytes, or 1 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 uses Unicode characters.

Example: Error Processing

cpW, 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 are unique to the threads (Chapter 7) 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.

ReportError, Program 2–1, 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. 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. This information is sufficient for our needs, but MSDN supplies complete details.

ReportError can simplify error processing, and nearly all subsequent examples use it. Chapter 4 extends ReportError to generate exceptions.

Program 2–1 introduces the include file Everything.h. As the name implies, this file includes windows.h, Environment.h, which has the UNICODE definition, and other include files.[6] It also defines commonly used functions, such as ReportError itself. All subsequent examples will use this single include file, which is in the *Examples* code.

Notice the call to the function LocalFree near the end of the program, as required by FormatMessage (see MSDN). This function is explained in Chapter 5. Previous book editions erroneously used GlobalFree.

See Run 2–2 for sample ReportError output from a complete program, and many other screenshots throughout the book show ReportError output.

---

[6] "Everything" is an exaggeration, of course, but it's everything we need for most examples, and it's used in nearly all examples. Additional special-purpose include files are introduced in later chapters.

Program 2–1 `ReportError`: Reporting System Call Errors

```
#include "Everything.h"

VOID ReportError(LPCTSTR userMessage, DWORD exitCode,
                 BOOL printErrorMessage)
/* General-purpose function for reporting system errors.
   Obtain the error number and convert it to the system error message.
   Display this information and the user-specified message to the
   standard error device, using the generic function _ftprintf.
   userMessage: Message to be displayed to standard error device.
   exitCode: 0 - Return.
             > 0 - ExitProcess with this code.
   printErrorMessage:Display the last system error message if set. */
{
   DWORD eMsgLen, errNum = GetLastError();
   LPTSTR lpvSysMsg;

   _ftprintf(stderr, _T("%s\n"), userMessage);
   if (printErrorMessage) {
      eMsgLen = FormatMessage(
             FORMAT_MESSAGE_ALLOCATE_BUFFER |
                FORMAT_MESSAGE_FROM_SYSTEM,
             NULL, errNum,
             MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
             (LPTSTR)&lpvSysMsg, 0, NULL);
      if (eMsgLen > 0) {
         _ftprintf(stderr, "%s\n", lpvSysMsg);
      } else {
         _ftprintf(stderr, _T("Last Error Number; %d.\n"), errNum);
      }
      if (lpvSysMsg != NULL) LocalFree(lpvSysMsg); /* See Ch 5. */
   }

   if (exitCode > 0)
      ExitProcess(exitCode);
   return;
}
```

Standard Devices

Like UNIX, a Windows process has three standard devices for input, output, and error reporting. UNIX uses well-known values for the file descriptors (`0`, `1`, and `2`), but Windows requires `HANDLE`s 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.

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 in the process with the same device argument return the same handle value. Closing a standard device handle makes the device unavailable for future use within the process. For this reason, the examples often obtain a standard device handle but do not close it.

Chapter 7's `grepMT` example and Chapter 11's `pipe` example illustrate `GetStdHandle` usage.

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

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

Parameters

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

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 use `CreateFile` to open handles to `"CONIN$"` or `"CONOUT$"`. The "Console I/O" section at the end of this chapter covers the subject.

UNIX standard I/O redirection is considerably different (see Stevens and Rago [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. The first method is:

```
close(STDIN_FILENO);

dup(fd_redirect);
```

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

Example: Copying Multiple Files to Standard Output

cat, the next example (Program 2–2), 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 2–2 includes complete error handling. Future program listings omit most error checking for brevity, but the *Examples* contain the complete programs with extensive error checking and documentation. Also, notice the Options function, which is called at the start of the program. This function, included in the *Examples* file 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.

