Random-Access Files

This chapter introduces the concept of random file access. Random file access enables you to read or write any data in your disk file without having to read or write every piece of data before it. You can quickly search for, add, retrieve, change, and delete information in a random-access file. Although you need a few new functions to access files randomly, you find that the extra effort pays off in flexibility, power, and speed of disk access.

This chapter introduces

- Random-access files
- File records
- The `seekg()` function
- Special-purpose file I/O functions

With C++'s sequential and random-access files, you can do everything you would ever want to do with disk data.

Random File Records

Random files exemplify the power of data processing with C++. Sequential file processing is slow unless you read the entire file into arrays and process them in memory. As explained in Chapter 30, however, you have much more disk space than RAM, and most disk files do not even fit in your RAM at one time. Therefore, you need a way to quickly read individual pieces of data from a file in any order and process them one at a time.

A record to a file is like a structure to variables.

Generally, you read and write file *records*. A record to a file is analogous to a C++ structure. A record is a collection of one or more data values (called *fields*) you read and write to disk. Generally, you store data in structures and write the structures to disk where they are called records. When you read a record from disk, you generally read that record into a structure variable and process it with your program.

Unlike most programming languages, not all disk data for C++ programs has to be stored in record format. Typically, you write a stream of characters to a disk file and access that data either sequentially or randomly by reading it into variables and structures.

The process of randomly accessing data in a file is simple. Think about the data files of a large credit card organization. When you make a purchase, the store calls the credit card company to receive authorization. Millions of names are in the credit card company's files. There is no quick way the credit card company could read every record sequentially from the disk that comes before yours. Sequential files do not lend themselves to quick access. It is not feasible, in many situations, to look up individual records in a data file with sequential access.

The credit card companies must use a random file access so their computers can go directly to your record, just as you go directly to a song on a compact disk or record album. The functions you use are different from the sequential functions, but the power that results from learning the added functions is worth the effort.

You do not have to rewrite an entire file to change random-access file data.

When your program reads and writes files randomly, it treats the file like a big array. With arrays, you know you can add, print, or remove values in any order. You do not have to start at the first

array element, sequentially looking at the next one, until you get the element you need. You can view your random-access file in the same way, accessing the data in any order.

Most random file records are *fixed-length* records. Each record (usually a row in the file) takes the same amount of disk space. Most of the sequential files you read and wrote in the previous chapters were variable-length records. When you are reading or writing sequentially, there is no need for fixed-length records because you input each value one character, word, string, or number at a time, and look for the data you want. With fixed-length records, your computer can better calculate where on the disk the desired record is located.

Although you waste some disk space with fixed-length records (because of the spaces that pad some of the fields), the advantages of random file access compensate for the "wasted" disk space (when the data do not actually fill the structure size).

TIP: With random-access files, you can read or write records in any order. Therefore, even if you want to perform sequential reading or writing of the file, you can use random-access processing and "randomly" read or write the file in sequential record number order.

Opening Random-Access Files

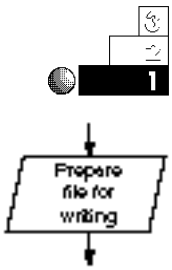
Just as with sequential files, you must open random-access files before reading or writing to them. You can use any of the read access modes mentioned in Chapter 30 (such as `ios::in`) only to read a file randomly. However, to modify data in a file, you must open the file in one of the update modes, repeated for you in Table 31.1.

Table 31.1. Random-access update modes.

Mode	*Description*
`app`	Open the file for appending (adding to it)
`ate`	Seek to end of file on opening it
`in`	Open file for reading
`out`	Open file for writing
`binary`	Open 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

There is really no difference between sequential files and random files in C++. The difference between the files is not physical, but lies in the method you use to access them and update them.

Examples



1. Suppose you want to write a program to create a file of your friends' names. The following `open()` function call suffices, assuming `fp` is declared as a file pointer:

```
fp.open("NAMES.DAT", ios::out);
if (!fp)
  { cout << "\n*** Cannot open file ***\n"; }
```

No update `open()` access mode is needed if you are only creating the file. However, what if you wanted to create the file, write names to it, and give the user a chance to change any of the names before closing the file? You then have to open the file like this:

```
fp.open("NAMES.DAT", ios::in | ios::out);
if (!fp)
   cout << "\n*** Cannot open file ***\n";
```

This code enables you to create the file, then change data you wrote to the file.

