4 Corruption

In this section, we will tell you everything we know about our hacked images.

Our original repository consisted of "plain" pgm image files (magic number: P2). It is critical that you understand this format in order to understand the hack that corrupted our files. You can read the full `pgm` spec at the above link or by accessing the Linux `man` page:

```
sunfire33{megan}: man pgm
```

The employed hack works on a given `pgm` file in three passes:

1. The header is deleted, including the magic number, the width and height, and the maximum gray value (Maxval). All that remains is the image raster.
2. Fake image rows are injected throughout the raster. So a `pgm` image that had a specified height of 50 rows, may now have a raster that exceeds 50 rows.
3. The whitespace in each row is replaced with a random infusion of non-digit bytes. That is, bytes that do not encode a digit in ASCII.

So a row that appeared as:

10 6 6 10 11

might become:

a10**b**6**c**6**d**10**e**11**fgh**

Suffice to say, the resulting, corrupted images cannot be read by our original reader module, `Pnmrdr`. However, we know four more things about the employed hack — oversights born of laziness, or simply an inferior CS program:

1. The original images included a single `\n` character at the end of each row of the image raster. This formatting has not been tampered with.
2. The original images all employed a Maxval of 255, which has not been altered.
3. The original images were all at least two pixels wide and two pixels high.
4. The original image rows (as opposed to the injected, duplicated rows) have all been infused with the same sequence of non-digit bytes. So the following rows:

   a10**b**6**c**6**d**10**e**11**fgh**
   a11**b**11**c**9**d**6**ef**6**gh**
   ab10**c**9**d**6**e**11**fgh**9

   are all original rows of the image because they employ the same infusion sequence of "**abcdefgh**". Injected rows each have a unique infusion sequence.

Obviously our days of using the "plain" format are over. Your job is to restore the original image information and output the result as a "raw" `pgm` (magic number: P5).

Our repository of corrupted images can be found at: `/comp/40/bin/images/corruption/`

5 Part A: Read a line

**** Note that for this implementation, 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 B, and only if you have time go back to enhance Part A for full credit**. You will lose much more credit for doing a poor job on Part B than for failing to handle long lines in Part A. ****

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 **not** rely on any 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 B 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 1000 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)`.

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 A nor Part B!

5.3 Hints to get you moving

- The `datapp` parameter to `readaline` is a *call by reference* 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.
- 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 B `restoration` 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.
- 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 B. As noted above, you will get significant partial credit for both Parts A and B if you can handle lines of at least 1000 characters (you will use your `readaline` implementation in Part B). We suggest you plan for longer lines, but for a start support only the 1000 character minimum. Go on to Part B and get that running. If you get all of that working, go back and extend `readaline` to handle longer lines. In principle, your Part B 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 B: Restoration

You are to write a program named `restoration`, the purpose of which is to restore a corrupted "plain" `pgm` file to a functional "raw" `pgm` file. A detailed specification of what the program must do follows below.

6.1 `restoration` specification

The specification for `restoration` is as follows:

- The `restoration` program takes at most one argument, which is the name of a corrupted `pgm`.
- If no argument is given, `restoration` reads from standard input, which should contain a corrupted `pgm`.
- The program extracts the original image information from the corrupted files.
- The program writes the reconstructed image to standard output as a "raw" `pgm`. Further details of this output are supplied 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 `restoration` program raises a Checked Runtime Error if any of the following occur:
  - More than one argument is supplied
  - The named input file cannot be opened
  - An error is encountered reading from an input file
  - Input is expected but not supplied
  - 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.
- Apart from the errors described above, you may otherwise assume that a supplied file name will be a corrupted "plain" `pgm`.
- You MUST use your implementation of `readaline` from Part B to read the data. If you are using the limited-length partial credit version of `readaline`, then you must rely on the error handling specified in Part B to exit from `restoration` if an unsupported long input line is encountered.

Output testing

As alluded to earlier, we have a `pgm` reader module, `Pnmrdr`. While it is useless on corrupted images, it should be able to read any "raw" `pgm` file that has been properly restored. 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.

Performance target

Your `restoration` program should perform well on large images as well as small. 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 small images, your program should respond more or less instantaneously.

6.2 Problem analysis and advice

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

- The key to solving the `restoration` 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):
  - How will you determine if a line read from your corrupted `pgm` file contains a row of the original image?
  - If it is, what information will you retain?
  - What data structure(s) will you create to collect restored image rows?
  - 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?
  - 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.
- You will be tempted to put the majority of your time into coding your program - 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 NULL character `'\0'` in the termination of C strings. If you just code without understanding this, you are in for hours of frustration.

- 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: **Hanson's Array data structure is *not* available for use in CS 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 Hanson'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 C (DESIGN): Restoration 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 CS 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 CS 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 CS 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 three parts:

Restoration 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 `restoration` 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 these items:

- Identify what data structures you will need to compute `restoration` 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 CS 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). The first three steps are suggestions to get you moving. You are welcome to alter them. The remaining steps are intended to serve as guidelines, but are too general. You will need to flesh them out into more detailed, incremental steps.

You must also include **time estimates** for each step of your implementation plan, i.e., an estimate of how long you think it will take you to implement that step. These estimates not only help you plan out your work, but also identify steps that exceeded your estimates 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` file for your `restoration` program. Write a `main` function that spits out the ubiquitous "Hello World!" greeting. Compile and run. **Time: 10 minutes**
2. Create the `.c` file that will hold your `readaline` implementation. Move your "Hello World!" greeting from the `main` function in `restoration` to your `readaline` function and call `readaline` from `main`. Compile and run this code.
3. Extend `restoration` to open and close the intended file, and call `readaline` with real arguments.
4. Build your `readaline` function. Extend `restoration` to print each line in the supplied file using `readaline`.
5. Route the output from `readaline` into the Hanson data structures that you have selected for your `restoration` implementation.
6. Retrieve the image information from your data structures and output your restored "raw" `pgm`.

Testing Plan (1 page): Explain in detail your testing plan. You'll get 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 use and their expected outputs. A recommended strategy is to interleave your testing plan with your implementation plan (these do not need to be separate sections). For each step in your implementation plan, what tests will you run 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 Submission

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 submit 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

As will be the case all semester, the programming parts of this assignment are due one minute before midnight on the day indicated on the course calendar. You may use our late token system to turn in assignments up to 48 hours after the due date. **If you spend a late 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. If either is two days late, that's two tokens.

If you are not using a late token, you may resubmit your work up until the deadline and we will grade your latest submission. **What you may not do is to submit before the deadline, then decide to use tokens and resubmit after the deadline.** Reason: we begin grading immediately after the the deadline and have no way of knowing if you're planning to resubmit. If you have an unusual problem, please post privately on Piazza to the course staff.

8.2 Submitting your design document

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

Your document should be a `pdf` file named `design.pdf`. This should be an actual `pdf` file and not, for example, a Word file renamed with a `.pdf` extension; export an actual `pdf`. When you are ready to submit, put your design document on the server (this can be done with the `scp` command or through VS Code). Then, `cd` into the directory containing your design document and run the following command:

```
submit40-filesofpix-design
```

This command will automatically upload your design doc to Gradescope. After running it, please check Gradescope to make sure that the correct document was submitted. Do **not** upload your design document to Gradescope manually. If you do not see the document on Gradescope after a few minutes, check the log that printed out when you submitted (using `prolog40` if necessary). If the submission was accepted but not uploaded and your partner's login is correct, then post a note on Piazza and we will look into it.

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`, `restoration.c`, `readaline.c`.

A carefully designed, modular solution for `restoration` 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: