HPPS 2024

Assignment Two

Interpreting Data

Due: 2024-12-08

Synopsis: Operate on numeric data stored in files.

1 Introduction

This assignment is about reading and writing numbers in various data formats. The goal is to demonstrate that you understand the difference between representation and interpretation, how to work with byte-oriented representations, and how to perform binary IO.

You will be implementing a library, `numlib`, that contains functions for reading and writing numbers in a range of data formats. This library is an extension of the one developed at the exercises. You are given a (rather large) set of programs that use this library to read data passed on *standard input*, and write results on *standard output*.

1.1 File formats

This assignment makes use of the following file formats. Reading and writing functions for some of them have already been implemented in the code handout, while implementing the rest is part of the assignment. The file formats are not self-describing, which means that we cannot just by looking at a sequence of bytes decide which format they should be interpreted as. Indeed, a sequence of bytes may be *interpretable* as a valid encoding of data in multiple formats.

ASCII integers

A file containing non-negative base-10 integers represented with ASCII digits (as by the `isdigit()` function). The numbers may be separated

by whitespace (as by the `isspace()` function). No other characters are allowed. Example of a valid file encoding three numbers:

```
0 123
456
```

A single number is described by the following regular expression[1]:

```
[0-9]+
```

ASCII decimal numbers

A file containing non-negative base-10 decimal numbers represented as optionally a sign (`'-'`), followed by one or more ASCII digits (as by the `isdigit()` function), followed by a full stop, followed by one or more ASCII digits. The numbers may be separated by whitespace (as by the `isspace()` function). No other characters are allowed. Example of a valid file encoding three numbers:

```
0.0 -123.456
0.890
```

A single number is described by the following regular expression[2]:

```
-?[0-9]+\.[0-9]+
```

Little-endian unsigned 32-bit integers

A file containing unsigned 32-bit integers (`uint32_t` in C) in little-endian byte order. Each integer is represented as four bytes. Any file whose size in bytes is divisible by four thus constitutes a valid sequence of numbers.

Big-endian unsigned 32-bit integers

A file containing unsigned 32-bit integers (`uint32_t` in C) in big-endian byte order. Each integer is represented as four bytes. Any file whose size in bytes is divisible by four thus constitutes a valid sequence of numbers.

[1] You are not expected to know what this is.
[2] You are still not expected to know what this is.

Binary floating point numbers

A file containing double-precision floating point numbers (`double` in C) in native byte order (on all machines you likely have access to, this means little-endian). Each number is represented as eight bytes. Any file whose size in bytes is divisible by eight thus constitutes a valid sequence of numbers.[3]

2 Your task

Your task for this assignment is to implement various additional functions in `numlib.c`.

2.1 Implement `read_uint_be()` and `write_uint_be()`

These are very similar to `read_uint_le()`/`write_uint_le()`, but read and write *big endian* rather than little endian integers.

Hints

- The actual IO logic is very similar to the little endian versions. Just decompose and assemble the `uint32_t` values differently.

2.2 Implement `read_double_bin()` and `write_double_bin()`

These are very similar to the binary IO functions you implemented for integers.

Hints

- Remember that our binary format for `double` always uses native byte order (little endian).
- `sizeof(double) == 8`.

[3] Although some of these may encode NaN, which is by definition *not a number*. That doesn't matter here.

- This can be implemented with a single call to `fread()` (although that is not the only correct way to do it).

2.3 Implement `write_double_ascii()`

This is very similar to the ASCII output function for integers.

Hints

- Unless you want to earn a place in Valhalla, definitely use `fprintf()` for this one. Implementing printing of floating point numbers from scratch is decidedly nontrivial.

2.4 Implement `read_double_ascii()`

This function reads a single number in the ASCII decimal format described above. Returns EOF if the provided file has no characters left. Returns 1 if an ASCII decimal number cannot be read.

Hints

- Remember to check whether the first character is a sign (`'-'`) and put it back with `ungetc()` if not.
- Use `read_int_ascii()` as a helper function.

2.5 Implement a program `avg_doubles`

Add a new file `avg_doubles.c` and modify the Makefile such that it compiles this file to a program `avg_doubles`, similar to the other programs. This program must read binary double-precision numbers on standard input until EOF, compute their arithmetic mean (average), and print that mean as a double-precision number in ASCII format, followed by a newline, to standard output.

Hints

- You will need to use `read_double_bin()` and `write_double_ascii()`.
- Consider should happen if the input contains no numbers.

Example

```
$ echo 1.2 2.3 3.4 4.5 5.6 6.7 | \
  ./double_ascii_to_double_bin | \
  ./avg_doubles
3.950000
```

3 Code handout

`Makefile`: How to build the source files. Already contains rules for various test programs, *except* `avg_doubles`. You may modify this file to add more test programs.

`numlib.h`: The header file with all necessary prototypes. Do not modify this file.

`numlib.c`: Implementations of the functions declared in `numlib.h`. In the handout, most of these immediately crash. Please do modify this file so that they work.

Various other test programs: These programs read data in some format on standard input, and write converted data on standard output. Initially not many of these will work, but as you implement more of `numlib.c`, they will gradually become operational.

4 Your Report

You are expected to comment on the *interesting* details of your implementation. You are *not* expected to give a line-by-line walkthrough of your code. Most importantly, you are expected to reflect on the *quality* of your code:

- Do you think it is functionally correct? Why or why not?
- Is there some improvement you'd have liked to make, but didn't have the time?

It is more important to be aware of the strengths or shortcomings of your solution, than it is to have a complete solution.

4.1 The structure of your report

Your report should be structured exactly as follows:

Introduction: Briefly mention very general concerns, your own estimation of the quality of your solution, and possibly how to run your tests.

A section for each of the five tasks: Briefly mention whether your solution is functional, which cases it fails for, and what you think might be wrong.

A section answering the following specific questions:

1. How much larger are data files in the ASCII integer format, compared to the binary integer format?
2. How much larger are data files in the ASCII floating-point format, compared to the binary floating-point format?
3. Would it be possible to add a function `read_uint()` that accepts *both* the binary and ASCII integer format; deciding the interpretation based on the actual input? Explain why or why not.
4. Are there any floating-point values that can be expressed in the binary format, but not the ASCII format?

All else being equal, **a short report is a good report**.

5 Deliverables for This Assignment

You must submit the following items:

- A single PDF file, A4 size, no more than 5 pages, describing each item from report section above.
- A single zip/tar.gz file with all code relevant to the implementation, including at least all the files from the handout. For this assignment it is not necessary to add additional files.

Remember to follow the general assignment rules listed on the course homepage.