**Name : Omkar Nikhal**

**Class : SE-3**

**Roll no : 21342**

**ASSIGNMENT NO.** 10

**TITLE** Sequential file to maintain the data.

**PROBLEM STATEMENT** Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular student. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to maintain the data.

**OBJECTIVE** 1. Understand how sequential files are organized.

2. Able to add, delete and update records in an ordered Sequential file using C++

3. Understand stream concepts and C++ classes hierarchy.

**OUTCOME** At the end of this assignment students will able to design and implement program which store student records in Sequential file and also able to perform add, delete operation successfully and system will be displays student records.

**S/W PACKAGES AND** 1. (64-bit)64-BIT Fedora 17 or latest 64-BIT Update of

**HARDWARE**  Equivalent Open source OS

**APPARATUS USED 2.** Programming Tools (64-Bit) Latest Open source update of

Eclipse Programming frame work, TC++, GTK

**CONCEPTS RELATED THEORY:**

**Sequential file:**

Up till now we have implemented console based programs. So when a Program is terminated, the Program data stored in main memory is lost. For Small Program like calculator, we don't need to store data permanently. But what's about some programs, where we need to retrieve result of Students, or calculate the Monthly salary of employee. Whenever we create a Program, and enter any data to process, after the program is terminated, the data stored in the memory will lost. So to store our data Permanent, C++ provide File Handling concept.

This File Handling provides a mechanism to store output of a program in a file and read from a file on the disk. So far, we have been using ostream Header file which provide function cin / cout to take input from console and write output to a console respectively.

**File Organization**

File organization refers to the way records are physically arranged on a storage device.

There are two main types of file organization.

1. Sequential file organization
2. Random file organization

In sequential file organization, records are arranged sequentially. A sequential file is a file whose records can be accessed on the order of their appearance in the file. The order in which the records are stored is determined by the order in which they are written when the file was prepared. This order does not charge. Records may be added at the end of file only. The records may be accessed in the order on which they were originally written into a file.

**Advantages of sequential file organization**

1. Simple to implement
2. Requires very low software support
3. A substantial amount of storage space on the disk can be saved.

**Disadvantages of sequential file organization**

1. Updates are not easily accommodated
2. Random access is not possible
3. All records must be structurally identical, if a new field has to be added, and then every record must be rewritten to provide space for the new field.
4. Continuous areas may not be possible because both the primary data file and the transaction file must be looked during merging.

**Applications**

Sequential files are most frequently used in commercial batch-oriented data processing applications where there is the concept of a master file to which details are added periodically. Example: - Payroll applications

**Implementation:**

**Input/output with files**

C++ provides the following classes to perform output and input of characters to/from files:

* ofstream: Stream class to write on files
* ifstream: Stream class to read from files
* fstream: Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream and ostream.

Write to a file

#include <iostream>

#include <fstream>

using namespace std;

int main ()

{

ofstream myfile;

myfile.open ("example.txt");

myfile << "Writing this to a file.\n";

myfile.close();

return 0;

}

**Open a file**

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to open a file.

An open file is represented within a program by a stream object (an instantiation of one of these classes, in the previous example this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it. In order to open a file with a stream object we use its member

**function open():**

**open (filename, mode);**

Where filename is a null-terminated character sequence of type const char \* (the same type that string literals have) representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

* ios::in -- Open for input operations.
* ios::out -- Open for output operations.
* ios::binary -- Open in binary mode.
* ios::ate -- Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the eginning of the file.
* ios::app -- All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
* ios::trunc -- If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one. All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open():

ofstream myfile;

myfile.open ("example.bin", ios::out | ios::app | ios::binary);

Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

Class default mode parameter

ofstream -- ios::out

ifstream -- ios::in

fstream -- ios::in | ios::out

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open() member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the open() member function and has the exact same parameters as this member.

