Assignment 1a: Scanning and Text Analysis

Assignment 1a was developed by Julie Zelenski.

Overview

Over the next three weeks, you will develop an advanced C program in stages. The program stresses several qualities which make it representative of advanced, professional level C work:

- it's large and complex
- it's been broken into separate modules which can be written and tested independently
- it uses `void *`, `malloc`, `realloc`, etc. to build data structures which are efficient and highly reusable.

For the first week, you will write a `Scanner` ADT that breaks apart a file of text into its component words and then put the scanner to work in a client program that reports on the word usage in the `leland` home pages. This first piece is a small piece isolated from the larger assignment and assigned early to give you a chance to refresh your C skills—in particular, those of dynamic allocation and string handling, as well as give you a chance to learn UNIX. You will use these utility components in the later parts of the assignment.

Due Tuesday, October 10th at 11:59 p.m.

The Scanner ADT

A token scanner is a tool that is used to divide a stream of text into words, or tokens, usually using the space and punctuation characters to delimit the words from each other. Some of you may already be familiar with the scanner abstraction from CS106 (covered in Chapter 10 of Eric Roberts' CS106A text). A scanner is designed to make it easy to dissect words, numbers, etc. from a stream instead of relying on C's formatted I/O function `scanf,` which is fairly awkward to use and inadequate for many sophisticated parsing needs.

Our particular version of the scanner will pull tokens out of a text file. A new scanner can be created from an already opened `FILE *` or by opening a file by name. The client can repeatedly call `ReadNextToken` on the scanner to retrieve tokens one by one from the file, starting at the beginning. This function returns a Boolean result which will be `false` when all the tokens have been extracted so the client can determine when the scanner is at the end of the input stream.

The interesting bit concerns how the scanner determines what constitutes a token. When a scanner is created, the client supplies a set of delimiter characters that is saved by the scanner for use later when forming tokens. The function `ReadNextToken` accumulates characters for a token, stopping when it encounters a character in the delimiter set. If the first character is itself a delimiter, then a single character token consisting of just the delimiter is formed. A configurable feature of the scanner determines whether these delimiter tokens are returned or whether they are discarded and the scanner continues on looking for the next non-delimiter token instead.

For example, if the scanner's delimiting set consisted of just the white space characters (space, tab, and newline), delimiters were being discarded, and the next characters coming up in the file were `"I love    CS!"` , a call to `ReadNextToken` would first extract the string `"I"`, the next call would get the string `"love"` (because the delimiter token " " was discarded) and finally `"CS!"`(discarding several intervening space tokens). If delimiters were not being discarded, each space character encountered would be returned as a one-character token. As another example, assume the delimiters are just the digits 0 through 9, delimiters are not being discarded, and the next characters are `"I am 9 years old`". The first call to `ReadNextToken` would produce the string `"I am "`, the next call would produce `"9"` (since delimiters were not discarded), and next gives the string `" years old"`. Note that whitespace can be included as part of token in this case, since it was not included in the set of delimiters. Each scanner has its own string of delimiting characters and a policy about discarding delimiters, both established when the scanner was created, and they do not change over the lifetime of that particular scanner when reading tokens.

One important difference between our scanner and the 106 version is the memory management for the token storage. In our version, the client is responsible for providing an array for the scanner to write the token characters into, rather than having the scanner allocate space and return a pointer to it. The 106 version was convenient, but is somewhat problematic in that the scanner allocates the memory, but can't free it because client is the only one who knows when the storage can be reclaimed. It is undesirable to split the allocation and free tasks in this way. Also, if the scanner allocates, the memory must come from the heap, which is expensive. If the client supplies the storage, they are free to use cheaper stack or reusable storage as appropriate. Given there is usually some upper bound on the maximum token length, the client knows best how to set that bound, so it also makes sense that they do the allocation. This type of arrangement (where client supplies a buffer when calling a library function) is a common idiom in standard C programming so it is good to become familiar with it, since the 106 approach you've seen is actually rarely used and you'll want to learn the more usual patterns.

Our scanner also has two "fast forward" functions that allow the client to quickly scan past characters when they know what they are looking for. `SkipOver` scans and discards all the characters in the given set passed as a parameter, stopping at the first character not in the set. `SkipUntil` operates similarly but skips over characters *not* in the given set and stops at the first character *in* the set. The two functions are quite similar, but rather than copy-and-paste between the two, it is a better design to avoid duplication and create one utility function that unifies the code common to both.

