University of Toronto
csc228 — File Structures and Data Management, Fall 2001

Assignment 1 — Basic File Processing

Part A Due: *at the beginning of class, Monday, September 24, 6:10 pm.*
Part B Due: *in the 228 class box, Tuesday, October 9, 6:10 pm.*
Total weight: *10% of your course grade. Part A is worth 2% of this total weight.*

You will do the assignment in two parts. Part A is merely a step on the way to part B. It will be handed in for marks simply to encourage you to get an early start on the design.

The purpose of this assignment is three-fold: 1) to expose you to some simple problems in file processing; 2) to ensure that you are comfortable with basic file input/output in C^{++}; and 3) to introduce you to some very powerful features of C^{++}, including objects, inheritance, etc. The tutors will spend time on these topics in the tutorials, but you must be prepared to consult your C^{++} book for more information.

Problem Statement

Many of you have undoubtedly toyed with a program called Napster, which allows users to share audio files with each other. The record industry is trying its best to shut down Napster, fearing it will lose income on CD sales if users are allowed to freely distribute audio files. The final outcome of this war has yet to be determined. In any case, being the software entrepreneur that you are, and noticing that Napster has its hands full fighting the battle with the corporate giants, you decide to build your own system that addresses a shortcoming of the Napster system. Your gamble is that when the dust settles, freedom of information access will rule, and you'll be well-positioned to introduce your new system.

You notice that Napster is song-based and not album-based. That is, if you want to bundle up the songs belonging to your favorite album, you need to know the songs ahead of time, explicitly search for them, and group them together in your own file system. You figure that if you can automatically gather the songs together that belong to an artist's particular album, your product will be providing an important service that will net you millions.

Here's what's involved. Your program will take two files as input. The first is a song file, containing fixed-length song records. Each song record includes a number of fields that specify the artist, genre, the album from which the song is taken, the size of the audio file, etc. The second input file is a "storage" file, that specifies an internet address and an available capacity, sorted by decreasing capacity. You will begin by reading in the song file and sorting it by artist. Since the file could be huge, you cannot bring it into memory. Instead, you must write out the songs to individual artist files. Next, you repeat the same process for each artist file, except that instead of sorting the artist's songs by album, you will build up a set of album objects in memory (since the number of albums per artist is not enough to tax your main memory), each specifying the songs on a particular album. You will write these album objects out to a temporary album file.

At this point, after deleting all the artist files that you've created, you're left with a temporary album file, along with the input song and storage files. Your last task is an important, but simple one. You must allocate albums to computers. You will cosequentially read both the album and storage files (remember you can only hold one album object and one storage object in memory at a time), and if the total album size can be accommodated by the storage device, you will assign the album to the storage device. If the storage device cannot accommodate the album, you will move to the next storage record. All album objects (whether updated with an assigned storage device or left unassigned to a storage device) are written to a new (and final) album file (the temporary one is then deleted).

The assignment will make you familiar with basic concepts, such as records, fields, input and output, and very basic cosequential file processing. Note that some of the above constraints (e.g., you can hold one thing in memory but not the other) may seem a bit contrived. They are for this particular application, but could be quite plausible for another application. They have been imposed to exercise certain skills and techniques on your part.

Assignment Details

Input and output file format

The "song" input file is a text file containing fixed-length records, each on a separate line. The records have the following format:

- name, a string of 15 characters specifying the song title
- genre, a string of 10 characters specifying the genre of the song (note that all songs from a given album are assigned the same genre). The genre categories include: Pop, Rock, Classical, Jazz, Folk, and Country.
- artist, a string of 15 characters specifying the artist/group name
- album, a string of 15 characters specifying the title of the album on which the song appears
- size, a 5-digit integer specifying the size (in MBytes) of the audio file
- audiofile, a string of 15 characters specifying the filename of the audio file

For example, a "song" input file might contain:

```
Let It Be      Pop       Beatles        Let It Be       2215lib.beatles
Dogs           Rock      Pink Floyd     Animals        10223dogs.pink.floyd
Old Man        Rock      Neil Young     Harvest         9988oldman.young
Rocket Man     Pop       Elton John     Honky Chateau  42669rocket.man
Symphony 5     Classical Beethoven      Karajan Box    90887beet.fifth
```

The "storage" input file is a text file fixed-length containing records, each on a separate line. The records have the following format:

- ipaddress, a string of 30 characters specifying the ip address of the storage device
- capacity, a 6-digit integer that specifies the available storage (in MBytes) on the device

For example, a storage input file might contain:

```
musicdisk.toronto.edu         100000
mydisk.rogers.com               4433
yourdisk.myhome.org            87655
```

The "album" output file is a text file containing variable-length records, each on a separate line. The records have the following format:

- album, a string of 15 characters specifying the title of the album on which the song appears
- artist, a string of 15 characters specifying the artist/group name
- genre, a string of 10 characters specifying the genre of the song (note that all songs from a given album are assigned the same genre). The genre categories include: Pop, Rock, Classical, Jazz, Folk, and Country.
- numsongs, a 2-digit integer specifying the number of songs on the album and the number of song fields to follow
- songrecnos, a sequence (of length "numsongs") of 4-digit integer fields, each containing the record number of a song contained on the album
- albumcapacity, an 8-digit integer that specifies the total size of the album (in MBytes)
- ipaddress, a string of 30 characters specifying the ip address of the storage device (disk) to which the album has been assigned

For example, the output file might contain (assuming that the Neil Young album, Harvest, has only one song on it):

```
Harvest         Neil Young     Rock       1    2     9988yourdisk.myhome.org
```

Note that for each of the three files above, records are terminated with an end-of-line (EOL) character, i.e., if the total length of the fields is n bytes, then in the file, there will be $n+1$ bytes (record plus EOL) before the next record starts. Having an EOL record delimeter means that the files are more easily viewable in your text editor.

