Sequential Files

So far, every example in this book has processed data that resided inside the program listing or came from the keyboard. You assigned constants and variables to other variables and created new data values from expressions. The programs also received input with `cin`, `gets()`, and the character input functions.

The data created by the user and assigned to variables with assignment statements is sufficient for some applications. With the large volumes of data most real-world applications must process, however, you need a better way of storing that data. For all but the smallest computer programs, disk files offer the solution.

After storing data on the disk, the computer helps you enter, find, change, and delete the data. The computer and C++ are simply tools to help you manage and process data. This chapter focuses on disk- and file-processing concepts and teaches you the first of two methods of disk access, *sequential file access.*

This chapter introduces you to the following concepts:

- An overview of disk files
- The types of files
- Processing data on the disk
- Sequential file access
- File I/O functions

After this chapter, you will be ready to tackle the more advanced random-file-access methods covered in the next chapter. If you have programmed computerized data files with another programming language, you might be surprised at how C++ borrows from other programming languages, especially BASIC, when working with disk files. If you are new to disk-file processing, disk files are simple to create and to read.

Why Use a Disk?

The typical computer system has much less memory storage than hard disk storage. Your disk drive holds much more data than can fit in your computer's RAM. This is the primary reason for using the disk for storing your data. The disk memory, because it is nonvolatile, also lasts longer; when you turn your computer off, the disk memory is not erased, whereas RAM is erased. Also, when your data changes, you (or more important, your users) do not have to edit the program and look for a set of assignment statements. Instead, the users run previously written programs that make changes to the disk data.

Disks hold more data than computer memory.

This makes programming more difficult at first because programs have to be written to change the data on the disk. Nonprogrammers, however, can then use the programs and modify the data without knowing C++.

The capacity of your disk makes it a perfect place to store your data as well as your programs. Think about what would happen if all data had to be stored with a program's assignment statements. What if the Social Security Office in Washington, D.C., asked you to write a C++ program to compute, average, filter, sort, and print each person's name and address in his or her files? Would you want your program to include millions of assignment statements? Not only would you not want the program to hold that much data, but it could not do so because only relatively small amounts of data fit in a program before you run out of RAM.

By storing data on your disk, you are much less limited because you have more storage. Your disk can hold as much data as you have disk capacity. Also, if your program requirements grow, you can usually increase your disk space, whereas you cannot always add more RAM to your computer.

NOTE: C++ cannot access the special extended or expanded memory some computers have.

When working with disk files, C++ does not have to access much RAM because C++ reads data from your disk drive and processes the data only parts at a time. Not all your disk data has to reside in RAM for C++ to process it. C++ reads some data, processes it, and then reads some more. If C++ requires disk data a second time, it rereads that place on the disk.

Types of Disk File Access

Your programs can access files two ways: through sequential access or random access. Your application determines the method you should choose. The access mode of a file determines how you read, write, change, and delete data from the file. Some of your files can be accessed in both ways, sequentially and randomly as long as your programs are written properly and the data lends itself to both types of file access.

A sequential file has to be accessed in the same order the file was written. This is analogous to cassette tapes: You play music in the same order it was recorded. (You can quickly fast-forward or rewind over songs you do not want to listen to, but the order of the songs dictates what you do to play the song you want.) It is difficult, and sometimes impossible, to insert data in the middle of a sequential file. How easy is it to insert a new song in the middle of two other songs on a tape? The only way to truly add or delete records from the middle of a sequential file is to create a completely new file that combines both old and new records.

It might seem that sequential files are limiting, but it turns out that many applications lend themselves to sequential-file processing.

Unlike sequential files, you can access random-access files in any order you want. Think of data in a random-access file as you would songs on a compact disc or record; you can go directly to any song you want without having to play or fast-forward over the other songs. If you want to play the first song, the sixth song, and then the fourth song, you can do so. The order of play has nothing to do with the order in which the songs were originally recorded. Random-file access sometimes takes more programming but rewards your effort with a more flexible file-access method. Chapter 31 discusses how to program for random-access files.

Sequential File Concepts

There are three operations you can perform on sequential disk files. You can

- Create disk files
- Add to disk files
- Read from disk files

Your application determines what you must do. If you are creating a disk file for the first time, you must create the file and write the initial data to it. Suppose you wanted to create a customer data file. You would create a new file and write your current customers to that file. The customer data might originally be in arrays, arrays of structures, pointed to with pointers, or placed in regular variables by the user.

