Programming Project 3
EE312 Fall 2021

String ADT

General

In this project, we are creating an Abstract Data Type (ADT) to improve upon the standard C way of representing strings. We will use malloc and free, and get some practice with pointers, structs, and macros.

On Strings

Strings in C are stored simply as an array (buffer) of ASCII-encoded characters with a zero (null character) on the end. This works well, but has some drawbacks. Importantly, strings are a common source of buffer overflow errors, a serious security vulnerability that happens when programmers forget how large the string's buffer actually is when writing to it.

Our new ADT will still store the actual string in basically the same way as before -- however, it will also store some extra information as well -- specifically, the length (number of characters) and the capacity (how many characters can it hold before buffer overflow would result). As some amount of extra safety, in the four bytes immediately following the string, we will store a not-sorandom looking byte sequence that we can check to confirm that nothing bad has happened (yet).

On UTStrings

You are to implement each of the functions declared inside UTString.h. You must use (without modifying) the UTString struct that is defined inside UTString.h. This struct consists of the following parts:

- **length** – The length of the string stored in the UTString. This is the number of characters in the string, and does not include the null character, nor anything after it. Keep in mind that the length of a string may be shorter than the buffer in which the string is stored. For example, if we had the string "hello", then the length would be 5 (one for each useful character), regardless of the size of the buffer.
- **capacity** – The length of the longest string that can be stored in the buffer. For example, if we had a string with a buffer of length 20, then capacity would be 15 (20, minus the null character and check), regardless of the length of the string.
- **string** – A pointer to the buffer where the string is stored. It must be allocated separately from the UTString itself.
- **check** –The signature value (~0xdeadbeef) stored after a string. It is not a true member of the struct, but should be in it regardless. We will check the value every time when working with a UTString to make sure that no buffer overflow has occurred yet. If it does not check out correctly, your program should fail an assert and crash immediately. UTString struct:

length (int, 4 bytes) = 11
capacity (int, 4 bytes) = 11 or greater
string (char*, 8 bytes) pointer to location of buffer on heap of at least capacity +5

The contents of the string buffer for the string “Hello World” (the single quotes and 0x are not shown for brevity):

H	e	l	l	o	‘ ’	W	o	r	l	d	\0	~0xdeadbeef

If the buffer is bigger, the bytes after 0xef are unknown.

UTStrings can be used much as an ordinary string. Anyone choosing to use our library should declare variables of type UTString*, and should only use our functions to change the stored string. Some notes:

- UTStrings can only be stored on the heap. Declaring a local or global variable of type “UTString” will not work. We can, of course, declare local variables of type “UTString*” (pointers to UTString). We’ll point these pointers at chunks from the heap.
- Our clients (anyone using our library) will only create strings by calling the *utstrdup* function. Our clients will not poke around inside our struct, or to mess with the characters inside the array. Our clients will call *utstrfree* with their UTStrings when they’re done with them.

Goals

Write the following functions:

- UTString* utstrdup(const char* src); -- An analog of the C stdlib function *strdup*. *src* is a string. Create a UTString on the heap that holds a copy of source, setting the length, capacity, and check appropriately. Return a pointer to the UTString.

- uint32_t utstrlen(const UTString* src); -- An analog of the C stdlib function *strlen*. *src* is a UTString. Find and return the length of the string.

- UTString* utstrcat(UTString* s, const char* suffix); -- An analog of the C stdlib function *strcat*. Append characters to *s* from *suffix* until out of capacity or done copying. Do not copy more than can be stored. Do not allocate further space. Do use a null terminator and update the check. Return *s* after appending.

- UTString* utstrcpy(UTString* dst, const char* src); -- An analog of the C stdlib function *strcpy*. Should replace characters in *dst* with characters from *src* until out of capacity or done copying. Do not copy more than can be stored. Do not allocate further space. Do use a null terminator and update the check. Return *dst* after copying.

- void utstrfree(UTString* self); -- Used to free a UTString. Similar to the C stdlib function *free*. Must deallocate both the string buffer and the UTString itself.

- UTString* utstrrealloc(UTString* s, uint32_t new_capacity); -- Used to reallocate space for a UTString. Similar to the C stdlib function *realloc*. If *new_capacity* is larger than the current capacity, create a buffer with *new_capacity* capacity, copy all the old contents, and deallocate the old buffer. Otherwise, do nothing. Either way, return *s* afterwards.

