Assignment 1a: Scanner Feedback

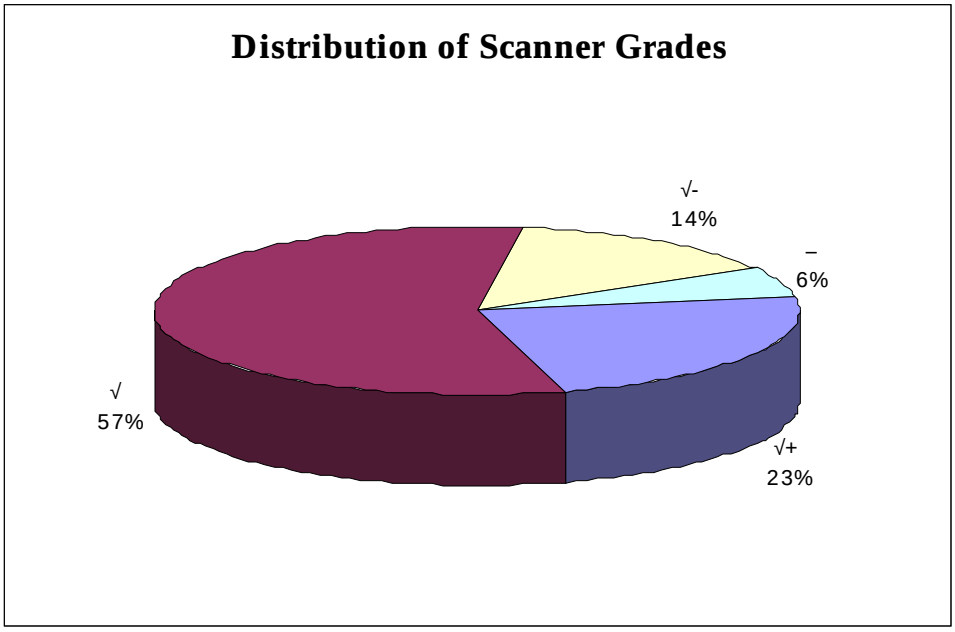
Feedback originally written by Julie and revised by Jerry.

General Comments

Overall, the programs were very nicely done. The abstraction wasn't the most difficult, and the analyze program didn't need all that much of a facelift, but this first assignment represented so much more work than that. In light of your time spent getting acclimated to UNIX, `emacs` and `gdb`, you should all be pretty happy with yourselves.

Grading Scale

CS107 has joined the CS106 program and adopted the bucket grading scheme. In the past, we've always graded on points, but I felt that less granularity was needed specifically because it's rarely the case that a 22/25 and a 23/25 represent something different. To recognize that one point difference as significant would be to allow that one point separation to possibly separate the lowest A- students from the highest B+ student, and I just think that's silly. With the bucket scheme, we have broader authority to make sure that a particular submission falls into the proper bucket.



The most annoyed contingent was, not surprisingly, the √ population. I understand that the mystique of the bucket system obscures exactly how far you were from getting a √+. My primary defense is this: If you received a √, it was because the grader noticed a significant gap in quality between your program and those receiving a √+. √+ programs nail everything—all oppurtunities for code reuse are taken, all edge cases of `GetNextToken` are considered and tested (now how many of you tested to see how your Scanner worked when the buffer was completely saturated?). Your `Scanner` isn't solid

just because it works well underneath `analyze`; `analyze` never discarded delimiters, nor did it advertise whether the last legitimate token read was ignored because `false` was returned one call too soon. Some of these errors are minor, and some aren't. When you only have six functions to write, and three of them are no-brainers, you best make sure the meatier ones are written cleanly and properly. At this point, you don't get points for good variable names or because you knew to use a `for` loop.

I think your primary handicap was your lack of a familiar debugger, and your demotion to UNIX and a run-time environment that issues vague segmentation faults. With that understanding, you have even more reason to program a more defensively than ever before. Write one function at a time, and then test it to death. Comment out my portion of the test code which doesn't work yet. Play with the code you keep. Set the buffer size to be 10 instead of 100. Then set it to be 5. Then 2. Write your own test functions; don't just use mine. Try to break your code, or if you don't have the heart and soul to do it, let someone else.

My TAs expressed frustration with the new grading system, but only to the extent that they want a bucket in between √ and √+. Recall than a √+ is A work and a √ represents A-/B+ work, so it's difficult to splice another one in. What we've decided on instead is the separation of the one grade into two, so from here on out you'll received a bucket grade for design and a separate bucket grade for correctness. With this change we'll hopefully capture assignment performances more accurately.

Finally, rest assured that adjustments will certainly be made in situations where one TA's median grade is statistically lower than anothers, provide that the exams suggest that adjustments should be made. In layman's terms, don't worry if you were unfortunately assigned to the gestapo grader of the millennium. I totally respect that grades are very important to all of you, and even though I can't give all As and A-s, what As I do give will be evenly distributed over all students.

