15-122: Principles of Imperative Computation, Spring 2013

Homework 7 Programming: Strbuflab

Due: Monday, April 15, 2013 by 23:59

“@”
“@@”
“@@@@@@”

For the programming portion of this week's homework, you'll write a small library for string manipulation in C.

- `strbuf.c` (implementation, described in Sections 2–6)
- `strbuf-test.c` (tests, described in Section 7)

You should submit these files electronically by the due date. Detailed submission instructions can be found below. The starter code does not contain these files; it is your job to write them.

Assignment: String Buffers (25 points in total)

Starter code. Download the file `hw7-handout.tgz` from the course website. When you untar it, you will find `strbuf.h` (the header file for the string buffer library for you to write), a `Makefile`, and a `lib/` directory with some provided libraries. You should not modify or hand in any of the given files. Instead, you should write `strbuf.c` and `strbuf-test.c`.

Compiling and running. You will compile and run your code using the standard `gcc` compiler. We have provided you with a `Makefile` that can save you from typing lengthy shell commands. You are not responsible for understanding the make file, but we invite you to read it anyway, since it is a common tool for C and Unix. You should compile and run with

```
% make strbuf-d  # Contracts checked.
% ./strbuf-d
% make strbuf    # Contracts not checked.
% ./strbuf
```

Submitting. Once you've completed some files, you can submit them to Autolab. There are two ways to do this:

From the terminal on Andrew Linux (via cluster or ssh) type:

```
% handin hw7 strbuf.c strbuf-test.c
```

Your score will then be available on the Autolab website.

Your files can also be submitted to the web interface of Autolab. To do so, please `tar` them, for example:

```
% tar -czvf sol.tgz strbuf.c strbuf-test.c
```

Then, go to `https://autolab.cs.cmu.edu/15122-s13` and submit them as your solution to homework 7 (Strbuflab).

You may submit your assignment up to 25 times. When we grade your assignment, we will consider the most recent version submitted before the due date.

Annotations. Be sure to include appropriate `REQUIRES`, `ENSURES`, and `ASSERT` macros in your program. If you write any auxiliary functions, include precise and appropriate pre- and postconditions.

You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Unit testing. You should write unit tests for your code. We recommend writing these early to check your code as you move forward. See Section 7.

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. If you find yourself writing the same code over and over, or find functions becoming very long, you should write a separate function to handle that computation and call it whenever you need it. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on Piazza if you're unsure of what constitutes good style. You may also find it helpful to refer to the course style guide at `http://www.cs.cmu.edu/~rjsimmon/15122-s13/rec/style_guide.pdf`.

Task 0 *As with the previous few assignments, we expect you to be writing code with good style; we have given you feedback on style for multiple assignments and we have shown you many examples of code with reasonable style. This assignment will initially have all 25 points assigned through the autograder, but up to 5 points may be deducted for style issues.*

1 Strings in C

Manipulation of strings in C varies considerably compared to many other common programming languages, including C0. Because of fixed-size allocations of memory, manual memory management with malloc and free, and required NUL-termination, programming with strings in C is tedious and error-prone. Many of the common buffer overflow attacks that exploit undefined behavior in C actually occur during string manipulation. Therefore, it is crucial to understand C string manipulation before beginning this assignment. The next few sections review some of the more important aspects of C strings.

1.1 Strings as Arrays of Characters

C does not actually have a `string` type. Instead, strings are represented as arrays of characters, `char*`. This means that we can manipulate them like any other array. For example, `s[0] = 'a'` sets the first character in the string `s` to `a`. Like all arrays in C, the length of the array is not stored in memory. Fortunately, we can use the `strlen` function to determine the length. This function simply iterates through the string until the NUL terminator is found. In the interest of runtime, it may still be desirable to explicitly store the length of the string yourself, instead of calling `strlen` repeatedly.

There are several ways to declare a string.

1. On the heap with `xmalloc` and `xcalloc`
   The same way that you would allocate an array of any other type:

   ```
   size_t len = 10;
   char *s0 = xmalloc(len * sizeof(char));
   char *s1 = xcalloc(len, sizeof(char));
   ```

2. On the program stack
   Similar to how you would for other types.

   ```
   char s[] = "C is fun.";
   ```

3. String Literals
   These strings are stored in read-only memory, so you cannot modify them.

   ```
   const char *s = "C is scary.";
   ```

   The `const` keyword isn't required, but the compiler may catch attempts to modify the string, instead of causing a segfault at runtime.