Over time, as your customer base grows, you can add new customers to the file (called *appending* to the file). When you add to the end of a file, you append to that file. As your customers enter your store, you would read their information from the customer data file.

Customer disk processing is an example of one disadvantage of sequential files, however. Suppose a customer moves and wants you to change his or her address in your files. Sequential-access files do not lend themselves well to changing data stored in them. It is also difficult to remove information from sequential files. Random files, described in the next chapter, provide a much easier approach

to changing and removing data. The primary approach to changing or removing data from a sequential-access file is to create a new one, from the old one, with the updated data. Because of the updating ease provided with random-access files, this chapter concentrates on creating, reading, and adding to sequential files.

Opening and Closing Files

Before you can create, write to, or read from a disk file, you must open the file. This is analogous to opening a filing cabinet before working with a file stored in the cabinet. Once you are done with a cabinet's file, you close the file drawer. You also must close a disk file when you finish with it.

When you open a disk file, you only have to inform C++ of the filename and what you want to do (write to, add to, or read from). C++ and your operating system work together to make sure the disk is ready and to create an entry in your file directory (if you are creating a file) for the filename. When you close a file, C++ writes any remaining data to the file, releases the file from the program, and updates the file directory to reflect the file's new size.

CAUTION: You must ensure that the FILES= statement in your CONFIG.SYS file is large enough to hold the maximum number of disk files you have open, with one left for your C++ program. If you are unsure how to do this, check your DOS reference manual or a beginner's book about DOS.

To open a file, you must call the `open()` function. To close a file, call the `close()` function. Here is the format of these two function calls:


```
file_ptr.open(file_name, access);
```

and

```
file_ptr.close();
```

`file_ptr` is a special type of pointer that only points to files, not data variables.

Your operating system handles the exact location of your data in the disk file. You don't want to worry about the exact track and sector number of your data on the disk. Therefore, you let `file_ptr` point to the data you are reading and writing. Your program only has to generically manage `file_ptr`, whereas C++ and your operating system take care of locating the actual physical data.

`file_name` is a string (or a character pointer that points to a string) containing a valid filename for your computer. `file_name` can contain a complete disk and directory pathname. You can specify the filename in uppercase or lowercase letters.

`access` must be one of the values from Table 30.1.

Table 30.1. Possible access modes.

Mode	*Description*
`app`	Open the file for appending (adding to it).
`ate`	Seek to end of file on opening it.
`in`	Open the file for reading.
`out`	Open the file for writing.
`binary`	Open the file in binary mode.
`trunc`	Discard contents if file exists
`nocreate`	If file doesn't exist, open fails.
`noreplace`	If file exists, open fails unless appending or seeking to end of file on opening.

The default access mode for file access is a *text mode.* A text file is an ASCII file, compatible with most other programming languages and applications. Text files do not always contain text, in the word-processing sense of the word. Any data you have to store can go in a text file. Programs that read ASCII files can read data you create as C++ text files. For a discussion of binary file access, see the box that follows.

Binary Modes

If you specify binary access, C++ creates or reads the file in a binary format. Binary data files are "squeezed"—they take less space than text files. The disadvantage of using binary files is that other programs cannot always read the data files. Only C++ programs written to access binary files can read and write to them. The advantage of binary files is that you save disk space because your data files are more compact. Other than the access mode in the `open()` function, you use no additional commands to access binary files with your C++ programs.

The binary format is a system-specific file format. In other words, not all computers can read a binary file created on another computer.

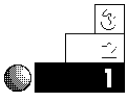
If you open a file for writing, C++ creates the file. If a file by that name already exists, C++ overwrites the old file with no warning. You must be careful when opening files so you do not overwrite existing data that you want to save.

If an error occurs during the opening of a file, C++ does not create a valid file pointer. Instead, C++ creates a file pointer equal to zero. For example, if you open a file for output, but use a disk name that is invalid, C++ cannot open the file and makes the file pointer equal to zero. Always check the file pointer when writing disk file programs to ensure the file opened properly.

TIP: Beginning programmers like to open all files at the beginning of their programs and close them at the end. This is not always the best method. Open files immediately before you access them and close them immediately when you are done with them. This habit protects the files because they are closed immediately after you are done with them. A closed file is more likely to be protected in the unfortunate (but possible) event of a power failure or computer breakdown.

This section contains much information on file-access theories. The following examples help illustrate these concepts.