Program interface

Your program should not be interactive. The names of the two input files and the output file should be command-line arguments to your program — arguments that the user gives when calling your program.

The original input files must be left intact by your program, and your program must remove any temporary files that it creates.

You need not do a lot of error checking in your program, but you should specify in your report what error checks would ideally be done, and which function would be responsible for each.

Overall algorithm

Your overall algorithm should be:

- Read a record of the song file into a memory buffer (an object). You should also define a song class hierarchy and, depending on the type of song (Rock, Pop, etc.), instantiate an appropriate song object, i.e., extract the information from the memory buffer object and use it to populate an appropriate song object. Note that this object is short-lived, since we don't have room in memory to keep it around. Although it might be more efficient to simply read the record into a single-record buffer and then later write the record back out to an appropriate file (see next step), I'd like to see you dynamically create an appropriate object based on the contents of the (single) buffer object.
- Whenever a new artist (in a song object) is encountered, you must create a new temporary artist file, and write the song object to that file. If the artist has already been encountered (you may assume that the set of artist names can be accommodated in memory), you must write the object out to the appropriate artist file.

- At this point, you have one or more artist files, each containing a number of song records. You have no song records in memory because they won't fit.

- Now, for each artist file, you will again read the song records, one by one, into your memory buffer object and then into a song object. Assuming the number of albums per artist is small enough to fit into memory, you can maintain a number of album objects pertaining to the artist file you're reading. When a song record is read, its album is compared to the album objects you've already defined. If the album object exists, the song is added and the album object updated. If the album object doesn't exist, a new album object is created, and the song added. When all the song records for a given artist file have been processed, all the album objects are written out as records to the temporary album file. The process is repeated for each artist file. The result is a single, temporary album file.

- Now for the cosequential process. You may assume that the storage records are sorted in decreasing order of capacity. You will cosequentially update a pair of read pointers, one for the storage file and one for the album file. When you read in the first storage record, you will create a storage object whose size is initialized with the initial capacity. Next, you will read in the first album record into an album object. If the album size can be accommodated by the storage device, you will assign the storage device ip address to the appropriate slot in the album object. After updating the remaining storage device capacity, you will move to the next album record and repeat. If the album record size cannot be accommodated by the storage device, then you will advance to the next storage record. If, when reading a new storage record (into a storage object), the initial capacity is less than the album size, that album cannot be stored anywhere, and its object's ip address slot is left empty. You will then advance to the next album record and repeat (note that the album records are not sorted by size). Each time an album object is processed (whether assigned to a storage device or not), you must write out the album object to a second (and final) album record file.

- Remember to delete any and all temporary files that you created. That would include all the artist files and the temporary album file.

Other design decisions

Any other decisions about the data structures and algorithms are left to you. Be sure to document your choices in your report.

If you notice a problem with the specifications, or if you feel you need more guidance, contact me.

Developing your program

I recommend that you implement the program in phases, testing your program at each phase before moving on. Below is a suggestion for one possible strategy. Of course, you should feel free to devise your own; my point here really is to encourage you to have *some* strategy.

It's probably best to first lay out the object hierarchy and the interface for each class. You can even compile this skeleton code. Next, try writing a program that solves a simplified version of the problem, and add more details to it until it does the entire thing. For example, you could:

1. Write a program that simply grabs the command-line arguments that you are to accept, and prints out their values.

2. Extend the program so that it opens the input files. For the song file, it could read a record into an appropriately instantiated object, and then print the contents of the newly created

object instance. For the storage file, it could read a record into an object instance, also printing its contents.

3. Extend the program so that, instead of printing the contents of the newly created object instances, it writes the song object instances to appropriate artist files.

4. Once you get that working, the next two phases, i.e., the temporary album file creation and the cosequential process, are independent. These routines can be designed, implemented, and tested independently from the first phase by creating their respective input files by hand. Again, incremental development allows you to isolate problems as you go.

What to hand in for part A

There is no specific requirement for Part A; it is merely a step on the way to part B. I'd like to see your class hierarchy, showing member variables and member functions for each class. I'd also like to see a pseudo-algorithm for your program, and how you plan on decomposing the task into simpler subtasks. This is probably the most important step in a design (the coding really is the final, rather mundane step). You should have made a good start on the program, and have some components working. You may need to create simple driver programs to test your code at each phase of development. Hand in a printout of your code (including any driver programs), and some test runs that demonstrate what works. Your code needn't work flawlessly for you to get some marks on part A. Do not hand in a report with part A.

Your tutor will evaluate your work on this part by looking at your object design, your algorithm, and your test runs (however simple), so be sure that they demonstrate what you have working. Explain your test results with brief, hand-written comments on your printout.

What to hand in for part B

For part B, you are to hand in your assignment in the form of a report, as described in the 228 Course Guide. The style of a 228 report is different from what you may be used to from earlier courses, so be sure to re-read the relevant sections of the Guide before getting far into the assignment. Your report must include all of your code and all relevant input and output test files.

Remember to justify the decisions you made in designing your program. This is a very important part of your report.

Your report should be thorough, well-written, concise, and honest. It should be no more than about five pages long, not including code and tests. Make sure that you run your report through a spelling checker.

You are to devise your own testing for this assignment. Although the program is designed to handle a HUGE input file, you can test it on relatively small ones. You will be marked for testing according to the standard set by the document on “Software Testing”, available on the course web site. Also note that if your testing is weak, you will lose marks not only for testing but also for correctness; the only way we will know that you program works is if you *demonstrate* that through your testing.