A few things to note when designing your `Scanner`:

- The `scanner.h` interface file (at end of this handout) has comments about the function details. Read this **very** carefully, since it contains all of the specifications your scanner implementation must conform to. You should become accustomed to reading program specifications supplied in this form, as this is an important and valuable skill.

- The `Scanner` is defined as an incomplete type, as we did for ADTs in CS106. Basically, the `typedef` that is given in `scanner.h` defines a new public type `Scanner` that is a pointer to a `struct ScannerImplementation`. The client does not and should not know what this `struct` looks like. Behind the wall, in the `scanner.c` file, the implementation completes the type by defining the `struct`:

  ```
  struct ScannerImplementation {
     /* your fields here */
  };
  ```

  What's interesting about this scheme is that it strictly enforces the separation between client and implementation by not giving the client enough information to break the wall.

- Note the use of the ANSI `const` qualifier on various parameters which denotes variables whose value cannot be changed. Although many compilers do not rigorously enforce this stipulation, it is valuable for documenting which pointer-valued parameters to a function can be altered and which can't. For example, consider the function `strcpy` whose prototype is `strcpy(char *a, const char *b)`, even without good parameter names, it is clear which string is the source and which is the destination.

- The `Scanner` should not scan the whole file into one long string and then pick it apart using string operations, instead it should read each character singly using `getc`. Reading ahead (either line by line or whole file) unnecessarily increases the memory requirements and complicates the code. You'll have to analyze each character as it passes through anyway, so it's easier to just deal with the incoming stream on a character by character basis.

- Note the two skipping functions have a return value of `int` instead of `char`. Although in most cases they will be returning a character, if they stop at the end of file, they need to return the value `EOF`, which cannot be represented in a `char` and requires an `int`.

- Feel free to use any of the standard C library's string, file, and memory routines to help you out when implementing the scanner. It's always better to use an optimized and debugged library routine than to re-invent the wheel. As a little hint, you will want to use `strchr` to deal with searching for a character in a string and you will need `ungetc` to push a character back onto a stream. Using the `man` command on Unix systems will give detailed information about any library call, for example, `man strchr` will give the prototype and description of the function.

Moving on from CS106

This assignment requires a bit more explicit handling of strings using ANSI string functions than those of you steeped in the CS106 string library are used to. For many of you, this will be your first time directly using the standard ANSI memory, string, and file functions (having grown up on `New`, `GetBlock`, etc.). The 106 libraries are layered on top of the ANSI libraries with the goal of providing a more user-friendly and safe interface to the underlying functionality. If you refer back to your 106 materials, you can learn exactly what is going on behind the scenes and you'll find that there's nothing magical in their implementation. We think it's important to learn how to use the C libraries in their standard and portable form, as you would in a professional setting.

Students often want to copy or adapt excerpts of the 106 libraries for their 107 programs. You'll learn the most by writing the code yourself so that's what we recommend, but it's certainly fine go back over the internals of the `genlib` and `strlib` to strengthen your understanding of how they are constructed. Directly including code from there is also okay with us as long as you properly cite the source and definitely make sure you understand any of the code that you include.

Putting the scanner to work

The rest of your assignment is to use the scanner to parse a collection of text files into their component words and provide a report on the frequency of words found. We will create a list of the leland ids of the class members and you will use your scanner to chew through their home pages, keep a tally of the words found, and then print out the list of word frequencies when done.

Each of their usernames is listed in the file `/usr/class/cs107/assignments/students` available on `leland`. (We will be updating this all week as more people register for the course mailing list). On `leland`, user accounts are divided into directories using the first two characters of the username. Within their home directory, a user's home page is found in `WWW/index.html`. For a given `leland` user, e.g. `cain`, construct the path like this: `/afs/ir/users/c/a/cain/WWW/index.html`. You only need to parse the main home page (`index.html`) for each user, don't worry about any other html files they may have in their WWW directory.

Rather than start you staring at a empty file, we've given you a leg up by providing you with a working program that uses the scanner. This starting program takes one argument, a username, and opens that user's home page, uses the scanner to extract the words, and then prints an alphabetized list of the unique words found. Your job is to take this program and change it so it instead takes a filename containing a list of usernames and will count the word frequencies on all of those users' home pages and report the collected result.

