

Example: A Simple Sequential File Copy

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

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

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

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

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

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

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

File Copying with the Standard C Library

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

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

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

```

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

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

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

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

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

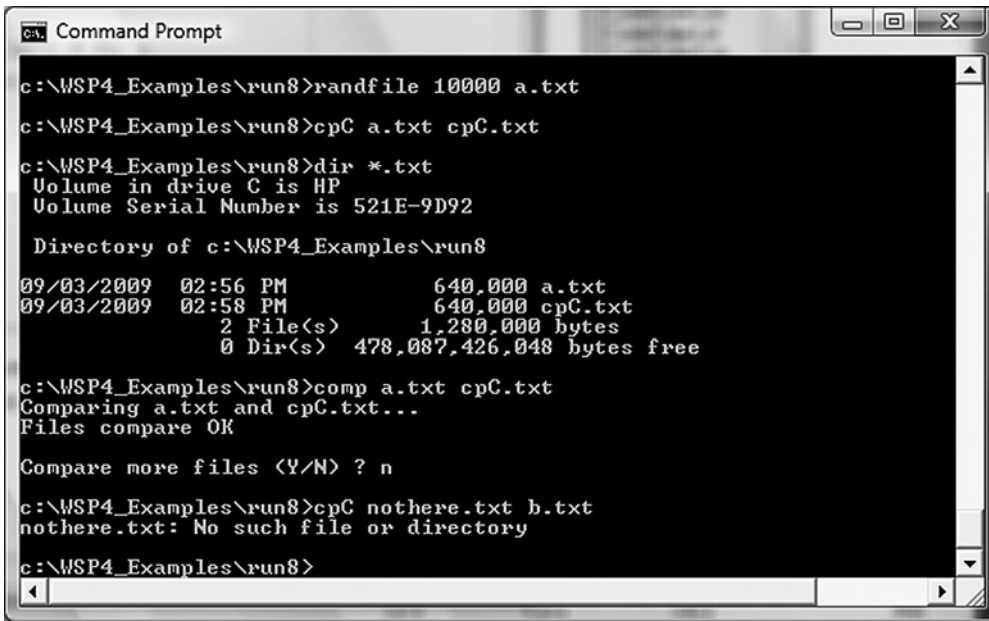
    outFile = fopen(argv[2], "wb");
    if (outFile == NULL) {
        perror(argv[2]);
        return 3;
    }

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

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

```

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

A screenshot of a Windows Command Prompt window titled "C:\> Command Prompt". The window has a black background with white text. The command history shows the following sequence of commands and their outputs:
1. `c:\WSP4_Examples\run8>randfile 10000 a.txt`
2. `c:\WSP4_Examples\run8>cpC a.txt cpC.txt`
3. `c:\WSP4_Examples\run8>dir *.txt`
 Output: "Volume in drive C is HP", "Volume Serial Number is 521E-9D92", "Directory of c:\WSP4_Examples\run8", a table of files, and "478,087,426,048 bytes free".
4. `c:\WSP4_Examples\run8>comp a.txt cpC.txt`
 Output: "Comparing a.txt and cpC.txt...", "Files compare OK".
5. `c:\WSP4_Examples\run8>cpC nothere.txt b.txt`
 Output: "nothere.txt: No such file or directory".
6. `c:\WSP4_Examples\run8>`
The window includes standard Windows window controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

Run 1-1 cpC: Execution and Test

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

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

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

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

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

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

File Copying with Windows

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

Program 1–2 cpw: File Copying with Windows, First Implementation

```

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

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

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

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

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

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

```

```

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

Directory of c:\WSP4_Examples\run8
09/03/2009  02:56 PM                640,000 a.txt
09/03/2009  02:59 PM                640,000 cpw.txt
               2 File(s)            1,280,000 bytes
               0 Dir(s)  478,087,307,264 bytes free

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

Compare more files (Y/N) ? n

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

c:\WSP4_Examples\run8>

```

Run 1-2 cpw: Execution and Test

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

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

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

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

File Copying with a Windows Convenience Function

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

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

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

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

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

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

```

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

```

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

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

Summary

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

```

cmd - Command Prompt
c:\WSP4_Examples\run8>timep cpCF a.txt cpCF.txt
Real Time: 00:00:00:079
User Time: 00:00:00:000
Sys Time: 00:00:00:015

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

Directory of c:\WSP4_Examples\run8

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

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

Compare more files (Y/N) ? n

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

c:\WSP4_Examples\run8>

```

Run 1–3 cpCF: Execution and Test, with Timing

ples clearly illustrate Windows programming style and conventions but only hint at the functionality available to Windows programmers.

Looking Ahead

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

Additional Reading

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

Windows API

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

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

Windows History

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

Windows NT Architecture

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

UNIX

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

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

Windows GUI Programming

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

Operating Systems Theory

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

The ANSI Standard C Library

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

Windows CE

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

Exercises

- 1–1. Compile, build, and execute the three file copy programs. Other possibilities include using UNIX compatibility libraries, including the Microsoft Visual C++ library (a program using this library is included in *Examples*). *Note:* All

source code is in the *Examples* file, along with documentation to describe how to build and run the programs using Microsoft Visual Studio.

- 1–2. Become familiar with a development environment, such as Microsoft Visual Studio 2005 or 2008. In particular, learn how to build console applications. Also experiment with the debugger on the programs in this chapter. *Examples* will get you started, and you will find extensive information on the Microsoft MSDN site and with the development environment's documentation.
- 1–3. Windows uses the carriage return–line feed (CR–LF) sequence to denote an end of line. Determine the effect on Program 1–1 if the input file is opened in binary mode and the output file in text mode, and conversely. What is the effect under UNIX or some other system?
- 1–4. Time the file copy programs using large files. Use `timep` to time program execution and use `randfile`, or any other technique, to generate large files. Obtain data for as many of the combinations as possible and compare the results. Needless to say, performance depends on numerous factors, but by keeping other system parameters the same, it is possible to get helpful comparisons between the implementations. *Suggestion:* Tabulate the results in a spreadsheet to facilitate analysis. Chapter 6 contains a program, `timep`, for timing program execution, and the executable, `timep.exe`, is in the *Examples* file *run* directories. Appendix C gives some experimental results.

This page intentionally left blank