# COMP 40 Assignment: Files, Pictures and Interfaces

## Contents

## 1 Summary of the Assignment

In this assignment you will design, build and test two application programs and one supporting file input routine. The first program reports the average brightness of an image file. The second program reports similarities in file data, and it uses an input routine called `readaline` that you will implement. That input routine must conform to an interface that we provide.

The primary specifications for these programs are contained in the sections Part A: Brightness, Part B: Read a Line and Part C: Similarities in Files below.

Note that for Part B, you get partial credit for a limited implementation that only handles short input lines. **We strongly urge you to implement that limited version first, then go on and complete Part C, and only if you have time go back to enhance Part B for full credit**. You will lose much more credit for doing a poor job on Part C than for failing to handle long lines in Part B.

## 2 Background and preparation

The sections below give information about the purpose of this assignment, as well as other background you should understand before starting work.

### 2.1 Purpose of this assignment

This assignment has several goals:

- To teach you to work in pairs on assignments that are more challenging than you have encountered before
- To help you make the transition from C++ programming to the kind of C programming we expect in COMP 40
- To start you thinking about the *interface* as a unit of *design*
- To give you practice in identifying interfaces and existing implementations that can help you solve problems
- To give you experience reading and conforming exactly to *specifications*, such as those contained in this document
- To start to convince you that writing good test cases, not just for correct or obvious input but also for edge cases and error cases, is as important as writing good program code
- To give you experience *teaching yourself* about languages like C, systems like Linux, and system features like `stdin`, `fopen`, etc.
- To give you experience working with a partner to design, document, implement and test a computer program
- To introduce you to multiple representations of numbers

We understand that assignments like this will take you out of your comfort zone. When you read these instructions there will be many things that at first you won't understand. *Not everything you need to know will be explained to you in detail.*

These are the challenges that professional programmers and computer scientists face every day. Stick with it, figure things out, use the system documentation, get help. You may not succeed completely at everything we ask of you even by the end of the assignment, although it's certainly possible and many students do. Experience shows that over time almost all of you will learn not just to build programs like these, but also to teach yourself about the language features and tools that you need.

## 2.2 Setting up your environment

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`:

```
use -q comp40
```

(Without the `-q`, you may have difficulties with `scp`, `ssh`, `git`, and `rsync`.)

To get started, run:

```
pull-code filesnpix
```

You will get one file: `Makefile`. Use `make` and the `Makefile` to build your programs. Depending on how you structure your code, you may have to make minor modifications to the `Makefile` so that it will compile and link together the correct source modules for your projects. The handout A simple introduction to Compile Scripts and Makefiles (concentrate on the `Makefile` part) should give you the information you need. In later projects you will likely have to make more extensive modifications to `Makefiles`, so you should read and learn about them so you understand what they?re doing. The `Makefile` for Lab 0 and this one are intended as gentle introductions: we will gradually make use of more sophisticated features of `make` as the semester progresses. (You may want to compare this one to the Lab 0 `Makefile` to see what we mean.)

## 2.3 C vs. C++

The C language is for the most part a proper subset of C++. Indeed, C came first and was only many years later extended to add object orientation, parameterized types and other higher-level features. Note that both languages are very widely used to this day.

So, why C in Comp 40? A primary goal of Comp 40 is to teach you how computers work, and to show you how modern software uses the hardware on which it runs. You will find that C, being a smaller and simpler language, translates much more directly to hardware primitives. Furthermore, working in C allows us to build ourselves many of the higher-level features whose implementation is hidden in the runtimes of more complex languages. So, using C we learn more about the internal workings of languages like C++ (and Java and Python, etc.).

If this is your first time programming in C, you may find Mark Sheldon?s Whirlwind Tour of C helpful. Although we will give you some hints like these about the differences between C and C++, we also expect you to teach yourselves many of the details. Other good sources include the books and links on our resources page. Help each other learn! In Comp 40 you must not share work on your solutions, but you are welcome to work with your fellow students to learn new languages and technologies.