2. As with sequential files, the only difference between using a binary `open()` access mode and a text mode is that the file you create is more compact and saves disk space. You cannot, however, read that file from other programs as an ASCII text file. The previous `open()` function can be rewritten to create and allow updating of a binary file. All other file-related commands and functions work for binary files just as they do for text files.

```
fp.open("NAMES.DAT", ios::in | ios::out | ios::binary);
if (!fp)
   cout << "\n*** Cannot open file ***\n";
```

The `seekg()` Function

C++ provides a function that enables you to read to a specific point in a random-access data file. This is the `seekg()` function. The format of `seekg()` is

```
file_ptr.seekg(long_num, origin);
```

You can read forwards or backwards from any point in the file with `seekg()`.

`file_ptr` is the pointer to the file that you want to access, initialized with an `open()` statement. `long_num` is the number of bytes in the file you want to skip. C++ does not read this many bytes, but literally skips the data by the number of bytes specified in `long_num`. Skipping the bytes on the disk is much faster than reading them. If `long_num` is negative, C++ skips backwards in the file (this allows for rereading of data several times). Because data files can be large, you must declare `long_num` as a long integer to hold a large amount of bytes.

`origin` is a value that tells C++ where to begin the skipping of bytes specified by `long_num`. `origin` can be any of the three values shown in Table 31.2.

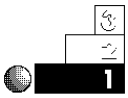
Table 31.2. Possible **origin** values.

Description	origin	*Equivalent*
Beginning of file	SEEK_SET	ios::beg
Current file position	SEEK_CUR	ios::cur
End of file	SEEK_END	ios::end

The origins SEEK_SET, SEEK_CUR, and SEEK_END are defined in stdio.h. The equivalents `ios::beg`, `ios::cur`, and `ios::end` are defined in fstream.h.

NOTE: Actually, the file pointer plays a much more important role than simply "pointing to the file" on the disk. The file pointer continually points to the exact location of the *next byte to read or write*. In other words, as you read data from either a sequential or random-access file, the file pointer increments with each byte read. By using `seekg()`, you can move the file pointer forward or backward in the file.

Examples



1. No matter how far into a file you have read, the following `seekg()` function positions the file pointer back to the beginning of a file:

   ```
   fp.seekg(0L, SEEK_SET); // Position file pointer at beginning.
   ```

   The constant `0L` passes a long integer `0` to the `seekg()` function. Without the `L`, C++ passes a regular integer and this does not match the prototype for `seekg()` that is located in fstream.h. Chapter 4, "Variables and Literals," explained the use of data type suffixes on numeric constants, but the suffixes have not been used until now.

   This `seekg()` function literally reads "move the file pointer 0 bytes from the beginning of the file."

2. The following example reads a file named MYFILE.TXT twice, once to send the file to the screen and once to send the file to the printer. Three file pointers are used, one for each device (the file, the screen, and the printer).

```
// Filename: C31TWIC.CPP
// Writes a file to the printer, rereads it,
// and sends it to the screen.

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

ifstream in_file;   // Input file pointer.
ofstream scrn;      // Screen pointer.
ofstream prnt;      // Printer pointer.

void main()
{
   char in_char;

   in_file.open("MYFILE.TXT", ios::in);
   if (!in_file)
   {
      cout << "\n*** Error opening MYFILE.TXT ***\n";
      exit(0);
   }
   scrn.open("CON", ios::out);    // Open screen device.
   while (in_file.get(in_char))
   { scrn << in_char; } // Output characters to the screen.
   scrn.close();  // Close screen because it is no
                  // longer needed.
   in_file.seekg(0L, SEEK_SET);   // Reposition file pointer.
   prnt.open("LPT1", ios::out);   // Open printer device.
   while (in_file.get(in_char))
     { prnt << in_char; }   // Output characters to the
                            // printer.
   prnt.close();          // Always close all open files.
   in_file.close();
   return;
}
```

You also can close then reopen a file to position the file pointer at the beginning, but using `seekg()` is a more efficient method.

Of course, you could have used regular I/O functions to write to the screen, rather than having to open the screen as a separate device.

3. The following `seekg()` function positions the file pointer at the 30th byte in the file. (The next byte read is the 31st byte.)

```
file_ptr.seekg(30L, SEEK_SET);   // Position file pointer
                                 // at the 30th byte.
```

This `seekg()` function literally reads "move the file pointer 30 bytes from the beginning of the file."

