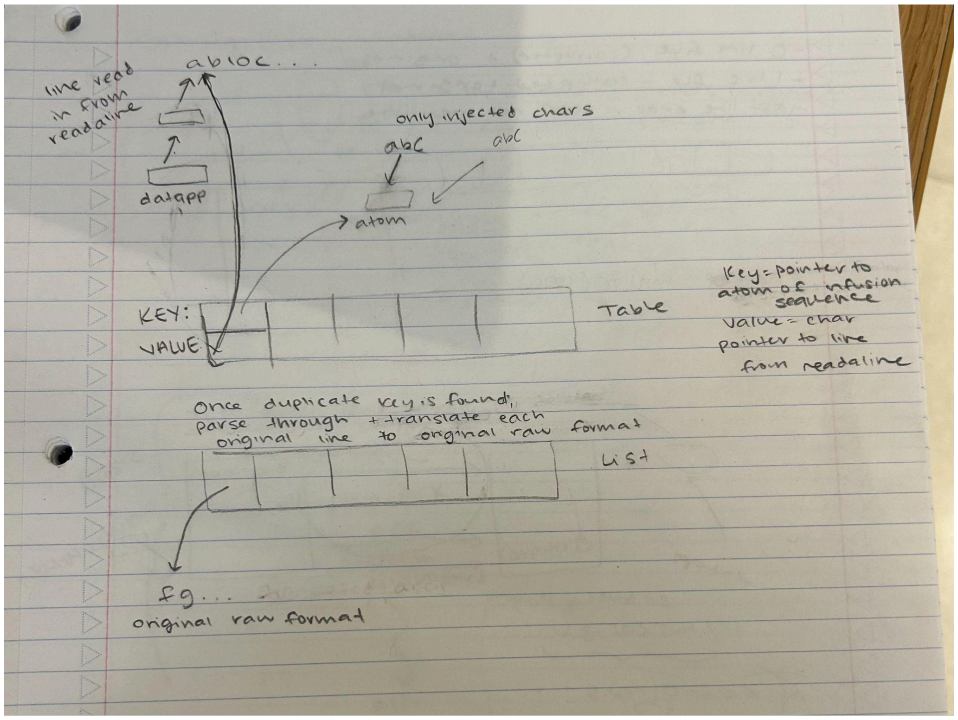
Filesofpix Design Doc – kcasey06, bdioni01

Restoration Architecture

We will use two main Hanson data structures for Restoration: a Table and a List. We will use the Table in order to determine the repeated infusion sequence in the original rows, and then store all corrected original rows in our List. Details of both can be found below, along with a diagram of the two data structures.

- Table
  - Using a table in able to determine the repeated infusion sequence
  - Key = pointer to atom of the non-digit infusion sequence from the line
    - Once there is a collision of identical keys, we have found our repeated infusion sequence
  - Value = char pointer to first char of line (directly from datapp from readaline)
  - Implementation functions
    - Allocate new table: extern T Table_new (int hint, int cmp(const void *x, const void *y), unsigned hash(const void *key));
      - Size hint – 1000
      - Comparison and hash functions: null function pointers NULL (b/c the key is an atom)
    - Deallocate: extern void Table_free(T *table);
    - extern void *Table_put (T table, const void *key, void *value);
      - *Table_put adds the key-value pair given by key and value to table. If table already holds key, value overwrites the previous value, and Table_put returns the previous value. Otherwise, key and value are added to table, which grows by one entry, and Table_put returns the null pointer.*
      - Pass an atom for key and char pointer for value
      - Returns NULL or a char pointer to line in file of first instance of an original line
    - extern void *Table_get (T table, const void *key);
      - Use to obtain second instance of an original line
      - Pass an atom for key
      - Returns a char pointer
- List
  - List of char pointers
    - Each string is the translated ascii chars for each line from readaline
    - Each entry is a char pointer – the address of the first char in the c-string
  - Implementation functions
    - Create new list: T List-list(void *x, …);

        - NULL as argument to create empty list
    - Deeallocate:
        - Void List_free (T *list)
    - Add element to front of list: T List_push(T list, void *x);
        - Pass char pointer of c-string to add to list
    - Int list_length(T list)
        - Returns length of list (height)
    - Reverse list: T list_reverse(T list);
    - remove first element from list for reading/printing: T list_pop (T list, void **x);
        - Void **x – double char pointer (pointer to c-string) in order to print the line (width)
            - Potential edge case: different line lengths


line read in from readaline
ab\0c...
datapp
only injected chars
abc
abc
atom
KEY:
VALUE
Table
Key = pointer to atom of infusion sequence
Value = char pointer to line from readaline
Once duplicate key is found, parse through + translate each original line to original raw format
List
fg...
original raw format

Implementation Plan

1. Create the .c file for the restoration program. Write a main function that spits out the ubiquitous “Hello World!” greeting. **Time: 10 minutes**
2. Create the .c file that will hold the readaline implementation. Write a main function that spits out the ubiquitous “Hello World!” greeting in readaline function and call readaline from main. **Time: 10 minutes**
3. Implement arguments within the restoration program main function. Handle arguments (argc should be 1) and attempt to open and close the intended file. **Time: 5 minutes**
4. Raise a checked runtime error for any of the following: **Time: 15 minutes**

