Data File Lab – Assignment 1

COMP202 Revised 1 August 2018

Len Hamey, len.hamey@mq.edu.au

This lab is the first of two assignments in COMP202.

Commences: Week 1.

Progress: Monday of weeks 4, 5, 6.

Due: 12 noon, Monday of week 7 – 10 September 2018

Value: 20% (17% for task, 3% weekly progress)

Overview of the Lab

Data files exist in various formats. In Unix, text files are common for simple data, but large data files are stored as binary data. In this lab, you will be developing programs to read, write, modify, and reformat the data in binary data files. The lab consists of a sequence of stages which build on each other. The first three stages develop your skills. In the final stage you will reverse engineer a data file using your knowledge of data representations. You have the choice between an easier final stage (stage 4) that can earn you at most 3 marks for the stage, or a more difficult final stage (stage 5) worth up to 4 marks for the stage. You may attempt both stages 4 and 5, but only the maximum of the two marks will count towards your total

The marking outline is:

Section	**Value**
Stage 1	3
Stage 2	3
Stage 3	3
Stage 4 (max 3 marks) or stage 5 (max 4 marks)	4
Code style	4
Progress	3
Total	**20**

Learning Outcomes

This lab will involve you in developing the following specific skills and capabilities.

- Able to write programs that use C data structures, pointers and arrays.
- Able to read and write binary data files, and write data in text format.
- Able to convert between different data representations.
- Able to use `malloc` and `free` to construct data structures using the heap.
- Able to implement simple command line parameters.
- Able to interpret and recognise binary data representations.
- Research Unix library and system calls

Maximising Your Mark

Here are some hints to get the most marks in this assignment.

1. Work on this assignment every week until the deadline. Don't wait until you've finished the assignments for your other units before you start this assignment. This assignment is intended to be worked on over a period of 5 weeks and almost certainly cannot be completed in a few days.
2. Achieve at least 2.3 marks in each stage of this assignment by the progress mark deadline. Progress marks reward you for consistently working on the assignment.
3. Start thinking about the next stage, and start working on it, once you have a reasonably good mark (at least 2.3) for the earlier stages. You may have an obscure bug that costs you 0.1 or 0.2 marks in the current stage, and you can earn more marks by working on the next stage than by spending all your time trying to perfect your current stage score.
4. Do your own testing as well as using the hints provided by the automarker. The automarker can give you a general idea of your problems, but running your program yourself allows you to examine the particular mistakes that you are making.

Individual work and information resources

The stages of the lab are based on data files that will be provided to you. Each student will have their own specific data files to work with, with their own unique data format.

This lab must be your own work. However, you may use resources on the Internet to obtain general information including information about the C language and libraries, information about binary and text data formats, and information about the operating system. If you obtain useful information from the Internet, you must include comments at the relevant points in your code acknowledging the source of the information (URL) and briefly describing the key idea(s) that you are using. (Exception: information from the Unix manual pages does not require citation in your program).

The Unix manual pages are available online on ash and iceberg – use the `man` command. You can also find Unix manual pages online through Google. For example, to find out about the `printf` library call, use the command "`man 3 printf`" or Google "man printf" and to find out about the directory listing command '`ls`', use the command "`man ls`" or Google "man ls". However, you should be wary of using information found online because sometimes there are differences between different Unix systems and our systems may not behave exactly the same as described in some online documentation.

The manual pages on the system (`man` command) are divided into sections:

1. System commands such as `ls`, `wc`, etc.
2. Unix system calls such as `read()`, `open()`, etc.
3. Unix library such as `printf()`, `fopen()`, etc.
4. Sections 4-8 contain other information.

For more information on the man command, use the command "`man man`" to read the manual pages about the man command.

