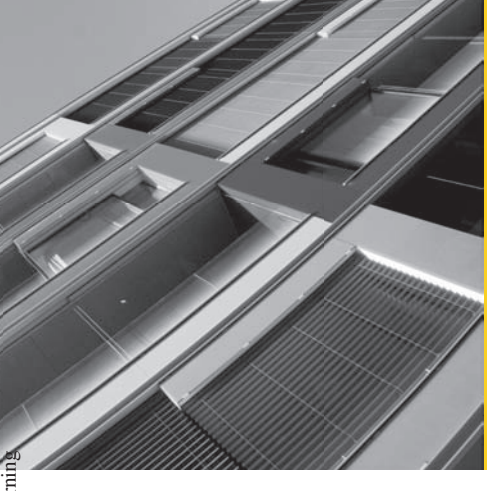


APPENDIX E
Additional C++ Topics

Binary (Base 2) Representation of a Nonnegative Integer

Converting a Base 10 Number to a Binary Number (Base 2)

Chapter 1 remarked that `A` is the 66th character in the ASCII character set, but its position is 65 because the position of the first character is 0. Furthermore, the binary number `1000001` is the binary representation of `65`. The number system that we use daily is called the **decimal number system or base 10 system**. The number system that the computer uses is called the **binary number system** or **base 2 system**. In this section, we describe how to find the binary representation of a nonnegative integer and vice versa.

Consider 65. Note that

$$65 = 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 + 2^1 + 1 \times 2^0$$

Similarly,

$$711 = 1 \times 2^9 + 0 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

In general, if m is a nonnegative integer, then m can be written as

$$m = a_k \times 2^k + a_{k-1} \times 2^{k-1} + a_{k-2} \times 2^{k-2} + \cdots + a_1 \times 2^1 + a_0 \times 2^0;$$

for some nonnegative integer k, and where $a_i = 0$ or 1, for each $i = 0, 1, 2, \ldots, k$. The binary number $a_k a_{k-1} a_{k-2} \cdots a_1 a_0$ is called the **binary** or **base 2 representation** of m. In this case, we usually write

$$m_{10} = (a_k a_{k-1} a_{k-2} \cdots a_1 a_0)_2$$

and say that m to the base 10 is $a_k a_{k-1} a_{k-2} \cdots a_1 a_0$ to the base 2.

For example, for the integer 65, $k = 6$, $a_6 = 1$, $a_5 = 0$, $a_4 = 0$, $a_3 = 0$, $a_2 = 0$, and $a_0 = 1$. Thus, $a_6 a_5 a_4 a_3 a_2 a_1 a_0 = 1000001$, so the binary representation of 65 is 1000001, that is,

$$65_{10} = (1000001)_2.$$

If no confusion arises, then we write $(1000001)_2$ as 1000001_2.

Similarly, for the number 711, $k = 9$, $a_9 = 1$, $a_8 = 0$, $a_7 = 1$, $a_6 = 1$, $a_5 = 0$, $a_4 = 0$, $a_3 = 0$, $a_2 = 1$, $a_1 = 1$, and $a_0 = 1$. Thus,

$711_{10} = 1011000111_2$.

It follows that to find the binary representation of a nonnegative integer, we need to find the coefficients, which are 0 or 1, of various powers of 2. However, there is an easy algorithm, described next, that can be used to find the binary representation of a nonnegative integer. First, note that

$0_{10} = 0_2$; $1_{10} = 1_2$; $2_{10} = 10_2$; $3_{10} = 11_2$; $4_{10} = 100_2$; $5_{10} = 101_2$; $6_{10} = 110_2$; and $7_{10} = 111_2$.

Let us consider the integer 65. Note that $65/2 = 32$ and $65 \% 2 = 1$, where % is the mod operator. Next, $32/2 = 16$, and $32 \% 2 = 0$, and so on. It can be shown that $a_0 = 6 \% 2 = 1$, $a_1 = 32 \% 2 = 0$, and so on. We can show this continuous division and obtaining the remainder with the help of Figure E-1.