If you write structures to a file, you can quickly seek any structure in the file using the `sizeof()` function. Suppose you want the 123rd occurrence of the structure tagged with `inventory`. You would search using the following `seekg()` function:

```
file_ptr.seekg((123L * sizeof(struct inventory)), SEEK_SET);
```

4. The following program writes the letters of the alphabet to a file called ALPH.TXT. The `seekg()` function is then used to read and display the ninth and 17th letters (*I* and *Q*).

```
// Filename: C31ALPH.CPP
// Stores the alphabet in a file, then reads
// two letters from it.

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

fstream fp;

void main()
{
   char ch;     // Holds A through Z.
```

```
    // Open in update mode so you can read file after writing to it.
    fp.open("alph.txt", ios::in | ios::out);
    if (!fp)
    {
      cout << "\n*** Error opening file ***\n";
      exit(0);
    }
    for (ch = 'A'; ch <= 'Z'; ch++)
    { fp << ch;  }  // Write letters.
    fp.seekg(8L, ios::beg);   // Skip eight letters, point to I.
    fp >> ch;
    cout << "The first character is " << ch << "\n";
    fp.seekg(16L, ios::beg);   // Skip 16 letters, point to Q.
    fp >> ch;
    cout << "The second character is " << ch << "\n";
    fp.close();
    return;
}
```

5. To point to the end of a data file, you can use the `seekg()` function to position the file pointer at the last byte. Subsequent `seekg()`s should then use a negative `long_num` value to skip backwards in the file. The following `seekg()` function makes the file pointer point to the end of the file:

```
file_ptr.seekg(0L, SEEK_END); // Position file
                              // pointer at the end.
```

   This `seekg()` function literally reads "move the file pointer 0 bytes from the end of the file." The file pointer now points to the end-of-file marker, but you can `seekg()` backwards to find other data in the file.

6. The following program reads the ALPH.TXT file (created in Exercise 4) backwards, printing each character as it skips back in the file.

```
// Filename: C31BACK.CPP
// Reads and prints a file backwards.
```

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

ifstream fp;

void main()
{
   int ctr;   // Steps through the 26 letters in the file.
   char in_char;

   fp.open("ALPH.TXT", ios::in);
   if (!fp)
   {
     cout << "\n*** Error opening file ***\n";
     exit(0);
   }
   fp.seekg(-1L, SEEK_END);   // Point to last byte in
                              // the file.
   for (ctr = 0; ctr < 26; ctr++)
   {
      fp >> in_char;
      fp.seekg(-2L, SEEK_CUR);
      cout << in_char;
   }
   fp.close();
   return;
}
```

This program also uses the `SEEK_CUR` `origin` value. The last `seekg()` in the program seeks two bytes backwards from the *current position*, not the beginning or end as the previous examples have. The `for` loop towards the end of the program performs a "skip-two-bytes-back, read-one-byte-forward" method to skip through the file backwards.

7. The following program performs the same actions as Example 4 (C31ALPH.CPP), with one addition. When the letters *I* and *Q* are found, the letter *x* is written over the *I* and *Q*. The `seekg()` must be used to back up one byte in the file to overwrite the letter just read.

```
// Filename: C31CHANG.CPP
// Stores the alphabet in a file, reads two letters from it,
// and changes those letters to xs.

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

fstream fp;

void main()
{
   char ch;   // Holds A through Z.

// Open in update mode so you can read file after writing to it.
   fp.open("alph.txt", ios::in | ios::out);
   if (!fp)
   {
     cout << "\n*** Error opening file ***\n";
     exit(0);
   }
   for (ch = 'A'; ch <= 'Z'; ch++)
      { fp << ch; }   // Write letters
   fp.seekg(8L, SEEK_SET);  // Skip eight letters, point to I.
   fp >> ch;
   // Change the Q to an x.
   fp.seekg(-1L, SEEK_CUR);
   fp << 'x';
   cout << "The first character is " << ch << "\n";
   fp.seekg(16L, SEEK_SET);   // Skip 16 letters, point to Q.
   fp >> ch;
   cout << "The second character is " << ch << "\n";
   // Change the Q to an x.
   fp.seekg(-1L, SEEK_CUR);
   fp << 'x';
   fp.close();
   return;
}
```

The file named ALPH.TXT now looks like this:

