Lecture#7

File Processing– Random Access Files

CENG 102- Algorithms and Programming II,

2024-2025, Spring

Contains materials from:
P. Deitel, H. Deitel, ‘’C How to Program with an Introduction to C++’’, 8th edition, Pearson

11.5 Random-Access Files

- Individual records of a random-access file are normally **fixed in length** and **may be accessed directly** (and thus quickly) without searching through other records.
- This makes random-access files appropriate for transaction-processing systems such as airline reservation systems, banking systems, and online shopping systems, and other system that require rapid access to specific data.

11.5 Random-Access Files (Cont.)

- There are other ways of implementing random-access files, but we will limit our discussion to this straightforward **approach using fixed-length records**.
- Because every record in a random-access file normally has the same pre-determined length, the exact location of a record relative to the beginning of the file can be calculated as a function of the record key.
- This facilitates ***immediate*** access to specific records, even in large files.

11.5 Random-Access Files (Cont.)

- Fixed-length records enable data to be inserted in a random-access file ***without destroying other data in the file***.
- Data stored previously can also be updated or deleted **without rewriting the entire file**.

11.6 Creating a Random-Access File

- Function `fwrite` transfers a specified number of bytes beginning at a specified location in memory to a file.
- The data is written beginning at the location in the file indicated by *the file position pointer*.

11.6 Creating a Random-Access File

- Function `fread` transfers a specified number of bytes from the location in the file specified by *the file position pointer* to an area in memory beginning with a specified address.

11.6 Creating a Random-Access File (Cont.)

- Now, when writing an integer, instead of using
  - `fprintf(fPtr, "%d", number);`

  which could print a single digit or as many as 11 digits (10 digits plus a sign, each of which requires 1 byte of storage) for a four-byte integer, we can use
  - `fwrite(&number, sizeof(int), 1, fPtr);`

  which *always* writes four bytes on a system with four-byte integers from a variable `number` to the file represented by `fPtr` (we'll explain the `1` argument shortly).

11.6 Creating a Random-Access File (Cont.)

- Later, `fread` can be used to read those four bytes into an integer variable `number`.
- Although `fread` and `fwrite` read and write data in fixed-size. The data they handle are processed in computer "raw data" format (i.e., bytes of data) rather than in `printf`'s and `scanf`'s human-readable text format.
- Because the "raw" representation of data is **system dependent**, "raw data" may not be readable on other systems, or by programs produced by other compilers or with other compiler options.

11.6 Creating a Random-Access File (Cont.)

- Functions `fread` and `fwrite` are **capable of reading and writing arrays** of data from and to disk.
- **The third argument** of both `fread` and `fwrite` is the number of elements in the array that should be read from or written to disk.

```
fread(&number, sizeof(int),1, fPtr);
fwrite(&number, sizeof(int),1, fPtr);
```

11.6 Creating a Random-Access File (Cont.)

```
fwrite(&number, sizeof(int),1, fPtr);
fread(&number, sizeof(int),1, fPtr);
```

- The `fwrite` function call writes a single integer to disk, so the third argument is `1` (as if one element of an array is being written).
- File-processing programs rarely write a single field to a file.
- Normally, they write one `struct` at a time, as we show in the following examples.

```
// C program for writing struct to file
#include <stdio.h>
#include <stdlib.h>
// a struct to be read and written
struct person {
		char fname[20];
		char lname[20];
};
int main() {
		FILE* outfile = fopen("person.txt", "wb");    // open file for writing in binary mode
		if (outfile == NULL) {
				fprintf(stderr, "\nError opened file\n");
				return(0);
		}
		struct person input1 = {"Linda", "Sharma" };
		int flag = 0;
		flag = fwrite(&input1, sizeof(struct person), 1,outfile);    // write struct to file and return the number of elements written
		if (flag)
				printf("Contents of the structure has been written to the file successfully");
		else
				printf("Error!");
		fclose(outfile);
		return 0;
}
```

```
// C program for reading struct from a file
#include <stdio.h>
#include <stdlib.h>
struct person {
        char fname[20];
        char lname[20];
};
int main() {
        FILE* infile
        infile = fopen("person1.txt", "wb+");   // Open person1.txt for writing and reading in binary mode
        if (infile == NULL) {
                fprintf(stderr, "\nError opening file\n");
                return(0);
        }
        struct person input1 = { "Linda", "Sharma" }; // writing to file
        fwrite(&input1, sizeof(input1), 1, infile);
        struct person output1;
        rewind(infile);                 // setting pointer to start of the file
        fread(&output1, sizeof(output1), 1, infile); // reading from file
        printf("%s %s", output1.fname, output1.lname);
        fclose(infile);
        return 0;
}
```

11.6 Creating a Random-Access File (Cont.)

- Consider the following problem statement:
  - Create a credit-processing system capable of storing up to 100 fixed-length records. Each record should consist of an **account number** that will be used as the **record key**, a **last name**, a **first name** and a **balance**. The resulting program should be able to insert a new account record, update an account, delete an account and list all the account records in a formatted text file for printing. Use a random-access file.
- The next several sections introduce the techniques necessary to create the credit-processing program.

