Lab3: RFS (Record File System)

Objectives:

- Learn how to work with files in C++.
- Be familiar with Relative RFS.
- Implement a RFS.

1- Disk File I/O with Streams

Most programs need to save data to disk files and read it back in. Working with disk files requires another set of classes: ifstream for input, fstream for both input and output, and ofstream for output. Objects of these classes can be associated with disk files, and we can use their member functions to read and write to the files.

Formatted File I/O

In formatted I/O, numbers are stored on disk as a series of characters. Thus 6.02, rather than being stored as a 4-byte type float or an 8-byte type double, is stored as the characters '6', '.', '0', and '2'. This can be inefficient for numbers with many digits, but it's appropriate in many situations and easy to implement. Characters and strings are stored more or less normally.

Writing Data

The following program writes a character, an integer, a type double, and two string objects to a disk file. There is no output to the screen.

```
/  formato.cpp : writes formatted output to a file, using <<
#include <fstream> // for file I/O
#include <iostream>
#include <string>
using namespace std;

int main()
{
        char ch = 'x';
        int j = 77;
        double d = 6.02;
        string str1 = "Kafka";
        string str2 = "Proust";
         ofstream outfile("fdata.txt");              //create ofstream object
        outfile   << ch
                  << j
                  << ' '                             //needs space between numbers
                  << d
                  << str1
                  << ' '                             //needs spaces between strings
                  << str2;
        cout << "File written\n";
        return 0;
}
```

Here we define an object called outfile to be a member of the ofstream class. At the same time, we initialize it to the file FDATA.TXT. This initialization sets aside various resources for the file, and accesses or *opens* the file of that name on the disk. If the file doesn' t exist, it is created.If it does exist, it is truncated and the new data replaces the old. The outfile object acts much as cout did in previous programs, so we can use the insertion operator (<<) to output variables of any basic type to the file. This works because the insertion operator is appropriately overloaded in ostream, from which ofstream is derived.
When the program terminates, the outfile object goes out of scope. This calls its destructor, which closes the file, so we don't need to close the file explicitly. There are several potential formatting glitches. First, you must separate numbers (such as 77 and 6.02) with nonnumeric characters. Since numbers are stored as a

sequence of characters, rather than as a fixed-length field, this is the only way the extraction operator will know, when the data is read back from the file, where one number stops and the next one begins. Second, strings must be separated with whitespace for the same reason. This implies that strings cannot contain imbedded blanks. In this example we use the space character (' ') for both kinds of delimiters.
You can verify that the program has indeed written the data by examining the FDATA.TXT file with the Windows WORDPAD accessory or Notepad.

Reading Data

We can read the file generated by FORMATO by using an ifstream object, initialized to the name of the file. The file is automatically opened when the object is created. We can then read from it using the extraction (>>) operator.
Here's the listing for the FORMATI program, which reads the data back in from the FDATA.TXT file:

```
// formati.cpp
// reads formatted output from a file, using >>
#include <fstream> //for file I/O
#include <iostream>
#include <string>
using namespace std;
int main()
{
        char ch;
        int j;
        double d;
        string str1;
        string str2;
        ifstream infile("fdata.txt");           //create ifstream object
                                                 //extract (read) data from it
        infile >> ch >> j >> d >> str1 >> str2;
        cout << ch << endl                       //display the data
             << j << endl
             << d << endl
             << str1 << endl
             << str2 << endl;
        return 0;
}
```

Here the ifstream object, which we name infile, acts much the way cin did in previous programs. Provided that we have formatted the data correctly when inserting it into the file, there's no trouble extracting it, storing it in the appropriate variables, and displaying its contents. The program's output looks like this:

```
x
77
6.02
Kafka
Proust
```

Of course the numbers are converted back to their binary representations for storage in the program. That is, the 77 is stored in the variable j as a type int, not as two characters, and the 6.02 is stored as a double.

Binary I/O

You can write a few numbers to disk using formatted I/O, but if you're storing a large amount of numerical data it's more efficient to use binary I/O, in which numbers are stored as they are in the computer's RAM memory, rather than as strings of characters. In binary I/O an int is stored in 4 bytes, whereas its text version might be "12345", requiring 5 bytes. Similarly, a float is always stored in 4 bytes, while its formatted version might be "6.02314e13", requiring 10 bytes.