## 2.4 Getting started with Hanson's Interfaces and Implementations

Although C itself does not include the sorts of high-level structures like Lists, Sets, or hash-based Tables/Maps you might find in C++ or Java, we in Comp 40 give you implementations by Dave Hanson. We expect you to use these rather than building your own, and we offer them as examples of interesting, well-written C code from which you can learn. **Note: the Hanson `Array` class is not available for**

**use in Comp 40, and is not needed for this assignment.** Of course, you are also encouraged where appropriate to use C language `structs`, `arrays`, etc.

For Part C: Similarities in files you will need several Hanson's structures. You'll also need a general understanding of Hanson's conventions, exceptions for assertions in error handling, etc. for Part A: Brightness.

To get started, 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 on pages: 45–52, 33–34, 103–107, 115–118 (pages 118–125 recommended), 137–140, 161–164, 171–173, and the first sections of Chapters 15 and 16.

### 2.5 Other Preparation

Be sure you have read the course policies. These policies are implicitly part of the specifications for all Comp 40 homework, including this one. Pay particular attention to the following sections:

- General Expectations for Comp 40 Homework
- Collaboration and Academic Honesty Policy
- Policies on Pair Programming
- Course Coding Standards
- Comp 40: Introduction to Exceptions

We emphasize again that pair programming is **required** for all Comp 40 assignments. **It is your responsibility to understand and abide by the Comp 40 policies on pair programming** . If you have not been assigned a partner or do not have permission to work separately (such permission is very rare), please contact the instructor immediately! You will get no credit if you work alone unless you have permission.

## 3 General Advice

This is a big assignment, and it's your first assignment. You will need to balance working in an orderly way with looking ahead so that you can plan your time and overlap thinking about some of the harder problems in Part C with some of the development work for Part A and Part B. As such, we offer the following pieces of advice:

- Read this document carefully! It contains the answers to many of the questions that you will have. TAs will often point you back to this document if you ask a question that is answered in these pages. If you have read this document carefully and still don't understand what is being asked of you, ask on Piazza or talk to a TA in the labs.
- Make an annotated copy of these instructions that you and your partner can share (either on paper or online), noting things you don't understand, and indicating specific areas that will require research, design work, etc. Note in your annotations everything that you are required to submit, and every specific case that you must cover. You will not be turning in your annotated copy, but it will serve as a valuable checklist as you work through the assignment.
- **Start early!** The implementation plan that we give you in Part D begins with some very basic steps that all of you should be able to accomplish quickly.
- Test, test, and test some more. Do not underestimate our ability to throw weird test cases at your submission. Think weird, test weird!

## 4 Part A: Brightness of a grayscale image

Please write a C program `brightness` that prints 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.

### 4.1 `brightness` specification

Print to standard output a single newline-terminated line containing the *average* brightness of the supplied image. The brightness should be printed in decimal notation with exactly one digit before the decimal point and three digits after the decimal point.

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 (see below).
- If a portable graymap is promised but not delivered, or if the supplied graymap has a pixel count of zero, `brightness` halts with an error (see below).
- Upon successful completion, your program must terminate with an exit code of `EXIT_SUCCESS` (from `stdlib.h`). This is true of all programs you write in this course unless otherwise specified.

Where the specification above requires that you halt with an error, you have two choices:

- You may print an error message on `stderr` and exit your program using `exit(EXIT_FAILURE)` from `stdlib.h`
- You may exit with a failed assertion or other Hanson-style Checked Runtime Exception (see explanation of CREs in the Hanson book)

Furthermore, you may rely on code that you have been given to do the same. Note: the Comp 40: Introduction to Exceptions may be very helpful in understanding these options for dealing with errors. When exiting due to such an error you **must not** produce any output on `stdout`.

### 4.2 Examples of Brightness in Use

Black cat in coal cellar

Polar bear in a snowstorm

See the photos on the previous page. If cellar.png 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 like this (of course, the prompts will be different, but your output should be just like this):

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

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*.

Megan's solution to this problem takes fewer than 50 lines of C code.

## 4.3 Help with image files

We provide code to help you read image files; you will find the `Pnmrdr` interface in `/comp/40/build/include` and the implementation in `/comp/40/build/lib`. If you use the supplied `Makefile`, these should be found automatically when your code needs them. 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.