    a. More than two arguments are supplied
    b. The named input file cannot be opened
5. Implement the partial readaline function – *size_t readaline(FILE *inputfd, char **datapp);* **Time: 45-50 minutes**
    a. read line of input from inputfd to the next newline character or until EOF into a cstring.
    b. “readaline: input line too long\n” and exit(4) if input line is too long
    c. if inputfd points to a character where there are no lines to be read, returns NULL
    d. set datapp to point to the first element of this array
    e. set inputfd to point to the first element of the next line (if not EOF)
    f. return the size of the array with sizeof.
    g. handle Checked Runtime Errors
6. Create restoration.h function declarations. Test functionality of readaline function from restoration file. **Time: 10 minutes**
7. Extend restoration to print each line in the supplied file using readaline. **Time: 15 minutes**
8. Implement a function such that when given a “plain” instance of a line, it returns a cstring containing only the injected chars in order. Test and verify the functionality of this function on its own. **Time: 20 minutes**
9. Route the output from readaline for each line into a Hanson Table data structure such that the key is a pointer to the atom that the order of injected chars (obtained by the function in the prior step) points to. The value is a cstring of the “plain” instance of the line (e.g., key : atom for cstring abc; value : pointer to a10b11c56).. **Time: 45 minutes**
10. For each inserted key-value pair, check if the Hanson Table does not return a null pointer. If so, we found the duplicate char sequence that points to the same atom and we print the replaced value (first original line). Then, print the value corresponding to the key of the atom that the current sequence of injected chars **datapp* is pointing to which is remaining in the table (second original line). **Time: 20 minutes**
11. At this point, free the table from the heap and continue routing the output from readaline, except now only printing the “plain” instance of the line if the sequence of injected chars matches that of *curr*. **Time: 10 minutes**
12. If everything is functioning properly, create a function that when given a “plain” instance of a line, transforms it into a cstring containing the “raw” instance of that line (using strtol). Test that this function works as intended. **Time: 25 minutes**
13. Now, instead of printing the “plain” instance of each line, we now store the transformed “raw” instance in a list with each element containing each “raw” line. Test that the list contains the proper information. **Time: 25 minutes**
14. Retrieve the magic number, width (length of “raw” line cstrings), length (length of list), maxval, to output list contents and restore the “raw” pgm. **Time: 15 minutes**
15. Enhance the part A to support input lines of any size. Create an expand function that doubles the allocated size when needed. **Time: 60 minutes**

- **Note:** Give or take +20 minutes for testing on each step

Testing Plan

- Step 1
  - test to see if restoration.c outputs “Hello World!” when ran.
- Step 2
  - test to see if readaline.c outputs “Hello World!” when ran.
- Steps 3 & 4
  - test running restoration with no arguments, one argument, and multiple arguments to see if the program handles these cases correctly.
  - check to see if the program opens a file that does exist and fails to open a file that doesn’t exist.
- Step 5
  - Try running the readaline function with null arguments. See if checked runtime error
  - See if the function handles not being able to read from the file correctly. Verify checked runtime error
  - See if the function handles when there are no more lines to be read correctly (datapp* set to null and return 0).
  - See for after every successful call of readaline that datapp* points to the first char of the next line of the file (and none if at EOF).
  - Try running readaline function with lines greater than 1000 characters in length to see if program properly prints out, “readaline: input line too long\n” and then exit(4)
  - Valgrind each test case and see if memory is getting allocated and deallocated properly. Further, test that the realloc function is reallocating memory properly.
- Step 6
  - Repeat the tests for the previous steps except call readaline from restoration.c.
- Step 7
  - Pass all 12 given demo corrupted files into the program and diff the output with the file to see if we produce the same output.
  - Create 4 of our own files to test and diff (easier to see the output on a smaller scale).
  - Possibly:
    - File with 1000+ chars on one line
    - Empty file
    - Multiple lines of a range of characters under 1000 chars/line
- Step 8
  - Give the function single c-string inputs to make sure that the “plain” lines transform to only the injected sequence of chars (e.g., “a35b345p23” -> “abp”)
  - Create our own files to pass through and make sure that the output contains only the injected sequence of chars and omits any numbers.
- Step 9
  - Assert the value of *datapp* for each pass through of readaline.
  - For our previous demo files, print out the contents of the hash table for each call of readaline to verify that the table is functioning as intended

    - Pass the 12 given corrupted files into the program and verify the hash table contents
    - Test for the case where the file only contains one line using a demo file. Check to see if the hash table works as intended
    - Valgrind to see if memory gets allocated and deallocated correctly (with the exception of atoms).
- Step 10
    - Repeat the testing steps from the last step except we should be testing to see if the first two injected lines with the duplicate sequence get printed.
- Step 11
    - Again, repeat the testing steps from the last two steps except we should be testing to see if all the injected lines with the correct sequence get printed.
- Step 12
    - Pass in example demo c-strings to verify whether the function correctly returns the c-string containing the chars corresponding to the sequence of ascii numbers.
        - e.g.
        - c-strings with injected characters in the front and back
        - c-strings with the ascii numbers in the front and back
        - empty c-strings
        - c-strings containing contiguous injected chars
        - c-strings with a single char or a single ascii number
- Step 13
    - Print out the contents of the list after every c-string insertion.
    - Test with the demo files we previously made (as the 12 given corrupted files will be too large to verify correctness).
- Step 14
    - Run program with 12 corrupted files to see if we can reproduce the correct image or if the correct error message is given
    - Run program with our own corrupted files
    - Run program with our own images that we find
- Step 15
    - Create demo files with lines containing over 1000 characters varying in contents
    - Test these files with the standalone readaline function to check and see if it can print out the contents without error. Diff with original file to verify this.
    - Use the given corrupted files that contain lines of over 1000 characters to check if the readaline can process these as well. Diff with original corrupted file to verify this.