Our next example shows how an array of chars is written to disk and then read back into memory, using binary format. We use two new functions: write(), a member of ofstream; and read(), a member of ifstream. These functions think about data in terms of bytes (type char). They don't care how the data is formatted, they simply transfer a buffer full of bytes from and to a disk file. The parameters to write() and read() are the address of the data buffer and its length. The address must be cast to type char*, and the length is the length in bytes (characters), *not* the number of data items in the buffer. Here's the listing for BINO:

```
// bino.cpp
// binary input and output with integers
#include <fstream> //for file streams
#include <iostream>
using namespace std;
int main()
{
	ofstream outfile ("edata.dat", ios::binary);
	char ch = 'x';
	int j = 77;
	double d = 6.02;
	char str1[6] = "Kafka";
	char str2[9] = "Proust 2";
	outfile<<ch<<j<<" "<<d;
	outfile.write(str1,6);
	outfile.write(str2,9);
	outfile.write((char*)&j,sizeof(int));
	cout << "File written\n";
	return 0;
}
```

The next program reads back into memory using binary format.

```
// bini.cpp
// binary input and output with integers
#include <fstream> //for file streams
#include <iostream>
using namespace std;
int main()
{
	ifstream infile ("edata.dat", ios::binary);
	char ch;
	int j;
	double d;
	char str1[6];
	char str2[9];
	infile	>> ch
		>> j
		>> d;
	infile.read(str1,6);
	infile.read(str2,9);

	cout<<ch<<endl
		<< j<<endl
		<< d<<endl
		<< str1<<endl
		<< str2<<endl;
	infile.read((char*)&j,sizeof(int));
	cout<<j<<endl;
	cout << "File written\n";
	return 0;
}
```

We can also use write to store array of integer as in binio

```
// binio.cpp
// binary input and output with integers
#include <fstream>                              //for file streams
#include <iostream>
using namespace std;
const int MAX = 100;                            //size of buffer
int buff[MAX];                                  //buffer for integers
int main()
{
        int j;
        for(j=0; j<MAX; j++)                    //fill buffer with data
                buff[j] = j;                            //(0, 1, 2, ...)
                                                //create output stream
        ofstream os("edata.dat", ios::binary);
                                                //write to it
        os.write( (char*)(buff), MAX*sizeof(int) );
        os.close();                     //must close it

        for(j=0; j<MAX; j++) //erase buffer
                buff[j] = 0;
                                                //create input stream
        ifstream is("edata.dat", ios::binary);
                                                //read from it
        is.read((char*)(buff), MAX*sizeof(int) );
        for(j=0; j<MAX; j++)                            //check data
                if( buff[j] != j )
                        { cout << "Data is incorrect\n"; return 1; }
        cout << "Data is correct\n";
        return 0;
}
```

Closing Files

So far in our example programs there has been no need to close streams explicitly because they are closed automatically when they go out of scope; this invokes their destructors and closes the associated file. However, in BINIO, since both the output stream os and the input stream is are associated with the same file, EDATA.DAT, the first stream must be closed before the second is opened. We use the close() member function for this.

Writing an Object to Disk

When writing an object, we generally want to use binary mode. This writes the same bit configuration to disk that was stored in memory, and ensures that numerical data contained in objects is handled properly. Here's the listing for OPERS, which asks the user for information about an object of class person, and then writes this object to the disk file PERSON.DAT:

```
// opers.cpp
// saves person object to disk
#include <fstream>                                      //for file streams
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class person                                            //class of persons
{
        protected:
                char name[80];                          //person's name
                short age;                               //person's age
        public:
        void getData()                                  //get person's data
        {
                cout << "Enter name: "; cin >> name;
                cout << "Enter age: "; cin >> age;
        }
};
////////////////////////////////////////////////////////////
int main()
{
      person pers;                                      //create a person
```

```
    pers.getData();                                     //get data for person
    ofstream outfile("PERSON.DAT", ios::binary);        //create ofstream object
    outfile.write((char*)(&pers), sizeof(pers));        //write to it
    return 0;
}
```

The getData() member function of person is called to prompt the user for information, which it places in the pers object. Here's some sample interaction:

```
Enter name: Coleridge
Enter age: 62
```

The contents of the pers object are then written to disk, using the write() function. We use the sizeof operator to find the length of the pers object.

Note: that this will not works good if a pointer is in the private data

Reading an Object from Disk

Reading an object back from the PERSON.DAT file requires the read() member function. Here's the listing for IPERS:

```
// ipers.cpp
// reads person object from disk
#include <fstream>                                      //for file streams
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class person                                            //class of persons
{
        protected:
                char name[80];                          //person's name
                short age;                              //person's age
        public:
        void showData()                                 //display person's data
        {
                cout << "Name: " << name << endl;
                cout << "Age: " << age << endl;
        }
};
////////////////////////////////////////////////////////////
int main()
{
        person pers;                                    //create person variable
        ifstream infile("PERSON.DAT", ios::binary);     //create stream
        infile.read( (char*)(&pers), sizeof(pers) );    //read stream
        pers.showData();                                //display person
        return 0;
}
```