Memory issues

- When allocating memory for a new scanner, the call to `malloc` should be sent the size of the structure, not of the pointer. Several people tried to allocate a scanner with the following incorrect code:

```
Scanner s;
s = (Scanner)malloc(sizeof(Scanner));        /* Not quite!!! */
```

  Since `Scanner` is just a pointer, this line will allocate only a four-byte chunk of memory (enough to hold the pointer), which is not enough for the full structure. The correct way to allocate the `Scanner` would be something like this:

```
s = (Scanner) malloc(sizeof(*s));
```

or

```
s = (Scanner) malloc(sizeof(struct ScannerImplementation));
```

Purify should be able to spot such an error as soon as you start writing to space past the first 4 bytes, so hopefully most of you found and fixed this one before it got too far.

- Get into the habit of checking the return value from `malloc`. (In fact, this is an excellent reason for decomposing to a `GetBlock` helper.) A `NULL` return value means your allocation request was not satisfied, and you certainly want to know about that sooner rather than later. On an operating system like UNIX, it actually isn't likely that you have run out of memory, but a `NULL` return value can signal that the heap's internal state is damaged, something you will want to know about as soon as possible.

- Be reasonable about memory use. It is not good practice to malloc a massive block of persistent memory, and then only use a few characters of it. Large space allocated on the stack is fine, since it is guaranteed to be temporary.

String handling

- Strings always need to have space reserved for the terminating null byte. This means allocating a buffer of size strlen +1, not just strlen. And by the way, `'\0'` is the null character, a zero byte. It is not quite the same thing as `NULL`, the zero pointer, which is a 4-byte zero pointer.

- In `ReadNextToken`, you need to handle a myriad of termination conditions: a single delimiter by itself, a token ended by a delimiter, a token ended by `EOF`, a token ended because it hit the maximum limit, no token because already at `EOF`, etc. Try to unify the code paths as much as possible, since the more special case code you have, the more code to deal with and debug. Be careful about handling all the edge cases in a loop like this. For example, some people dropped the last character read in when the buffer was full. While this seems like a small detail, it will result in errors for the next token read in. `ReadNextToken` was the only algorithmically challenging function in the scanner, and more care should have been taken to get this right.

- The comments for `ReadNextToken` indicated that the client supplies the buffer along with its size to be used the upper bound on the maximum token length. It is assumed that the client has chosen a sufficiently large upper bound so that it will practically never be encountered. However, that doesn't mean you can act as though the limit doesn't exist when writing the code! You must detect when this maximum is hit and truncate the token, not overrun the buffer and crash.

- You should have used `strlen`, `strcmp`, and `strchr` for the string manipulations required. These operations were used multiple times, and repeating the raw code everywhere is not a good idea. At the very least, you should have decomposed the work into reusable functions, but the best solution was to use the already-written and debugged ANSI library versions.

- Unlike the 106 `strlib` functions, all of the ANSI string functions expect you do the necessary allocation. So the first argument to `strcpy` or `strcat` must already be allocated to the correct size before trying to copy or concatenate to it. Also all strings passed to these routines must be properly null-terminated.

- String literals (i.e. compile-time constant strings inside double-quotes) are allocated in read-only memory. You can use a string literal as an argument to any function which just reads the string (such as the second argument of `strcat` or `strcpy`) but you cannot change or write to the string. Since UNIX implements proper memory protection, any attempt to write to a string literal (such by using it as the first argument to `strcat`) will immediately produce a protection fault. You also do not free string literals as they are not in the heap. Remember that you only free things that were `malloc'ed` earlier.

Scanner ADT

- You should only include variables in the struct that are needed to maintain scanner state between calls. For example, some Scanner structures had a character field for the most recently read character or a char * for the current token, but these variables should just be local variables of the `ReadNextToken` function since they are only used within that context.

- Note that the interface comment for the two `NewScanner` functions made it clear that the implementation was to make a new copy of the delimiters string, not just store a pointer to the same string being passed in. This was to avoid any problems with aliasing between the two strings.

- It was disappointing to see how many of you failed to unify the common code from `NewScannerFromFilename` and `NewScannerFromFile`. In fact, the first should just use `fopen` and then call the second, simple as that! Before you repeat or write any code, first consider if you've got some similar functionality elsewhere that you can re-use, and we don't mean "copy and paste" re-use.

- It was a poor design decision to scan the entire file into a string and then parse tokens out of it, or worse, scan line by line. For a large file, this dramatically increases the memory requirements, and can degrade the performance, especially when the user only wants to scan a few tokens and stop. It also tended to make the code more complicated. The desire to improve performance by this sort of batch read rather

than reading one character at a time is understandable, however, you should realize that `getc` already does this for you— it reads in a large block and then parcels out characters as needed, so this approach isn't gaining anything.

