15-122: Principles of Imperative Computation, Fall 2013

Homework 7 Programming: String Buffers

"@"
"@@"
"@@@@@@"

Update 7
Due: Tuesday, November 5, 2013 by 22:00

In this programming assignment we will explore a useful data structure for working with strings in an imperative language, a *string buffer*. We'll write and test code for string buffers in both C0 and C.

The code handout for this assignment is at

`http://www.cs.cmu.edu/~rjsimmon/15122-f13/hw/15122-prog7.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a 25 handin limit for this assignment. Additional handins will incur a 1-point penalty per handin.

1 Strings in as Arrays of Characters

C does not actually have a `string` type. Instead, strings are represented as arrays of characters. It is possible to represent strings as arrays of characters in C0 as well.

The length of an array is not accessible outside of contracts in C0 and is not available at all in C, so one problem with representing strings as arrays of characters is that we have to be able to tell when the string stops! For this, we use the special ASCII value `'\0'`, which has the numerical value 0 and is also called the "NUL terminator" or the "null character." This means that it actually takes an array of length at least 9 to store a string, like `"Hi world"`, that has length 8:

0	1	2	3	4	5	6	7	8	9	10	11	12
'H'	'i'	' '	'w'	'o'	'r'	'l'	'd'	'\0'				

The convention that we will adopt from C is to not count the NUL terminator when we describe the length of this string.

Algorithms that deal with strings in C generally do not know the length of the strings they are working with; instead, they just read until the NUL terminator and then make sure not to read or write any further. If the NUL terminator is missing, many of these algorithms will keep right on reading or writing past the end of the array.

1.1 String manipulation in C0 and C

The built-in C0 program `string` library function includes three functions which may be useful. The function `string_terminated(str, n)` checks that `str[0..n)` contains a NUL-terminated string. (Note that this string can have any length from 0 to $n - 1$ depending on where the NUL terminator is.) The function `string_from_chararray` constructs a C0 string from a NUL-terminated character array, and the function `string_to_chararray` constructs a NUL-terminated character array from a C0 string. This last function will be very helpful as you are writing tests in C0. In C, string literals are treated as *constant arrays*, memory that you can read from but not write to, and the NUL terminator is added automatically by C. So writing `assert("Hello World"[6] == 'W')` is technically valid C code. [Update 7: fixed read/write switch.]

In C, the library `string.h` can be very useful in manipulating strings, and a translation of part of this library into C0 is given in `lib/cstring.c0` for use in your code; you should make sure to look at and understand the functions in this library. A reference for the complete C version of this library is available at `http://www.cplusplus.com/reference/cstring/`. You can also find more information in the unix manual pages, accessible with the command `man`. 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 is a possible source of bugs. However, here are some common pitfalls of some library functions:

- Be aware of the runtimes, especially for `strcat` and `strncat`.
- As mentioned previously, `strlen` does not include the NUL terminator.
- Using `strcpy` may access an array out of bounds if the source string is longer than the destination array.
- 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`.

For this library and the translation into C0 to really make sense, it helps to think of a C string as the combination of a C0 array and an *offset* describing where the string starts in the array. This strategy lets us store multiple strings in the same array and even save space by having strings overlap:

0	1	2	3	4	5	6	7	8	9	10	11	12
'H'	'i'	' '	'w'	'o'	'r'	'l'	'd'	'\0'	'B'	'y'	'e'	'\0'

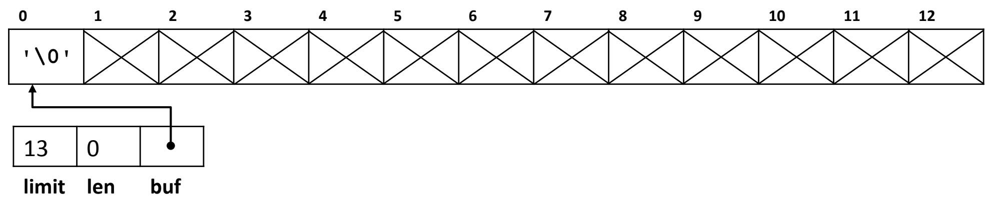
If the array above is `str`, then we could say that `str+0` points to the string `"Hi world"`, that `str+3` points to the string `"world"`, that `str+9` points to the string `"Bye"`, and that both `str+8` and `str+12` point to the empty string. In C0 this *does not work syntactically*, so `lib/cstring.c0` takes two arguments per string - the base array and the offset. In C, *pointer arithmetic* and the conflation of pointers and arrays will allow us to actually add positive integers to pointers and get occasionally meaningful results:

```
// In C
char *str1 = "Hi world";
assert(0 == strcmp("world", str1 + 3));
```