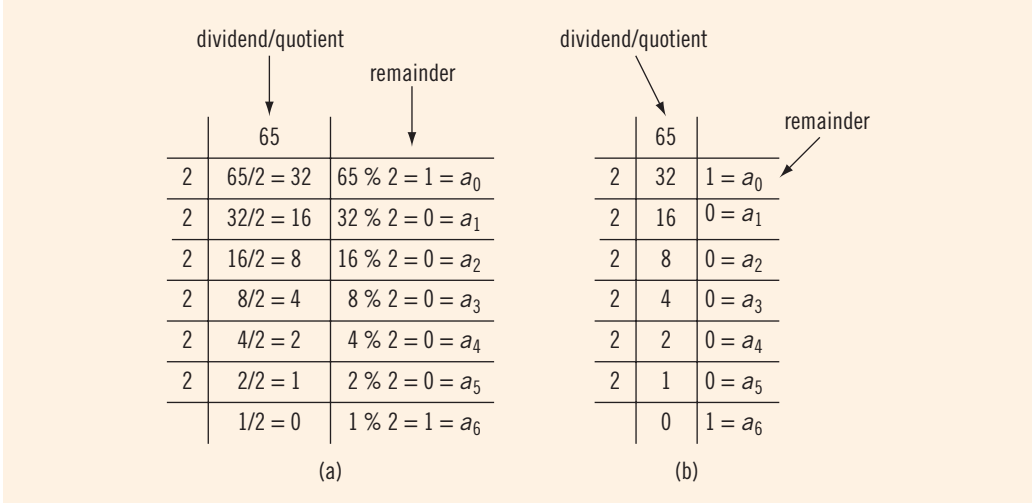


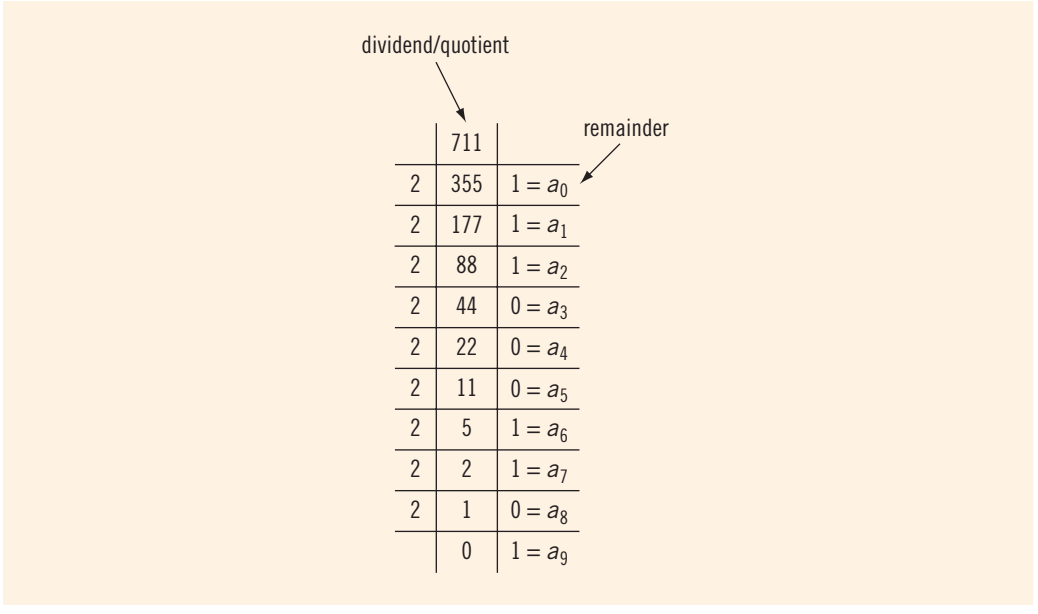
FIGURE E-1 Determining the binary representation of 65

Notice that in Figure E-1(a), starting at the second row, the second column contains the quotient when the number in the previous row is divided by 2 and the third column contains the remainder of that division. For example, in the second row, $65/2 = 32$ and $65 \% 2 = 1$. In the third row, $32/2 = 16$ and $32 \% 2 = 0$, and so on. For each row, the number in the second column is divided by 2, the quotient is written in the next row, below the current row, and the remainder is written in the third column. When

using a figure, such as E-1, to find the binary representation of a nonnegative integer, typically, we show only the quotients and remainders as in Figure E-1(b). You can write the binary representation of the number starting with the last remainder in the third column, followed by the second to last remainder, and so on. Thus,

$65_{10} = 1000001_2$.

Next, consider the number 711. Figure E-2 shows the quotients and the remainders.



	711	
2	355	$1 = a_0$
2	177	$1 = a_1$
2	88	$1 = a_2$
2	44	$0 = a_3$
2	22	$0 = a_4$
2	11	$0 = a_5$
2	5	$1 = a_6$
2	2	$1 = a_7$
2	1	$0 = a_8$
	0	$1 = a_9$

FIGURE E-2 Determining the binary representation of 711

From Figure E-2, it follows that

$711_{10} = 1011000111_2$.