A trick for creating a string on the heap from the contents of a string literal, which may help in your testing code, is to use `strcpy`:

```
const char *s_lit = "Hello World!";
char *s_heap = xmalloc((strlen(s_lit) + 1) * sizeof(char));
strcpy(s_heap, s_lit);
```

1.2 The NUL Terminator

A common source of confusion when working with strings in general with C (and especially true on this assignment) is managing the NUL terminator. The NUL terminator essentially signifies the end of the string. It has an ASCII value of 0 and is represented as the character `'\0'`. Many string library functions, such as `strlen` rely on the NUL terminator; calling them when the NUL terminator is missing leads to undefined behavior.

Since the NUL terminator is a character itself, you need to ensure that there is sufficient memory to store it. For example, if I want to store "cat" in a string, 4 bytes are needed: 3 for the contents of the string and 1 for the NUL terminator. This is reflected in the diagrams shown in Section 2. Therefore, you need to take this into consideration when allocating heap memory. Note that `strlen` does not include the NUL terminator in its result. There are some situations when this behavior is desirable, but others where you must add one to the result. If you observe invalid memory accesses using `valgrind`, mishandling the NUL terminator is a common cause.

Strings that are created from double-quotes automatically include the NUL terminator, so you don't need to worry about those cases. To illustrate these points, consider this code:

```
int main () {
  char s[] = "Hello World!";
  printf("strlen: %zu sizeof: %zu\n", strlen(s), sizeof(s));
  return 0;
}
```

When run, this code prints `strlen: 12 sizeof: 13`. Recall that `sizeof` behaves differently with stack arrays compared to heap arrays.

1.3 Standard `string.h` Functions

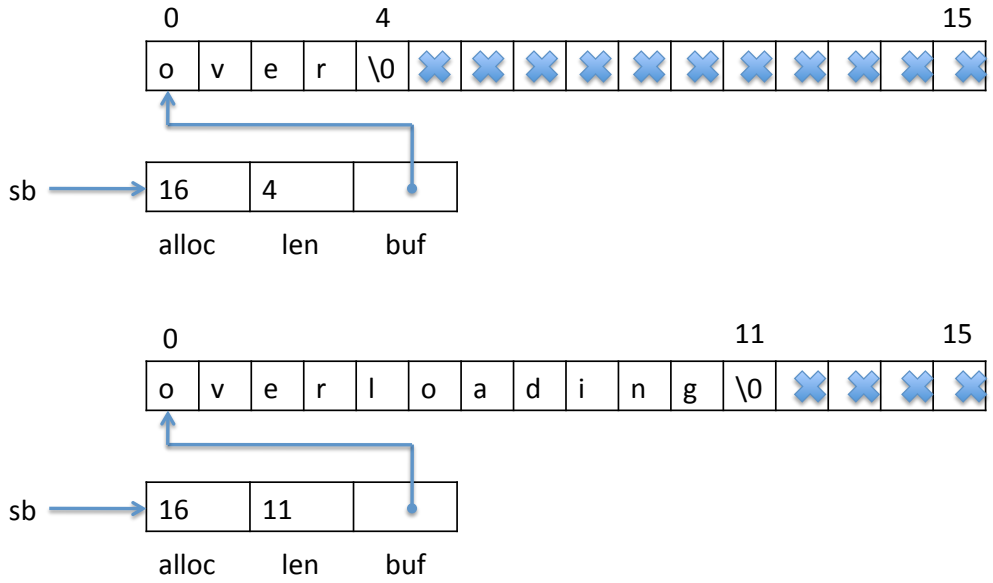
You will find it helpful to use some `string.h` library functions when writing your code. A reference is available at `http://www.cplusplus.com/reference/cstring/`. You can also find more information in the unix man pages. For example, type `man strcpy` into the unix shell. You should familiarize yourself with these functions to avoid duplicating their functionality in your code. This would be considered bad style and a possible source of bugs. However, here are some common pitfalls of some library functions:

- Be aware of the runtimes. An example using `strncat` is given in Section 5. In some cases, the man pages provide a sample simple implementation.
- As mentioned previously, `strlen` does not include the NUL terminator.
- Using `strcpy` may continue past the end of an array if `src` is longer than `dest`.
- Using `strncpy` does not guarantee that `dest` will be NUL terminated if a NUL byte is not copied from the first `n` bytes of `src`.