The output from IPERS reflects whatever data the OPERS program placed in the PERSON.DAT file:

```
Name: Coleridge
Age: 62
```

I/O with Multiple Objects

The OPERS and IPERS programs wrote and read only one object at a time. Our next example opens a file and writes as many objects as the user wants. Then it reads and displays the entire contents of the file. Here's the listing for DISKFUN:

```
// diskfun.cpp
// reads and writes several objects to disk
#include <fstream>                                      //for file streams
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class person                                            //class of persons
{
        protected:
                char name[80];                          //person's name
                int age;                                //person's age
        public:
        void getData()                                  //get person's data
        {
                cout << "\n Enter name: "; cin >> name;
```

```
                cout << " Enter age: "; cin >> age;
        }
        void showData()                                         //display person's data
        {
                cout << "\n Name: " << name;
                cout << "\n Age: " << age;
        }
};
//////////////////////////////////////////////////////////////
int main()
{
        char ch;
        person pers;                                            //create person object
        fstream file;                                           //create input/output file
                                                                //open for append
        file.open("GROUP.DAT", ios::app | ios::out |ios::in | ios::binary );
        do                                                      //data from user to file
        {
                cout << "\nEnter person's data:";
                pers.getData();                                 //get one person's data
                                                                //write to file
                file.write( (char*)(&pers), sizeof(pers) );
                cout << "Enter another person (y/n)? ";
                cin >> ch;
        }
        while(ch=='y');                                          //quit on 'n'
        file.seekg(0);                                           //reset to start of file
                                                                //read first person
        file.read( (char*)(&pers), sizeof(pers) );
        while( ! file.eof() )                                   //quit on EOF
        {
                cout << "\nPerson:";                    //display person
                pers.showData();                                //read another person
                file.read( (char*)(&pers), sizeof(pers) );
        }
        cout << endl;
        return 0;
}
```

Here's some sample interaction with DISKFUN. The output shown assumes that the program has been run before and that two person objects have already been written to the file.

```
Enter person's data:
        Enter name: McKinley
        Enter age: 22
Enter another person (y/n)? n
Person:
        Name: Whitney
        Age: 20
Person:
        Name: Rainier
        Age: 21
Person:
        Name: McKinley
        Age: 22
```

Here one additional object is added to the file, and the entire contents, consisting of three objects, are then displayed.

The Mode Bits

We've seen the mode bit ios::binary before. In the open() function we include several new mode bits. The mode bits, defined in ios, specify various aspects of how a stream object will be opened. The following table shows the possibilities.

Mode Bit	*Result*
in	Open for reading (default for ifstream)
out	Open for writing (default for ofstream)
ate	Start reading or writing at end of file (AT End)
App	Start writing at end of file (APPend)
Trunk	Truncate file to zero length if it exists (TRUNCate)
nocreate	Error when opening if file does not already exist
Noreplace	Error when opening for output if file already exists, unless ate or app is set
Binary	Open file in binary (not text) mode

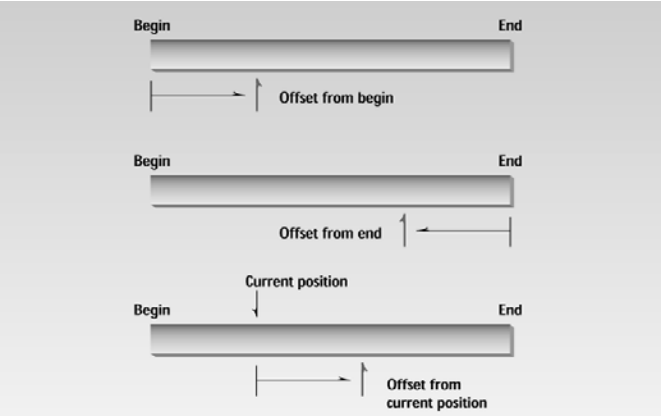
File Pointers

Each file object has associated with it two integer values called the *get pointer* and the *put pointer*. These are also called the *current get position* and the *current put position*, or simply the *current position*. These values specify the byte number in the file where writing or reading will take place. (The term *pointer* in this context should not be confused with normal C++ pointers used as address variables.)

Often you want to start reading an existing file at the beginning and continue until the end. When writing, you may want to start at the beginning, deleting any existing contents, or at the end, in which case you can open the file with the ios::app mode specifier. The seekg() and tellg() functions allow you to set and examine the get pointer, and the seekp() and tellp() functions perform these same actions on the put pointer.

