Project 1

- CSci 2021-010, Spring 2020.
- Released February 9th, 2020.
- Due Wednesday February 19th, 2020 by 11:59 PM.

Introduction

If you get an email address that goes with a job, your employer's rules will probably say that the employer has the right to monitor your email for compliance with the company's rules, since the email system is owned by the company and supposed to be used only for company purposes. In this assignment, we put you in the shoes of the boss of such a company who wants to ensure the employees aren't discussing inappropriate subjects in the emails. Specifically you are going to create a program named `email-guardian` that takes a list of forbidden words, and a file full of email messages, and reports the names of anyone who sent an email message containing a forbidden word. (How the employees are to be disciplined is outside the scope of this project.)

The list of forbidden words, which we'll call the *blacklist*, is replaceable as an input to the program because the policy about what words to disallow might change at the boss's whim. But because the blacklist might be long, your program will need to store it in a data structure that allows efficient lookups. The names of the offending employees will really just be the "From:" header lines of emails, but because we don't want the program's output to be overwhelming if there are repeat offenders, each name should only be reported once in a run of the program. To get you started, we've given you some framework code that handles the command line arguments to the program and the basic I/O operation of reading a file into memory. Your job is to write the rest. In particular, any places where you see `TODO` in a comment are places that you should add code.

This is the sort of program that you could write in any general-purpose language, and it doesn't depend on any 2021-specific concepts; in fact it might seem like it could have come from CSci 1933. That's on purpose: we're doing this project first to give you some practice writing programs in C in our Linux environment, to help you prepare for later projects where you'll have to work with C while also exploring new concepts.

The data structure we're requiring you to use to store the blacklist is a binary search tree, a linked data structure that is set up to allow efficient search by subdividing the alphabetical order. Efficiency is not so important for the data structure keeping track of which users have already been reported, but a plain C array wouldn't work well because we don't know in advance how many misbehaving employees there will be. We'll ask you to use a singly-linked list for this purpose. The sizes of these data structures won't be known in advance, so you need to use `malloc` to allocate them. But because it's not so important for a short-running program and to save you from some kinds of bugs, you don't need to use `free` to deallocate any of the memory you allocate. (The memory will all be deallocated when the program finishes.)

The compiled program will be called `email-guardian` and have the following usage:

```
email-guardian <blacklist_file> <messages_file>
```

The angle brackets aren't part of the syntax; they indicate that the arguments described as `dictionary_file` and `input_file` need to be replaced by appropriate file names.

Other than error messages, the program should produce its output to the standard output (e.g., using `printf`). The only output will be `From:` lines from the messages in the mailbox: one per line, including everything from the `From:` at the beginning of the line up through the end of the line. These lines should be printed in the order of appearance in the messages file of the first email from a given user containing a forbidden word, but every subsequent forbidden message from the same user should not cause the user to be printed again: a given user should appear at most once in the output.

The blacklist file is a text file with one word per line; i.e. each word is terminated by a newline. You shouldn't need to make any more assumptions about what characters can appear in a word, for instance a "word" might contain space characters. But you can assume that a word will not contain the null character. The words don't need to appear as complete words in the message: for instance, if the word `steak` is on the blacklist, messages that mention `beefsteak` or `steakhouse` should be flagged. The words in the blacklist file may appear in any order.

The messages file is a text file containing email messages one after the other; this is actually a de-facto standard format used on many older systems. Each message in the file begins with a line that starts with `"From "` (i.e., the five letters capital F, lowercase r, o, and m, and a space). The rest of this line will often appear to contain an email address called the "envelope sender" and a date, but you should ignore these. The rest of the message follows on additional lines: first there are a number of "header" lines with information about the message. Each header line is introduced by a name of the header and a colon, and the end of the header lines is indicated by a blank line. The only header that is important for this program is the `From:` header, which is what we will use to determine the person who sent the message and should therefore be responsible if it contains forbidden words. The main location we expect to find forbidden words is in the body of the messages (after the headers), so to make things simple, just have your code identify uses of forbidden words that occur after the `From:` header, but before the `From` header that indicates the next message.