Converting a Binary Number (Base 2) to Base 10

To convert a number from base 2 to base 10, we first find the weight of each bit in the binary number. The weight of each bit in the binary number is assigned from right to left. The weight of the rightmost bit is 0. The weight of the bit immediately to the left of the rightmost bit is 1, the weight of the bit immediately to the left of it is 2, and so on. Consider the binary number 1001101. The weight of each bit is as follows:

weight	6	5	4	3	2	1	0
	1	0	0	1	1	0	1

We use the weight of each bit to find the equivalent decimal number. For each bit, we multiply the bit by 2 to the power of its weight and then we add all of the numbers. For the above binary number, the equivalent decimal number is

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 64 + 0 + 0 + 8 + 4 + 0 + 1$$
$$= 77.$$

Converting a Binary Number (Base 2) to Octal (Base 8) and Hexadecimal (Base 16)

The previous sections described how to convert a binary number to a decimal number (base 2). Even though the language of a computer is binary, if the binary number is too long, then it will be hard to manipulate it manually. To effectively deal with binary numbers, two more number systems, octal (base 8) and hexadecimal (base 16), are of interest to computer scientists.

The digits in the octal number system are 0, 1, 2, 3, 4, 5, 6, and 7. The digits in the hexadecimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. So A in hexadecimal is 10 in decimal, B in hexadecimal is 11 in decimal, and so on.

The algorithm to convert a binary number into an equivalent number in octal (or hexadecimal) is quite simple. Before we describe the method to do so, let us review some notations. Suppose a_b represents the number a to the base b. For example, $2A0_{16}$ means 2A0 to the base 16, and 63_8 means 63 to the base 8.

First we describe how to convert a binary number into an equivalent octal number and vice versa. Table E-1 describes the first eight octal numbers.

TABLE E-1 Binary representation of first eight octal numbers

Binary	Octal	Binary	Octal
000	0	100	4
001	1	101	5
010	2	110	6
011	3	111	7

Consider the binary number 1101100010101. To find the equivalent octal number, starting from right to left we consider three digits at a time and write their octal representation. Note that the binary number 1101100010101 has only 13 digits. So

when we consider three digits at a time, at the end we will be left with only one digit. In this case, we just add two 0s to the left of the binary number; the equivalent binary number is 001101100010101. Thus,

$$\begin{aligned} 1101100010101_2 &= 001\ 101\ 100\ 010\ 101 \\ &= 15425_8 \text{ because } 001_2 = 1_8,\ 101_2 = 5_8,\ 100_2 = 4_8,\ 010_2 = 2_8, \\ &\qquad \text{and } 101_2 = 5_8 \end{aligned}$$

Thus, $1101100010101_2 = 15425_8$.

To convert an octal number into an equivalent binary number, using Table E-1, write the binary representation of each octal digit in the number. For example,

$$\begin{aligned} 3761_8 &= 011\ 111\ 110\ 001_2 \\ &= 011111110001_2 \\ &= 11111110001_2 \\ \text{Thus, } 3761_8 &= 11111110001_2. \end{aligned}$$

Next we discuss how to convert a binary number into an equivalent hexadecimal number and vice versa. The method to do so is similar to converting a number from binary to octal and vice versa, except that here we work with four binary digits. Table E-2 gives the binary representation of the first 16 hexadecimal numbers.

TABLE E-2 Binary representation of first 16 hexadecimal numbers

Binary	Hexadecimal	Binary	Hexadecimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Consider the binary number 1111101010001010101_2. Now,

$$\begin{aligned} 1111101010001010101_2 &= 111\ 1101\ 0100\ 0101\ 0101_2 \\ &= 0111\ 1101\ 0100\ 0101\ 0101_2, \text{ add one zero to the left} \\ &= 7D455_{16} \end{aligned}$$

Hence, $1111101010001010101_2 = 7D455_{16}$

Next, to convert a hexadecimal number into an equivalent binary number, write the four-digit binary representation of each hexadecimal digit into that number. For example,

$$\begin{aligned} A7F32_{16} &= 1010\ 0111\ 1111\ 0011\ 0010_2 \\ &= 10100111111100110010_2 \\ \text{Thus, } A7F32_{16} &= 10100111111100110010_2. \end{aligned}$$

More on File Input/Output

In Chapter 3, you learned how to read data from and write data to a file. This section expands on the concepts introduced in that chapter.

Binary Files

In Chapter 3, you learned how to make a program read data from and write data to a file. However, the files that the programs have used until now are called text files. Data in a text file is stored in the character format. For example, consider the number `45`. If `45` is stored in a file, then it is stored as a sequence of two characters—the character `'4'` followed by the character `'5'`. The eight-bit machine representation of `'4'` is `00000100` and the eight-bit machine representation of `'5'` is `00000101`. Therefore, in a text file, `45` is stored as `0000010000000101`. When this number is read by a C++ program, it must first be converted to its binary format. Suppose that the integers are represented as 16-bit binary numbers. The 16-bit binary representation of `45` is then `0000000000101101`. Similarly, when a program stores the number `45` in a text file, it must first be converted to its text format. It thus follows that reading data from and writing data to a text file is not efficient, because the data must be converted from the text to the binary format and vice versa.

On the other hand, when data is stored in a file in the binary format, reading and writing data is faster because no time is lost in converting the data from one format to another format. Such files are called binary files. More formally, **binary files** are files in which data is stored in the binary format. Data in a text file is also called **formatted data**, and data in a binary file is called **raw data**.

