

COMP 40 Assignment: Understanding and Using Interfaces

Design for part B only due Monday, September 12 at 11:59PM (parts A and C do not require a written design).
Full assignment (parts A, B, and C) due Thursday, September 15 at 11:59PM.

Please read the entire assignment before starting work.

Purpose

This assignment has four goals:

1. To help you make the transition from C++ programming to the kind of C programming we expect in 40:
 - ◆ Instead of taking basic algorithms and data structures from the C++ Standard Template Library, you will take them from Dave Hanson's C library, which has been **specially modified for COMP 40**.
There is an online Quick Reference guide at <http://ciibook.webhop.net/pdf/quickref.pdf>.
 - ◆ Instead of doing input and output using the `<iostream>` C++ library with its convenient `<<` operator and the `cout` and `cin` streams, you will use the `<stdio.h>` C library, with its less convenient `printf` and `fgets` functions and the `stdout` and `stdin` file handles.
(In C++, the operators `<<` and `>>` are overloaded at multiple types, but C does not support operator overloading, so another approach is needed.)
2. To give you practice in **identifying interfaces that can help you solve problems**
3. To start you thinking about the *interface* as a unit of *design*
4. To introduce you to multiple representations of numbers

Pair programming

As will be discussed in class, pair programming is an effective way to *more than double your programming productivity*. Pair programming is **required** for at least two assignments, and it is **strongly recommended** for all assignments. Pair programming will help most with your *design*. For most of the term, you will choose your own pairs, but for this assignment, pairs will be assigned before the first lab.

Preliminaries

- To add the course binaries to your execution path, run

```
use comp40
```

You may want to put this command in your `.cshrc` or your `.profile`, but to work around a bug in `use` (reported 25 September 2008; still uncorrected as of 28 August 2011) you will need the line

```
use comp40 > /dev/null
```

Without the redirection to `/dev/null` you may have difficulties with `scp`, `ssh`, `git`, and `rsync`.

- You'll want to get a copy of the compile script from the handout on writing compile scripts, which you can do by running

```
git clone linux.cs.tufts.edu:/comp/40/git/intro
```

- **IMPORTANT NOTE:** Like all programming assignments for this class, the programming parts of this assignment are due one minute before midnight before a class day. You may turn them in *up to 48 hours after the due date*, which will cost you one or two extension tokens. If you wish not to spend an extension token, then when midnight arrives submit whatever you have. We are very willing to give partial credit.
- **The design of part B is due early**, at 11:59 PM on Monday, September 12.

COMP40 Assignment: Intro

If you spend an extension token on any part of the assignment, it automatically applies to *all* parts of the assignment.

Part A: Brightness of a grayscale image

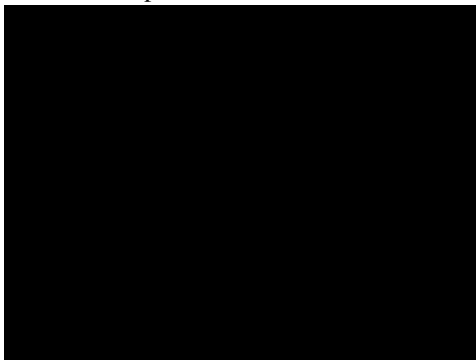
Please write a C program `brightness`, which should print the average brightness of a grayscale image. Every pixel in a grayscale image has a brightness between 0 and 1, where 0 is black and 1 is as bright as possible. Print the *average* brightness using decimal notation with exactly one digit before the decimal point and three digits after the decimal point. Print *only* the brightness.

The program takes at most one argument:

- If an argument is given, it should be the name of a portable graymap file (in `pgm` format).
- If no argument is given, `brightness` reads from standard input, which should contain a portable graymap.
- If more than one argument is given, `brightness` halts with an error message.
- If a portable graymap is promised but not delivered, `brightness` should halt with some sort of error message on `stderr`. (Any message will do, but do print a message on standard error, and don't produce anything on standard output—especially not a senseless answer.)

Examples:

- Here are two photos:



Black cat in coal cellar



Polar bear in snowstorm

If `cellar.pgm` is a picture of a black cat in a coal cellar at midnight, and if `bear.jpg` is a picture of a polar bear in an snowstorm, then output should look something like this:

```
sunfire33{nr}: brightness cellar.pgm
0.000
sunfire33{nr}: djpeg -grayscale bear.jpg | brightness
0.972
```

The first example takes its input from a file named on the command line, and the second example takes its input from standard input, as part of a Unix *pipeline*.

My solution to this problem takes less than 35 lines of C code.