Examples



1. Suppose you want to create a file for storing your house payment records for the previous year. Here are the first few lines in the program which creates a file called HOUSE.DAT on your disk:

```
#include <fstream.h>

main()
{
   ofstream file_ptr;   // Declares a file pointer for writing
   file_ptr.open("house.dat", ios::out);  // Creates the file
```

The remainder of the program writes data to the file. The program never has to refer to the filename again. The program uses the `file_ptr` variable to refer to the file. Examples in the next few sections illustrate how. There is nothing special about `file_ptr`, other than its name (although the name is meaningful in this case). You can name file pointer variables `XYZ` or `a908973` if you like, but these names would not be meaningful.

You must include the fstream.h header file because it contains the definition for the `ofstream` and `ifstream` declarations. You don't have to worry about the physical specifics. The `file_ptr` "points" to data in the file as you write it. Put the declarations in your programs where you declare other variables and arrays.

TIP: Because files are not part of your program, you might find it useful to declare file pointers globally. Unlike data in variables, there is rarely a reason to keep file pointers local.

Before finishing with the program, you should close the file. The following `close()` function closes the house file:

```
file_ptr.close();      // Close the house payment file.
```

2. If you want, you can put the complete pathname in the file's name. The following opens the household payment file in a subdirectory on the D: disk drive:

```
file_ptr.open("d:\mydata\house.dat", ios::out);
```

3. If you want, you can store a filename in a character array or point to it with a character pointer. Each of the following sections of code is equivalent:

```
char    fn[ ] = "house.dat";  // Filename in character array.
file_ptr.open(fn, ios::out);   // Creates the file.

char    *myfile = "house.dat";   // Filename pointed to.
file_ptr.open(myfile, ios::out);   // Creates the file.

// Let the user enter the filename.
cout << "What is the name of the household file? ";
gets(filename); // Filename must be an array or
                // character pointer.
file_ptr.open(filename, ios::out);   // Creates the file.
```

No matter how you specify the filename when opening the file, close the file with the file pointer. This `close()` function closes the open file, no matter which method you used to open the file:

```
file_ptr.close();   // Close the house payment file.
```

4. You should check the return value from `open()` to ensure the file opened properly. Here is code after `open()` that checks for an error:


```
#include <fstream.h>

main()
{
   ofstream file_ptr;   // Declares a file pointer.
```

```
file_ptr.open("house.dat", ios::out); // Creates the file.
if (!file_ptr)
{  cout << "Error opening file.\n"; }
else
{
   // Rest of output commands go here.
```

5. You can open and write to several files in the same program. Suppose you wanted to read data from a payroll file and create a backup payroll data file. You have to open the current payroll file using the `in` reading mode, and the backup file in the output `out` mode.

   For each open file in your program, you must declare a different file pointer. The file pointers used by your input and output statement determine on which file they operate. If you have to open many files, you can declare an array of file pointers.

   Here is a way you can open the two payroll files:

```
#include <fstream.h>

ifstream       file_in;       // Input file
ofstream       file_out;      // Output file

main()
{
   file_in.open("payroll.dat", ios::in);   // Existing file
   file_out.open("payroll.BAK", ios::out); // New file
```

   When you finish with these files, be sure to close them with these two `close()` function calls:

```
file_in.close();
file_out.close();
```

Writing to a File

Any input or output function that requires a device performs input and output with files. You have seen most of these already. The most common file I/O functions are

```
get() and put()
gets() and puts()
```

You also can use `file_ptr` as you do with `cout` or `cin`.

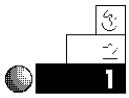
The following function call reads three integers from a file pointed to by `file_ptr`:

```
file_ptr >> num1 >> num2 >> num3;    // Reads three variables.
```

There is always more than one way to write data to a disk file. Most the time, more than one function will work. For example, if you write many names to a file, both `puts()` and `file_ptr <<` work. You also can write the names using `put()`. You should use whichever function you are most comfortable with. If you want a newline character (`\n`) at the end of each line in your file, the `file_ptr <<` and `puts()` are probably easier than `put()`, but all three will do the job.

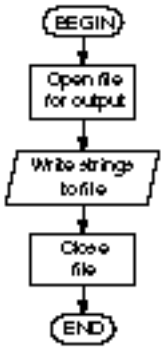
TIP: Each line in a file is called a *record*. By putting a newline character at the end of file records, you make the input of those records easier.