C++ allows a programmer to create binary files. This section explains how to create binary files and also how to read data from binary files.

To create a binary file, the file must be opened in the binary mode. Suppose **outFile** is an **ofstream** variable (object). Consider the following statement:

```
outFile.open("employee.dat", ios::binary);
```

This statement opens the file **employee.dat**. Data in this file will be written in its binary format. Therefore, the file opening mode **ios::binary** specifies that the file is opened in the binary mode.

Next, you use the stream function **write** to write data to the file **employee.dat**. The syntax to use the function **write** is

```
fileVariableName.write(reinterpret_cast<const char *> (buffer),
                       sizeof(buffer));
```

where **fileVariableName** is the object used to open the output file, and the first argument **buffer** specifies the starting address of the location in memory where the data is stored. The expression **sizeof(buffer)** specifies the size of the data, in bytes, to be written.

For example, suppose **num** is an **int** variable. The following statement writes the value of **num** in the binary format to the file associated with **outFile**:

```
outFile.write(reinterpret_cast<const char *> (&num), sizeof(num));
```

Similarly, suppose **empSalary** is an array of, say, 100 components and the component type is **double**. The following statement writes the entire array to the file associated with **outFile**:

```
outFile.write(reinterpret_cast<const char *> (empSalary),
              sizeof(empSalary));
```

Next, let us discuss how to read data from a binary file. The operation of reading data from a binary file is similar to writing data to a binary file. First, the binary file must be opened. For example, suppose **inFile** is an **ifstream** variable, and a program has already created the binary file **employee.dat**. The following statement opens this file:

```
inFile.open("employee.dat");
```

or:

```
inFile.open("employee.dat", ios::binary);
```

To read data in the binary format, the stream function **read** is used. The syntax to use the function **read** is

```
fileVariableName.read(reinterpret_cast<char *> (buffer),
                      sizeof(buffer));
```

The first argument `buffer` specifies the starting address of the location in memory where the data is to be stored. The expression `sizeof(buffer)` specifies the size of the data, in bytes, to be read.

The program in the following example further explains how to create binary files and read data from a binary file.

EXAMPLE E-1

```
//Creating and reading binary files
#include <iostream>                                          //Line 1
#include <fstream>                                           //Line 2

using namespace std;                                         //Line 3

struct studentType                                           //Line 4
{                                                            //Line 5
    char firstName[15];                                      //Line 6
    char lastName[15];                                       //Line 7
    int ID;                                                  //Line 8
};                                                           //Line 9

int main()                                                   //Line 10
{                                                            //Line 11

        //create and initialize an array of students' IDs
    int studentIDs[5] = {111111, 222222, 333333,
                         444444, 555555};                    //Line 12

          //declare and initialize the struct newStudent
    studentType newStudent = {"John", "Wilson",
                              777777};                       //Line 13

    ofstream outFile;                                        //Line 14

          //open the output file as a binary file
    outFile.open("ids.dat", ios::binary);                    //Line 15

          //write the array in the binary format
    outFile.write(reinterpret_cast<const char *> (studentIDs),
                  sizeof(studentIDs));                       //Line 16
          //write the newStudent data in the binary format
    outFile.write(reinterpret_cast<const char *> (&newStudent),
                  sizeof(newStudent));                       //Line 17

    outFile.close();  //close the file                     //Line 18

    ifstream inFile;                                         //Line 19
    int arrayID[5];                                          //Line 20
    studentType student;                                     //Line 21
```

```
     //open the input file
    inFile.open("ids.dat");                                    //Line 22

    if (!inFile)                                               //Line 23
    {                                                          //Line 24
        cout << "The input file does not exist. "
             << "The program terminates!!" << endl;            //Line 25
        return 1;                                              //Line 26
    }                                                          //Line 27

        //input the data into the array arrayID
    inFile.read(reinterpret_cast<char *> (arrayID),
                sizeof(arrayID));                              //Line 28
        //output the data of the array arrayID
    for (int i = 0; i < 5; i++)                                //Line 29
        cout << arrayID[i] << " ";                             //Line 30
    cout << endl;                                              //Line 31

        //read the student's data
    inFile.read(reinterpret_cast<char *> (&student),
                sizeof(student));                              //Line 32

        //output studentData
    cout << student.ID << " " << student.firstName
         << " " << student.lastName << endl;                   //Line 33

    inFile.close();   //close the file                        //Line 34

    return 0;                                                  //Line 35
}                                                              //Line 36
```

Sample Run:

```
111111 222222 333333 444444 555555
777777 John Wilson
```

The output of the preceding program is self-explanatory. The details are left as an exercise for you.

NOTE In the program in Example E-1, the statement in Line 2 declares the `struct` variable **`newStudent`** and also initializes it. Because **`newStudent`** has three components and we want to initialize all the components, three values are specified in braces separated by commas. In other words, `struct` variables can also be initialized when they are declared.