Fetching your lab

The lab files are accessed through the lab command which can be found at

```
/home/unit/group/comp202/lab
```

There is no Unix man page for the lab command (it is not a Unix system command) but there is documentation on iLearn and if you don't give it any command-line parameters or options then it will print out some brief documentation itself. This feature is common in many Unix programs. To see how this works, try the following command (where the $ symbol represents the Unix command-line prompt – you should type the command that is underlined in this example).

```
$ /home/unit/group/comp202/lab
```

The option `-g` is used to **get** a lab stage. For example, to get lab 1 stage 1, do:

```
$ /home/unit/group/comp202/lab -g 1.1
```

For stage 2, the option would be `-g 1.2` instead. Please see the *Lab Command Manual* in iLearn for more information about the lab command, including options for submitting assignments, getting marking reports, checking due dates and claiming your free extension days. Also, you can set up your Unix account so that you can abbreviate the command and just type "`lab`" instead of the full path name "`/home/unit/group/comp202/lab`". For the rest of this document, we will use the abbreviated name.

The lab get command downloads your lab data as a tar file. For stage 1, the tar file is `stage1.tar`.

Tar is an archive utility (like zip) – it stores many files packed into one file. Use `tar` to extract the contents of this file. You can read all about tar in the Unix man page

```
$ man 1 tar
```

Here is the command to extract the contents of `stage1.tar`.

```
$ tar xvf stage1.tar
```

This will create a directory called `stage1` and put the downloaded files in that directory.

Submitting your lab solution

Your lab solution can be submitted using the `lab` command. The option `-s` is used to submit a solution to a lab stage. After the option, list all the files that you want to submit. Each time you submit, it is treated as a fresh submission, so you must list all the files that you want to submit every time. (If you find that tedious, learn about wildcards in the bash shell.) For example:

```
$ lab -s 1.1 stage1.c sub.c defs.h
```

The `lab` utility sends your submitted files to a server which compiles the C files together into a program, runs it, and tests that it works correctly for your particular lab assignment. The server records information about your submission and also sends back information to you through the `lab` command.

You can submit as many times as you like. As a matter of personal achievement, you should aim to achieve a really good score on your initial submit, having checked that your program compiles without errors and performs correctly on the provided sample data files. However, if there are problems identified by the auto marker, you can resubmit without penalty.

You **must** download each stage before you attempt to submit a solution to that stage. You need to download each stage anyhow because the download provides you with the input and output data files that you need in order to test your program yourself.

Progress marks [3 marks]

Each lab assignment includes marks that are awarded for progress on the task each week. The lab assignments are to be done both during lab sessions (with the assistance of lab supervisors) and in your own time. Each week that the lab is out, you earn a progress mark if you achieve the specified milestone by 12 noon on the specified date. You can earn the progress marks early, but you cannot earn them late.

If you do not achieve the milestone for a progress mark by the specified date then you lose that week's progress mark and the milestone "slips" and becomes due on the next progress date. All the later milestones also slip back by one week, but the last milestone is lost. If you achieve the slipped milestone by the new progress date then you receive the progress mark for that date, but you have lost the progress mark for the missed date and you cannot make it up later.

Milestones

- Monday of Week 4: Stage 1 achievement mark of at least 2.3/3.0
- Monday of Week 5: Stage 2 achievement mark of at least 2.3/3.0
- Monday of Week 6: Stage 3 achievement mark of at least 2.3/3.0
- Monday of week 7: Lab closes

Detailed marking guides for each stage

When you extract the files for a stage you will also find a file called `marking-guide.txt` in the extracted files. This text file contains a detailed marking rubric for the stage. The auto marker uses this rubric to mark your submission for the stage. The marking guide includes detailed notes that describe how each mark is calculated and what is being marked.

In later stages, some auto marker checks are thresholds. Threshold conditions may not earn you marks, but are required for your program to be eligible to earn other marks. The marking report will display if any threshold has failed, and it will indicate which marks are suppressed due to the failed threshold. Thresholds and marks that require thresholds are indicated in the marking guide `marking-guide.txt`.

Stage 1: Initialising a C struct and printing it out as text [3 marks]

In this stage you will declare a C data structure, create an instance of it and statically initialise it (declare it as a static or global variable and initialise it in one statement using braces). You will then print out the instance. This stage develops the following specific skills:

- Declaring a C `struct`.
- Initialising a C `struct`
- Printing various data types using `printf`

Note: Do *not* use bit fields in your `struct`. All the data types that are specified correspond to ordinary C data types.