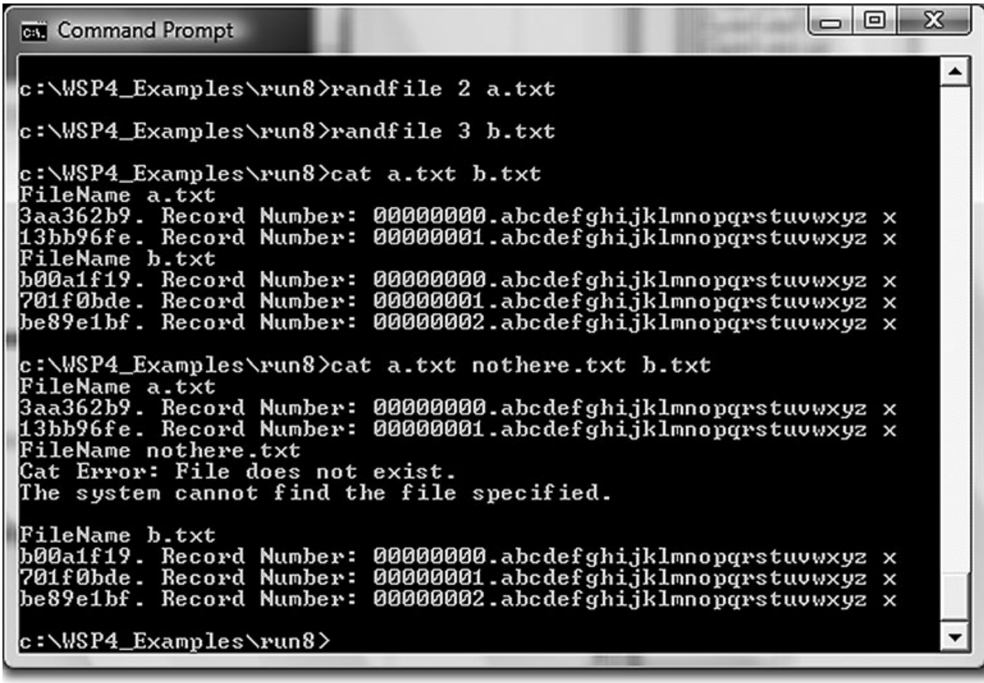
Program 2–2 cat: File Concatenation to Standard Output