In its final version, your client program should take one argument, the filename containing the usernames, and open the user's home pages one at a time (skipping those who don't have a home page) and scan through each page extracting tokens. For each new word found, add an entry into your array of found words and continue to count how many times that word appears in all the home pages. Your final program should have the same basic features as the starting version: you should scan using the same set of token delimiters; you should only count the frequency of words that are at least four letters long and that pass the supplied `ContainsOnlyAscii` function; any words longer than 100 will be chopped into pieces during tokenizing; you only need to track the first 10,000 unique words you encounter; your list should be case-insensitive (so "Stanford" and "stanford" are considered the same word) and you should ignore all words within HTML tags (i.e. skipping over everything inside angle brackets).

After parsing all the home pages, sort your array of word counts by decreasing frequency and sort ties alphabetically. The ANSI function qsort should come in handy here, check out the man page if you need a refresher on the syntax. After sorting, print the list out in order of decreasing frequency, words that tie for frequency should be sorted alphabetically. Your list should simply be a list of frequency and word, just like this:

```
epic5> analyze /usr/class/cs107/assignments/students
   121   Stanford
   110   this
      ...
    38   university
      ...
     1   zygote
```

Your program is using a fairly naive and limited approach that we will greatly improve in our next assignment. Don't worry about creating a better data structure or faster lookup, there is no need to do anything clever with it this time; that will come later.

The `assert` macro

When you are implementing a reusable library component, it is best if you can guarantee that your code will behave reasonably, even if it is used improperly by a client. For example, if a client attempts to scan a file that doesn't exist or use an empty set of delimiters, it is much better to gracefully report the problem than to blunder through and dereference `NULL` pointers.

The standard `assert` macro can be used to detect and report exceptional conditions. This macro takes one argument, an expression which is evaluated and if true (non-zero) execution continues on, but if false (zero) results, it will print an error and terminate the program. The idea is to **`assert`** what must be true before carrying out an operation that depends on those assumptions.

```
#include <assert.h>     // need to include this to see macro definition

{
   int i, scores[MAX_SCORES];

   i = SomethingComplicated();  // i should be in bounds, but is it really?

   assert(i >= 0 && i < MAX_SCORES);   // assert before we access
   scores[i] = ...
```

The scanner has a few useful places for `asserts` to ensure no important assumptions are being violated, such as checking that the parameters are valid or verifying the return value from `malloc` (which is something `GetBlock` did in the 106 libraries and a function you may want to recreate for yourself.) Getting in the habit of coding "defensively" from the beginning is an important component of bringing discipline to your programming. And, a few seconds putting in `assert` statements can save you hours of debugging later.

Some general design guidelines

We will distribute some handouts that help to identify our C style, decomposition, and documentation standards. You will definitely want to look those over to familiarize yourself with our standards, especially if you didn't come through the CS106 classes. Also the starting home page client program gives you a good example to learn from about what we consider well-written code. In addition, here is some basic information about our expectations about your programs:

- Your code should be reasonably well-documented. You don't need to write a book about every function, but do provide information about your design choices and definitely explain the more intricate parts of your code.

- Code repetition is to be avoided whenever possible. Factor common code into helper routines, unify similar passages, avoid copy-and-paste coding with a passion.

- Decomposition is essential. Always break tasks down into small, self-contained, well-named subroutines. Not only does it please us, it will make your job much easier. Long, complicated functions are a mess to write, test, debug, and maintain.

- In general, global variables are rarely appropriate, and in this program, there is no compelling need for any.

- You should free all dynamically allocated memory when it is no longer in use, including cleaning everything up at program termination. Even a small leak for each word or page could grow to mammoth proportions if indexing many pages because of the large amount of data to process. You also want to be careful about closing files, since many UNIX systems won't allow you to have more than 128 files open at a time.

- We expect that your code compile cleanly— i.e. without any warnings— and that it run under Purify (see below) without finding any errors.
- Please do not modify our `.h` files or specifications (i.e. your scanner should have same return values, your analyze program should use the same delimiter set and maximum token length we start you with, etc.) Think of an assignment as a consulting job and the `.h` is the contract to which you are held. You must meet its exact specification to please your demanding employer and are not free to change things to suit your design.