**Closing a file**

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function close (). This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

myfile.close();

**get and put stream pointers:**

All i/o streams objects have, at least, one internal stream pointer:

ifstream, like istream, has a pointer known as the get pointer that points to the element to be read in the next input operation.

ofstream, like ostream, has a pointer known as the put pointer that points to the location where the next element has to be written.

Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

tellg() and tellp()

These two member functions have no parameters and return a value of the member type pos\_type, which is an integer data type representing the current position of the get stream pointer(in the case of tellg) or the put stream pointer (in the case of tellp).

seekg() and seekp()

These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype is:

seekg ( position );

seekp ( position );

Using this prototype the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is the same as the one returned by functions tellg and tellp: the member type pos\_type, which is an integer value.

The other prototype for these functions is:

seekg ( offset, direction );

seekp ( offset, direction );

**WRITING AND READING OBJECTS TO AND FROM THE FILE:**

**Step 1**: Create the stream of desire type so for reading and writing both create object of fstream. For reading from file create instance of ifstream and for writing to file create stream instance of ofstream type

fstream fio;

ifstream fin;

**Step 2**: Attach the desired file to the declared stream

After declaring streams, the next step is to link the file with the declared stream. This is done by opening the desired file.

For example, if a file named Student.dat is to be opened in input mode i.e., linked to a stream of ifstream type, we can do it in the following two ways :

**1st Way:**

ifstream fin("Student.dat", ios::in) ; //using constructor

**2nd Way:**

fin.open("Student.dat", ios::in) ; //using open()

To open a file, say Student.txt, in I/O mode (stream of fstream) type, we may write as

fstream fio("Student.txt", ios::in | ios::out) ; //using constructor

Or as fstream fio:

fio.open("Student.txt", ios::in | ios::out) ; //using open

**Step 3**: Check whether file is opened or not. If error occurs display the appropriate messeage and exit.

if(!fio)

{

cout<<"Error occurred in opening the file..!!\n";

cout<<"Press any key to exit...\n";

getch();

exit(1);

}

**Step 4**: Write and read the Student class objects to the file and from the file

Use functions read () and write () for reading and writing class objects. These functions handle the entire structure of an object as a single unit, using the computer's internal representation of data. For example, the function write () copies a class object from memory byte by byte with no conversion. But one thing that must be remembered is that only data members are written to the disk file and not the member functions.

The length of an object is obtained by sizeof operator and it represents the sum total of lengths of all data members of the object. Here is an example:

Student st[noOfStudents];

Student outst[ noOfStudents];

for(int i=0; i<noOfStudents; i++)

{

st[i].getdata();

fio.write((char \*)&st[i], sizeof(st[i]));

}

fio.seekg(0); /\* seekg(0) resets the file to start, to access the file from the beginning \*/

for(i=0; i<noOfStudents; i++)

{

fio.read((char \*)&outst[i], sizeof(outst[i]));

outst[i].display();

}

**Step 5**: Close the stream by calling close method.

fio.close();

Test-Cases

|  |  |  |  |
| --- | --- | --- | --- |
| Desciption | Input | Output | Result |
| User input and write | Tyrion Lann  26  A  Kings Landing  -------  Jon Snow  31  B  Winterfell  -------  Arya Stark  10  A  Winterfell | Student data added to the file successfully | Pass |
| Read from the file | - | 26 A Tyrion Lann Kings Landing  31 B Jon Snow Winterfell  10 A Arya Stark Winterfell | Pass |
| Delete | 10 | Arya Stark deleted fromo database | pass |
| Read from the file | - | 26 A Tyrion Lann Kings Landing  31 B Jon Snow Winterfell | Pass |
| If record exists | 31 | Student found with roll number 31  Name : Jon Snow  Div : B  Addr : Winterfell | Passss |
| If record exists | 10 | Student with roll 10 not found in database | Pass |

**CONCLUSION**: Thus, we have successfully studied Sequential file to maintain the student data to add, delete information of student.