Examples



1. The following program creates a file called NAMES.DAT. The program writes five names to a disk file using `file_ptr <<`.

```
// Filename: C3OWR1.CPP
// Writes five names to a disk file.
#include <fstream.h>

ofstream fp;
```




```
void main()
{
   fp.open("NAMES.DAT", ios::out);  // Creates a new file.

   fp << "Michael Langston\n";
   fp << "Sally Redding\n";
   fp << "Jane Kirk\n";
   fp << "Stacy Wikert\n";
   fp << "Joe Hiquet\n";
   fp.close();   // Release the file.
   return;
}
```

To keep this first example simple, error checking was not done on the `open()` function. The next few examples check for the error.

NAMES.TXT is a text data file. If you want, you can read this file into your word processor (use your word processor's command for reading ASCII files) or use the MS-DOS TYPE command (or your operating system's equivalent command) to display this file on-screen. If you were to display NAMES.TXT, you would see:

```
Michael Langston
Sally Redding
Jane Kirk
Stacy Wikert
Joe Hiquet
```

2. The following file writes the numbers from 1 to 100 to a file called NUMS.1.

```
// Filename: C30WR2.CPP
// Writes 1 to 100 to a disk file.

#include <fstream.h>

ofstream       fp;

void main()
```

```
{
   int ctr;

   fp.open("NUMS.1", ios::out);   // Creates a new file.
   if (!fp)
     {  cout << "Error opening file.\n"; }
   else
     {
        for (ctr = 1; ctr < 101; ctr++)
           {  fp << ctr << " "; }
     }
   fp.close();
   return;
}
```

The numbers are not written one per line, but with a space between each of them. The format of the `file_ptr <<` determines the format of the output data. When writing data to disk files, keep in mind that you have to read the data later. You have to use "mirror-image" input functions to read data you output to files.

Writing to a Printer

Functions such as `open()` and others were not designed to write only to files. They were designed to write to any device, including files, the screen, and the printer. If you must write data to a printer, you can treat the printer as if it were a file. The following program opens a file pointer using the MS-DOS name for a printer located at LPT1 (the MS-DOS name for the first parallel printer port):

```
// Filename: C30PRNT.CPP
// Prints to the printer device

#include <fstream.h>

ofstream prnt;   // Points to the printer.

void main()
```

```
{
   prnt.open("LPT1", ios::out);
   prnt << "Printer line 1\n";     // 1st line printed.
   prnt << "Printer line 2\n";     // 2nd line printed.
   prnt << "Printer line 3\n";     // 3rd line printed.
   prnt.close();
return;
}
```

Make sure your printer is on and has paper before you run this program. When you run the program, you see this printed on the printer:

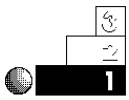
```
Printer line 1
Printer line 2
Printer line 3
```

Adding to a File

You can easily add data to an existing file or create new files, by opening the file in append access mode. Data files on the disk are rarely static; they grow almost daily due to (hopefully!) increased business. Being able to add to data already on the disk is very useful, indeed.

Files you open for append access (using `ios::app`) do not have to exist. If the file exists, C++ appends data to the end of the file when you write the data. If the file does not exist, C++ creates the file (as is done when you open a file for write access).

Example



The following program adds three more names to the NAMES.DAT file created in an earlier example.

```
// Filename: C30AP1.CPP
// Adds three names to a disk file.

#include <fstream.h>
```

```
ofstream    fp;

void main()
{
   fp.open("NAMES.DAT", ios::app);   // Adds to file.
   fp << "Johnny Smith\n";
   fp << "Laura Hull\n";
   fp << "Mark Brown\n";
   fp.close();                  // Release the file.
   return;
}
```

Here is what the file now looks like:

```
Michael Langston
Sally Redding
Jane Kirk
Stacy Wikert
Joe Hiquet
Johnny Smith
Laura Hull
Mark Brown
```

NOTE: If the file does not exist, C++ creates it and stores the three names to the file.

Basically, you only have to change the `open()` function's access mode to turn a file-creation program into a file-appending program.

Reading from a File

Files must exist prior to opening them for read access.

Once the data is in a file, you must be able to read that data. You must open the file in a read access mode. There are several ways to read data. You can read character data one character at a time or one string at a time. The choice depends on the format of the data.

Files you open for read access (using `ios::in`) must exist already, or C++ gives you an error. You cannot read a file that does not exist. `open()` returns zero if the file does not exist when you open it for read access.

Another event happens when reading files. Eventually, you read all the data. Subsequent reading produces errors because there is no more data to read. C++ provides a solution to the end-of-file occurrence. If you attempt to read from a file that you have completely read the data from, C++ returns the value of zero. To find the end-of-file condition, be sure to check for zero when reading information from files.