Help with image files

We provide code to help you read image files; you will find the `Pnmrdr` interface in `/comp/40/include` and the implementation in `/comp/40/lib64`. Creating a `Pnmrdr_T` will read the graymap header for you, and from the header you can compute how many pixels are in the image. (You should read exactly as many pixels as are there—no more, no fewer.) Don't forget that the brightness of each pixel is represented as a *scaled integer*, as described in the `Pnmrdr` interface.

Getting images

You can get images to play with by using one or more of the following programs:

- `djpeg` (use the `-grayscale` option)
- `pngtopnm`
- `pstopnm`
- `ppmtopgm`

Problem analysis and advice

The main issues here are:

- In place of much of the C++ technique you already know, you have new C technique to learn. The ideas are all similar, like old wine; C is a new bottle.
- You will have to read and understand the interface for `Pnmrdr`, and you will have to understand a little bit about the `pgm` image format.
- You will have to do something sensible if somebody hands you input that is *not* a portable graymap.
- You will have to understand the compile-time and link-time options that `gcc` needs to work with the `Pnmrdr` interface (`-I/comp/40/include` and `-L/comp/40/lib64 -lpnmrdr`). A good start would be to understand the `compile` script from the libraries handout and to use

```
git clone linux.cs.tufts.edu:/comp/40/git/intro
```

or

```
git clone /comp/40/git/intro
```

to get your own personal copy in an `intro` directory.

- You'll have to deal with **two representations** of real numbers between 0 and 1.

There is also an important issue of style:

- When using an enumeration literal such as `Pnmrdr_gray`, refer to it by *name*, **not** by number.

Part B: Finding fingerprint groups

In this problem you are to write a program `fgroups` (short for "fingerprint groups"), which when given a set of names with fingerprints will identify groups of names that share a fingerprint. The real object of the exercise is **to familiarize you with the CII library** and with C I/O.

Your input is always on standard input. Input is a **sequence of lines** where each line has the following format:

- The line begins with one or more non-whitespace characters. This sequence of characters is the **fingerprint**.
- The fingerprint is followed by one or more whitespace characters, not including newline.
- The **name** begins with the next non-whitespace character and continues through the next newline. **A name may contain whitespace**, but a name never contains a newline.

Finally, you may also assume that **each name appears at most once**.

You may assume that a fingerprint is at most 512 characters long (2048 bits represented in hexadecimal notation), but there is no *a priori* upper bound on the length of a name.

What to do with good input

By the nature of the input, every fingerprint you read will be associated with at least one name.

- If a fingerprint is associated with exactly one name, ignore the fingerprint (and the name).
- If a fingerprint is associated with two or more names, those names constitute an **fingerprint group**. A fingerprint group always has at least two members.

Your program should **print all the fingerprint groups** in the following format:

- If there are no fingerprint groups, print nothing.
- If there is exactly one fingerprint group, print it.
- If there are multiple fingerprint groups, print them separated by newlines.

Groups may be printed in any order.

To print a fingerprint group, print each name in the group. Put a newline after each name, as if by

```
printf("%s\n", name);
```

Names within the group may be printed in any order.

For example, if the input is

```
A   Ron
A   Poppy
B   Bill
A   Dubya
B   Barry
```

Then one possible output is

```
Ron
Dubya
Poppy

Bill
Barry
```

(This example output contains exactly one blank line.)

My solution to this problem takes about 150 lines of C code. About 25 lines deal with input; about 25 lines are devoted to printing output; and about 50 lines are memory management and I/O. The actual algorithm is under 20 lines. The rest is comments, whitespace, `#include` directives, and the like.

What to do with bad input

- If you get an input line that is badly formed, write an error message to `stderr`, discard the badly formed line, and continue.
- If you get a fingerprint of more than 512 characters, you may choose to write an error message to `stderr`, discard the line, and continue—or you may handle the oversize fingerprint correctly.
- Your program should handle any name that can appear in the filesystem; if you get a name that is longer than you can handle, write a suitable message to `stderr`, truncate the name, and use the truncated name with the given fingerprint. **Bonus if your code can handle a name of any length.**
- If a name appears more than once, your program may silently give wrong answers, but it must not crash or commit memory errors. **Bonus if your program issues a suitable error message.**

Performance target

Your `fgroups` program should perform well on **inputs of up to several hundred thousand lines**. On the lab machines, it should be capable of processing a half a million lines per minute.

Problem analysis and advice

This problem boils down to simple string processing and standard data structures.

- C strings are different from C++ strings, and C's string library is nearly useless. You've got two sensible choices:
 - ◆ Roll your own string processing using pointer arithmetic and the standard header file `<ctype.h>`.
 - ◆ Use either the `Str` or the `Text` interface from *C Interfaces and Implementations*.
- When it comes to string comparison, **what you know is wrong**. In C, writing

```
if (s1 == s2) { ... }
```

does *not* compare equality of two strings—it compares equality of *pointers*. To compare two strings for equality, you must write

```
if (strcmp(s1, s2) == 0) { ... }
```

Many experts choose to write this code more briefly:

```
if (!strcmp(s1, s2)) { ... }
```

but the briefer style requires a sharp eye for the exclamation mark.