- Some students assumed that all whitespace characters were automatically included in the delimiter set, but that is not what the specification dictated. A token can contain embedded space characters if those characters are not in the set of delimiters given by the client.

- As you accumulate the token, you need to add characters one by one to a buffer. Some of you went about this in a rather awkward and inefficient manner by creating a one character string for each character and then using `strcat` to append to the token so far. Why not just assign characters in the buffer using array notation?

- The `SkipXXXX` functions only seem to be different until you write the code for them. They are actually logical complements of each other, and should be written in terms of a single helper function. For example:

```
static int SkipHelper(FILE *in, bool untilP, const char *set)
{
   int ch, inSet;

   while ((ch = fgetc(in)) != EOF) {
      inSet = (strchr(set, ch) != NULL);
      if ((inSet && untilP) || (!inSet && !untilP)) {
         ungetc(ch, in);
         break;
      }
   }

   return ch;
}

int SkipOver(Scanner s, const char *skipSet) {
   return(SkipHelper(s->file, FALSE, skipSet));
}

int SkipUntil(Scanner s, const char *untilSet) {
   return(SkipHelper(s->file, TRUE, untilSet));
}
```

By leveraging the fact that the logical tests return exactly 0 or 1, you can even combine the two tests into something like this (although this is bordering on unreadable):

```
if (inSet == untilP) {
```

- Remember that you need to work with `int`s not `char`s, when using `fgetc` to read from a `FILE`, since `EOF` needs to be distinguished from other valid characters. (see discussion p.16 K & R).

- You do not use `ungetc` to put `EOF` back into a stream. `EOF` is not a marker character that is read and needs to be put back. It is a return value from `getc` that indicates there are no more characters to be read. Many implementations do not do anything wrong in this situation, but it is not guaranteed to work and thus you shouldn't write code that depends on it.

- Don't forget to `fclose` the file in `FreeScanner`, if you opened it, as the comments required. Many UNIX systems have a hard-wired 128-entry file table and any attempt to open a file after 128 are already open will fail silently and return `NULL`, so you must close the ones you are no longer using so that other files can be opened. On a related note, you should not have closed any file you didn't open. If a client uses `NewScannerFromFile`, the file was already opened and closing when you free the scanner is exactly the wrong thing to do. (For example, consider if the file being used was stdin, which comes pre-opened and never should be closed...)

Home page analyzer

- The analyze client should have taken one argument, which was the filename containing the usenames. Your program was supposed to read the file from `argv[1]`, and not just hard-code a specific path. Programs that don't implement the required interface are tough for our automated tools to test and require manual adjustment by the TAs, which is annoying and inconvenient. Please read the handout carefully and make sure your program conforms to our specification.

- Although capitalizing on the `Scanner` for parsing the individual home pages was obvious, many students missed the opportunity to also use it for extracting the usernames from the `students` file. That meant their client had a gob of file-reading code which could have been entirely replaced by use of the handy `Scanner`. Solving a problem a second time is inadvisable not just because it adds effort and little or no value, it opens up many more opportunities to introduce previously removed errors back into the program.

- When passing a function pointer as a parameter, the compiler requires that the function prototype exactly match the declaration. For example, `qsort` is declared like this:

```
void qsort(void *base, size_t num, size_t width,
           int (*cmpfn)(const void *, const void *));
```

  That last argument is a function pointer for a function that takes two const `void*` arguments and returns an int. Any function pointer passed to `qsort` must **exactly** match that prototype without exception. You couldn't pass `strcmp`, for example, since it takes two char *s. And the `const` matters, so a function that took two `void *`s (not declared `const`) also won't match.

- Starting with a large body of already-commented code is a big leg up, but if you end up changing that code, you really should take the time to update the comments to match. There's nothing less helpful than an out-of-date comment that confuses the reader more than if there was no comment at all.

Other issues

- Please do not modify our .h files or specifications, e.g. don't change the prototypes or use a different delimiter set, etc., although it is okay to make small additions (expand the comments, add a utility function, and the like). Think of each assignment as a consulting job and the .h is the contract to which you are beholden. You must meet its exact specification to please your demanding employer and are not free to change things to suit your design. Also, since you will not edit our supplied .h file, you should not need to print it and hand it in.

- Internal functions should be declared to be `static`. Otherwise, they default to be globally visible, which may cause "multiply defined" problems for the client at link-time because of conflicts in the global namespace.

- If you want to define a helper function and use it in more than one module, you don't really want to copy and paste the code to duplicate the function. Instead, you can make a third file which contains any shared utility functions and use them in both. You can probably figure out how to edit the `Makefile` to include this extra file; if not, just ask one of the staff to show you.