## 4.4 Getting image files

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

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

## 4.5 Problem analysis and advice

The main issues here are:

- In place of much of the C++ technique you already know, you have new C techniques 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 appropriate if somebody hands you input that is *not* a portable graymap.

  There is a subtlety here: we are asking you to use `Pnmrdr` to read image files, and we are also sending a strong signal that your programs should be careful to check for and handle erroneous input. What if `Pnmrdr` fails to detect an error in what is supposed to be an image file? Answer: for this assignment and others in Comp 40, don't worry about it. The whole purpose of using `Pnmrdr` is to hide from you details of how the image file is represented on disk. Therefore, it's inappropriate to ask you to do more checking than `Pnmrdr` does.

Of course, if this were a commercial program that might not be good enough: you would have to do due diligence to check that `Pnmrdr` was indeed detecting all significant errors. For our purposes, just trust that it does.

- You are still responsible for making sure that when `Pnmrdr` does detect an error in the image file, your program reports the error in an appropriate way. Maybe or maybe not this will involve writing nontrivial code.
- You will have to understand the compile-time and link-time options that gcc needs to work with the `Pnmrdr` interface (`-I/comp/40/build/include` and `-L/comp/40/build/lib -lpnmrdr`). A good start would be to understand the `Makefile` you received for this assignment (and you can also look in A simple introduction to Compile Scripts and Makefiles, which is for a previous version of this assignment, but is carefully explained there).
- 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.**

If you are a little unsure how to get started with the `brightness` program, the `brightness` lab will help to get you moving. In the meantime, focus on the design work for Parts B and C.

## 5 Part B: Read a line

Please create a single source file named `readaline.c`. Within that file you must:

- Include the header file `readaline.h` using `#include`. (The header file itself is provided for you in `/comp/40/build/include`, which is in the include path if you use the provided `Makefile`.)
- Implement a single function named `readaline`, conforming to the function declaration in the header file, which is:

```
size_t readaline(FILE *inputfd, char **datapp);
```

- This file **must be self-contained**, i.e. it must rely on no other source files (except, of course, the provided `readaline.h`). Do not create separate files for any helper functions you might create.

We will separately test and grade the correctness of your implementation of this function by compiling just this file and linking it with our test code. We will not link with any of your other files, so the code must be self-contained.

You will also use the function in Part C below, so problems with this function can also affect your grade on that. Test carefully (and then test again)!

**Do not** make a copy of `readaline.h` in your project directory! Your submission will not be accepted if you do. If your compiles fail when including that header file there is almost surely something wrong with your `Makefile`.

### 5.1 `readaline` specification

The purpose of this function is to read a single line of input from file `inputfd`, which is presumed to have been opened for reading. As is common in specifications for computer programs and interfaces, we carefully define some terms, and then use those to specify the behavior of `readaline`:

- The term *character* refers to any of the 256 characters of ISO Latin-1 extended ASCII. The bytes in the input file are interpreted as such *characters*.
- The characters comprising each file are grouped into zero or more *lines* as follows:
  - Each line contains at least one character
  - New lines begin at the start of the file, and after each newline character (`'\n'`)
  - Each newline character is included in the line that it terminates
- Each invocation of `readaline` retrieves the next unread line in the file. The characters comprising the line are placed into a contiguous array of bytes, and `*datapp` is set to the address of of the first byte. `readaline` returns the number of bytes in the line.
- The array of bytes is allocated by `readaline` using `malloc` (or the related allocation functions described in `man 3 malloc`.) It is the responsibility of the caller of `readaline` to free the array using `free`.
- `readaline` leaves the file seek pointer at the first (i.e., unread) character of the following line (if any) or at EOF
- If `readaline` is called when there are no more lines to be read, it sets `*datapp` to NULL and returns 0.
- `readaline` terminates with a Checked Runtime Error in any of the following situations:
  1. Either or both of the supplied arguments is NULL
  2. An error is encountered reading from the file
  3. Memory allocation fails
- `readaline` must not cause memory leaks. That is, it must not leave allocated any dynamically acquired memory other than that returned to the caller through `*datapp`.