The program in the following example further explains how to create binary files and then read the data from the binary files.

EXAMPLE E-2

```
//Creating and reading a binary file consisting of
//bank customers' data

#include <iostream>                                          //Line 1
#include <fstream>                                           //Line 2
#include <iomanip>                                           //Line 3

using namespace std;                                         //Line 4

struct customerType                                          //Line 5
{                                                            //Line 6
    char firstName[15];                                      //Line 7
    char lastName[15];                                       //Line 8
    int ID;                                                  //Line 9
    double balance;                                          //Line 10
};                                                           //Line 11

int main()                                                   //Line 12
{                                                            //Line 13
    customerType cust;                                       //Line 14
    ifstream inFile;                                         //Line 15
    ofstream outFile;                                        //Line 16

    inFile.open("customerData.txt");                         //Line 17

    if (!inFile)                                             //Line 18
    {                                                        //Line 19
        cout << "The input file does not exist. "
             << "The program terminates!!" << endl;          //Line 20
        return 1;                                            //Line 21
    }                                                        //Line 22

    outFile.open("customer.dat", ios::binary);               //Line 23

    inFile >> cust.ID >> cust.firstName >> cust.lastName
           >> cust.balance;                                  //Line 24

    while (inFile)                                           //Line 25
    {                                                        //Line 26
        outFile.write(reinterpret_cast<const char *> (&cust),
                      sizeof(cust));                         //Line 27
        inFile >> cust.ID >> cust.firstName
               >> cust.lastName >> cust.balance;             //Line 28
    }                                                        //Line 29

    inFile.close();                                          //Line 30
    inFile.clear();                                          //Line 31
    outFile.close();                                         //Line 32
```

```
    inFile.open("customer.dat", ios::binary);          //Line 33

    if (!inFile)                                       //Line 34
    {                                                  //Line 35
        cout << "The input file does not exist. "
             << "The program terminates!!" << endl;    //Line 36
        return 1;                                      //Line 37
    }                                                  //Line 38

    cout << left << setw(8) << "ID"
         << setw(16) << "First Name"
         << setw(16) << "Last Name"
         << setw(10) << " Balance" << endl;            //Line 39
    cout << fixed << showpoint << setprecision(2);     //Line 40

         //read and output the data from the binary
         //file customer.dat
    inFile.read(reinterpret_cast<char *> (&cust),
                sizeof(cust));                         //Line 41
    while (inFile)                                     //Line 42
    {                                                  //Line 43
         cout << left << setw(8) << cust.ID
              << setw(16) << cust.firstName
              << setw(16) << cust.lastName
              << right << setw(10) << cust.balance
              << endl;                                 //Line 44
        inFile.read(reinterpret_cast<char *> (&cust),
                    sizeof(cust));                     //Line 45
    }                                                  //Line 46

    inFile.close();     //close the file               //Line 47

    return 0;                                          //Line 48
}                                                      //Line 49
```

Sample Run:

```
ID      First Name      Last Name        Balance
77234   Ashley          White              4563.50
12345   Brad            Smith            128923.45
87123   Lisa            Johnson            2345.93
81234   Sheila          Robinson            674.00
11111   Rita            Gupta             14863.50
23422   Ajay            Kumar             72682.90
22222   Jose            Ramey             25345.35
54234   Sheila          Duffy             65222.00
55555   Tommy           Pitts               892.85
23452   Salma           Quade              2812.90
32657   Jennifer        Ackerman           9823.89
82722   Steve           Sharma            78932.00
```

Random File Access

In Chapter 3 and the preceding section, you learned how to read data from and write data to a file. More specifically, you used **ifstream** objects to read data from a file and **ofstream** objects to write data to a file. However, the files were read and/or written sequentially. Reading data from a file sequentially does not work very well for a variety of applications. For example, consider a program that processes customers' data in a bank. Typically, there are thousands or even millions of customers in a bank. Suppose we want to access a customer's data from the file that contains such data, say, for an account update. If the data is accessed sequentially, starting from the first position and read until the desired customer's data is found, this process might be extremely time consuming. Similarly, in an airline's reservation system to access a passenger's reservation information sequentially, this might also be very time consuming. In such cases, the data retrieval must be efficient. A convenient way to do this is to be able to read the data randomly from a file, that is, randomly access any record in the file.

In the preceding section, you learned how to use the stream function **read** to read a specific number of bytes, and the function **write** to write a specific number of bytes.

The stream function **seekg** is used to move the read position to any byte in the file. The general syntax to use the function **seekg** is

```
fileVariableName.seekg(offset, position);
```

The stream function **seekp** is used to move the write position to any byte in the file. The general syntax to use the function **seekp** is

```
fileVariableName.seekp(offset, position);
```

The **offset** specifies the number of bytes the reading/writing positions are to be moved, and **position** specifies where to begin the offset. The offset can be calculated from the beginning of the file, end of the file, or the current position in the file. Moreover, **offset** is a long integer representation of an offset. Table E-3 shows the values that can be used for **position**.