Every UTString passed to any of these functions should have a valid *check*. If it does not, your program should fail an assert and crash immediately.

Submit all files created or edited for the assignment except testing or debugging files. Submit no more, and no less.

Testing

The provided tests are useful, but not very thorough (particularly with regards to utstrcpy). Please write your own tests to verify your own code.

The stage 1 and stage 2 tests are intended to convey how UTStrings work. The stage 3 test is trying to ensure you've done nothing silly. If all goes well, this test should run in less than a second and should not have any buffer overflows. Otherwise, you've probably done something silly.

The stage 4 test is trying to ensure that you are actually checking things as intended.

We will use valgrind to grade your program on memory leaks.

Hints

- You could write your own functions within Project3.cpp to avoid repeating code, for modularity, or for debugging.
- Look for off-by-1 errors in setting length, setting capacity, or copying characters.
- Think of valid but unexpected test cases – for instance, appending a string of length 0.
- Avoid writing code that does the same thing many times. If you need to, say, find the length of a regular string several times during the project, you should write a function to do so.

FAQ

Q: Will null pointer inputs be checked?
A: No.

Q: May we use the standard C string library?
A: Yes, you may include string.h, but only use them with C strings and not UTStrings. The standard functions are not compatible with UTStrings because of the extra data. For performance reasons, it might be better to make your own functions rather than use the string.h functions.

Q: How long is a "long time" for stage 3?

A: If it takes more than 2-3 seconds, there's probably something wrong.

Q: If we have a UTString, how do we find the signature?

A: Use the macro defined at the top of the file.

Example, where str is a valid UTString*: CHECK(str)

Q: When initializing the String struct on the heap from a char*, what should the initial capacity be?
A: The capacity should be initialized to the length of the char* in the strdup function, since we need to allocate at least that much space to fit the string.

Q: How do length and capacity work?
A: Length refers to the meaningful length of the utstring. Capacity is the maximum amount of meaningful information the utstring can hold.
The null terminator is not a meaningful piece of information, as every string has one at the end and is just a way to communicate that the string has ended. Neither is the check, for similar reasons.

The length and capacity are numbers in the String struct. These values do not count the null character. However, when allocating space, you do need to allocate 1 byte more than the length, to include the null character (plus the space for the header/metadata, of course). And you have to actually set the character after the last character in the string to null.

Q: What is the ptr->data arrow for?
A: This is just shorthand for (*ptr).data. We write this a lot, so the arrow saves us some characters. If you want to use * instead, make sure to include the parentheses, or else it will look for data in the pointer and not find anything meaningful.

Q: Do we have to check if malloc() or realloc() fails?
A: For this project you don't have to check if malloc returns a null pointer.

Q: Do we have to check if we're given valid UTStrings for functions that operate on UTStrings?

A: Yes. You should #include <assert.h> and use assert() to check if the UTString is valid.

Example:
assert(isOurs(my_string));
This will be the first thing in functions where you need to check, and then you can write the rest of the function afterward as normal.
Q: What about Stage 4?

A: Stage 4 will be tested.

Stage 4 checks if you used assert() properly.

Adding a print function for UTStrings might be useful for debugging. Make sure to use Valgrind to eliminate all memory leaks!

Q: My gcc/g++ is not working on kamek with the Makefile.
A: In order to run the provided Makefile on the LRC servers, please log on to kamek and enter the following command: module initadd gcc
Then log out of the server and log back in.
This will permanently update gcc and g++ on your account on the server so that they will be compatible with our Makefile.

Q: How do I write the 'signature' into a character array?

```
CHECK(s) = SIGNATURE;
```

where s is a UT string. Since SIGNATURE is a 32 bit Int, and CHECK(s) is a dereference of an int pointer, this will set the 4 bytes after the null character correctly.

CHECKLIST – Did you remember to:

- ☐ Re-read the requirements after you finished your program to ensure that you meet all of them?
- ☐ Make sure that your program does not need modifications of main.cpp or UTString.h to work?
- ☐ Make sure that your program passes all our testcases?
- ☐ Seal all memory leaks?
- ☐ Make up your own testcases?
- ☐ Include the header file (filled out) at the top of your submitted file?
- ☐ Upload your solution to Canvas (Project3.cpp)? If you have multiple files, you should zip them up into a file called Project3_<EID>.zip or .gzip or .gz before uploading.
- ☐ Download your uploaded solution into a fresh directory on the ECE server and re-run all testcases?