Note: The automarker checks your `struct` definition against expected ways of writing it and awards marks for correctness. Field names must be exactly correct. Types should be the common C language data types as defined in ANSI C.

Resources

The following documents on iLearn may be helpful:

- *Compile, Run, Make C Programs on Linux*
- *C Programming Notes for Data File Lab*

Your downloaded `stage1.tar` file contains the following files:

- `filestruct-description.txt`: A simple description of the fields that are in your struct – their names and type description.
- `initialisation-specification.txt`: Specifies the initial value for each field of your struct. The initial value has to be formatted in a specific way in your source code – this may mean that you have to convert one representation to another. See the lab note *Decimal, Binary, Octal and Hex*. Note: It makes no difference to the data that is stored inside the computer whether you initialise the field with decimal or the equivalent hexadecimal or octal. However, as an exercise, we require you to make the appropriate type conversions and the automarker will check your code.
- `expected-output.txt`: Stage 1 expected output file. Use the example in this file to work out what formatting options to use in `printf`.

Useful Unix commands

You might find the following Unix system commands helpful.

- `cat`
- `diff`

Task

Write a C program that declares your particular data structure as described in the C structure description file. Statically initialise an instance of the data structure to the initial values as specified in the file – use the data formats as specified in the file such as hexadecimal, decimal or octal constants. In the main program, print out the data structure using `printf` formatting to make it exactly match the provided sample output file. Note that you may need to use various formatting options with `printf` to control the appearance of the output. You are expected to read about `printf` and work out how to format the data so that it exactly matches the expected output.

Submit your program for automatic assessment using the `lab` command. Your program style may be assessed according to the coding standards in the documents *Some Important Comments on Code Style* and *Systems Programming Style* which are available on iLearn.

Stage 2: Reading a binary data file and printing it out [3 marks]

In this stage you will read a binary data file in a known format, storing the information into instances of a C data structure which you will then print out. This stage develops the following specific skills:

- Reading binary data
- Opening and closing files
- Printing various data types using `printf`.
- Using a simple command-line parameter.

Resources

- `filestruct-description.txt`: Describes the members of the C data struct which correspond to fields in the records of the data file.
- `input-*.bin`: Sample binary input files.
- `output-*.txt`: Sample text output files corresponding to the input files.

Useful Unix commands

You might find the following Unix system commands helpful.

- "`more`" or "`less`"
- `diff`
- `od`

Task

Write a C program that reads a file of binary data records as described in the structure description file. The program will read and print all the records in a binary data file where each record has the format described in `filestruct-description.txt`. You already developed code to print out a single record in stage 1, so the focus of this stage is reading a binary data file into memory.

The output formatting requirements for this stage are the same as in stage 1. However, it is possible that you **may need to modify your record printing code** – it could be that your `printf` call worked correctly for the single initialised record in stage 1 but it may not be correct for all the data records in the files. You should check the output against the expected output using `diff`, and improve your `printf` statement in whatever way is needed to get the correct output.

Your program must accept one command-line parameter which is the name of the input file.

The fields of the records are stored using the types specified in the data file description. The fields are stored packed next to each other in the data file. You cannot read the entire record directly into a C struct in one call because C inserts additional unused space between some of the fields in the `struct` (this is called alignment padding; we will discuss it later in COMP202 lectures)[1]. You must read the data record one field at a time. It is suggested to use `fread` to read each field.

Each record that you read should be printed out as text. Your output should exactly match the sample output files.

Remember that coding style is important: use good modularisation, and use header files appropriately. Your program's style may be assessed according to the coding standards in the documents *Some Important Comments on Code Style* and *Systems Programming Style*.

[1] The C compiler has a special way of creating structs that are packed, but this is a non-standard extension and the automarker does not accept programs that use it.

Submit your program for marking using the `lab` command. We may use additional data files for testing, including files that are larger than the samples provided to you.