TABLE E-3 Values of **position**

position	Description
ios::beg	The offset is calculated from the beginning of the file.
ios::cur	The offset is calculated from the current position of the reading marker in the file.
ios::end	The offset is calculated from the end of the file.

EXAMPLE E-3

Suppose you have the following line of text stored in a file, say,

```
digitsAndLetters.txt:
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Also, suppose that `inFile` is an `ifstream` object and the file `digitsAndLetters.txt` has been opened using the object `inFile`. One byte is used to store each character of this line of text. Moreover, the position of the first character is `0`.

Statement	Explanation
`inFile.seekp(10L, ios::beg);`	Sets the reading position of `inFile` to the 11th byte (character), which is at position `10`. That is, it sets the reading position just after the digit `9` or just before the letter `A`.
`inFile.seekp(5L, ios::cur);`	Moves the reading position of `inFile` five bytes to the right of its current position.
`inFile.seekp(-6L, ios::end);`	Sets the reading position of `inFile` to the sixth byte (character) from the end. That is, it sets the reading position just before the letter `U`.

The program in the following example further explains how the functions `seekg` and `seekp` work.

EXAMPLE E-4

```
#include <iostream>                                          //Line 1
#include <fstream>                                           //Line 2

using namespace std;                                         //Line 3

int main()                                                   //Line 4
{                                                            //Line 5
    char ch;                                                 //Line 6
    ifstream inFile;                                         //Line 7

    inFile.open("digitsAndAlphabet.txt");                    //Line 8

    if (!inFile)                                             //Line 9
    {                                                        //Line 10
       cout << "The input file does not exist. "
            << "The program terminates!!" << endl;           //Line 11
       return 1;                                             //Line 12
    }                                                        //Line 13
```

```
    inFile.get(ch);                                    //Line 14
    cout << "Line 15: The first byte: " << ch
         << endl;                                      //Line 15

      //position the reading marker six bytes to the
      //right of its current position
    inFile.seekg(6L, ios::cur);                        //Line 16
    inFile.get(ch);  //read the character             //Line 17
    cout << "Line 18: Current byte read: " << ch
         << endl;                                      //Line 18

      //position the reading marker seven bytes
      //from the beginning
    inFile.seekg(7L, ios::beg);                        //Line 19
    inFile.get(ch);  //read the character             //Line 20
    cout << "Line 21: Seventh byte from the "
         << "beginning: " << ch << endl;               //Line 21

      //position the reading marker 26 bytes
      //from the end
    inFile.seekg(-26L, ios::end);                      //Line 22
    inFile.get(ch);    //read the character           //Line 23
    cout << "Line 24: Byte 26 from the end: " << ch
         <<  endl;                                     //Line 24

    return 0;                                          //Line 25
}                                                      //Line 26
```

Sample Run:

```
Line 15: The first byte: 0
Line 18: Current byte read: 7
Line 21: Seventh byte from the beginning: 7
Line 24: Byte 26 from the end: A
```

The input file contains the following line of text:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The following program illustrates how the function **seekg** works with **struct**s.

EXAMPLE E-5

Suppose **customerType** is a **struct** defined as follows:

```
struct customerType
{
    char firstName[15];
    char lastName[15];
    int ID;
    double balance;
};
```

The program in Example E-2 created the binary file **customer.dat** consisting of certain customers' data. You can use the function **seekg** to move the reading position of this file to any record. Suppose **inFile** is an **ifstream** object used to open the binary file **customer.dat**.

The following statement calculates the size of a **customerType** **struct** and stores it in the variable **custSize**:

```
long custSize = sizeof(cust);
```

We can use the value of the variable **custSize** to move the reading position to a specific record in the file. For example, consider the following statement:

```
inFile.seekg(6 * custSize, ios::beg);
```

This statement moves the reading position just after the sixth customer's record, that is, just before the seventh customer's record.

The following program further illustrates how the function **seekg** works with **struct**s.

EXAMPLE E-6

```
//Reading a file randomly

#include <iostream>                                          //Line 1
#include <fstream>                                           //Line 2
#include <iomanip>                                           //Line 3

using namespace std;                                         //Line 4

struct customerType                                          //Line 5
{                                                            //Line 6
    char firstName[15];                                      //Line 7
    char lastName[15];                                       //Line 8
    int ID;                                                  //Line 9
    double balance;                                          //Line 10
};                                                           //Line 11

void printCustData(const customerType& customer);            //Line 12

int main()                                                   //Line 13
{                                                            //Line 14
    customerType cust;                                       //Line 15
    ifstream inFile;                                         //Line 16

    long custSize = sizeof(cust);                            //Line 17

    inFile.open("customer.dat", ios::binary);                //Line 18
    if (!inFile)                                             //Line 19
```