For handling input lines you MUST NOT use the system library routines `getline` or `getdelim`, and you must not consult the `man` pages or other documentation relating to them. Reason: we want you to learn to do the work of reading input lines yourself.

## 5.2 Partial credit

For full credit, your `readaline` implementation must support input lines of any size. Significant partial credit is available if you support input files in which no line exceeds 200 characters in length, including any newline character. If you read a line that is longer than your implementation can handle, your `readaline` MUST write the message to `stderr`:

```
readaline: input line too long\n
```

This must immediately cause the program to exit with status code = 4 by calling the system function `exit(4)`. (You learned to call `exit()` in Part A.)

If you begin with a partial credit version of `readaline`, be sure to **save a copy of the code** before you tackle the full credit implementation. That way, if you get in trouble with the enhancement you'll still have something that works. Without that, you will have neither a working Part B nor Part C!

## 5.3 Hints to get you moving

- The `datapp` parameter to `readaline` is a *call by reference* (CBR) parameter, i.e., it's used for function *output*.

  Analogy: "Please slip the address of the party under my door."; In this case, there are two locations involved: the location of the party, and the location where we want location of the party to be stored (under the door).

Application: For this function, the caller wants access to the data in the next line. The `readaline` function will collect the data and store it somewhere (that's where the party is). The caller is asking `readaline` to store the location of the data in a location it has access to. That's what `*datapp` refers to.

Draw a stack diagram. Stack diagrams are not analogies or "the way you think about how functions work": They are precise descriptions of what *actually happens in the computer*. Therefore, you want to get these right, and there is a definite right way to draw these. Please ask the course staff for help, but try to draw it out first. Then you can explain your thinking to a member of the course staff, and we can applaud your insight or correct misunderstandings as appropriate.

- Under the covers, Hanson's `ALLOC` and `NEW` macros use `malloc`. So, you have a choice: if you are more comfortable using `malloc` and friends directly, go ahead. If you prefer the extra error-checking provided by Hanson's macros, you may use them. (FWIW: Norman Ramsey prefers Hanson's versions, Noah Mendelsohn is generally more comfortable with `malloc`. Either will get you full credit if used properly.)
- You will want to come up with some strategy for carefully testing your `readaline` function. Of course, you could just use it in your Part C `simlines` program and hope everything works, but we think you'll find that debugging is actually much harder that way. When something goes wrong (as it almost surely will), you will have to look everywhere to find the problem. Therefore: we *strongly urge* you to come up with a strategy for carefully testing `readaline` by itself.
- Did you *really* look at that ASCII table? It's not obvious how to even *type* some of those characters, much less test them against your `readaline` implementation. Just saying.
- Question: According to the specification, is there *any* file you cannot read in its entirety by repeatedly calling your `readaline` function? Think about it. Why or why not? (You do not need to submit your answer.) Make sure your implementation can handle every file that the specification requires. Design your test cases accordingly.
- Handling lines of arbitrary size may be trickier than you think. DO NOT spend all your time trying to implement that at the cost of not getting to Part C. As noted above, you will get significant partial credit for both Parts B and C if you can handle lines of at least 200 characters (you will use your `readaline` implementation in Part C). We suggest you plan for longer lines, but for a start support only the 200 character minimum. Go on to Part C and get that running. If you get all of that working, go back and extend `readaline` to handle longer lines. In principle, your Part C program should immediately start working with longer lines too!
- The `size_t` return type is an (unsigned) integer large enough to hold the number of bytes in the largest supported file on our Linux system. It is a standard type defined in `stddef.h` and used by system library routines such as `fread`.

## 6 Part C: Similarities in files

You are to write a program named `simlines`, the purpose of which is to read any number of files and detect lines in the files that are similar to each other. Below we give you a detailed specification of what the program must do. Informally, it reads through one or more files looking for cases in which two or more lines contain the same words in the same order. It produces on standard output a report with a section for each such list of words, indicating which line numbers in which files contain those words.

