

Homework 6- MPCS 51040

(last modified: May 9, 2017)

Issued: May 8, 2017

1 General Instructions

1.1 Compiling

Your code must compile with `gcc -std=c11 -Wall -pedantic`. There should be no warnings or errors when compiling with `gcc` (as installed on `linux.cs.uchicago.edu`).

1.2 Handing in

To hand in your homework, you need to commit all requested files (with correct filenames!) to your personal git repository.

Make a subdirectory called 'homework/hw6' and place your files under that directory. Don't forget to commit and push your files! You can check on <http://mit.cs.uchicago.edu> to make sure all files were committed to the repository correctly.

The deadline for this homework is May 15, 2017. To grade the homework, the contents of your repository at exactly the deadline will be considered. Changes made after the deadline are not taken into account.

1.3 Code samples

This document, and any files you might need to complete the homework can be found in the git repository <https://mit.cs.uchicago.edu/mpcs51040-spr-17/mpcs51040-spr-17/>.

1.4 Grading

Your code will be graded based on the following points (in order of descending importance):

- Correctness of the C code: there should be no compiler errors or warnings when compiling as described in 1.1. There should be no memory leaks or other problems (such as those detected by `valgrind`).
- Correctness of the solution. Your code should implement the required functionality, as specified in this document.
- Code documentation. Properly documented code will help understand and grade your work.
- Code quality: your code should be easy to read and follow accepted good practices (avoid code duplication, use functions to structure your program, ...). This includes writing portable code (which will work on both 32 bit and 64 bit systems).
- Efficiency: your code should not use more resources (time or space) than needed.
- Proper use of Makefiles

2 Assignment

For homework 6, we will be writing a header file and corresponding implementation for accessing the `http://yann.lecun.com/exdb/mnist/` MNIST database of handwritten digits. These digits are typically used to train and test machine learning algorithms to recognize digits. For this homework, we only care about accessing the data in a format we can later use; this is not a homework about machine learning.

2.1 Preparation

Go to `http://yann.lecun.com/exdb/mnist/` and download the 4 files (train and t10k images and labels). These files are compressed using gzip. (You can uncompress them using the `gunzip` command on `linux.cs.uchicago.edu`). Note the warning about some browsers decompressing the files!

Note: **You do not need to add these files to your repository when handing in your homework!**

Read the page (you can ignore details about classifiers – this homework is about accessing the data, not about machine learning).

In particular, make sure you fully understand the section ‘FILE FORMATS FOR THE MNIST DATABASE’, as this is a description of the structure of the files that we will be accessing.

2.2 Byte Order

When we have a datatype which consist out of more than one byte, such as `int`, the C standard does not specify in which order these bytes have to be stored in memory. For example, consider a 32 bit integer on a machine which uses 8-bit wide memory locations. A 32-bit integer stored in memory will take up 4 8-bit bytes. One way of storing this integer would be to start with the most-significant bits of the 32 bit integer and store the first 8 bits into the first memory location (let’s say address 1000). The next 8 bits of the integer would then go into memory location 1001, and so on. An alternative way of storing this integer could be the reverse: start with the least significant 8-bits and store those at memory location 1000, followed by the next 8 bits and so on. The former is called big-endian byte order, while the latter is called little-endian byte order.

Note that *the 8 bit sequences are not reversed*; Only the order of each 8-bit group is changed. In other words, in little endian, bit 0 of the 32 bit integer (i.e. the least significant bit of the integer) will also be the least significant bit of the byte in memory containing that bit. For example, the integer 305419896, which in can be represented in hexadecimal as `0x12345678` or `0001 0010 0011 0100 0101 0110 0111 1000` in binary would, in little endian, store the least significant bits first: `0111 1000`, followed by `0101 0110` in the second byte, etc.

A well written, portable program should function correctly regardless of the byte-order used by the compiler and machine the program runs on.

The C11 standard lists a number of possible ways an integer can be represented as a binary sequence – see section 6.2.6.2 in the C11 standard) – regardless of how that binary sequence is subsequently distributed over a set of bytes. However, for this homework, you can assume that for our C compiler, integers are stored in two’s complement mode, and that this matches the encoding (but not byte order!) of the integers stored in the MNIST database. (See https://en.wikipedia.org/wiki/Two's_complement for a discussion of two’s complement encoding.)

Most-significant byte first is also called network byte order. You can use the `hton` and `ntoh` family of functions (defined in `arpa/inet.h`, see the manpage for `htonl`).

2.3 MNIST Database

We want to create an easy way to access the images provided by the MNIST database. The public part (i.e. the header file) is already provided for you. It contains a description of how each function should behave, which arguments the function takes and what the return value should be. You are not allowed to modify `mnist.h` in any way. Your task is to write the corresponding `mnist.c` file.

When opening an existing dataset, your code must load all images into memory before returning from `mnist_open`. Furthermore, *the images must be stored as a linked list*. (These requirements are for educational purposes)

DO NOT MODIFY `mnist.h` or `mnist2pgm.c`!

Some hints and tips:

- The data files are binary files; So far, we've dealt with text files. Consult the `fopen` manpage for information on how to open a binary file.
- The header `stdint.h` provides fixed-width integer types which will ensure that your code is portable and does not depend on a specific compiler and platform.
- You can use functions from `string.h` to construct the filenames. Make sure to use the *safe* versions, i.e. ensuring that no out of bounds accesses to the output buffer will be made.
- You can use `assert` (from header `assert.h`) to help you catch problems in your code.
- Your makefile can help automate certain things (such as downloading the database files and unzipping them).

2.4 Unit Tests

You will be testing your code by writing a unit test *for each function in `mnist.h`*. Your unit test code should be in `test_mnist.c`. It is a good idea to write the unit tests first (which you should be able to do based on `mnist.h` and this document. Once you've written your unit test, add the needed functions in `mnist.c` to make the tests compile (but not necessarily pass). At this point you can start implementing each function, using the unit tests as an easy method to test and debug your code.



Where possible, your unit test should exercise the failure conditions of each function as well. For example, trying to open a dataset which does not exist or passing in `MNIST_IMAGE_INVALID` or `MNIST_DATASET_INVALID`.

2.5 Example program using your library

Additionally, an example program is provided which will exercise the functions exported by `mnist.h`. Make sure your `mnist.c` works with the provided test program (`mnist2pgm`). Check to make sure that there are no memory leaks or other valgrind errors when running the test program on the MNIST datasets.

The test program reads an image from the database and writes out a `.pgm` file. (See <http://netpbm.sourceforge.net/doc/pgm.html> for more information about the PGM file format.) You can use the `convert` program on `linux.cs.uchicago.edu` to convert a PGM file into a JPEG or PNG file for viewing.

If your code is correct, the last image of each dataset should look like:

- `train-59999.pgm`: 
- `t10k-09999.pgm`: 

2.6 Handing in

Commit `mnist.c`, `test_mnist.c` and a Makefile for building the `mnist2pgm` program to `homework/hw6`. In addition, commit files `train-36666.pgm` and `t10k-06666.pgm` (which you can obtain by running `mnist2pgm`). *It is not needed to commit `mnist.h` or `mnist2pgm.c` or any of the dataset files, as these should not be modified by you.*

You should commit a single Makefile. The makefile should have at least the following targets:

- `all`: build `mnist2pgm`, `test_mnist`.
- `test`: build **and execute** `test_mnist`

- program name, e.g. `mnist2pgm` etc.: build the specified executable
- `clean`: remove all object files and executables that have been built by the other targets.

The default action when nothing is specified (i.e. executed simply using `'make'`) should be to build the `'all'` target.