On the course web site near where you found this handout, you'll be able to download a compressed tar file `proj1.tar.gz` containing the files for this assignment, such as sample blacklist and messages files, and a framework `email-guardian.c` containing the code we've already written for you.

To get started, copy the tar file to a location in your home directory and then expand it with this command:

```
tar -xzvf proj1.tar.gz
```

Part A - Binary search tree data structure

The most important data structure you'll build for this project is a binary search tree containing the blacklist, to allow checking for forbidden words without having to compare every blacklist word with every position in an email message.

You likely encountered binary search trees in a previous programming course, but here's a quick review. A binary tree is a tree in which each node has at most two children, which we distinguish as the left child and the right child. A binary search tree represents a set of data objects that have an ordering. Each node represents one data object, but in addition that item determines where other items are located in the tree. If an item is earlier in the order, it is reached following the left child, while items later in the order are reached by following the right child. Binary search trees allow an item to be looked up by following a single path through the tree, which is most efficient if the tree is *balanced*; that is, if the left and right subtrees of nodes contain similar numbers of nodes.

A binary search tree is a better data structure for this program than a hash table because the program can't easily tell in advance how long a word might match at a given position: for instance if a message mentions `steakhouse fries`, it should be flagged if any of `s`, `st`, `ste`, `stea`, `steak`, `steakh`, and so on, is in the blacklist. But you can check whether any of these strings is in the blacklist along a single path through the tree comparing with `steakhouse fries`.

If you are unfamiliar with or rusty on binary search trees, you can look up an old textbook or an web resource for a review, but remember that such outside sources should only be helping you with concepts: you need to write your own code. As a starting point, here are some web resource we like:

```
https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm
https://medium.com/@mbetances1002/data-structures-binary-search-trees-explained-5a2eeb1a9e8b
https://yourbasic.org/algorithms/binary-search-tree/
https://www.youtube.com/watch?v=i_Q0v_Ct5lY
```

This project can use a relatively simple kind of binary search tree because the set of items in the tree does not need to change over time: the entire blacklist is available from the beginning of the program and fixed. The two operations you'll need to implement are building a balanced binary search tree from a sorted array of strings, and looking up whether a binary search tree contains a string that is a prefix of a query string. Both these operations can be conveniently implemented as recursive functions.

First, declare a `struct` for the nodes in your binary search tree. Since the number of tree nodes to create depends on the size of the blacklist, your code should create such structures dynamically with `malloc` and refer to them with pointers.

One good interface for the function that builds a binary search tree is to accept an array of strings (`char` pointers) using a pointer to the beginning of the array and a length giving the number of the elements in the array. Under the assumption that the strings in the array are sorted, one of the elements closest to the middle of the array is a good choice for the root of the binary search tree. After making that choice, think about how the rest of the process of building the tree can be performed recursively. As you may have found with other kinds of data-structure programming problems, you'll probably find it helpful in thinking about this problem to start with a small example, and draw a picture of what you'd like the structures to look like.

The lookup routine can be expressed as a function that takes a pointer to a binary search node, and a pointer to the query string. It should compare the node's string to see if it equals the corresponding number of characters in the query string, and if so, the function has found a match. If the node's string is not a match, then the direction of the comparison (which string was alphabetically before the other), tells which subtree should be searched next.

Implementing just part A won't give you a complete checking program, but it would probably be a good idea to stop after writing part A to check that the code you've written works correctly with some separate tests. You may also find it helpful to write a debugging function that prints the contents of a binary search tree.

Part B - Loading the blacklist

The second area of functionality you'll need to implement is loading the binary search tree from part A with blacklist words from a file. We've already given you a basic function named `slurp_file` for reading input from a file, which reads the entire contents of a file into a single character array. (You don't need to understand the details of how this function works, but if you're curious, all the functions it calls have manual pages.) But the buffer this function creates isn't the same format required to put the words into a binary search tree, so your code in this part will need to bridge that gap.

The first format difference is that in a single large character array, it isn't convenient to refer to each word separately, but we want character pointers for each word to put into the binary search tree. The second difference is that the words in the blacklist file are not in any particular order, but they need to be sorted before making them into a binary search tree.