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

#include "Everything.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 files. */
   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) {
         if (!dashS) ReportError(_T("Error: File does not exist."),
                              0, TRUE);
      } else {
         CatFile(hInFile, hStdOut);
         if (GetLastError() != 0 && !dashS)
            ReportError(_T("Cat Error."), 0, TRUE);
         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;
}
```

Run 2–2 shows `cat` output with and without errors. The error output occurs when a file name does not exist. The output also shows the text that the `randfile` program generates; `randfile` is convenient for these examples, as it quickly generates text files of nearly any size. Also, notice that the records can be sorted on the first 8 characters, which will be convenient for examples in later chapters. The “x” character at the end of each line is a visual cue and has no other meaning.

Finally, Run 2–2 shows `cat` displaying individual file names; this feature is not part of Program 2–2 but was added temporarily to help clarify Run 2–2.



Run 2–2 `cat:` Results, with `ReportError` Output

Example: Simple File Encryption

File copying is familiar by now, so Program 2–3 also converts a file byte-by-byte so that there is computation as well as file I/O. The conversion is a modified "Caesar cipher," which adds a fixed number to each byte (a Web search will provide extensive background information). The program also includes some error reporting. It is similar to Program 1–3 (`cpCF`), replacing the final call to `CopyFile` with a new function that performs the file I/O and the byte addition.

The shift number, along with the input and output file, are command line parameters. The program adds the shift to each byte modulo 256, which means that the encrypted file may contain unprintable characters. Furthermore, end of line, end of string, and other control characters are changed. A true Caesar cipher only shifts the letters; this implementation shifts all bytes. You can decrypt the file by subtracting the original shift from 256 or by using a negative shift.

This program, while simple, is a good base for numerous variations later in the book that use threads, asynchronous I/O, and other file processing techniques.

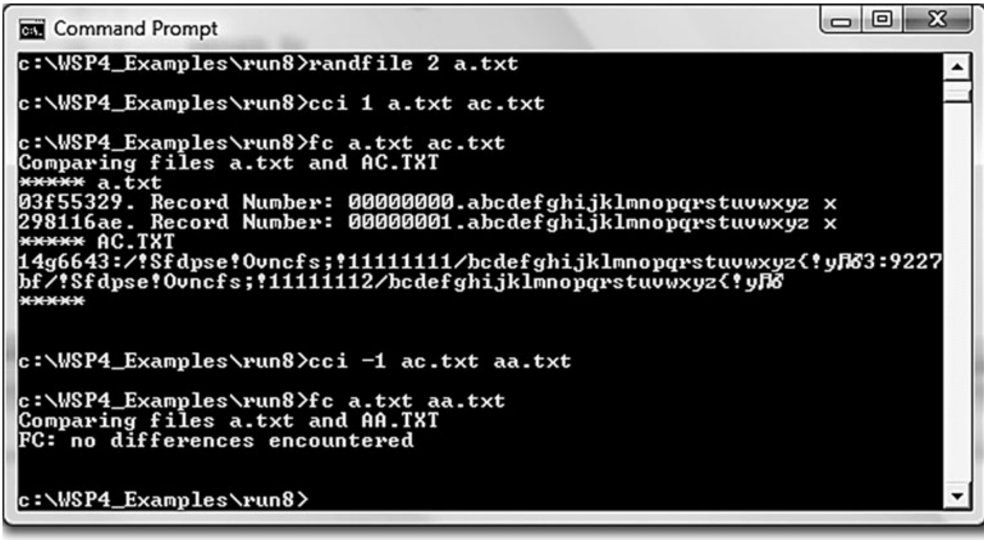
Program 2–4, immediately after Program 2–3, shows the actual conversion function, and Run 2–3 shows program operation with encryption, decryption, and file comparison using the Windows `FC` command.

Comment: Note that the full *Examples* code uses the Microsoft C Library function, `_taccess`, to determine if the file exists. The code comments describe two alternative techniques.

Warning: Future program listings after Program 2–3 omit most, or all, error checking in order to streamline the presentation and concentrate on the logic. Use the full *Examples* code if you want to copy any of the examples.

Program 2–3 `cci`: File Encryption with Error Reporting

```
/* Chapter 2. cci Version 1. Modified Caesar cipher */
/* Main program, which can be linked to different implementations */
/* of the cci_f function. */

/* cci shift file1 file2
 *     shift is the integer added mod 256 to each byte.
 * Otherwise, this program is like cp and cpCF. */

/* This program illustrates:
 *     1. File processing with converstion.
 *     2. Boilerplate code to process the command line.
*/

#include "Everything.h"
#include <io.h>

BOOL cci_f(LPCTSTR, LPCTSTR, DWORD);

int _tmain(int argc, LPTSTR argv[])
{
   if (argc != 4)
      ReportError (_T("Usage: cci shift file1 file2"), 1, FALSE);

   if (!cci_f(argv[2], argv[3], atoi(argv[1])))
      ReportError(_T("Encryption failed."), 4, TRUE);

   return 0;
}
```



Run 2–3 `cci`: Caesar Cipher Run and Test

Program 2–4 is the conversion function `cci_f` called by Program 2–3; later, we'll have several variations of this function.

Program 2–4 `cci_f`: File Conversion Function

```
/* Chapter 2. Simple cci_f (modified Caesar cipher) implementation */
#include "Everything.h"

#define BUF_SIZE 256

BOOL cci_f(LPCTSTR fIn, LPCTSTR fOut, DWORD shift)