Stage 3: Sorting a binary data file [3 marks]

In this stage you will sort files of binary data in a known format. This stage develops the following specific skills:

- Reading and writing binary data files.
- Opening and closing files.
- Working with pointers to structures.
- Memory allocation, dynamically sizing an array.
- Using system library routines (specifically, a system library sort routine).
- Writing code to compare structures with a lexical sort order.
- Using a function pointer in C.

Resources

- `filestruct-description.txt`: Describes the members of the C data structure which correspond to fields in the records of the data file.
- `filestruct-sort.txt`: Specifies the sorting order.
- `input-*.bin`: Sample binary input files.
- `output-*.bin`: Sample binary output files corresponding to the input files. The output files contain the same data as the input files, but the records are sorted.

Useful Unix commands

You might find the following Unix system commands helpful.

- `od`
- `cmp`

Task

Modify your program from stage 2 so that it reads the input file (parameter 1), storing all the records into a dynamic array in memory. The program should then sort the data records and write the output file (parameter 2) in sorted order.

Use the Linux library sort routine `qsort` to perform the sorting. Use the Unix manual (section 3) to find out how to call the `qsort` library routine. Hint: you must write a comparison routine that can compare two structures according to the sort order specified for your lab.

Your program will need to store all the records in memory in order to sort them. The program will allocate a dynamic array of structs (or some other data structure), and read the data file into the array. You do not know how large the file may be, so you must accommodate different file sizes. Here are two possible approaches (there are others).

1. Dynamic sized array: Allocate an initial array of some size (e.g. 100 records) and then if (while reading the file) you find that the array is not large enough then use `realloc` to increase (e.g. double) the size of it. `Realloc` allocates a new larger array in memory and copies the data from the existing array to the new larger array, before freeing the original array. Repeatedly doubling the size allows you to accommodate arbitrarily large data files without copying the data too many times. See the Unix manual pages for `malloc` and `realloc`.

2. Compute the number of records from the file size: This is a systems approach that will require some reading to find out how to achieve. There is a system call `stat` that can tell you the total number of bytes in a file. There are also other ways to find out how many bytes are in a file but you should NOT read the entire file just to find out how big it is! Your file description gives you the information about how long each record is, so you can compute the number of records in the file from the number of bytes. You can then allocate an array of `struct` to the exact correct size using `malloc`. See the Unix manual pages for `stat` and `malloc`.

After sorting the records, write them out in binary form. It is suggested to use `fwrite` to write each field individually.

> **Students aiming for D or HD grade:** It is more efficient to sort an array of pointers to the structs than to sort the structs themselves, because it is cheaper to move pointers than to move entire records. Therefore, top marks are awarded for sorting pointers. However, it is suggested to first sort the array itself and then implement pointer sorting if you have time.

The output files must exactly match the sample output files provided.

Remember that coding style is important: use good modularisation, and use header files appropriately. Your program's style may be assessed according to the coding standards in the documents *Some Important Comments on Code Style* and *Systems Programming Style*.

Submit your program for auto marking. We will test your program on additional sample files that have not been provided to you.

Lexical sorting

The records are to be sorted according to the values in various fields of the records. The sort order specification lists the fields that should be considered, and for each field it specifies whether that field is sorted in ascending or descending order. If you are familiar with sorting in Excel, this works similarly.

For example, consider the following simple text file, shown with line numbers. The first line is the header line that gives the name of each member of the record structure.

```
1.   horse, cat, paper, train
2.   3, word, 1.4, 1
3.   4, wood, 1.4, 1
4.   1, word, 1.7, 0
5.   2, word, 1.5, 0
```

Suppose that this file is sorted in the following way: First, by train in ascending order, then by cat in descending order, then by paper ascending and finally by horse descending. The data to be sorted is lines 2 through 5. Examining the last column (train), lines 4 and 5 have the value 0 whereas lines 2 and 3 have the value 1. Therefore, lines 4 and 5 will be sorted before lines 2 and 3. Now, comparing lines 2 and 3, which have the same value for train, the values for cat are different. Sorting these records by the field cat in descending order, “wood” should come after “word” because the sort is reverse of alphabetical, so record 3 is to be sorted after record 2. Finally, comparing records 4 and 5, they are the same for fields train and cat, but differ in the field paper which is to be sorted ascending. Record 4 is therefore sorted after record 5.