### 6.1 `simlines` specification

The specification for `simlines` is as follows:

- The `simlines` program accepts zero or more command line arguments, each of which is the name of a file.
- The program identifies all cases in which two or more lines in any of the input files are "similar" to each other, where similarity is determined by the definition below.
- Similarities are reported if they occur within any single file and/or if lines in more than one of the files are similar.
- The program writes its results to standard output in exactly the form specified for output below.
- Upon successful completion, your program must terminate with an exit code of `EXIT_SUCCESS` (from `stdlib.h`). This is true of all programs you write in this course unless otherwise specified.
- The `simlines` program raises a Checked Runtime Error if any of the following occur:
  - Any one or more of the named input files cannot be opened
  - An error is encountered reading from any of the input files
  - Memory cannot be allocated using `malloc`

  If any of these situations arise, the program produces no output on `stdout`. The output on `stderr` must be only what is produced by the default Checked Runtime Error exception handler.
- If the same file is named more than once on the command line, then it is read and processed repeatedly, once for each command line reference. Note that this is very likely to result in similarities being detected and reported, since files are mostly similar to themselves!
- You MUST use your implementation of `readaline` from Part B to read the data. If you are using a partial credit version of `readaline` that supports input lines of limited length, then you must rely on the error handling specified in Part B to exit from `simlines` if an unsupported long input line is encountered.

**Definition of line similarity**

The following gives the rules for determining whether two lines are *similar*:

- The terms *character* and *line* have the same definition as in the specifications for `readaline`.
- A *word character* is any of lowercase 'a' – 'z', uppercase 'A' – 'Z', the digit characters '0' – '9', and the underscore character '_' (which is ASCII code 95 decimal). All other characters are *non-word characters*.
- Any contiguous grouping of word characters is a *word*.[1] As common sense would suggest the term always refers to the largest possible groupings, thus the line:

  ```
  'abc  def'
  ```

  contains the two words `abc` and `def`, but not the words `ab`, `ef` or `a`.
- Thus any line contains zero or more words, optionally preceded, followed by and/or separated by non-word characters.
- Two lines are similar if:
  - They contain the same number of words
  - Each word in either line is the same as the corresponding word in the other line (i.e. the first word must match the first word, etc.)

  Note that non-word characters are significant only as separators. Thus, the following lines are all similar to each other:

  ```
  'abc  def'
  'abc     def'
  '  abc    def '
  'abc,def'
  ```

[1]Note that this definition implies that `Cat` and `cat` and `cAt` are all different words.

None of the following lines is similar to the others:

```
‘abc  def’
‘abc  def   def’
‘abc  abc   def’
```

- Lines containing no words (including empty lines) are *not* similar to other lines containing no words

## `simlines` output specification

We define the term *match group* to refer to any set of two or more lines in the input file(s) that are *similar*. Such lines contain the same words in the same order, and we refer to that ordered list of words as the *matching words* list.

The output written to `stdout` by `simlines` must conform *exactly* to the following specification:

- There is one *output section* for each match group.
- If there is more than one match group, the corresponding output sections are separated from each other by a single additional newline (\n). However, there must **not** be an additional newline following the last match group.
- The sections may appear in *any* order.
- If a match group contains *n* matches, then the corresponding output section consists of *n + 1* newline-terminated lines:
  - The first line of each output group contains the list of match words, separated from each other (if there is more than one) by a single space `‘ ’` character.
  - For each match, a line consisting of:
    - The name of the file in which the match occurred, exactly as specified in the corresponding command line argument. The name is printed left justified (i.e. starting at the beginning of the line), and padded to the right with blanks to fill a total of 20 characters. If the filename is longer than 20 characters, then it is written in its entirety, but with no additional padding.
    - A single additional space (typically the 21st character, unless the file name was long)
    - A right justified seven digit integer giving the line number in which the match occurred. Line numbering is 1-based, i.e. the first line in the file is line 1, the second line 2, etc. No leading zeros are included in the line numbers.
  - Within a match group the lines corresponding to each match are written in order. That is all matches in the file named by the first command line argument appear ahead of those in the files that follow. If there are multiple matches in a single file, they are written in order of increasing line number.
- As noted above, the same file named repeatedly on the command line results in no special processing: the file is read repeatedly, almost surely resulting in similar lines in the output, and those lines must be reported in order according to where the file references were in the command line. E.g. the command

  ```
  simlines testfile otherfile testfile
  ```

  might well result in an output group like this:

  ```
  hello world
  testfile-------------------2
  otherfile------------------7
  testfile-------------------2
  ```