Specifying the Position

We saw an example of positioning the get pointer in the DISKFUN program, where the seekg() function set it to the beginning of the file so that reading would start there. This form of seekg() takes one argument, which represents the absolute position in the file. The start of the file is byte 0, so that's what we used in DISKFUN. The following figure shows how this looks.



Specifying the Offset

The seekg() function can be used in two ways. We've seen the first, where the single argument represents the position from the start of the file. You can also use it with two arguments, where the first argument represents an offset from a particular location in the file, and the second specifies the location from which the offset is measured. There are three possibilities for the second argument: beg is the beginning of the file, cur is the current pointer position, and end is the end of the file. Examples:

```
seekp(-10, ios::end);   // will set the put pointer to 10 bytes before the end of the file.
seekp(10, ios::cur);            // will set the put pointer to 10 bytes after the current position of the pointer.
seekp(-10, ios::cur);           // will set the put pointer to 10 bytes before the current position of the pointer.
seekp(10, ios::beg);            // will set the put pointer to 10 bytes after the beginning of the file.
seekp(10);              // will set the put pointer to 10 bytes after the beginning of the file.
```

Here's an example that uses the two-argument version of seekg() to find a particular person object in the GROUP.DAT file, and to display the data for that particular person. Here's the listing for SEEKG:

```
// seekg.cpp
// seeks particular person in file
#include <fstream>                                      //for file streams
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class person                                            //class of persons
{
        protected:
                char name[80];                          //person's name
                int age;                                //person's age
        public:
        void getData()                                  //get person's data
        {
                cout << "\n Enter name: "; cin >> name;
                cout << " Enter age: "; cin >> age;
        }
        void showData(void)                     //display person's data
        {
                cout << "\n Name: " << name;
                cout << "\n Age: " << age;
        }
};
////////////////////////////////////////////////////////////
int main()
{
        person pers;                                    //create person object
        ifstream infile;                                //create input file
        infile.open("GROUP.DAT", ios::in | ios::binary);        //open file
        infile.seekg(0, ios::end);                      //go to 0 bytes from end
        int endposition = infile.tellg();               //find where we are(size of file)
        int n = endposition / sizeof(person);           //number of persons
        cout << "\nThere are " << n << " persons in file";
        cout << "\nEnter person number: ";
        cin >> n;
        int position = (n-1) * sizeof(person);          //number times size
        infile.seekg(position);                         //bytes from start
                                                        //read one person
        infile.read( (char*)(&pers), sizeof(pers) );
        pers.showData();                                //display the person
        cout << endl;
        return 0;
}
```

Here's the output from the program, assuming that the GROUP.DAT file is the same as that just accessed in the DISKFUN example:

```
        There are 3 persons in file
        Enter person number: 2

                Name: Rainier
                Age: 21
```

For the user, we number the items starting at 1, although the program starts numbering at 0; so person 2 is the second person of the three in the file.

Error-Status functions

The following table containing function used to detect errors in file operations and reset it.

Name	*Meaning*
int eof();	Returns true if EOF flag set
int fail();	Returns true if an error occurs
int good();	Returns true if everything OK; no flags set
void clear();	clears all error bits

```
// ferrors.cpp
// checks for errors opening file
#include <fstream> // for file functions
#include <iostream>
using namespace std;
int main()
{
        ifstream file;
        file.open("a:test.dat");
        if( !file )
                cout << "\nCan't open GROUP.DAT";
        else
                cout << "\nFile opened successfully.";
        cout << "\ngood() = " << file.good();
        cout << "\neof() = " << file.eof();
        cout << "\nfail() = " << file.fail();
        file.close();
        return 0;
}
```

This program first checks the value of the object file. If its value is zero, the file probably could not be opened because it didn't exist. Here's the output from FERRORS when that's the case:

```
Can't open GROUP.DAT
good() = 0
eof() = 0
fail() = 4
```

2- Record file System (Relative RFS)

Data of real life problems are usually in the form of interrelated **objects** or **entities**. To deal with the entities of the problem we need first to store its describing data called **attributes**.

Usually the attributes of a given entity are stored together to form what we often call a record and each attribute is stored in a logical part of the record called field. Records for similar entities are stored together in files and are located in it using some unrepeatable value (explicit or implicit) called key or record identifier.

Records can be divided into two main types:

1- **Fixed-length records**: in which case, all the records of a given file have the same length, and this type of records will make implementing the algorithms of the file-system very easy and also will make the performance of the generated algorithms very near to optimal if designed carefully. This observation may lead some file-system designers to employ what we can name Fixed-At-Max-Length approach in which they convert a variable length record system to a fixed length record system by fixing the record size at the maximum possible size and this approach may be very handful in many situations.