2 String Buffers: Overview

In this programming assignment we will explore a common data structure for imperative languages like C for manipulating strings called a *string buffer*.

A *string buffer* is based on the idea behind *unbounded arrays*. When we allocate a string buffer, we allocate an array of a given initial size, but leave it empty. Then we repeatedly add strings to the end of the buffer. For example after we add the string `"over"` to an empty string buffer `sb` that was initially allocated with size 16, we obtain the state shown at the top, where uninitialized memory locations are marked with an `X`.



After adding the string `"loading"`, we would get the state at the bottom of the picture.

The corresponding calls with the functions explained later in this writeup would be

```
struct strbuf *sb = strbuf_new(16);
strbuf_addstr(sb, "over");
strbuf_addstr(sb, "loading");
```

At any time we can also read from or write to an arbitrary (in-bounds) index position in the buffer.

When we run out of space in the current buffer we resize it by allocating a larger array and copy the current elements to the new array. The size increase has to be sufficient to guarantee a worst-case amortized time of $O(k)$ to add a string of length k to the buffer.

3 String Buffer Representation

The interface to string buffers is different from many of our prior examples because we do not keep the representation abstract. Instead, we announce the precise form of the internal representation. One advantage is that this allows the client to directly access the fields of a string buffer, avoiding a proliferation of interface functions. A disadvantage is that now the client is partially responsible for maintaining the data structure invariants. Moreover, we lock ourselves into a particular representation. In this particular case, however, it seems the advantages of a concrete representation may outweigh its disadvantages.

We have in the file `strbuf.h`:

```
struct strbuf {
  size_t alloc;    /* alloc > 0, bytes allocated for buf */
  size_t len;      /* len < alloc */
  char *buf;       /* buf != NULL, buf[len] == '\0', strlen(buf) == len */
};
bool is_strbuf(struct strbuf *sb);
```

Because the client can see the implementation, we also publish the interface to the `is_strbuf` function.

A string buffer must satisfy the following properties:

1. `buf` is not `NULL`
2. The number of characters allocated in `buf` must be equal to `alloc`. Note that this cannot be checked dynamically, when the code executes.
3. The segment `buf[0..len]` is a valid C string of length `len`. This means all the characters in `buf[0..len)` must be non-`NUL` (`'\0'`), and `buf[len]` must be `NUL`.

Task 1 (6 pts) *In a file* `strbuf.c`*, write the function*

```
bool is_strbuf(struct strbuf *sb);
```

to check that `sb` *is a pointer to a valid string buffer (to the extent that this is possible in C). Make sure you* `#include` *all appropriate library headers (*`.h` *files), including* `strbuf.h`*.*

4 Allocation and Deallocation

Allocation of a string buffer is straightforward, given what we have said above. An initial allocation size for `buf` is supplied by the client.

```
struct strbuf *strbuf_new(size_t alloc_init);
```

Deallocation is a little bit trickier. Throughout the lifetime of a string buffer, *it always owns the* `buf` *array.* It is essential that you understand this invariant. It is necessary so that the string buffer implementation can resize the `buf` array as needed; it could not do so if it didn't own the array.

When we deallocate the string buffer, we return the embedded `buf` array and pass ownership of it to the client, who becomes responsible for (eventually) freeing it. This allows us to detach the current contents of the string buffer without making a copy.

```
char *strbuf_dealloc(struct strbuf *sb);
```

Also, at any time we can ask for a string that's a copy of the one currently in the string buffer. This copy should only occupy the exact memory needed, and may be much shorter than the buffer itself. The client also obtains ownership of this string and will eventually be responsible for freeing it.

```
char *strbuf_str(struct strbuf *sb);
```

Task 2 (7 pts) *Inside the file* `strbuf.c`*, implement functions* `strbuf_new`*,* `strbuf_dealloc`*, and* `strbuf_str` *according to the description above.*

5 Adding a String

We can read or write to individual characters at index `i` in the string buffer `sb` simply with `sb->buf[i]`. To add a string to a string buffer we concatenate it at the end of the string already in the buffer when there is enough space. Of course, we must do this in such a way that all invariants of the string buffer data structure are preserved. When there is not enough room we need to allocate more space so that the result fits into the array. This should exploit ideas from our implementation of *unbounded arrays* to make sure adding a string of length k is $O(k)$ worst-case amortized time, regardless of the size of the existing contents, *len*.