```
ABCDEFGHxJKLMNOPxRSTUVWXYZ
```

This program forms the basis of a more complete data file management program. After you master the `seekg()` functions and become more familiar with disk data files, you will begin to write programs that store more advanced data structures and access them.

The mailing list application in Appendix F is a good example of what you can do with random file access. The user is given a chance to change names and addresses already in the file. The program, using random access, seeks for and changes selected data without rewriting the entire disk file.

Other Helpful I/O Functions

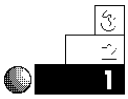
There are several more disk I/O functions available that you might find useful. They are mentioned here for completeness. As you perform more powerful disk I/O, you might find a use for many of these functions. Each of these functions is prototyped in the fstream.h header file.

- `read(array, count)`: Reads the data specified by `count` into the array or pointer specified by `array`. `read()` is called a *buffered I/O* function. `read()` enables you to read much data with a single function call.
- `write(array, count)`: Writes `count array` bytes to the specified file. `write()` is a buffered I/O function. `write()` enables you to write much data in a single function call.
- `remove(filename)`: Erases the file named by `filename`. `remove()` returns a `0` if the file was erased successfully and `-1` if an error occurred.

Many of these (and other built-in I/O functions that you learn in your C++ programming career) are helpful functions that you could duplicate using what you already know.

The buffered I/O file functions enable you to read and write entire arrays (including arrays of structures) to the disk in a single function call.

Examples



1. The following program requests a filename from the user and erases the file from the disk using the `remove()` function.

```
// Filename: C31ERAS.CPP
// Erases the file specified by the user.

#include <stdio.h>
#include <iostream.h>

void main()
{
   char filename[12];

   cout << "What is the filename you want me to erase? ";
   cin >> filename;
   if (remove(filename) == -1)
   {  cout << "\n*** I could not remove the file ***\n"; }
   else
   {  cout << "\nThe file " << filename << " is now removed\n";}
   return;
}
```

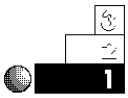
2. The following function is part of a larger program that receives inventory data, in an array of structures, from the user. This function is passed the array name and the number of elements (structure variables) in the array. The `write()` function then writes the complete array of structures to the disk file pointed to by `fp`.

```
void write_str(inventory items[ ], int inv_cnt)
{
   fp.write(items, inv_cnt * sizeof(inventory);
   return;
}
```

If the inventory array had 1,000 elements, this one-line function would still write the entire array to the disk file. You could use the `read()` function to read the entire array of structures from the disk in a single function call.

Review Questions

The answers to the review questions are in Appendix B.

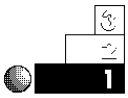


1. What is the difference between records and structures?
2. True or false: You have to create a random-access file before reading from it randomly.
3. What happens to the file pointer as you read from a file?

4. What are the two buffered file I/O functions?
5. What is wrong with this program?

```
#include <fstream.h>
ifstream  fp;
void main()
{
   char in_char;
   fp.open(ios::in | ios::binary);
   if (fp.get(in_char))
   {  cout << in_char;  }   // Write to the screen
   fp.close();
   return;
}
```

Review Exercises



1. Write a program that asks the user for a list of five names, then writes the names to a file. Rewind the file and display its contents on-screen using the `seekg()` and `get()` functions.

2. Rewrite the program in Exercise 1 so it displays every other character in the file of names.

3. Write a program that reads characters from a file. If the input character is a lowercase letter, change it to uppercase. If the input character is an uppercase letter, change it to lowercase. Do not change other characters in the file.

4. Write a program that displays the number of nonalphabetic characters in a file.

5. Write a grade-keeping program for a teacher. Allow the teacher to enter up to 10 students' grades. Each student has three grades for the semester. Store the students' names and their three grades in an array of structures and store the data on the disk. Make the program menu-driven. Include options of adding more students, viewing the file's data, or printing the grades to the printer with a calculated class average.

Summary

C++ supports random-access files with several functions. These functions include error checking, file pointer positioning, and the opening and closing of files. You now have the tools you need to save your C++ program data to disk for storage and retrieval.

The mailing-list application in Appendix F offers a complete example of random-access file manipulation. The program enables the user to enter names and addresses, store them to disk, edit them, change them, and print them from the disk file. The mailing-list program combines almost every topic from this book into a complete application that "puts it all together."