2- **Variable-length records**: in which case, not all the records of a given file have the same length, and this type of records will make implementing the algorithms of the file-system much more difficult compared with fixed-length records systems (except for very simple file-systems such as sequential chronological files as will explained later) and also will normally gives less performance than a comparable fixed-length record system. There are three kinds of variety that may cause records to have a variable length:

   I- **Variable-length fields**: When some of the fields of the record stores an attribute that can has variable length value such as alphanumeric strings (names, addresses ... etc). This is the only type of variable-record systems that can be encountered in modern relational systems (and some considers this as one of the main definitive properties of database systems).

   II- **Variable-format records**: When the format of the records (its fields) are not the same for every entity (for example if there is a file to store employees' data and some of them has a commission and others not a designer may decide to preserve space to store a commission field just with those who has a commission value and eliminate this field entirely –not just store a special value or zero on it- from the records of other employees). This type of variable-length records is very uncommon and usually can be avoided because of its added complexity (sometimes records may has a logical fixed-format while it is stored under skin as variable-format records to preserve space). If this type of records is used usually a special field is added to the record to specify its type.

   III- **Repeating groups:** In this case a set of fields could be repeated a variable number of times for a single entity (for example a set of child describing fields –Name, Birth date, ... etc- may be included in the records describing citizens of a city which are repeated for every child). In this case usually a special field is added to store the number of repeats of each possible repeating group (but this introduces some complications to ensure that this field is always correct or otherwise the whole file will be corrupted).

The constraints of physical Devices

If computers had perfect memories then the study of files and file-systems as a separate topic would never exists, but computer memories had two imperfect characteristics (in this context):

1- They are limited in size so that they may not be able to store all the data needed by a process.
2- They are volatile that is they lose data stored on them if their supply of power has been interrupted.

For these reasons a need for secondary storage emerged that can overcome these two limitations. Thus instead of storing information in memory we use this secondary storage which is typically much slower than the main memory (1000 slower). An interface between your program and the secondary storage is needed to organize the access to this slow storage system.

File-System types:

Sequential-Chronological Files: It is formed by appending records at the end of the file and in the order in which they arrive. It is useful only in limited circumstances. Although records may be logically deleted from the file they may not been physically deleted from the file, they may be logically deleted by flagging the records It is necessary occasionally to reorganize (defragment) the file to remove them physically.

The relative Files: It is one in which records can be accessed directly using their record number as an external key. We are no longer restricted to accessing record in the sequence in which they appear in the file. This type of access is some time referred to as random access. The file system assumes the burden of calculating the address of the block containing a particular record. In this respect the relative file looks very much like one-dimensional array where each element of the array is a record and the index is the record number.

Ordered Files: They are files constrained so that the records are ordered by the value of a key. Specifically, for each rescored i, where i is any record except the first or the last one, it is always true that KEY[i-1]<=KEY[i]<=Key[i+1]. This constrain has a large effect on the way the file is built, but also allows more effective algorithm to be applied.