The final sorted text file is:

```
horse, cat, paper, train
2, word, 1.5, 0
1, word, 1.7, 0
3, word, 1.4, 1
4, wood, 1.4, 1
```

Note: You are expected to use a system library sorting routine, not to write your own sorting algorithm.

Stages 4 and/or 5: Reading and sorting an unknown data format [3 or 4 marks]

In the final stage(s) you will reverse engineer an unknown file format containing the same data fields that you are familiar with but stored using different representations. These stage(s) develop the following specific skills:

- Recognising binary data formats
- Interpreting and converting binary data formats
- Exploring binary files with dumping tools
- Reading and writing binary data files
- Converting data from one format to another

Options for your final stage

You have the choice of which stage(s) to attempt to complete this assignment. The following are suggested guidelines, but the choice is entirely yours.

- **Most students** should complete stage 4 as the last stage of this assignment. This option is the easiest option. You can earn at most 3.0 marks for stage 4. You can still achieve a very good total mark for the lab.
- **Students aiming for HD grades** may choose to skip stage 4 and complete stage 5 as the last stage of this assignment. This option may be the most difficult option. You can earn at most 4.0 marks for stage 5.
- **Students aiming for D or HD grades** may first complete stage 4 and then attempt stage 5 as the last stage of this assignment. This option is the most work because the input files for stages 4 and 5 are completely different. Stage 4 is marked out of 3.0 and stage 5 is marked out of 4.0, but your final mark will only include either your stage 4 mark or your stage 5 mark – whichever is greater. For example, if you complete stage 4 and earn 2.9 marks and also earn 3.5 marks for stage 5, your final mark will include the 3.5 marks for stage 5 but not the 2.9 marks for stage 4. On the other hand, if your stage 4 mark was 2.9 and your stage 5 mark was only 2.2 then your final mark would include the 2.9 marks earned for stage 4 but not the 2.2 marks for stage 5.

You can download both stages using the `lab` command and then decide which stage you want to attempt first. You can change your decision at any time, but the structures of stages 4 and 5 input files are completely different so there will be additional work involved if you work on both stages.

Resources

- C structure description
- Data file description
- Sort order specification
- Stage 4 or 5 sample input and output files. The output files contain the same data as the input files, but the output files are converted to the original file format. For stage 4, the output files are not sorted – this makes it possible to award partial marks if your program correctly converts only part of each input record. For stage 5, the output files are sorted, and correct sorting requires correctly converting all the fields of the input records.
- Debug output files to assist with stage 4. These files contain the text version of the output files. You should be able to produce the same files by running your stage 2 program over the binary output files, so these files are provided only as a convenience.

Task

Study the provided sample data files. The input files are binary files in a new file format, while the output files are in your known file format (and, for stage 5, the output files are sorted). Your first task is to identify the input file format by comparing the information contained in it with the information contained in the sample files. Fortunately, there are some small files provided which will make it a lot easier to work out what information in the input file corresponds with what information in the output file.

You will need to spend some time examining hex dumps (and possibly other dump formats) of the sample input files and comparing the information in the bytes with what you might expect to find. Feel free to use whatever tools you can find to examine the bytes in your files. The input files and output files contain the same data values, but the binary formats of individual fields are different, so you are looking for the correspondences between the data values in two files. For example, a 16-bit signed integer in your original file format might be represented as 32-bit unsigned or as a floating-point in the new file format.

> **Hint:** On iLearn, there is a separate document *Hints on Reverse Engineering a Data File* which provides helpful hints and suggestions for completing stage 4 and/or stage 5 of this lab assignment.

Once you have identified your input file data record's format, modify your program from stage 3 so that it can read a binary file in the new file format. As each record is read, convert the data into the correct form to store in your C structure. For stage 4, your program will then output the data as an unsorted binary file in the original file format. For stage 5, your program will sort the records and output a new sorted binary file in the original file format. For full marks, the output files must exactly match the sample output files provided, and your program must work correctly on additional test data that we do not provide to you. For stage 4, if you cannot work out how to convert a small number of fields of the input file, you can still earn partial marks by correctly converting the other fields, and simply outputting zeros in the unconverted fields.