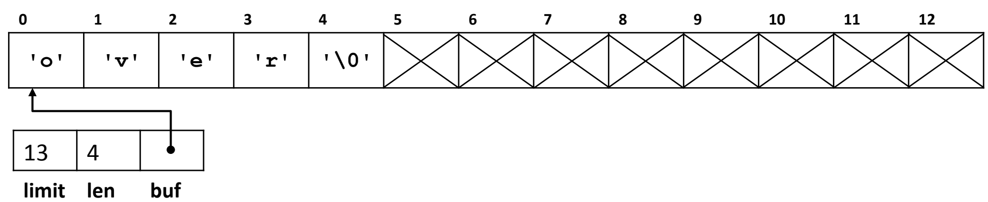
Therefore the `string.h` library only has to take one `char*` argument per string. Pointer arithmetic like this is often a bad idea, but it's important to be aware of it.

2 String Buffers: Overview

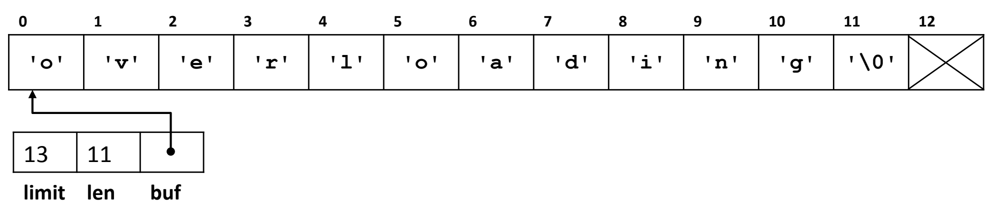
In practice, manipulation of strings as mutable arrays is tedious and error-prone, which is one of the reasons that string buffers are useful. A *string buffer* is fundamentally an adaptation of an unbounded array. When we allocate a string buffer, we allocate an array of a given initial size, but leave it empty. This is the initial state of a string buffer `sb` allocated with size 13:



Then we repeatedly add strings to the end of the buffer. For example after, if we add the string `"over"` to `sb`, we get this picture:



If we next add the string `"loading"` to the buffer, we would get this picture:



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

```
struct strbuf *sb = strbuf_new(13);
strbuf_add(sb, "over", 4);     // In C0 use string_to_chararray("over")
strbuf_add(sb, "loading", 7);  // ...same thing here.
```

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. [Update 2: Fixed function declarations.] [Update 4: Changed function calls from addstr to add.]

Unlike most of the other data structures we have considered in 15-122, we will expose the representation of our string buffers to clients of our library. One advantage of this approach is that it allows the client to directly access the fields of a string buffer, avoiding a proliferation of interface functions. One disadvantage is that this approach locks us into a particular representation. Furthermore, the client is partially responsible for maintaining the data structure invariants, and we have to be very careful about those invariants. For example, the client can write to position `12` in the buffers above without violating the data structure invariants, so we must treat the unspecified contents of the array as really unspecified.

3 String Buffers in C0

The C0 type `struct strbuf` of string buffers is declared as follows:

```
struct strbuf {
  int limit;
  int len;
  char[] buf;
};
```

A string buffer `sb`, a pointer to a `struct strbuf`, must satisfy the following properties:

1. The buffer `sb` must not be `NULL`.
2. The number of characters allocated in `buf` (the size of the array) must be equal to `limit`. [Update 1: changed `alloc` to `limit`]
3. The segment `buf[0..len]` is a valid NUL-terminated string of length `len`. This means all the characters in `buf[0..len)` must be non-NUL (`'\0'`), and `buf[len]` must be NUL.

Note that these invariants do *not* say anything about the unspecified portion of the array after the NUL terminator. Your data structure invariants should not either.

Task 1 (3 pts) *In the file* `strbuf.c0`*, write the function*

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

to check that `sb` *is a pointer to a valid string buffer. It should return* `false` *rather than failing a contract whenever possible.*

Allocation for a string buffer is straightforward: the buffer initially contains the empty string and the initial size for `buf` is supplied by the client. We will also write a function `strbuf_str`, which should return a fresh copy of the string-occupied part of the string buffer. This returned array should be NUL-terminated and no longer than necessary. [Update 3: Clarified NUL termination.]

Task 2 (3 pts) *In the file* `strbuf.c0`*, write the functions*

```
struct strbuf* strbuf_new(int initial_limit);
char[] strbuf_str(struct strbuf* str);
```

according to the description above.

3.1 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 (but we don't want to allocate more space until we are forced to). This should exploit ideas from our implementation of unbounded arrays to make sure adding a string of length k takes amortized time in $O(k)$.