Example: Simple School RFS

```
// Record.h: interface for the Record class.
//
//////////////////////////////////////////////////////////////////////
#if !defined RecoreInclude
#define RecoreInclude
typedef unsigned int uint;
class RFS;
class Record
{
protected:
        uint RecordNumber;
        bool active;        //lazy delete
        friend class RFS;
public:
        Record();
        virtual ~Record();
        bool isActive()     {return active;}
};
#endif
```

```
// Record.cpp: implementation of the Record class.
//
//////////////////////////////////////////////////////////////////////
#include "Record.h"
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
Record::Record()
{
}
Record::~Record()
{
}
```

```
// RFS.h: interface for the RFS class.
//
//////////////////////////////////////////////////////////////////////
#if !defined RFSInclude
#define RFSInclude
#include <iostream>
#include <fstream>
#include "Record.h"
using namespace std;
typedef unsigned int uint;
class RFS
{
protected:
        uint numberOfRecords;
        uint sizeOfRecord;
        fstream F;
        uint recodLocation(uint i) {return (i*sizeOfRecord);}
public:
        RFS(int sor,char filename[]);
        bool get(Record * r, uint i);
        bool put(Record * r, uint i=-1); // single input= append;
        bool update(Record * r);
        bool del(Record * r);
        bool good();
        uint GetNumberOfRecords(){return
numberOfRecords;}
        virtual ~RFS();
};
#endif
```

```
// RFS.cpp: implementation of the RFS class.
//
//////////////////////////////////////////////////////////////////////
#include "RFS.h"
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
RFS::~RFS()
{
        F.close();
}
//////////////////////////////////////////////////////////////////////
RFS::RFS(int sor,char filename[])
{
        numberOfRecords=0;
        sizeOfRecord=sor;
        F.open(filename,ios::app|ios::out|ios::in|ios::binary);
        F.close();   // to create it if not exist
                     // a bug in c++
        F.open(filename,ios::out|ios::in|ios::binary);
        if(good()){
                cout<< filename<<" is Open"<<endl;
                F.seekg(0,ios::end);
                numberOfRecords=F.tellg()/sizeOfRecord;
                }
        else{
                cout<< "could not open"<<filename<<endl;
        }
}
//////////////////////////////////////////
bool RFS::get(Record * r,uint i)
{
        if (!good())
        {
                cout<< "error reading from file"<<endl;
                return false;
        }
        if (i<numberOfRecords)
        {
                F.seekg(recodLocation(i),ios::beg);
                F.read((char *)r,sizeOfRecord);
                return true;
        }
        else
                return false;
}
//////////////////////////////////////////
bool RFS::put(Record * r,uint i)
{
        if (!good())
        {
                cout<< "error writting to file"<<endl;
```

```
                return false;
        }
        if (i==-1)
        { //append
                r->active=true;
                r->RecordNumber=numberOfRecords;
                F.seekg(recodLocation(numberOfRecords)
                        ,ios::beg);
                F.write((char *)r,sizeOfRecord);
                numberOfRecords++;
                return true;
        }
        else
                if (i<numberOfRecords)
                {
                        r->RecordNumber=i;
                        F.seekg(recodLocation(i),ios::beg);
                        F.write((char *)r,sizeOfRecord);
                        return true;
                }
                else
                        return false;
}
////////////////////////////////////////////
bool RFS::del(Record * r)
{
        r->active=false;
        return(put(r,r->RecordNumber));
}
////////////////////////////////////////////
bool RFS::update(Record * r)
{
        return(put(r,r->RecordNumber));
}
////////////////////////////////////////////
bool RFS::good()
{
        return F.good();
}
```

```
// Student.h: interface for the CStudent class.
//
//////////////////////////////////////////////////////////////////////
#if !defined StudentInclude
#define StudentInclude
#include "RFS.h"
class CStudent : public Record
{
public:
        unsigned int m_Number;
        char m_Name [100];
        float m_SubjectGrade [10];
public:
        CStudent();
        virtual ~CStudent();
};
#endif
```

```
// Student.cpp: implementation of the CStudent class.
//
//////////////////////////////////////////////////////////////////////
#include "Student.h"
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
CStudent::CStudent()
{
        strcpy(m_Name,"");
        for (int i=0; i<10; i++)
                m_SubjectGrade[i]=0.0;
}
CStudent::~CStudent()
{
}
```

```
// School.h: interface for the CSchool class.
//
//////////////////////////////////////////////////////////////////////
#if !defined SchoolInclude
#define SchoolInclude
#include "RFS.h"
class CSchool
{
        private:
                RFS rfs;
        public:
                CSchool(char filename[]);
                bool AddNewStudent();
                bool RemoveStudent();
                bool ModifyStudent();
                bool SearchForStudent();
                bool Sort();
};
#endif
```

```
// School.cpp: implementation of the CSchool class.
//
//////////////////////////////////////////////////////////////////////
#include "School.h"
#include "Student.h"
#include "RFS.h"
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
CSchool::CSchool(char filename[])
                : rfs(sizeof(CStudent),filename)
{  }

/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
// Add New Student Function
// getting new student data and add the new student to school
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
bool CSchool::AddNewStudent()
{
        CStudent NewStudent;
        cout<<"\t\t\tAdding New Student\n\n";
        cout<<"Student Number : ";
        cin>>NewStudent.m_Number;
        cin.ignore(); // to avoid bug in getline
        cout<<"Student Name : ";
        cin.getline(NewStudent.m_Name,100);
        for(int i=0;i<10;i++)
        {
                cout<<"% Grade of Subject # "<<i+1<<" : ";
                cin>>NewStudent.m_SubjectGrade[i];
        }
        return(rfs.put(&NewStudent));
                //append new student to school
}
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
// Remove Student Function
// remove student from school.
// Removes the first occurrence of the student number
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