11.7 Creating a Random-Access File (Cont.)

- Figure 11.10 shows how to open a random-access file, define a record format using a `struct`, write data to the disk and close the file.
- This program initializes all 100 records of the file `"credit.dat"` with empty `struct`s using the function `fwrite`.
- Each empty `struct` contains `0` for the account number, `""` (the empty string) for the last name, `""` for the first name and `0.0` for the balance.
- The file is initialized in this manner to create space on the disk in which the file will be stored and to make it possible to determine whether a record contains data.

```c
// Fig. 11.10: fig11_10.c
// Creating a random-access file sequentially
#include <stdio.h>

// clientData structure definition
struct clientData {
   unsigned int acctNum; // account number
   char lastName[15]; // account last name
   char firstName[10]; // account first name
   double balance; // account balance
};

int main(void)
{
   FILE *cfPtr; // accounts.dat file pointer

   // fopen opens the file; exits if file cannot be opened
   if ((cfPtr = fopen("accounts.dat", "wb")) == NULL) {
      puts("File could not be opened.");
   }
```

Fig. 11.10 | Creating a random-access file sequentially. (Part 1 of 2.)

```
21      else {
22         // create clientData with default information
23         struct clientData blankClient = {0, "", "", 0.0};
24
25         // output 100 blank records to file
26         for (unsigned int i = 1; i <= 100; ++i) {
27            fwrite(&blankClient, sizeof(struct clientData), 1, cfPtr);
28         }
29
30         fclose (cfPtr); // fclose closes the file
31      }
32   }
```

Fig. 11.10 | Creating a random-access file sequentially. (Part 2 of 2.)

11.6 Creating a Random-Access File (Cont.)

- *Note:* Figures 11.11, 11.14 and 11.15 use the data file created in Fig. 11.10, so you must run Fig. 11.10 before running Figs. 11.11, 11.14 and 11.15.

11.7 Writing Data Randomly to a Random-Access File

- Figure 11.11 writes data to the file `"credit.dat"`.
- It uses the combination of `fseek` and `fwrite` to store data at specific locations in the file.
- Function **`fseek` sets the file position pointer** to a specific position in the file, then `fwrite` writes the data.

```c
// Fig. 11.11: fig11_11.c
// Writing data randomly to a random-access file
#include <stdio.h>

// clientData structure definition
struct clientData {
   unsigned int acctNum; // account number
   char lastName[15]; // account last name
   char firstName[10]; // account first name
   double balance; // account balance
}; // end structure clientData

int main(void)
{
   FILE *cfPtr; // accounts.dat file pointer

   // fopen opens the file; exits if file cannot be opened
   if ((cfPtr = fopen("accounts.dat", "rb+")) == NULL) {
      puts("File could not be opened.");
   }
   else {
      // create clientData with default information
      struct clientData client = {0, "", "", 0.0};

```

Fig. 11.11 | Writing data randomly to a random-access file. (Part 1 of 3.)

```
25          // require user to specify account number
26          printf("%s", "Enter account number"
27             " (1 to 100, 0 to end input): ");
28          scanf("%d", &client.acctNum);
29
30          // user enters information, which is copied into file
31          while (client.acctNum != 0) {
32             // user enters last name, first name and balance
33             printf("%s", "\nEnter lastname, firstname, balance: ");
34
35             // set record lastName, firstName and balance value
36             fscanf(stdin, "%14s%9s%lf", client.lastName,
37                client.firstName, &client.balance);
38
39             // seek position in file to user-specified record
40             fseek(cfPtr, (client.acctNum - 1) *
41                sizeof(struct clientData), SEEK_SET);
42
43             // write user-specified information in file
44             fwrite(&client, sizeof(struct clientData), 1, cfPtr);
45
```

Fig. 11.11 | Writing data randomly to a random-access file. (Part 2 of 3.)

```
46            // enable user to input another account number
47            printf("%s", "\nEnter account number: ");
48            scanf("%d", &client.acctNum);
49         }
50
51         fclose(cfPtr); // fclose closes the file
52      }
53   }
```

Fig. 11.11 | Writing data randomly to a random-access file. (Part 3 of 3.)

```
Enter account number (1 to 100, 0 to end input): 37
Enter lastname, firstname, balance: Barker Doug 0.00
Enter account number: 29
Enter lastname, firstname, balance: Brown Nancy -24.54
Enter account number: 96
Enter lastname, firstname, balance: Stone Sam 34.98
Enter account number: 88
Enter lastname, firstname, balance: Smith Dave 258.34
Enter account number: 33
Enter lastname, firstname, balance: Dunn Stacey 314.33
Enter account number: 0
```

Fig. 11.12 | Sample execution of the program in Fig. 11.11.