The `slurp_file` function reads the contents of a file into memory in just the same format it has on a disk, a sequence of bytes in an array. This data structure isn't organized by lines, but you can tell where each line ends because of the presence of newline (`\n`) characters: each newline indicates the end of one line, and then the following character starts the next line. For instance this is the output of the command `od -Ad -c meat.lst`, which shows the file as an array of 111 characters (in rows of 16):

```
0000000   h   o   t       d   o   g  \n   h   a   m   b   u   r   g   e
0000016   r  \n   s   t   e   a   k  \n   b   e   e   f  \n   P   o   r
0000032   k  \n   c   h   i   c   k   e   n  \n   F   I   S   H  \n   s
0000048   a   l   m   o   n  \n   T   u   n   a  \n   s   a   u   s   a
0000064   g   e  \n   t   u   r   k   e   y  \n   f   o   i   e       g
0000080   r   a   s  \n   p   e   p   p   e   r   o   n   i  \n   p   r
0000096   o   s   c   i   u   t   t   o  \n   B   A   C   O   N  \n
0000111
```

In the first step, you need to make an array of character pointers, each pointing to the beginning of one of the words in the blacklist. You will need to allocate the array of pointers dynamically after figuring out how many entries it needs. But you don't need to make a copy of the words themselves: the copies in the character array produced by `slurp_file` will work fine, as long as you make sure they have null terminators at the right places. You can modify the contents of the input array, so for instance you can turn the newline terminators into null terminators.

To put the string array in order, you don't have to implement a sorting algorithm yourself, because the C standard library already includes a suitable function named `qsort` (which uses either quicksort or another similarly-fast algorithm). The one thing that's a bit tricky about using `qsort`, though, is that it is designed to be able to sort items of any size and with any comparison function, so you have to supply the item size and comparison function for it to work correctly. It's easier to move around pointers than the strings they point to, so you should apply `qsort` to an array of pointers. The final argument to `qsort` should be the name of a function that will be used to compare two items in the array to decide which one should come first. You'll have to write your own comparison function that appears to take two arguments of the generic type `const void *`, but these will actually be pointers to two character pointers in your array, so you should convert them to the right type in the body of the function before using them.

Part C - Mailbox processing

The last major area of functionality for the program is processing the messages, using the blacklist data structure and lookup function from the previous parts. We'll use the same `slurp_file` input routine as before, but you will need to write code to traverse that array, keeping track of what user is responsible for the current message and looking for forbidden words at each location. You will also need to implement an additional data structure, a singly-linked list, to keep track of the misbehaving users who have already been reported.

The two signposts your code will use to determine what user is responsible for text in the mailbox file are the lines that start with `"From "` and `"From:"`. (Remember that these are only significant when they are the first thing in the line. In this mailbox format the word `From` is changed to `>From` if it appears at the beginning of a line in a message body, so it isn't confused with the special lines.) These represent the end and the beginning of an area of responsibility respectively: a user is responsible for the contents of a message after the `"From:"` header containing their name, but a `"From "` indicates the beginning of a new message they aren't responsible for.

To avoid printing duplicate messages about a user, your code should also create a linked list of the `From:` lines of users that have already been printed, so your code can skip printing them if they have been printed in the past. The linked list is a linked data structure similar to the binary search tree, but simpler.

Useful library functions

Here is a list of functions from the C standard library that you may find useful for the project. We've given their function prototypes and brief descriptions here, but you can find more details about them in the manual pages on any Linux machine, or other sources.

- `malloc`: `void *malloc(size_t size);` The `malloc` function allocates a memory region large enough to store at least SIZE bytes, and returns a pointer to it. The `void *` type is a generic pointer, but you should store it into a pointer variable of an appropriate type. If you aren't going to use the allocated memory region anymore, you can deallocate it with the `free` function, but you don't need to do that for this project.
- `puts`: `int puts(const char *s);` The `puts` function writes the contents of a string, up to but not including the terminating to null byte, to standard output. Then it outputs a newline to standard output. In other words, `puts(str)` is the same as `printf("%s\n", str)`.
- `qsort`: `void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));` The `qsort` function sorts an array into ascending order. The elements of the array can be any type: the caller needs to specify the size SIZE of each element and a function

COMPAR to compare them. The array itself is passed via a starting pointer BASE and number of elements NMEMB. The comparison function takes two pointers to the locations of two elements in the array, and should return a negative number if the first element is less than the second, a positive number if the first element is greater, and zero if they are equal. The syntax with the extra parentheses indicates that the argument COMPAR is a function pointer. For the purposes of this project you can just pass the name of a function for this argument, as long as it has a prototype like `int compare_fn(const void *a,   const void *b)`.