/* Simplified Caesar cipher implementation
 *     fIn:   Source file pathname
 *     fOut:  Destination file pathname
 *     shift: Numerical shift
 * Behavior is modeled after CopyFile */
{
   HANDLE hIn, hOut;
   DWORD nIn, nOut, iCopy;
   CHAR aBuffer[BUF_SIZE], ccBuffer[BUF_SIZE];
   BOOL writeOK = TRUE;

   hIn = CreateFile(fIn, GENERIC_READ, 0, NULL, OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL, NULL);
   if (hIn == INVALID_HANDLE_VALUE) return FALSE;
```

```
    hOut = CreateFile(fOut, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);
    if (hOut == INVALID_HANDLE_VALUE) return FALSE;

    while (ReadFile(hIn, aBuffer, BUF_SIZE, &nIn, NULL) &&
            nIn > 0 && writeOK) {
        for (iCopy = 0; iCopy < nIn; iCopy++)
            ccBuffer[iCopy] = (aBuffer[iCopy] + shift) % 256;
        writeOK = WriteFile(hOut, ccBuffer, 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`. Later chapters show more advanced techniques to enhance this simple program.

File and Directory Management

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

File Management

Windows provides a number of file management functions, which are generally straightforward. The functions described here delete, copy, and rename files. There is also a function to create temporary file names.

File Deletion

You can delete a file by specifying the file name and calling the `DeleteFile` function. Recall that all absolute pathnames start with a drive letter or a server name.

```
BOOL DeleteFile (LPCTSTR lpFileName)
```

Copying a File

Copy an entire file using a single function, `CopyFile`, which was introduced in Chapter 1's `cpCF` (Program 1–3) example.

```
BOOL CopyFile (
   LPCTSTR lpExistingFileName,
   LPCTSTR lpNewFileName,
   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`. `CopyFile` also copies file metadata, such as creation time.

Hard and Symbolic Links

Create a hard link between two files with the `CreateHardLink` function, which is similar to a UNIX hard link. With a hard link, a file can have two separate names. Note that there is only one file, so a change to the file will be available regardless of the name used to open the file.

```
BOOL CreateHardLink (
   LPCTSTR lpFileName,
   LPCTSTR lpExistingFileName,
   BOOL lpSecurityAttributes)
```

The first two arguments, while in the opposite order, are used as in `CopyFile`. The two file names, the new name and the existing name, must occur in the same file system volume, but they can be in different directories. The security attributes, if any, apply to the new file name.

Windows Vista and other NT6 systems support a similar symbolic link function, but there is no symbolic link in earlier Windows systems.

```
BOOL CreateSymbolicLink (
   LPTSTR lpSymlinkFileName,
   LPTSTR lpTargetFileName,
   DWORD dwFlags)
```

`lpSymlinkFileName` is the symbolic link that is created to `lpTargetFileName`. Set `dwFlags` to 0 if the target is a file, and set it to `SYMBOLIC_LINK_FLAG_DIRECTORY` if it is a directory. `lpTargetFileName` is treated as an absolute link if there is a device name associated with it. See MSDN for detailed information about absolute and relative links.

Renaming and Moving Files

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

```
BOOL MoveFile (
   LPCTSTR lpExistingFileName,
   LPCTSTR lpNewFileName)

BOOL MoveFileEx (
   LPCTSTR lpExistingFileName,
   LPCTSTR lpNewFileName,
   DWORD dwFlags)
```

`MoveFile` fails if the new file already exists; use `MoveFileEx` to overwrite existing files.

Note: The `Ex` suffix is common and represents an extended version of an existing function in order to provide additional functionality. Many extended functions are not supported in earlier Windows versions.

The `MoveFile` and `MoveFileEx` parameters, especially the flags, are sufficiently complex to require additional explanation:

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

`lpNewFileName` specifies the new file or directory name, which cannot already 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. Wildcards are not allowed in file or directory names. Specify the actual name.

`dwFlags` specifies options as follows: