**CPSC 261: Lab 2**

* [Goal](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#goal)
* [Introduction](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#introduction)
* [Overview](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#overview)
* [The Details](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#details)
* [Provided Materials](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#provided)
* [Deliverables and Handing In](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#deliverables)

**Goal**

The goals of this lab are to:

* get you more familiarity with the Linux environment that you will use for the rest of the labs in this course,
* gain some experience with pointers in C
* enhance your C programming expertise
* Practice using UNIX File I/O.

**Introduction**

In today's highly interconnected world, many computer programs are required to share data with other programs running on other computers. This sharing of data can be a source of difficulty when the programs are written in different programming languages or the computers on which they run have different data representations. An External Data Representation (often abbreviated XDR) is a specification of how data should be represented so that it can be universally understood by every program running on every computer. One way to solve this problem is to convert all of the data into text and encode it in some standard way. A very popular encoding standard for text data is XML. However, converting data to text in XML wastes a great deal of space and processing time. Therefore, we are going to encode binary data so that it can be universally understood.

**Endian-ness**

When computers store values that are larger than a single byte they must decide in which order the multiple bytes that make up the value are stored in memory. This determination is known as the "endian-ness" of the computer and is discussed in Sidebar 4.3 on page 158 of the text. You should read it. The Intel x86 family of processors are little-endian. The Sun/Oracle Sparc family of processors are big-endian. In order to communicate binary data effectively, programs must be able to determine the endian-ness of the computer on which they are executing as well as the endian-ness that has been used to encode the data that they are reading.

**Unix file I/O**

The standard C library defines a set of functions for reading and writing files. These include:

* FILE \*fopen(char \*filename, char \*mode);

Open a file and return a "handle" that can be used to read and write from the file. When you want to open a file for reading binary data, the mode that you use is "rb". The "r" indicates your desire to read from the file, and the "b" indicates your desire to read binary data (as opposed to textual data) from the file.

* int fclose(FILE \*file);

Close the file when reading and writing is complete.

* int fread(void \*ptr, size\_t size, size\_t nitems, FILE \*file);

Read some number of values from the file. The size of each value is in size, and the number of values to read is in nitems.

* int fwrite(void \*ptr, size\_t size, size\_t nitems, FILE \*file);

Write some number of values to the file. The size of each value is in size, and the number of values to write is in nitems.

* int feof(FILE \*file);

Returns true (1) when the end of an input file has been read. Note that feof only returns true after a failed read has been attempted. It will not return true after you read the last item from the file; you will have to do one more read - which will fail - before feof returns true.

Documentation for all of these functions is available in any C reference, or use Google: "man Linux fopen", for example.

**Overview**

This lab has a number of parts:

* First you must write a function that takes a long value as an argument and returns as its long result the argument value with its bytes in reverse order. It should be called swapLong. Note that on the x86\_64 computers that we are using, a long value is 8 bytes (64 bits).
* Next you must write a function that takes an int value as an argument and returns as its int result the argument value with its bytes in reverse order. It should be called swapInt. Note that on the x86\_64 computers that we are using, a int value is 4 bytes (32 bits).
* Next you must write a function that determines the endian-ness of the computer on which it is invoked. It is to be called isLittleEndian and should return true (or 1) if the computer is little-endian and false (or 0) if the computer is big-endian.
* Next you must use the provided test programs that call the previous 3 functions and ensure that they work correctly.
* Next you must write a set of functions that can read binary data from a file. For this lab, we will require only that you support 5 data types:
  + characters
  + strings
  + ints
  + longs
  + pointers
* Finally, you must run the driver program that we have provided that uses the functions above to read and decode all of the data from a given input file, and ensure that it executes correctly.

**The Details**

* [Step 1](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#step1)
* [Step 2](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#step2)
* [Step 3](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#step3)
* [Step 4](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#step4)
* [Step 5](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab2.html#step5)
* Step 1 - Implement swapLong() The prototype for the function is

long swapLong(long x)

This function should be defined in a source file named swapLong.c. If you think about the arguments and results in hexidecimal, it is easy to see what the function should do. Consider a long value whose hex representation is:

0x0102030405060708

Since a long is 64 bits, and since each hex digit encodes 4 bits, encoding a long value requires 16 hex digits. When you reverse the bytes in this value, you should get the long value:

0x0807060504030201

This function should not manipulate data in memory or use pointers. The C operators that are most likely to be useful to you are: <<, >>, &, |.

We have provided a testing program called testSwap.c that exercises your swapLong() function with a small set of values. If it passes our tests, it is likely that it works correctly. You will need to at least have a "stub" for the swapInt() function described in step 2 in order to compile our test program.

Note that a "stub" for a function is a function that has the same argument and return types as the function for which it is a stub, but has no useful body. Normally a stub just returns an arbitrary constant of the right type (0 is my personal favourite arbitrary constant).

* Step 2 - Implement swapInt()

This is just like swapLong, except that it manipulates a 4 byte int rather than an 8 byte long. This function should be defined in a source file named swapInt.c.

Once you have implemented swapInt() all of the tests in our provided testSwap.c file should pass.

* Step 3 - Implement isLittleEndian()

The prototype for the function is:

long isLittleEndian(void)

This function takes no arguments and returns a long (either 0 or 1) indicating whether the current computer represents data in little-endian format.

You should write this function in C and put it in a file named isLittleEndian.c.

The key for doing this is to look at a single multi-byte value (an int or long for example) as a sequence of bytes. You will need to use pointers in a type-unsafe way to accomplish this.