- `strcmp`: `int strcmp(const char *s1, const char *s2);'   The`strcmp‘ function takes two pointers to strings, and returns an integer indicating which one is larger (according to an alphabetical order by byte values). A negative return value means that the first string is less than the second string, a zero return value means that the strings are equal, and a positive return value means the first string is greater.
- `strncmp`: `int strncmp(const char *s1, const char *s2, size_t n);` The `strncmp` function is similar to `strcmp`, but the parameter N limits how much of the strings it examines. Only up to the first N characters of each string are examined, so if the first N characters of the strings are equal, the return value will be 0.

Testing

We have included some sample blacklists and mailboxes that you can use for testing. But these aren't enough to reveal every possible bug in your code, so you should also create some small examples of your own.

The blacklist file `meat.lst` contains a relatively short list of words related to meat, and the messages file `restaurant.mbox` contains some messages between the employees of a vegetarian restaurant. Running your program with these files should reveal a list of employees who talk about meat. Specifically you should see the following output:

```
From: Charlie <charlie@vegout.example.com>
From: Ethan <ethan@vegout.example.com>
From: Gwen <gwen@vegout.example.com>
```

As an example of a larger blacklist file, the file `non-french.lst` contains a list of almost 50,000 words, which are all English words that are not also French words. Thus this list could be used to check whether English words are sneaking into email discussions that are supposed to be held in French. The sample mailbox `french.mbox` has a few different messages about Linux written in French, but they will all be flagged (because of English words in the outputs of programs being discussed, and English words that appear as substrings of French words). You can use this blacklist to check that your implementation deals okay with large blacklists: for instance, checking the `non-french.lst` blacklist against the messages in `french.mbox` should still run just about instantaneously.

Expectations

This project must be done individually. It is not a lab activity, and it is meant to test your C programming abilities, not the skill of your friend or the code written by other people and posted online. But the course staff want to help you, especially if there are parts of the instructions that seem unclear or incomplete. The best place for these discussions, as long as they don't have significant spoilers, in on the Piazza forum where everyone can contribute to and benefit from them. Questions that unavoidably include spoilers should best go to the staff mailing list or office hours.

One of the most important things you can do to do better on a project like this is to start it early. This project doesn't need an enormous amount of code (it can be done in less than 300 lines, including blank lines and reasonable comments). But you'll need to think carefully about what that code should do, and allow time for getting temporarily stumped, asking clarifying questions, and debugging problems.

Grading

The project will be graded in two ways.

1. Automated grading scripts to test the accuracy of the output and the efficiency of the program.
2. Manual grading by course staff reading and trying to understand your code.

The grade breakdown will be:

- Binary search tree operations: 30%
- Loading the blacklist: 20%
- Processing messages, avoiding duplicate reports: 30%
- Code clarity, style, and comments: 20%

Note

A portion of the grading will be automated, so it is important that your program compiles correctly as submitted, takes its inputs in the format specified we specified, and produces its output in the format we specified. You will get only partial credit if you have the right concepts, but your program does not do what it is supposed to do.

On the other hand, we also need to be able to read your code. Writing code that others can read is important for making software in the real world. And in this grading context, we will more likely to be able to award partial credit for code that doesn't always behave correctly if we can see that you understood other parts of the task correctly.

Submitting

This assignment is due on Wednesday, February 19th by 11:59 pm. Please review the course syllabus for the late submission policy.

Please submit your solution to this lab on Canvas. We ask that you only submit one .c file that contains all of your code. Please name the file:

`email-guardian_<X500>.c`

Where `<X500>` is the part of your UMN email address before the `@umn.edu`, also the same as your CSE Labs username and the sometimes called an Internet ID or X.500 identifier.

For example, if your email address is `goldy001@umn.edu`, your filename should look like:

`email-guardian_goldy001.c`