  If `testfile` contained the words “hello world” on line 2, and `otherfile` contained the same words on line 7. (The gray dashes in the sample output above are spaces in the actual output of `simlines`; the dashes are shown above to make the spaces easier to count.)

A consequence of the above rules is that `simlines` produces no output if there are no matches.

**Output hints**

Here are some hints that may help you with your `simlines` output:

- Writing that left-justified 20 character filename is easy if you know how to use `printf`. Look online for hints about printing fixed width left-justified strings or ask for help. If you do this right, all the formatting including correct handling of very long filenames, the proper right justification of the line number, etc. can easily be handled by a single `printf`.
- If you read carefully you will note that, except for allowing the output groups to be written in any order, the specification determines character by character exactly what your output must be.
- **Be careful!** If your output does not exactly match the specification, you will not get credit. Note that formatting errors tend to impact all of your output, and can easily result in failure of multiple tests or even of all tests. **Read the specification and check your output carefully!**

## 6.2 Performance target

Your `simlines` program should perform well on inputs resulting in at least 10,000 match groups with up to hundreds of thousands of lines of input. (Due to limitations in the implementations of Hansons libraries, if you use the data structures we think likely, you may find performance slows dramatically if the number of match groups grows much larger than 10,000.) By "perform well," we mean completing in under 20 seconds or so on an unloaded server, if the output is redirected to a file or to `/dev/null`. (If you look up `/dev/null` you'll find that it is a pseudo-file that throws away whatever is written to it. Writing to that won't let you check your output, but it's a good way to time your program without waiting for hundreds of thousands of lines of output to be written to your display or even to a real file.) Of course, for shorter inputs of hundreds of lines with tens of matches, your program should respond more or less instantaneously.

## 6.3 Problem analysis and advice

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

- The key to solving the `simlines` problem is to think very carefully about the data structures you will be creating and about *how* those data structures will result in a solution. Ask yourself questions like the following (much of your design submission ask you to answer these questions):
  - What data structure(s) will you create to represent a match group?
  - How will you efficiently find out whether a given line from an input file belongs in a match group?
  - If it does, what information will you retain about the match?
  - For which of these are Hanson's datatype implementations such as List, Table, Set, Sequence, etc. useful, or when is it more appropriate to use C-language arrays, structs, etc.?
  - Which data, if any, should be converted to Hanson Atoms, and why?
  - When the time comes to write out a match group, how will you write the matches in order? (Hint: you should *not* try to sort them using a sort routine. There are much easier and more efficient ways if you use the right data structures. Be very careful to check the ordering rules for the various data structures you might be tempted to use!)
  - How is the memory for each of your data structures allocated? Do you have a way to free it to avoid memory leaks (remember, there is no way to free the memory for Atoms, and you will not be penalized for using Atoms. All other memory leaks must be avoided.)

Draw pictures! Take some interesting but small test cases and draw out in detail a picture of all your data structures and how they connect to each other. Once you have this picture absolutely clear and correct, organizing your code and then writing and debugging it will become much easier.

- We've said it before, and you'll hear us say it again and again in Comp 40: you will be tempted to put the majority of your time into coding your program. You'll be tempted to build all or most of it, test the whole thing, and then try to find where the bugs are. *This will almost surely waste a lot of your time!* When you test all of your work together, the bugs could be almost anywhere. A bug in one routine might not cause an immediate crash, but might produce bad results or cause your program to fail after executing hundreds of thousands of lines of additional code.