If you want to figure out how to do this on your own, then don't read the following hint!

If you store an int in memory, each of whose bytes hold a different value, such as 0x01020304, then the value of the first byte in memory can tell you whether the int has been stored in little-endian or big-endian fashion.

If you are trying to avoid the hint, start reading again here.

We have provided a testing file named testEndian.c, that will call your isLittleEndian() function and print out a message. It should print either:

This computer is little endian

or

This computer is big endian

based on what isLittleEndian() returns.

Test your program. All x86\_64 computers (including the ones in the lab) are little endian. However, you should try to make sure that your program "does the right thing" on a big-endian computer. How can you test your program without a big-endian computer to run it on?

* Step 4 - Implement the read functions

There will be 5 read functions

* + void readChar(FILE \*f, int swap, char \*c);
  + void readString(FILE \*f, int swap, char \*c);
  + void readInt(FILE \*f, int swap, int \*c);
  + void readLong(FILE \*f, int swap, long \*c);
  + void readPtr(FILE \*f, int swap, void \*\*c);

Each one reads the corresponding value from the input file, and based on the value of the swap parameter swaps the bytes after they have been read. The value is written to the variable whose address is provided in the c parameter.

All 5 of these read functions will use the standard C input function fread to read the necessary data from the file. The documentation of the fread function can be found [here](http://linux.die.net/man/3/fread).

A single file named readData.c should contain the implementation of all 5 of these functions.

**Notes**

* + The swap parameter is unnecessary and irrelevant for readChar and readString. It is included in these functions to increase their similarity with the other three functions.
  + A string consists of all of the characters from the input file up to and including the null character '\0' that is used to terminate strings in C. This is the only function that will read a variable number of bytes from the input file. The readString function assumes that the caller has passed a pointer to a large enough buffer to hold the string that is being read.
* Step 5 - Test your read functions using the provided decode program

We have provided a driver that calls your read functions to decode files stored in either the big or little endian data formats. The program takes a single command line argument, the name of the file to read. Upon opening the file the program will read an int (4 bytes). The value of this int determines if the contents of the remainder of the file are to be interpreted using big endian or little endian rules. The first 4 bytes will be the 32 bit int value 0x01020304 in the natural byte order for the host that wrote the file. From this the driver program figures out whether the file is in big-endian or little-endian format.

After the first 4 bytes, the remainder of the file consists of a sequence of bytes that are logically grouped into triples. A triple consists of a 1 byte value that identifies the type of the element, a 4 byte value (the repeat count) stored in the endian format of the file, that indicates how many elements of the type indicated by the first element are in the triple, and finally the data itself. Depending upon the type and the repeat count, the final element of the triple will occupy a certain number of bytes and then the next triple will start. The allowed types for the tuples are character, string, int, long, and pointer. Characters are stored in single bytes, strings are stored as null terminated strings of bytes (no null bytes are allowed in the middle of a string), ints are stored as 4 byte quantities in the byte order of the file, longs are stored as 8 byte quantities in the byte order of the file, and finally pointers are stored as 8 byte quantities in the byte order of the file.   
The type codes are:

* + 1 for a character
  + 2 for a string
  + 3 for a int
  + 4 for a long
  + 5 for a pointer
  + Any other value is an error.

As an example, the triple for a single 2 stored as a long, in a little endian file, would be (in hex)

04 01000000 0200000000000000

(the spaces are shown here for clarity - they will not be in the file. In a big endian file, the same triple would be (in hex)

04 00000001 0000000000000002

Three test files been provided for you to try our program on to test our read functions. The first two of them, little.dat and big.dat, both have the content that is described below. They differ only in that they were created on a little-endian machine (little.dat) and on a big-endian machine (big.dat). If you want to see what the contents of the files look like at the byte level then use the od command with the -x option to print the contents of the file in hex format. More details on using the od command can be found in the man pages.   
The decoder program should produce output that follows this example:   
Sample session:   
   deas:~/ > decode big.dat   
   This computer is little endian  
   The data file is big endian  
   string: abcdefghijklmnopqrstuvwxyz   
   int: 2   
   long: 4   
   long: -4   
   pointer: 0xfffe0024540   
   string: zyxwvutsrqponmlkjihgfedcba   
   deas:~/   
  
From the sample output, observe that the computer is little endian, while the file uses the big endian format.   The file big.dat contains 2 strings (a-z, z-a), an int (2), 2 longs (4, -4), and a pointer (0xfffe0024540). When you print a number you just print it out.

If our test program doesn't provide the output detailed above, then your read functions are the most likely suspects. Check them carefully.

You should not have to modify our driver program or testing programs, but you should read them and make sure that you understand what they do. You will be writing some similar programs next lab.

**Provided Materials**

You should use subversion to check out the lab2 directory from your repository, in the same way that you did for lab1.

Your lab2 directory should contain the following files:

* + Makefile

The provided Makefile contains instructions for compiling and linking your programs. You should not need to, and should not, change the Makefile that we have provided for you.

* + endian.h

This file defines prototypes of the functions that you will be implementing. All of your source program files should:

#include "endian.h"

* + testSwap.c

Our test driver to test your swapLong() and swapInt() functions.

* + testEndian.c

Our test driver to test your isLittleEndian() function.

* + decode.c

Our driver program to exercise your read functions.

* + little.dat
  + big.dat

These are two sample input files that your decoder should be able to decode and print. They contain the same information, but are encoded in opposite endian-ness.

* + complex.dat

A little bit more complex data file.

You will need to create all of the other files.