Random Access Files in C++

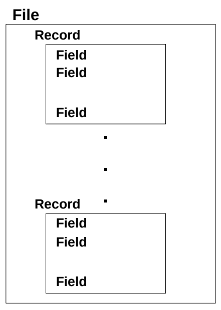
C++ allows the user to randomly access a file at any point in the file.

Of course to use this ability in a meaningful way, the file should be given some structure.

A conceptual look at files

A file can be broken down into 3 layers of abstraction

1. File
2. Record
3. Field



A file may contain many records and each record can contain many fields.

Reading and writing to and from the file is done on the **record level**.

To work at the record level a data structure we have to be defined in the programming language that matches that of the field structure found in the records.

In most cases, the records are homogenous in a file.

Working with Records in a file

To read a record from the file:

1. Open the file
2. Move the file pointer to the correct record
3. Read the record into a data structure defined in the program
4. Close the file

To write a record to the file:

1. Open the file
2. Move the file pointer to the correct position (append would be the end)
3. Write the data structure out to the file
4. Close the file

To Up-date a record in the file:

1. Open the file
2. Move the file pointer to the correct record
3. Read the file into the data structure defined in the program

4. Modify the data structure
5. Move the file pointer to the correct record
6. Write the data structure out to the file
7. Close the file

To Delete a record from the file:

"Deferred until a later discussion"

Working with a File of Records in C++

To create a file of records in C++ we must first create the data structure that will represent the records in the file. In C++ we can do this with a Struct definition.

Example: Assume we want to keep track of a product we sell in a store. We might create a struct definition like:

```
struct product_type
{
   char  name[50];
   float price;
             :
};
```

We will then be able to create variables of this type

```
product_type  product_record;
```

The product_record is then the level at which we will read and write to and from the file.

Creating a new File of Records   (Binary Files in C++)

When using binary files in C++, the programmer is responsible for:

1. Including the correct header file.
2. Opening the file in binary mode
3. Setting the correct file I/O pointers..
4. Using the file correctly
5. Closing the file.

1. To use binary files in C++ you must include the header file: **fstream   or fstream.h**

2. The file mode describes how a file is to be used: read it, write to it, append it, and so on.
   When you associate a stream with a file, either by initializing a file stream object with a file name or by using the open() method, you can provide a second argument specifying the file mode:

   ```
   ifstream fin("data.dat", <mode>);

   ofstream fout;
   fout.open("data2.dat", <mode>);
   ```

   The file mode is of type int and you can choose from several constants defined in the ios class.

The ifstream has a default mode of ios::in   The ofstream has a default mode of ios::out

Opening a file in ios::out mode also opens it in the **ios::trunc mode by default!!!**

Note: this means you usually want to also open it for reading to prevent this   ios::in

To open a file in binary mode you need to use ios::binary. Some systems may not have ios:binary defined because the underlying storage is binary already.

Constant	Meaning
ios::in	Open file for reading
ios::out	Open the file for writing
ios::ate	Seek to eof upon opening file
ios::app	Append to the file
ios::trunc	Truncate file if it exists
ios::nocreate	Open fails if file does not exist
ios::replace	Open fails if the file does exist
ios::binary	Binary file

You can use several constant together to set the mode by using the bitwise 'or' operator.

ios::binary | ios::app

You can open a binary file for reading or writing or both. For our program we will want to open the file so that we can both read from the file and write to the file. We must also make sure that when we open the file that we do not truncate it since we would lose the other employee records.

```
#include <fstream.h>
char empl_file_name [ ] = "data.dat";
    :
fstream emplfile;
emplfile.open(emply_file_name,ios::in | ios::out | ios::ate | ios::binary);

// the ios:ate prevents the program from truncating the file!!
if (emplfile.good())  // use a stream state method to check the state of
{                            //  the stream
          :
}
```

Method	Returns
eof()	Nonzero on end-of-file
fail()	Nonzero if last I/O operation failed or if invalid operation attempted or if there's been an unrecoverable error
good()	Nonzero if all stream state bits are zero
rdstate()	The stream state
clear(int n = 0)	Returns nothing, but sets stream states to n; the default value of n is 0;
bad()	Nonzero if invalid operation attempted or if there's been an unrecoverable error

Positioning the File Pointer

Once we have opening the file correctly, we must position the file pointer(s) to point to the correct record. We can use the **seekg** method to do this.

```
emplfile.seekg( <offset>,<location>)
```

The offset is an integer and the location is one of three

Location	Action
ios::beg	Set to beginning of the file
ios::end	Set to end of file
ios::cur	Use the current pointer position

Example: emplfile.seekg(30, ios::beg) // go to byte number 30 in the file.