Purify

Purify is a development tool from Pure-Atria Software.[1] It checks your program at run-time for many common errors, such as memory leaks, use of memory after it has been freed, overflow and underflow of memory regions, use of uninitialized or illegal addresses, etc. Purify works by "instrumenting" your compiled code with its checkpoints and creating a new executable which will run your program while Purify keeps tabs on things. If your terminal has windowing capabilities, it will show the status graphically in another window. If not, it will report the errors by printing on your terminal. Purify can be very helpful in tracking down those difficult-to-find memory errors and can find lurking errors that you didn't even know existed. We will be running your submissions against Purify to check for errors, so you will want to find them first and fix any problems before submitting. Please don't think of Purify as a bothersome extra step to do at the very end before submitting, it is an excellent tool that can help you all along the development cycle to rid your program of memory problems.

We have included a target for Purify in the starting makefile. You can make a Purify-ed version by saying `make pure` and then run the result as `analyze.purify`. Be default, it will place its output in the text file `purify.log`. You can learn all about Purify from its on-line documentation (see `man purify`). We will do a quick demo of Purify in an upcoming discussion section. One small detail to make sure of is that you copy the `.purify` file from the starting project directory with the rest of the files since it contains important set-up information for correctly running Purify.

Getting started

We have a class directory on leland /usr/class/cs107 where we will place course materials. For this assignment, there is a project directory hw1a which contains the scanner.h and the skeleton scanner.c and analyze.c, along with a makefile to build the project. You will want to make your own copy of the project directory (i.e. cp -r /usr/class/cs107/assignments/hw1a ~) to get started. You can also get the starting files via anonymous ftp to ftp.stanford.edu or from the class Web site

[1] We are eternally grateful to Pure-Atria for freely donating their software for students to use in Stanford classes. Purify is a neat tool and we definitely recommend using it early and often.

http://cse.stanford.edu/classes/cs107. Be sure to copy over all the files, including the hidden .purify file which is needed to set up the Purify environment correctly.

Electronic submission

You will deliver your project directory on-line and the instructions for electronic submission are at http://cse.stanford.edu/class/cs107/submit.html. In addition to the e-submit, you also need to bring in a paper copy. If you're totally on the ball, you might complete it in time to bring your printouts to Wednedays' lecture. Otherwise, slip it under my door as soon as possible.

Assignment 1a Deliverables

Thus, here are three things you need to do for the assignment due Tuesday by midnight.