  The way to avoid this is to put as much energy into your test plan and design as you put into the coding of your solution. Find ways to test individual pieces of your code. Create test cases that explore not just the obvious paths, but the unusual ones.
- C strings are different from C++ strings and C's string library can be tricky to use properly. Make absolutely certain you understand the role of the NUL character `'\0'` in the termination of C strings. If you just code without understanding this, you are in for hours of frustration. Hanson offers some string processing libraries that you may (or may not) find helpful. My solution does not use them, but Norman Ramsey *did* use them in the solution to a somewhat similar problem given in years past.
- 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 as keys or for similar purposes with Hanson's data structures, you **must** use the `Atom` interface. (When using a string as a *value* (as opposed to a key), then use of the `Atom` interface is optional.)
- Note: **Hansons's Array data structure is *not* available for use in Comp 40**. As you will see in the next assignment, we have developed an improved (or at least more interesting) version that we will introduce then. You should not need to use Hansons's array for this first homework. (Of course, C character strings are C arrays, so you will definitely use C arrays when working with those.)
- Don't forget to run `valgrind` on your code!

Repeat: **the data structures are already built for you**; your job is to figure out which ones will be useful. We are looking for a clean, straightforward design.

## 7 Part D (DESIGN): Simlines Design

**DUE EARLY!** (see our course calendar). An overview of design documents in general and the details of what is required for this assignment specifically are given below.

### 7.1 Design overview

The key to writing a good program of any significant complexity is to think thoroughly through two related questions *in advance* of commencing coding work:

1. How will data in the program be represented and interconnected?
2. How will the program logic be organized, and how will the computation be done?

Writing down the answers to these questions *before* writing your code addresses two fundamental aspects of large programming efforts:

**Efficiency:** Nobody likes deleting large swaths of code. Nobody likes backtracking and starting over. While it is not possible to foresee every challenging nuance that lies ahead, the design process helps to uncover and avoid large-scale missteps that could cost hours of coding work. A driving analogy feels appropriate here: it often takes less time to look at a map than it takes to find your way back from a wrong turn.

**Communication:** Two minds are greater than one. Ten minds are greater than two. Success in Comp 40 hinges on the speed and clarity with which you are able to communicate with your partner and the course staff. A written design that succinctly lays out your coding plan ensures that you and your partner are on the same page and allows the course staff to warn of looming problems before they derail you.

## 7.2 Homework 1 Design Specifics

Because design is such an essential part of Comp 40, you may find it helpful to read Norman Ramsey?s treatise on the matter. For a large-scale industrial project, a design plan can easily reach this level of detail.

For each Comp 40 assignment, however, you will be asked to submit only a reduced subset of these design components. Specifically, we will require only the components that best address the challenges of the assignment at hand. For this first assignment, your design document should consist of the following three parts:

**Simlines Architecture (1 page):** This section must convey your high-level plan for using Hanson's data structures to both store and retrieve information in your `simlines` implementation. You may find it helpful to draw pictures, make bulleted lists, write English paragraphs, etc. The medium does not matter as much as conveying your plan *clearly*; note that it's *much* harder to convey a portion of your design document as a paragraph than in bulleted list form. We highly recommend using lists and pictures instead of English paragraphs to ensure as much clarity as possible. We are specifically looking for you to address the following items:

- Identify what data structures you will need to compute `simlines` and **what each data structure will contain**.
- Hanson's data structures are *polymorphic*, so you will have to **specify what each `void *` pointer will point to**.
- You are supplied with a set of methods for each of the Hanson structures. Which ones will you use to **get information in and out** of your structures?

**Implementation Plan (1 page):** A detailed implementation plan should be part of every Comp 40 assignment, regardless of whether or not you are required to submit it. It lays out a step-wise progression of the programming aspect of the assignment, allowing you to focus on one thing at a time instead of being overwhelmed by the assignment as a whole.

Since this is the first assignment, we are providing you with a bare bones implementation plan (see below). You may alter this plan if you'd like, and will likely need to flesh out portions of it to match the plans you laid out in the architecture portion of your design.