- Hanson's `Atom` interface maps equal strings to identical pointers, so pointer equality is OK on strings created with `Atom_string` or `Atom_new`. To use strings with Hanson's data structures, you **must** use the `Atom` interface.
- Hanson provides sets, finite maps, and a wealth of sequence abstractions. Any or all of these may be useful for solving the fingerprint problem. Your job is to figure out **what data structures will be useful** and how to combine them into a **clean design**.

Repeat: **the data structures are already built for you**; your job is to figure out which ones will be useful.

Getting started:

- To get immigrated into *C interfaces and implementations*, read Chapters 1 and 2 (pages 1–31) of Hanson's book.
- To learn what Hanson has built for you, skim the beginnings of the relevant chapters: pages 45–52, pages 33–34, pages 103–107, pages 115–118 (pages 118–125 recommended), pages 137–140, pages 161–164, pages 171–173, and the first sections of Chapters 15 and 16, which I don't have the page numbers for.
- Write the design document described in the next section.

Design document for `fgroups`—due Monday, September 12

The heart of a software design lies in its representation of data. For this assignment, we are asking you to identify **what data structures you will need** to compute fingerprint groups, and **what each data structure will contain**.

- Hanson's data structures are *polymorphic*, so you will have to explain what each `void *` pointer will point to.
- Please **describe the invariant** that will hold when you are partway through reading input lines.
- Based on these two explanations, explain briefly how you will compute the fingerprint groups once all input has been read.

Your design document is **due Monday, September 14** at 11:59PM.

Part C: Solving problems with fingerprint groups

The example above is not entirely facetious, but in this problem we ask you **list problems you could solve** with a working version of `fgroups`. Be imaginative. For one set of ideas, look up the man page for `find`, especially the `-exec` option. And it might help you to know that in the arcane jargon of computer science, a fingerprint is sometimes called a "message digest". This information is especially useful if you also know how to use the `apropos` command.

Please include your answer to part C in your `README` file.

General advice for new C programmers

Don't forget to initialize each C pointer variable. In C++, a new pointer variable may be initialized implicitly by a constructor. In C, you should use `NEW` to initialize each new pointer variable. (You might wish to take three minutes to view, or review, *Pointer Fun with Binky*, and compare the code you see there with Hanson's `NEW` macro.)

Use the **programming idioms** on the class web page!

Organizing and submitting your solutions

- On Monday, submit a brief design document that explains what data structures you will use to write `fgroups` and how you will use them. Your document should be a plain ASCII document called `DESIGN` and formatted to fit in 80 columns. If you prefer to use a word processor you can submit `design.pdf`. to submit your design, change to the directory that contains the file and run the `submit40-intro-design` command. For this to work, you will have had to run `use comp40` either by hand or in your `.cshrc` or your `.profile`.
- In your final submission on Thursday, don't forget to include a `README` file which
 - ◆ Identifies you and your programming partner by name and **tells me how to pronounce your names** (my name is pronounced NORE-mun RAM-zee)
 - ◆ Acknowledges help you may have received from or collaborative work you may have undertaken with classmates, programming partners, course staff, or others (at office hours you can ask me about Otis the debugging dog)
 - ◆ Identifies what has been correctly implemented and what has not
 - ◆ Tells me what kinds of problems you can solve using `fgroups` (part C of this assignment)
 - ◆ Says **approximately how many hours you have spent** completing the assignment
- Your submission should include at least these files:

```
README
brightness.c
fgroups.c
```

A carefully designed, modular solution will probably include at least two other files.

- When you get everything working, `cd` into the directory you are submitting and type `submit40-intro` to submit your work.

Avoid common mistakes

In this assignment, the most common mistakes involve producing output in a format other than what the specification calls for. You will learn to read specifications *very* carefully. Consider these questions:

- When a number is called for, how many digits of that number should be printed?
- Which parts of the assignment call for words to be printed, and which call for numbers?
- Does output go to standard output or go to a file?
- Does input come from standard input or from a file? Or both?
- In the output, exactly where should you place blank lines?

Here are some other common mistakes:

COMP40 Assignment: Intro

- When proposing applications of fingerprint groups, it's easy to overlook that a "group" containing only one member might as well not exist.
- It's easy to think that you might have to implement several data structures. All the data structures you need are **already implemented** in Hanson's library.
- It's possible to believe that you have to write code that reads images in the PNM formats. This work is already done for you; just call the functions in the `Pnmrd` interface.
- When your code is working, it's easy to forget to run `valgrind`. Your code should not have any memory errors.