11.7 Writing Data Randomly to a Random-Access File (Cont.)

- Lines 40–41 position the file position pointer for the file referenced by `cfPtr` to the byte location calculated by `(client.accountNum - 1) * sizeof(struct clientData)`.
- The value of this expression is called the offset or the displacement.
- Because the account number is between 1 and 100 but the byte positions in the file start with 0, 1 is subtracted from the account number when calculating the byte location of the record.
- Thus, for record 1, the file position pointer is set to byte 0 of the file.

11.7 Writing Data Randomly to a Random-Access File (Cont.)

- The symbolic constant `SEEK_SET` indicates that the file position pointer is positioned **relative to the beginning of the file by the amount of the offset**.
- The function prototype for `fseek` is
  - `int fseek(FILE *stream, long int offset, int whence);`

where `offset` is the number of bytes to seek from `whence` in the file pointed to by `stream`—a positive `offset` seeks forward and a negative one seeks backward.

- Argument `whence` can get the values `SEEK_SET`, `SEEK_CUR` or `SEEK_END` (all defined in `<stdio.h>`), which indicate the location from which the seek begins.

11.7 Writing Data Randomly to a Random-Access File (Cont.)

- `SEEK_SET` indicates that the seek starts at the *beginning* of the file;
- `SEEK_CUR` indicates that the seek starts at the *current location* in the file
- `SEEK_END` indicates that the seek starts at the *end* of the file.

- For simplicity, the programs in this chapter do not perform error checking.
- Industrial-strength programs should determine whether functions such as `fscanf`, `fseek` and `fwrite` operate correctly by checking their return values.

11.7 Writing Data Randomly to a Random-Access File (Cont.)

- Function `fseek` *returns a nonzero value if the seek operation cannot be performed*. (So, 0 refers to a successful operation)
- Function `fwrite` *returns the number of items it successfully write*.
- Function `fread` *returns the number of items it successfully read*.
- If these numbers are less than the *third argument* in the functions call, then a write error occurred.

11.8 Reading Data from a Random-Access File

- Function `fread` reads a specified number of bytes from a file into memory.
- For example,
  - `fread(&client, sizeof(struct clientData), 1, cfPtr);`

  reads the number of bytes determined by `sizeof(struct clientData)` from the file referenced by `cfPtr`, stores the data in `client`.
- The bytes are read from the location specified by the file position pointer.

11.8 Reading Data from a Random-Access File (Cont.)

- Function `fread` can read several fixed-size array elements by providing a pointer to the array in which the elements will be stored and by indicating the number of elements to be read.
- The statement

```
fread(&client, sizeof(struct clientData), 1, cfPtr);
```

  reads *one* element.
- To read *more than one*, specify the number of elements as `fread`'s *third argument*.

11.8 Reading Data from a Random-Access File (Cont.)

- Figure 11.14 reads sequentially every record in the `"credit.dat"` file, determines whether each record contains data and displays the formatted data for records containing data.
- Function `feof` determines when the end of the file is reached.

```c
// Fig. 11.14: fig11_14.c
// Reading a random-access file sequentially
#include <stdio.h>

// clientData structure definition
struct clientData {
   unsigned int acctNum; // account number
   char lastName[15]; // account last name
   char firstName[10]; // account first name
   double balance; // account balance
};

int main(void)
{
   FILE *cfPtr; // accounts.dat file pointer

   // fopen opens the file; exits if file cannot be opened
   if ((cfPtr = fopen("credit.txt", "rb")) == NULL) {
      puts("File could not be opened.");
   }
```

Fig. 11.14 | Reading a random-access file sequentially. (Part 1 of 3.)

```
21      else {
22         printf("%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
23            "First Name", "Balance");
24
25         // read all records from file (until eof)
26         while (!feof(cfPtr)) {
27            // create clientData with default information
28            struct clientData client = {0, "", "", 0.0};
29
30            int result = fread(&client, sizeof(struct clientData), 1, cfPtr);
31
32            // display record
33            if (result != 0 && client.acctNum != 0) {
34               printf("%-6d%-16s%-11s%10.2f\n",
35                  client.acctNum, client.lastName,
36                  client.firstName, client.balance);
37            }
38         }
39
40         fclose(cfPtr); // fclose closes the file
41      }
42   }
```

Fig. 11.14 | Reading a random-access file sequentially. (Part 2 of 3.)

```
Acct  Last Name       First Name    Balance
29    Brown           Nancy          -24.54
33    Dunn            Stacey         314.33
37    Barker          Doug             0.00
88    Smith           Dave           258.34
96    Stone           Sam             34.98
```

Fig. 11.14 | Reading a random-access file sequentially. (Part 3 of 3.)