bool CSchool::RemoveStudent()
{
        CStudent Student;
        unsigned int number,i;
```

```
        cout<<"\t\t\tRemove Student\n\n";
        cout<<"Enter Student Nnumber to be removed : ";
        cin>>number;    //get student number
        i=0;
        if( ! rfs.get(&Student,i++) )
                return false;
        //loop if not inteded student or student is deleted
        while((number!=Student.m_Number)||
                                (!Student.isActive()))
        {
                if( ! rfs.get(&Student,i++) )
                        return false;
        }
        // at this point the CStudent object holds the intended
        // student  otherwise if the student does not exist the
        // function fails

        //now removes the student from school
        return( rfs.del(&Student) );
}
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
// Modify Student Function
// modify student data. The function displays old data
// and asks for new one. Then it updates the data
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
bool CSchool::ModifyStudent()
{
        CStudent Student;
        unsigned int number,location;
        cout<<"\t\t\tModifying Student\n\n";
        cout<<"Enter Student Number to be modified : ";
        cin>>number; //get student number to be modified
        location=0;
        if( ! rfs.get(&Student,location++) )
                return false;
        //loop if not inteded student or student is deleted
        while((number!=Student.m_Number)||
                                (!Student.isActive()))
        {
                if( ! rfs.get(&Student,location++) )
                        return false;
        }
        location--;
                // the CStudent object holds the intended
                // student to be modified
        cout<<"Old Number : "<<Student.m_Number<<endl;
        cout<<"New Number : ";
        cin>>Student.m_Number;
        cout<<"Old Student Name : "<<Student.m_Name
              <<endl;
        cout<<"New Student Name : ";
        cin.getline(Student.m_Name,100);
        for(int i=0;i<10;i++)
        {
                cout<<"Old % Grade of Subject # "<<i+1<<
                      " : "<<Student.m_SubjectGrade[i]<<endl;
                cout<<"New % Grade of Subject # "<<i+1<<
                      " : ";
                cin>>Student.m_SubjectGrade[i];
        }
        return( rfs.put(&Student,location) );
                // update student data
}
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
// Search for Student Function
// search for the first matched student name and displays its
data
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
bool CSchool::SearchForStudent()
{
        CStudent Student;
        unsigned int location;
        char name[100];
        cout<<"\t\t\tSearch for Student\n\n";
        cin.ignore(); // for getline bug
        cout<<"Enter Student Name to Search for : ";
        cin.getline(name,100);
        location=0;
        if( ! rfs.get(&Student,location++) )
                return false;
        //loop if not inteded student or student is deleted
        while((strcmp(name,Student.m_Name))||
                                (!Student.isActive()))
        {
                if( ! rfs.get(&Student,location++) )
                        return false;
        }
        location--;
        // CStudent object holds inteded student
        // display student data on screen
        cout<<"Student Number : "
              <<Student.m_Number<<endl;
        cout<<"Student Name : "<<Student.m_Name<<endl;
        for(int i=0;i<10;i++)
                cout<<"% Grade of Subject # "<<i+1<<
                      " : "<<Student.m_SubjectGrade[i]<<endl;
        return true;
}
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
//                      Sort Function
// sort students according to their results in a specified subject
// sorting is done using bubble sort (the most easy algorithm)
// Note: deleted records will be moved to the end of file
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/*/
bool CSchool::Sort()
{
        CStudent Student1,Student2;
        unsigned int Subject,Loc1,Loc2;
        unsigned int NRecords;
        cout<<"\t\t\t\tSort\n\n";
        cout<<"Enter Subject Number to sort according to : ";
        cin>>Subject;
        Subject--;
        NRecords= rfs.GetNumberOfRecords();
        cout<<"Sorting ...\n";
        for(Loc1=0;Loc1<NRecords-1;Loc1++)
        {
              for(Loc2=Loc1+1;Loc2<NRecords;Loc2++)
                    {
                          if(! rfs.get(&Student1,Loc1) )
                                return false;
                          if(! rfs.get(&Student2,Loc2) )
                                return false;
                          if(!Student2.isActive())
                                continue;
                         if((!Student1.isActive())||
                         (Student2.m_SubjectGrade[Subject]>
                          Student1.m_SubjectGrade[Subject]))
                          {
                              rfs.put(&Student2,Loc1);
                              rfs.put(&Student1,Loc2);
                          }
                    }
        }
        cout<<"Sorting completed.\n";
        Loc1=0;
        while((rfs.get(&Student1,Loc1++))
                        &&(Student1.isActive()))
```

```
        {
                cout<<"Student Number : "<<
                        Student1.m_Number;
                cout<<"\t\tStudent Name : "<<
                        Student1.m_Name<<endl;
                cout<<"\t\t\t% Grade of Subjects\n";
                for(Subject=0;Subject<10;Subject++)
                {
                        cout<<" # "<<Subject+1<< "\t";
        cout<<Student1.m_SubjectGrade[Subject]<<"\t";
                cout<<endl;
                }
        }
        return true;
}
```

```
#include <conio.h>
#include "RFS.h"
#include "Student.h"
#include "School.h"
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/
// Functions header
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/
void CleanScreen(void);
int ShowMenu(void);
char againYON();
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/
int  main ( )
{
      CSchool School("School.db");
      char again; //a flag to indicate do selected option again
      int choice; //user selected option
      bool exit=false; // flag to terminate execution
      while(true) //loop forever
      {
            CleanScreen();
            choice=ShowMenu();
                  //show menu and get user option
            switch(choice) //handling user input