Task 3 (6 pts) *Inside the file* `strbuf.c`*, implement functions*

```
void strbuf_add(struct strbuf *sb, char *str, size_t str_len);
void strbuf_addstr(struct strbuf *sb, char *str);
```

The first form can be used by the client if it happens to know the length `str_len` *of the string* `str`*; the second if that information is not readily available. However, you should take into account their similarity to simplify your code.*

Important: It may be very tempting to use the `string.h` functions `strcat` or `strncat` when writing these functions. However, this will fail to achieve the required $O(k)$ runtime. We're told in `man strcat`:

A simple implementation of strncat() might be:

```
char *strncat(char *dest, const char *src, size_t n) {
  size_t dest_len = strlen(dest);
  size_t i;
  for (i = 0 ; i < n && src[i] != '\0' ; i++) {
    dest[dest_len + i] = src[i];
  }
  dest[dest_len + i] = '\0';
  return dest;
}
```

The key is that there is a call to `strlen`, which runs in $O(len)$. Therefore, any solution using `strcat` or `strncat` would run in $O(len + k)$ time, which is not acceptable.

6 Contracts

Your functions should have contracts on them that are as precise as is possible in C. We have already discussed the string buffer data structure invariants in Section 3.

Since we cannot properly check the length of arrays, you should assume that arguments of type `char*` are either `NULL` or valid C strings. This means they are allocated somewhere in memory (heap, stack, or data segment) and they are properly terminated by the `NUL` character `'\0'`.

If a function receives a string `s` and its purported length `len`, your precondition should check that `strlen(s) == len` (after ascertaining that `s` is not `NULL`). Similarly, if a function returns a string of known length, your postcondition should check the length.

When we test your code, we may supply `NULL` for any pointers to make sure appropriate contract exceptions are raised, but we will only apply it to strings of type `char*` that are terminated by `'\0'`.

7 Test Cases

This time, we require you to write and hand in test cases. To simplify matters, we only request a single file `strbuf-test.c` defining a function `main()` that does black-box testing of the string buffer implementation, using only what is exposed in the interface in file `strbuf.h`. Tests should have the form of `assert()` statements that verify the correctness of the implementation on correct input.

Task 4 (6 pts) *In the file* `strbuf-test.c`*, write test cases that are exercised when the function* `main()` *is called.*

Your test should not pass inputs to the function that would violate the function's preconditions, or modify the data representation in ways that's inconsistent with the invariants.

We will link your testing code with various plausibly faulty implementations to verify reasonable coverage, using the black-box testing techniques we discussed in lecture.

You should revisit the specification for each of the above tasks and determine how to write a test that would catch an implementation that does not follow the specification. Be sure to consider not only the contents of the string buffer, but also memory, e.g. when new memory should and shouldn't be allocated. Naturally, incorrect memory management that cannot be checked by your code will **not** be tested.

You may find it helpful to refer back to your twitterlab code as a reminder of how to structure your tests. Similar to that assignment, you do not need to identify all bugs to receive full credit.

To help you with this task, you will be able to try to catch some of the bugs without submitting to Autolab. However, the remaining bugs are only accessible by submitting to Autolab. See README.txt for details.

8 Advice

All your functions should be relatively short. The difficulty is to reason properly about invariants, string lengths, allocation sizes, and effects of various operations to get it *exactly* right. We will test your code thoroughly, in the following respects:

1. The strength of your contracts, specifically, the preconditions and data structure invariants.

2. The correctness of the answers.

3. Memory safety (including null-pointer dereferences, reading or writing to memory that has not been allocated or has been freed, and reading from uninitialized memory).

4. Memory leaks.

5. Memory footprint (including allocating more memory than you need).

To help debug some of the above points, you may find `valgrind` gives valuable information.

We found it very helpful to draw diagrams, illustrating the state of the string buffer at various stages when operations are performed on them.

Also, we recommend the use of the (local) `xalloc` library which defines `xmalloc` and `xcalloc` that abort rather than returning `NULL` when no more memory is available.

For this assignment, we have set up a `Makefile` for you. You may add new targets to this, but do not hand it in, and do not change the indicated targets or your files may not compile on the Autolab server.

Good luck!