Reading and Writing Records in C++

To read a record from a file the read method of the file object is used.

```
emplfile.read( (char *) &emplrec, sizeof(emplrec) );
```

where

emplfile *: the file which has been opened for input.*

emplrec *: the record struct created to hold the record.*

the read method has two parameters

1. The address (as a pointer to a char) of where the bytes are to be read into.

   In the example the address of emplrec, type cast as a pointer to a char, is passed to the method.

   ```
   (char *) &emplrec
   ```

2. The number of bytes to read in.

   The number of bytes is computed and passed in to the read method.

   ```
   sizeof(emplrec)
   ```

Writing a record to the file uses the write method with the same parameters as the read method.

```
emplfile.write( (char *) &emplrec,sizeof(emplrec) );
```

Using an index for random access of binary files in C++

An index is a method for allowing us to randomly read and write records from files.

Using random access requires that we know the name of the file to read from, the type of records in the file, and the position of the record in the file.

The sequence is then to open the file in the correct mode, seek (that is move) to the correct record, read the record into memory, edit fields of the record, seek again to the correct record and finally write the record out. Of course making sure to close the file when done.

The index will allow us to keep track of where a record is in the file. Each entry in the index will include a key value which uniquely identifies the record along with the record position in the file. By searching the index for the key, we can find the record position to seek to.

Lets assume we have a employee_rec structure declared. If we have opened the file for I/O and know the record number (lets say recnum) we can do the following.

```
employee_rec  emplrec;  // a location to read the record into

//  assuming we have opened the file
empfile.seekg ( recnum * sizeof (emplrec), ios::beg); // move to the record in the file

empfile.read ( (char *) &emplrec, sizeof (emplrec)); // read one record from the file

//  modify emplrec
empfile.seekg ( recnum * sizeof (emplrec), ios::beg);  // move back to the record in the file

empfile.write( (char *) &emplrec, sizeof (emplrec));  // write one record out to the file
```

If we create an array of structures which hold the key and recnum of each of our employees, then all we have to do is a sequential search of the array to find the recnum.

Managing an Index

To manage an index for a file, a data structure will have to be created to hold the index.

One possible structure is an array of index structures where each index structure holds the key value and the record position in the file.

Example of index structure

```
struct index_rec
{
   int empl_num;    //  the key used to look up an employee
   int rec_num;     //  the record number of the record in the file
};
```

Example of the array to hold the index_rec(s)

```
  index_rec  index[MAX_RECORDS];
```

What we want to do is then keep the index updated as employee records are added and deleted to the file.

A major design decision is how the index will be filled. There are two common ways for this to happen.

1. **Create the index from the employee file each time the program runs.**

   This method requires the sequential traversal of the whole employee file each time the program is run. It does however guarantee that the index is correct at the start of each program run

2. **Save the index in a file and read it in each time the program runs and save it whenever the file has been updated.**

   This method only needs to read the index in at the start of the program run which will be much faster than building it. However, it does require that the index file be updates as the employee file is updated. If a mistake is made or an error occurs on the disk, the index and employee file may no longer match!!!!

Building an index

Lets assume we have an employee file and we want to build an index on it where the key will be the empl_num. Assuming the index structure given earlier, how would we build the index.

We will need to sequentially traverse the employee file, reading in one record at a time, and making an entry in the index for each record until all records have been read in.

Assume the employee file is called empl.dat with a record format of

```
struct empl_rec
{
  char empl_name[25];
  int    empl_num;
         :
};
```

```
// open the employee file for input.

// assert to see if the open worked correctly

// read in an employee record from the file

while ( ! empl_file.eof( ) )
{
   // use the empl_rec to update the index

   // read in an employee record from the file
}

// close the file

//  the index is now ready to be used !!!
```