Task 3 (4 pts) *In the file* `strbuf.c0`*, implement functions*

```
void strbuf_add(struct strbuf* sb, char[] str, int 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` (implemented in `lib/cstring.c0`) when writing these functions. However, this will fail to achieve the required $O(k)$ runtime. We're told in `man strcat`:

```
A simple [C] 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.

3.2 Unit Testing

We will run your tests both against your own code, correct implementations of string buffers in C0, and buggy implementations of string buffers in C0.

You should write tests that will pass for *any* correct implementation of string buffers, and write tests that will fail for as many buggy implementations as possible. While it is a requirement that `strbuf_add` and `strbuf_addstr` only allocate a larger buffer when they are absolutely required to, the specific amount that the buffer grows is up to the implementation.

Therefore, don't submit unit tests that try to check the specific new size of the post-increase buffer, or the autograder will complain that these tests fail on correct implementations.

Your tests should use the `assert()` function instead of using contracts so that the unit tests run whether or not you are compiling with `-d`.

Task 4 (3 pts) *Write and submit test cases in* `strbuf-test.c0` *that test your C0 implementation of string buffers.*

To get these three points, your test cases must catch an implementation that is correct except for a specific bug in Task 1, a second implementation that is correct except for a specific bug in Task 2, and a third implementation that is correct except for a specific bug in Task 3.

3.3 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 at least the following respects:

1. The strength of your contracts, specifically, the preconditions and data structure invariants.
2. The correctness of the answers.
3. Amortized running time and memory usage.
4. Memory footprint (including allocating more memory than you need).

4 String buffers in C

For the last part of the assignment, we will turn our C0 string buffers into C string buffers. In C, it is *not possible* to check the length of an array, so it will no longer be possible to enforce certain invariants of your data structure.

4.1 Adapting C0 code to C

Here is an *incomplete* list of the changes you will need to make as you adapt your C0 string buffers to C:

- Change array types like `char[]` to pointers `char*`. Be careful: this means that you now have to check that arrays are non-NULL in your code and data structure invariants!
- Modify uses of the `lib/cstring.c0` library to be uses of the standard `string.h` library.
- Change calls to from `alloc` and `alloc_array` to their C analogues. We strongly recommend the use of the (local) `xalloc` library which defines `xmalloc` and `xcalloc`. These functions abort rather than returning `NULL` when no more memory is available.

- Change `//@requires`, `//@ensures`, and `//@assert` C0 contracts into `REQUIRES()`, `ENSURES()`, and `ASSERT()` C contracts.
- The string buffer is responsible for managing its character array, so when `strbuf_add` or `strbuf_addstr` need to increase the size of that array, it's necessary for the functions to make sure that the old array gets freed. *Your test cases should free all allocated memory.*
- Adapt to the use of `size_t` instead of `int` for quantities that are supposed to be array offsets. The type `size_t` is unsigned, so you don't need to check that they're greater than zero.

As a stylistic issue, remember that we write

```
int* x = alloc(int);
char[] A = alloc_array(char, 10);
```

in C0 but write

```
int *x = xmalloc(sizeof(int)); // Update 5 xalloc->xmalloc
char *A = xcalloc(10, sizeof(char));
```

in C. Attaching the `*` to the variable instead of the type is consistent with the C idea of making the definition of a variable look like the way it is used. We won't be picky about this stylistic issue on this assignment, though.

Task 5 (5 pts) *Copy the implementation of* `strbuf.c0` *to* `strbuf.c`, *making sure to include the interface by writing* `#include "strbuf.h"` *within this file.*

A word of warning: our tests for `strbuf.c` are not exactly the same as our tests for `strbuf.c0`, and it's possible that things that we missed with earlier testing will be caught by the C tests!

4.2 Handling Deallocation

As is common with C0 to C translations, we have to write a function that deallocates the memory reserved for our data structure. Rather than having the deallocation function free both the `struct strbuf` and the buffer `buf`, we will return the embedded `buf` array without freeing it 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.

Task 6 (2 pts) *In the file* `strbuf.c`, *implement the function*

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

according to the description above.

4.3 Unit Testing

Your C unit tests should again use `assert()` rather than contracts. To use the `assert()` function in C, you need to `#include <assert.h>`. In addition to running your tests, you should use the `valgrind` tool to check for invalid memory access and memory leaks. When you compile with the `-g` flag, `valgrind` can give you valuable information about where undefined behavior and memory leaks are occurring.

For your C0 tests, you can create C-style strings easily with `string_to_chararray`. In C, 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. C lets you declare and initialize arrays on the stack that are automatically freed when the function returns. One way of initalizing such stack-allocated arrays is to write a string. [Update 6: previous description was misleading.]

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

3. String literals. These strings are stored in read-only memory; you can't write to 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 undefined behavior 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);
```