You must also include **time estimates** for each step of your implementation plan. That is, an estimate of how long you think it will take you to implement that step. These estimates not only allow you to plan out your work, but also to identify steps that exceeded your estimates in order to better prepare for future assignments. We have listed a time estimate for the first step of the provided implementation plan as a reference.

1. Create the `.c` files for both your `brightness` and `simlines` programs. In each, write a `main` function that spits out the ubiquitous "Hello World!" greeting. Compile them with the provided script and make sure that they run correctly. **Time: 10 minutes**
2. Extend your `brightness` implementation to read through each pixel of a `pgm` image and print out its grayscale value.

3. Incorporate the necessary computations to print out the average brightness value.
4. Create the `.c` file that will hold your `readaline` implementation. Move your "Hello World!" greeting from the `main` function in `simlines` to your `readaline` function and call `readaline` from `main`. Compile and run this code.
5. Extend `simlines` to open and close the intended file, and call `readaline` with real arguments.
6. Build your `readaline` function. Extend `simlines` to print each line in the supplied file using `readaline`.
7. Route the output from `readaline` into the Hanson data structures that you have selected for your `simlines` implementation.
8. Retrieve the necessary information from your data structures and output your match groups in the specified format.

**Testing Plan (1 page):** Explain in detail your testing plan for Parts B and C (you do not need to specify testing plans for `brightness`). You'll get essentially no credit for saying "I plan to try it with lots of inputs and see if it works." We need to know the specific inputs you'll be using and their expected outputs. A recommended strategy is to organize your testing plan according to your implementation plan. For each step in your implementation plan, what tests will you run in order to confidently move forward with your implementation? Some implementation steps may require only one or two tests, while other steps will be more involved.

## 8 Organizing and submitting your solutions

You will make two submissions to complete this assignment. Several days before the final submission, you will submit your design document. At the end of your work, you will make a final submission that includes your code.

Only one partner makes each submission, and the same partner should submit both the design and the final code submission. When you submit, you will identify your partner by their CS login (i.e. `mmonro2`).

### 8.1 Deadlines and tokens

Like all programming assignments for this class, the programming parts of this assignment are due one minute before midnight on the day indicated in the course calendar. You may turn in assignments *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 (make sure it compiles and passes valgrind first!).

**If you spend an extension token on any part of the assignment, it automatically applies to *all* parts of the assignment.** I.e., if you submit either your design or your code a day late you use a token; submit them both a day late and it's still only one token total. Of course, if either is two days late, that's two tokens.

You may resubmit until the original deadline (i.e., not counting token extensions) and we will grade the latest version available at the deadline. *What you may not do is to submit before the deadline, then decide to use tokens and resubmit.* Reason: at the deadline we gather your work and start grading. If you resubmit, we will likely not notice, and would then have to regrade. If you are planning to use tokens, please do not submit at the regular deadline — i. e., do not submit something until you are really ready to submit for grading. If you have an unusual problem, please email comp40-staff@cs.tufts.edu.

## 8.2 Submitting your design document

By the design deadline, submit the design document described in Part D: Design.

Your document should be a `pdf` file named `design.pdf`. You will submit your document via Gradescope. If you have not received an e-mail to join the class on Gradescope, please contact the course staff.

Your submission via Gradescope will be comprised of three steps:

- Choose the assignment "Filesnpix" and upload your document
- Indicate the partner that you worked with

The Gradescope interface makes these steps fairly straightforward, but contact the course staff if you have any issues. Note that if you resubmit an improved version of your design document, **you will need to re-specify who your partner is**.

## 8.3 Submitting your completed code

In your final submission, don't forget to include a `README` file which

- Identifies you and your programming partner by name and CS login
- Acknowledges help you may have received or collaborative work you may have undertaken with classmates, programming partners, course staff, or others
- Identifies what has been correctly implemented and what has not
- Says **approximately how many hours you have spent** completing the assignment

Your final submission should include at least these files:

```
README
brightness.c
simlines.c
readaline.c
```

A carefully designed, modular solution for `simlines` will probably include at least two other files. When you are ready to submit, `cd` into the directory you are submitting and type the following command:

```
submit40-filesnpix
```

Congratulations on making it through your first Comp 40 assignment!