            {
            case 1: // Adding a new student to the school
                  again='Y';
                  while(again=='Y')
                              //loop as user wants to continue
                  {
                     if(School.AddNewStudent())
                     {
                        cout<<"a New Student has been"
                              <<" successfully added\n";
                        cout<<"Do you want to add another"
                              <<" (y/n)?";
                           again=againYON();
                     }
                     else //failed to add a student to library
                     {
                           again='N';
                           cout<<"couldn't add a New Student\n";
                           cout<<"Press any key to continue";
                           _getch();
                     }
                  }
                  break;
            case 2: //remove a student from the school
                  again='Y';
                  while(again=='Y')
                  {
                        if(School.RemoveStudent())
                        {
                              cout<<"a Student has been"
                                    <<" successfully removed\n";
                              cout<<"Do you want to remove another"
                                    << (y/n)?";
                              again=againYON();
                        }
                        else // failed to remove a student from school
                        {
                              again='N';
                              cout<<"couldn't remove a Student\n";
                              cout<<"Press any key to continue";
                              _getch();
                        }
                  }
                  break;
            case 3: // modify student data
                  again='Y';
                  while(again=='Y')
                  {
                        if(School.ModifyStudent())
                        {
                              cout<<"a Student has been"
                                    <<" successfully modified\n";
                              cout<<"Do you want to modify another"
                                    <<" (y/n)?";
                              again=againYON();
                        }
                        else // failed to modify student data
                        {
                              again='N';
                              cout<<"couldn't modify a Student\n";
                              cout<<"Press any key to continue";
                              _getch();
                        }
                  }
                  break;
            case 4: // search for a student by name
                     //(the first matched will be displayed)
                  again='Y';
                  while(again=='Y')
                  {
                        if(School.SearchForStudent())
                        {
                              cout<<"Do you want to search"
                                    <<" for another (y/n)?";
                              again=againYON();
                        }
                        else // failed to find a matching student name
                        {
                              again='N';
                              cout<<"couldn't find specified student\n";
                              cout<<"Press any key to continue";
                              _getch();
                        }
                  }
                  break;
            case 5: // Sort student according to a certain subject
                     // results and display students data
                  again='Y';
                  while(again=='Y')
                  {
                        if(School.Sort())
                        {
                              cout<<"Do you want to do another Sort"
                                    << "(y/n)?";
                              again=againYON();
                        }
                        else // failed to sort students
                        {
                              again='N';
```

```
                        cout<<"couldn't do Sort\n";
                        cout<<"Press any key to continue";
                        _getch();
                    }
                }
                break;
            case 6: // terminate execution
                exit=true; //set exit flag
                break;
            }
            // break infinite loop to terminate execution
            if(exit)
                break;
        }
}
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/
char againYON()
{
    char again=0;
    while((again!='Y')&&(again!='N'))
    {
        again=_getch();
        again=toupper(again);
    }
    return again;
}



/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/
// Clean Screen Function
// print an empty 25 lines
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/
void CleanScreen(void)
{
    cin.ignore();
    cout.flush();
    for(int i=0;i<25;i++)
        cout<<endl;
}
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/
// Show Menu Function
// print the menu options and get user input
// return user input to calling function for processing
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/
int ShowMenu(void)
{
    int choice;
    cout<<"\t\t\t\t MainMenu\n\n";
    cout<<"\t\t\t (1) Add New Student.\n";
    cout<<"\t\t\t (2) Remove Student.\n";
    cout<<"\t\t\t (3) Modify Student Data.\n";
    cout<<"\t\t\t (4) Search for a Student.\n";
    cout<<"\t\t\t (5) Sort by Subject Results.\n";
    cout<<"\t\t\t (6) Exit.\n\n";
    cout<<"\t\t Enter your choice (1...6) : ";
    cin>>choice;
    return choice;
}
```

3-Lab Work

Write a relative RFS that stores books information of a library.
The book information is its number, title, author, Publisher, available quantity, number of pages, cost.
Your program should be able to:

1- Add a new book to the library.
2- Search for a book by the title or number.
3- Find all books written by an author.
4- Find all books publisher by a publisher.
5- Change cost of a book. The book number should be used to reference the book.
6- Change the quantity of a book. The book number should be used to reference the book.
7- Delete a book. The book number should be used to reference the book.
8- Sort books by their titles.
9- Find all books with zero quantity.
10- Find all books that are more expensive than a given value

Write a good user interface to your program.