Remember that coding style is important: use good modularisation, and use header files appropriately. Your program's style may be assessed according to the coding standards in the documents *Some Important Comments on Code Style* and *Systems Programming Style*.

Submit your program for auto marking. We will test your program on additional sample files that have not been provided to you.

About the New Data File Format

The new format input data files will contain the same information as the original format output data files, but the records of the input files are not sorted. If you are attempting stage 4, the order of the fields in each record will be the same as in the original format. On the other hand, if you are attempting stage 5, the order of the fields in the records will be different from the original format. This makes stage 5 more difficult than stage 4 because you have less knowledge of the unknown file format.

The data formats of individual fields can be different in the new data format compared to the original data format. In particular:

- Numeric fields can be represented as any of the numeric types: signed or unsigned integers in 8, 16 or 32 bits; float or double.
- Booleans can be represented in 8, 16, or 32 bits, or several Boolean fields can be packed as specific bits in a field of 8 bits.
- Strings can have a different length and characters can be converted to strings in the new format.

In cases where a field has a different numeric type in the new format, we guarantee that the sample data values present in the input and output files will semantically correspond (i.e. they will have the same meaning). For example, if the input file has 16-bit signed integers but the output file has 8-bit unsigned integers then the sample data values for that field will all be positive values in the range 0 to 255, since these values can be represented in both 16-bit signed integers and 8-bit unsigned integers. As another example, if the input file has 8-bit signed integers and the output file has 32-bit unsigned integers, then the data values for that field will all be positive values in the range 0 to 127, since these values can be represented as both 8-bit signed integers and 32-bit unsigned integers.

Useful Unix commands

You might find the following Unix system commands helpful.

- `od`
- `cmp`

Code Style [4 marks]

We will mark one of your submitted programs for code style. We recommend that you adhere to code style guidelines for all your programs. See the documents *Some Important Comments on Code Style* and *Systems Programming Style*.

Each time you submit a lab 1 solution using the `lab` command, you will be notified which version of your program will be marked for style. The decision is made by an algorithm (see below). We prefer to mark later stages of the lab where your programs will be more sophisticated. However, we prefer not to mark programs where you have not yet solved the stage.

> **The stage selected for style marking** is the latest stage for which you achieved a mark of at least 2.3. For example, if you earn 3.0 marks in stage 1, 2.8 marks in stage 2, 2.3 marks in stage 3 and 2.1 marks in stage 4, we will mark your stage 3 submission for style. We will mark the last successful or forced submission to that stage. Whenever you submit, the `lab` command clearly tells you what stage will be marked for style, and whether it is the program that you just submitted that will be marked for style.

All your programs should be written with good style. If you write with consistently good style then you won't be caught out with a poor style mark if (for example) you manage to achieve 2.3 marks in stage 4 at the last minute and have no time left to improve the style of your program!

Details of Achievement Marks

All marks are computed to 1 decimal place as displayed in the marking reports that you receive from the `lab` command.

There are three types of achievement marks, as explained in the `marking-guide.txt` files.

- Ordinary marks are proportional, computed as a percentage and scaled to the maximum mark. After scaling, the mark is rounded *down* to a multiple of 0.1. For example, if the percentage mark is 98% and the mark is scaled to a maximum of 1.0, then the rounded mark would be 0.9 (not 1.0). Rounding down ensures that full marks are only awarded for perfect scores of 100% on the particular marking item.
- Error count marks deduct a fixed amount (usually 0.1 or 0.15) from the maximum mark for each error that is counted, until the mark reaches 0.0. Error count marks are typically used for error checking such as checking your structure definition.
- Boolean marks are used for test conditions which are either success or failure. The mark is awarded either as the full mark or as 0.0. The full mark is awarded when the test condition is satisfied, and 0.0 is awarded when the test fails. Boolean marks typically have small values (such as 0.1 or 0.2) and are awarded for specific tests such as ensuring that your program exits without an error status in normal operation.