```
    {                                                          //Line 20
        cout << "The input file does not exist. "
             << "The program terminates!!" << endl;            //Line 21
        return 1;                                              //Line 22
    }                                                          //Line 23

    cout << fixed << showpoint << setprecision(2);             //Line 24

      //randomly read the records and output them
    inFile.seekg(6 * custSize, ios::beg);                      //Line 25
    inFile.read(reinterpret_cast<char *> (&cust),
                sizeof(cust));                                 //Line 26
    cout << "Seventh customer's data: " << endl;               //Line 27
    printCustData(cust);                                       //Line 28

    inFile.seekg(8 * custSize, ios::beg);                      //Line 29
    inFile.read(reinterpret_cast<char *> (&cust),
                sizeof(cust));                                 //Line 30
    cout << "Ninth customer's data: " << endl;                 //Line 31
    printCustData(cust);                                       //Line 32

    inFile.seekg(-8 * custSize, ios::end);                     //Line 33
    inFile.read(reinterpret_cast<char *> (&cust),
                sizeof(cust));                                 //Line 34
    cout << "Eighth (from the end) customer's "
         << "data: " << endl;                                  //Line 35
    printCustData(cust);                                       //Line 36

    inFile.close();     //close the file                       //Line 37

    return 0;                                                  //Line 38
}                                                              //Line 39

void printCustData(const customerType& customer)              //Line 40
{                                                              //Line 41
    cout << "  ID: " << customer.ID <<endl
         << "  First Name: " << customer.firstName << endl
         << "  Last Name: " << customer.lastName << endl
         << "  Account Balance: $" << customer.balance
         << endl;                                              //Line 42
}                                                              //Line 43
```

Sample Run:

```
Seventh customer's data:
  ID: 22222
  First Name: Jose
  Last Name: Ramey
  Account Balance: $25345.35
Ninth customer's data:
  ID: 55555
  First Name: Tommy
  Last Name: Pitts
  Account Balance: $892.85
```

```
Eighth (from the end) customer's data:
  ID: 11111
  First Name: Rita
  Last Name: Gupta
  Account Balance: $14863.50
```

The program in Example E-6 illustrates how the function **seekg** works. Using the function **seekg**, the reading position in a file can be moved to any location in the file. Similarly, the function **seekp** can be used to move the write position in a file to any location. Furthermore, these functions can be used to create a binary file in which the data is organized according to the values of either a variable or a particular component of a `struct`. For example, suppose there are at most, say, 100 students in a class. Each student has a unique ID in the range 1 to 100. Using the students' IDs, we can create a random access binary file in such a way that in the file, a student's data is written at the location specified by its ID. This is like treating the file as an array. The advantage is that, once the file is created, a student's data from the file can be read, directly, using the student's ID. Another advantage is that in the file, the data is sorted according to the IDs.

Here, we are assuming that the student IDs are in the range 1 to 100. However, if you use, say, a three-, four-, or five-digit number as a student ID and there are only a few students in the class, the data in the file could be scattered. In other words, a lot of space could be used just to store only a few students' data. In such cases, more advanced techniques are used to organize the data so that it can be accessed efficiently.

The program in Example E-7 illustrates how to use the students' IDs to organize the data in a binary file. The program also shows how to output the file.

EXAMPLE E-7

```
//Creating and reading a random access file

#include <iostream>                                          //Line 1
#include <fstream>                                           //Line 2
#include <iomanip>                                           //Line 3

using namespace std;                                         //Line 4

struct studentType                                           //Line 5
{                                                            //Line 6
    char firstName[15];                                      //Line 7
    char lastName[15];                                       //Line 8
    int ID;                                                  //Line 9
    double GPA;                                              //Line 10
};                                                           //Line 11

void printStudentData(const studentType& student);           //Line 12
```

```
int main()                                                    //Line 13
{                                                             //Line 14
    studentType st;                                           //Line 15
    ifstream inFile;                                          //Line 16
    ofstream outFile;                                         //Line 17

    long studentSize = sizeof(st);                            //Line 18

        //open the input file, which is a text file
    inFile.open("studentData.txt");                           //Line 19

    if (!inFile)                                              //Line 20
    {                                                         //Line 21
        cout << "The input file does not exist. "
             << "The program terminates!!" << endl;           //Line 22
       return 1;                                              //Line 23
    }                                                         //Line 24