Examples

1. This program asks the user for a filename and prints the contents of the file to the screen. If the file does not exist, the program displays an error message.

```
// Filename: C30RE1.CPP
// Reads and displays a file.

#include <fstream.h>
#include <stdlib.h>

ifstream fp;

void main()
{
   char filename[12]; // Holds user's filename.
   char in_char;      // Input character.

   cout << "What is the name of the file you want to see? ";
   cin >> filename;
   fp.open(filename, ios::in);
   if (!fp)
   {
     cout << "\n\n*** That file does not exist ***\n";
     exit(0);    // Exit program.
   }
   while (fp.get(in_char))
     {  cout << in_char;  }
   fp.close();
   return;
}
```

Here is the resulting output when the NAMES.DAT file is requested:

```
What is the name of the file you want to see? NAMES.DAT
Michael Langston
Sally Redding
Jane Kirk
Stacy Wikert
Joe Hiquet
Johnny Smith
Laura Hull
Mark Brown
```

Because newline characters are in the file at the end of each name, the names appear on-screen, one per line. If you attempt to read a file that does not exist, the program displays the following message:

```
*** That file does not exist ***
```

2. This program reads one file and copies it to another. You might want to use such a program to back up important data in case the original file is damaged.

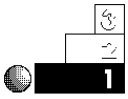
   The program must open two files, the first for reading, and the second for writing. The file pointer determines which of the two files is being accessed.

```
// Filename: C3ORE2.CPP
// Makes a copy of a file.

#include <fstream.h>
#include <stdlib.h>

ifstream in_fp;
ofstream out_fp;

void main()
{
   char in_filename[12];   // Holds original filename.
   char out_filename[12];  // Holds backup filename.
   char in_char;           // Input character.
```

```
   cout << "What is the name of the file you want to back up?
";
   cin >> in_filename;
   cout << "What is the name of the file ";
   cout << "you want to copy " << in_filename << " to? ";
   cin >> out_filename;
   in_fp.open(in_filename, ios::in);
   if (!in_fp)
   {
     cout << "\n\n*** " << in_filename << " does not exist
***\n";
     exit(0);   // Exit program
   }
   out_fp.open(out_filename, ios::out);
   if (!out_fp)
   {
      cout << "\n\n*** Error opening " << in_filename << "
***\n";
      exit(0);    // Exit program
   }
   cout << "\nCopying...\n";   // Waiting message.
   while (in_fp.get(in_char))
     { out_fp.put(in_char); }
   cout << "\nThe file is copied.\n";
   in_fp.close();
   out_fp.close();
   return;
}
```

Review Questions

Answers to the review questions are in Appendix B.

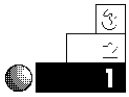


1. What are the three ways to access sequential files?
2. What advantage do disk files have over holding data in memory?
3. How do sequential files differ from random-access files?

4. What happens if you open a file for read access and the file does not exist?
5. What happens if you open a file for write access and the file already exists?
6. What happens if you open a file for append access and the file does not exist?

7. How does C++ inform you that you have reached the end-of-file condition?

Review Exercises



1. Write a program that creates a file containing the following data:

   Your name
   Your address
   Your phone number
   Your age

2. Write a second program that reads and prints the data file you created in Exercise 1.
3. Write a program that takes your data created in Exercise 1 and writes it to the screen one word per line.

4. Write a program for PCs that backs up two important files: the AUTOEXEC.BAT and CONFIG.SYS. Call the backup files AUTOEXEC.SAV and CONFIG.SAV.

5. Write a program that reads a file and creates a new file with the same data, except reverse the case on the second file. Everywhere uppercase letters appear in the first file, write lowercase letters to the new file, and everywhere lowercase letters appear in the first file, write uppercase letters to the new file.

Summary

You can now perform one of the most important requirements of data processing: writing and reading to and from disk files. Before this chapter, you could only store data in variables. The short life of variables (they only last as long as your program is running) made long-term storage of data impossible. You can now save large amounts of data in disk files to process later.

Reading and writing sequential files involves learning more concepts than actual commands or functions. The `open()` and `close()` functions are the most important functions you learned in this chapter. You are now familiar with most of the I/O functions needed to retrieve data to and from disk files.

The next chapter concludes the discussion of disk files in this book. You will learn how to create and use random-access files. By programming with random file access, you can read selected data from a file, as well as change data without having to rewrite the entire file.