1. Electronically submit your project. The electronic script has no mercy for things submitted even just 10 seconds past midnight, so be sure to give yourself time to complete this step.
2. Bring in a printout of `scanner.c` and `analyze.c` files. Your paper copies are due at the next lecture following when you e-submitted (will likely be Wednesday.) Late days count on paper copies, even when the e-submit was on time, so please don't forget. Also your paper copy must match the electronically submitted version. Making changes in a paper copy handed in later than your e-submit will result in your homework being counted as submitted at the later date.

```
#ifndef _SCANNER_H
#define _SCANNER_H

#include <stdio.h>

/*
 * bool type
 * ---------
 * A true boolean type is a little detail sorely missed in C. Here's a
 * handy typedef for a simple enum version of it. This is just like the
 * one you used back in CS106.
 */

typedef enum {false, true} bool;

/*
 * Scanner type
 * ------------
 * The Scanner ADT is a simple token scanner that provides the ability to
 * divide up a file token by token.  The client can declare variables of
 * type Scanner, but they must be initialized with NewScannerFromFilename or
 * NewScannerFromFile. The Scanner is implemented with pointers, so all client
 * copies in variables or parameters will be "shallow"- they will all
 * actually point to the same Scanner structure.  Note that the Scanner is
 * defined as an incomplete type, a pointer to a struct that will only
 * be detailed in the scanner.c file, far from the prying eyes of any
 * Scanner client trying to break the abstraction wall.
 */

typedef struct ScannerImplementation *Scanner;

/*
 * NewScannerFromFilename
 * ----------------------
 * Allocate and return a new Scanner. The specified filename specifies the
 * file to open for scanning.  If the filename is NULL or the named file
 * doesn't exist or can't be opened, a NULL scanner is returned.  The
 * delimiter string lists the characters to be treated as token delimiters.
 * The boolean discardDelimiters controls whether delimiters are returned as
 * tokens. (see more info about delimiters in the spec for the ReadNextToken
 * function). A copy of the delimiter string should be made and kept by the
 * scanner. (i.e. you should not just store the pointer since you don't know
 * what will happen to that space later.) There is no limit on the number of
 * characters that the client may give in the delimiter set, so the scanner
 * should not make assumptions that introduce any fixed bound. If the
 * delimiter parameter is NULL, an assert is raised.
 */

Scanner NewScannerFromFilename(const char *filename, const char *delimiters,
                               bool discardDelimiters);
```

```
/*
 * NewScannerFromFile
 * ------------------
 * Like the creation function above, but used to scan a file which has already
 * been opened. This is needed when scanning pre-opened files such as stdin.
 * If a null file pointer is passed, a NULL scanner is returned.  See docs
 * for NewScannerFromFilename above for information about delimiters.
 */

Scanner NewScannerFromFile(FILE *fp, const char *delimiters,
                           bool discardDelimiters);
/*
 * FreeScanner
 * -----------
 * Free all the storage for the given scanner. If the scanner was the one
 * to open the file (i.e. it was created via NewScannerFromFilename not
 * NewScannerFromFile), it also closes the underlying file. (It would not
 * be appropriate for the scanner to close a file that itself did not open.)
 * It is an error for the client to continue to use a scanner after it has
 * FreeScanner has been called on it.
 */

void FreeScanner(Scanner s);

/*
 * ReadNextToken
 * -------------
 * Scans characters from the file to form a string in the client-supplied
 * char buffer. The client must pass a buffer which is valid storage and
 * gives the length of the buffer in the bufLen parameter. This function
 * extracts the next token from the scanner and writes it into the buffer.
 * The token written into the buffer is null-terminated by this function.
 *
 * Return value:
 * This function returns a boolean result which indicates whether anything was
 * written into the buffer. If a valid (i.e. non-empty) token was found
 * and written into the client's buffer, the function returns true. If
 * the scanner is at EOF and there are no more tokens to read (or if the
 * the only characters left are delimiters and they are being discarded),
 * it will return false.  Think of the boolean result as communicating to
 * the client whether there was a token written into the buffer that needs to
 * be processed. If the scanner writes a token, it returns true, false
 * otherwise. Note that if the scanner reads the last token and stops at EOF
 * it still returns true (since a token was written to buffer), but the next
 * call to ReadNextToken will return false (since there are no tokens left).
 *
 * How it forms tokens:
 * ReadNextToken starts by examining the next character in the file.  If
 * the char is a member of the delimiter set, a single-character string
 * of just that delimiter will be formed. If discardDelimiters is false,
 * it returns it. If discardDelimiters is true, the delimiter token is
 * discarded and it continues on looking for a non-delimiter token. When a
 * non-delimiter character is found, it keeps scanning characters and
 * accumulating a multi-character token into the client buffer until the
 * first delimiter character (or EOF) is seen. The stopping delimiter
 * char is left in the input stream for next time.
 *
 * What happens when client buffer is full:
 * The client provides the buffer and indicates how big it is. If while
 * reading a token, the scanner exhausts the buffer before end of the token,
 * it truncates to what fits in the client buffer (leaving space to null-
 * terminate). The next call to ReadNextToken will pick up where it left off.
 * Thus, long tokens are chopped into pieces. No error message is printed or
 * assert raised when this happens. For example, if chars coming are
 * "antidisestablishmentarianism" and the clients scans into a 10-char buffer,
 * it would first pull out "antidises", then "tablishme", etc.
 *
 * Error conditions:
 * If the buflen parameter is less than 2, an assert is raised.
 */

bool ReadNextToken(Scanner s, char buffer[], int bufLen);

/*
 * SkipOver
 * --------
 * Skip over characters in the file so long as they occur in skipSet. For
 * example, to skip over the common white-space characters, pass skipSet
 * as " \n\r\t" -- the function should keep reading characters as long
 * as they are white-space and stop at the first non white-space character.
 * The character that halted the skipping is left in the input stream so
 * that it will be the very next character read (i.e. it is not skipped).
 * This character is returned as the function result in case the caller
 * wants to know about it. Hitting the end of file also stops the skipping
 * and returns the value EOF. An assert is raised if skipSet is NULL.
 */

int SkipOver(Scanner s, const char *skipSet);

/*
 * SkipUntil
 * ---------
 * Like SkipOver, but it skips UNTIL it gets to a character in the untilSet
 * For example, you could use SkipOver to jump to the end of the sentence
 * by using the untilSet of ".!?" which would scan over characters until it
 * encountered the first period, exclamation or question mark. As with
 * SkipOver, the halting character is left in the input stream and returned
 * as the result from the function.  Again, it will also halt and return EOF
 * if it encounters the end of file. An assert is raised if untilSet is NULL.
 */

int SkipUntil(Scanner s, const char *untilSet);

#endif
```