       //open a binary output file
    outFile.open("student.dat", ios::binary);                 //Line 25

    inFile >> st.ID >> st.firstName
           >> st.lastName >> st.GPA;                          //Line 26

    while (inFile)                                            //Line 27
    {                                                         //Line 28
        outFile.seekp((st.ID - 1) * studentSize,
                      ios::beg);                              //Line 29
        outFile.write(reinterpret_cast<const char *> (&st),
                      sizeof(st));                            //Line 30
        inFile >> st.ID >> st.firstName
               >> st.lastName >> st.GPA;                      //Line 31
    }                                                         //Line 32

    inFile.close();                                           //Line 33
    inFile.clear();                                           //Line 34
    outFile.close();                                          //Line 35

    cout << left << setw(3) << "ID"
         << setw(16) << "First Name"
         << setw(16) << "Last Name"
         << setw(12) << "Current GPA" << endl;                //Line 36
    cout << fixed << showpoint << setprecision(2);            //Line 37

         //open the input file, which is a binary file
    inFile.open("student.dat", ios::binary);                  //Line 38

    if (!inFile)                                              //Line 39
    {                                                         //Line 40
       cout << "The input file does not exist. "
            << "The program terminates!!" << endl;            //Line 41
       return 1;                                              //Line 42
    }                                                         //Line 43
```

```
        //read the data at location 0 in the file
    inFile.read(reinterpret_cast<char *> (&st),
                sizeof(st));                                    //Line 44

    while (inFile)                                              //Line 45
    {                                                           //Line 46
        if (st.ID != 0)                                         //Line 47
            printStudentData(st);                               //Line 48

            //read the data at the current reading position
        inFile.read(reinterpret_cast<char *> (&st),
                    sizeof(st));                                //Line 49
    }                                                           //Line 50

    return 0;                                                   //Line 51
}                                                               //Line 52

void printStudentData(const studentType& student)               //Line 53
{
    cout << left << setw(3) << student.ID
         << setw(16) << student.firstName
         << setw(16) << student.lastName
         << right << setw(10) << student.GPA
         << endl;                                               //Line 54
}                                                               //Line 55
```

Sample Run:

```
ID First Name      Last Name       Current GPA
2  Sheila          Duffy                 4.00
10 Ajay            Kumar                 3.60
12 Ashley          White                 3.90
16 Tommy           Pitts                 2.40
23 Rita            Gupta                 3.40
34 Brad            Smith                 3.50
36 Salma           Quade                 3.90
41 Steve           Sharma                3.50
45 Sheila          Robinson              2.50
56 Lisa            Johnson               2.90
67 Jose            Ramey                 3.80
75 Jennifer        Ackerman              4.00
```

The data in the file `studentData.txt` is as follows:

```
12 Ashley White 3.9
34 Brad Smith 3.5
56 Lisa Johnson 2.9
45 Sheila Robinson 2.5
23 Rita Gupta 3.4
10 Ajay Kumar 3.6
67 Jose Ramey 3.8
2 Sheila Duffy 4.0
16 Tommy Pitts 2.4
```

```
36 Salma Quade 3.9
75 Jennifer Ackerman 4.0
41 Steve Sharma 3.5
```

Naming Conventions of Header Files in ANSI/ISO Standard C++ and Standard C++

The programs in this book are written using ANSI/ISO Standard C++. However, there are two versions of C++—ANSI/ISO Standard C++ and Standard C++. For the most part, these two standards are the same. The header files in Standard C++ have the extension `.h`, while the header files in ANSI/ISO Standard C++ have no extension. Moreover, the names of certain header files, such as `math.h`, in ANSI/ISO Standard C++ start with the letter `c`. The language C++ evolved from C. Therefore, certain header files such as `math.h`, `stdlib.h`, and `string.h` were brought from C into C++. The header files such as `iostream.h`, `iomanip.h`, and `fstream.h` were specially designed for C++. Recall that when a header file is included in a program, the global identifiers of the header file also become the global identifiers of the program. In ANSI/ISO Standard C++, to take advantage of the `namespace` mechanism, all of the header files were modified so that the identifiers are declared within a `namespace`. Recall that the name of this `namespace` is std.

In ANSI/ISO Standard C++, the extension `.h` of the header files that were specially designed for C++ was dropped. For the header files that were brought from C into C++, the extension `.h` was dropped and the names of these header files start with the letter `c`. Following are the names of the most commonly used header files in Standard C++ and ANSI/ISO Standard C++:

Standard C++ Header File Name	ANSI/ISO Standard C++ Header File Name
`assert.h`	`cassert`
`ctype.h`	`cctype`
`float.h`	`cfloat`
`fstream.h`	`fstream`
`iomanip.h`	`iomanip`
`iostream.h`	`iostream`
`limits.h`	`climits`
`math.h`	`cmath`
`stdlib.h`	`cstdlib`
`string.h`	`cstring`

To include a header file, say, **iostream**, the following statement is required:

```
#include <iostream>
```

Furthermore, to use identifiers, such as **cin**, **cout**, **endl**, and so on, the program should use either the statement

```
using namespace std;
```

or the prefix **std::** before the identifier.