Programming Project 3
EE312 Fall 2017

String ADT

General: In this project, we'll improve upon the standard C method for representing strings. We're going to do a little work with malloc and free, and along the way, get some practice with pointers, structs and even some macros.

The Big Picture: Strings in C are stored using an array of ASCII-encoded characters with a zero on the end. This is well and good, most of the time, but has some drawbacks. For one, in order to determine how long a string is, we have to start counting at the beginning, examining each character until we get to the zero. Even more significant, strings are a common source of buffer overflow errors in programs because programmers often forget how large the array actually is when filling it up with characters.

So, we're going to create an improved version of the string. Our string will be "backwards compatible" with C-style strings. However, we'll store some extra information along with every string. Specifically, in the four bytes immediately preceding the string we'll store the length (number of characters) in the string. In the four bytes immediately preceding the length, we'll store the size of the array (i.e., the capacity for the string, how many characters can it hold before buffer overflow would result). As a small amount of extra safety, in the four bytes immediately preceding the capacity, we'll store a not-so-random looking byte sequence that we can check to confirm that nothing bad has happened (yet). These three extra **uint32**s worth of information can be accessed using pointers (**uint32_t*** pointers specifically), or using a *String* struct that we define for you.

Your Mission: You are to implement each of the functions declared inside the String.h file. Hopefully I've remembered to describe each of those functions in this handout. But, the official list of what you need to do is contained in the String.h file.

You must use (and don't modify) the String struct that is defined inside String.h. This struct is designed to align the 12 bytes immediately preceding a character string into three integers. You'll note that this struct defines four components to each and every string. These components are:

- **length** – obvious enough, this is the length of the string. Note that the length is the number of characters in the string, and does not include the zero at the end. The length does not include any bogus values that happen to be array after the zero either. Keep in mind that the length of a string may be shorter than the array in which the string is stored.
- **capacity** – the capacity is (one less than) the size of the array in which the characters are stored. So, for example, if we had the string "apple" stored in an array with 20 characters, then capacity would be 19, length would be 5. The first

six elements of that array would be { ‘a’, ‘p’, ‘p’, ‘l’, ‘e’, 0 }, and the remaining elements would be random junk. In other words, capacity is the maximum number of characters that can be safely stored inside our string array (and still leave room for the zero on the end).

- **check** – The signature value stored in a string should always be ~0xdeadbeef. This signature serves an unusual purpose. It’s a marker that we place in all our strings. We’ll look for this signature every time we’re asked to work with a string. If signature has exactly the right value (~0xdeadbeef), then the data almost certainly comes from one of our strings. If the signature has the wrong value, then the string probably wasn’t created by us – i.e., someone made a mistake.
- **data** – As a cute trick, our struct won’t contain a pointer; it will contain an actual array. We’re declaring this array to have a size of one. But, when we allocate space for a String, we’ll always be sure to allocate an extra big chunk.

Here’s some C code that will create and initialize a string for “yo!” using an array with capacity 10.

```
// Example only, this is not great code
char* s; // should be “yo!” when we’re done
String* new_string = malloc(sizeof(String) + 10 + 1);
(*new_string).length = 3; // 3 characters in “yo!”
(*new_string).capacity = 10; // malloc’d 10 bytes
(*new_string).check = ~0xdeadbeef;
(*new_string).data[0] = ‘y’;
(*new_string).data[1] = ‘o’;
(*new_string).data[2] = ‘!’;
(*new_string).data[3]  = 0;
s = (*new_string).data;
printf(“the string is %s\n”, s);
```

The first executable statement in the example calls malloc and requests a chunk with enough space to store a String (that’s what sizeof(String) does) plus an extra 10 bytes for the character array at the end (our capacity of 10), plus one extra byte for the zero that we will place at the end of the string. We initialize the length, capacity and signature values for this struct, and then start copying our sequence of characters into the array at the end. Since we requested an extra 10 bytes, we know there’s room to spare for these three characters and the zero. Finally, we make the char* pointer s point at the first byte in that array and we’re done.

Here's some functionally equivalent C code without the use of the String struct.

```
// Example only, this is not great code
char* s; // should be "yo!" when we're done
uint32_t* p; // access the ints before the string
p = (uint32_t*) malloc(3 * sizeof(uint32_t) + 10 + 1);
s = (char*) (p + 3);
s[0] = 'y';
s[1] = 'o';
s[2] = '!';
s[3] = 0;
p = (uint32_t*) s;
p = p - 1;
*p = 3; // length immediately before the string
P = p - 1;
*p = 10; // capacity immediately before the length
P = p - 1;
*p = ~0xdeadbeef; // signature before capacity
printf("the string is %s\n", s);
```

Using Strings: Our strings can be used any place that an ordinary string can be used. Anyone choosing to use our string library should declare variables of type char* and work with our strings almost as if they were ordinary strings. There are a few exceptions.

- Our strings can only be stored on the heap. Declaring a local or global variable of type "String" will not work. We can, of course, declare local variables of type "String*" (pointers to String). We'll point these pointers at chunks from the heap.
- Our "clients" (programmers who decide to use our strings) agree to create strings calling the *utstrdup* function. Our clients further agree not to poke around inside our struct, or to mess with the characters inside the array. Our clients promise to call *utstrfree* with their strings when they're done with them. Note that clients will never declare any variables of type String or type String*. Client programmers will access strings using variables of type char*.

The Functions: You are to write each of the following six functions. Note that the parameters and return values for these functions are **char*** types. You will, of course, actually be creating structs on the heap – which leads to some good practice for us.

char* utstrdup(char* source); -- This function is a mirror of the standard C library function called *strdup*. The parameter is a string that we wish to duplicate. The returned pointer will point onto the heap. When you write this function, create a String struct on the heap that holds a copy of source. Set the length and capacity of your String equal to the number of characters in source. Be sure to return the address to the first character in your String (and not the address of the String struct itself). The argument source may or may not be a utstring (i.e., it might be an ordinary C string without our extra info in front).

void utstrfree(char* p); -- This function is used by our clients when they are finished working with one of our strings. The function should deallocate the chunk on the heap that holds this String struct (by calling free). Just be sure to compute the correct address before calling free. Our clients are required to never call this function unless the argument p is a utstring. They're also required to always call this function (eventually) for every utstring that they create.

uint32_t utstrlen(char* str); -- This function is a mirror of the standard C library function called *strlen*. NOTE: str must actually be a String struct. Your function should perform the pointer arithmetic and return the length stored inside this string struct – in other words, this function had better be a whole lot faster than the standard strlen function (i.e., no loops (and, please, no recursion)). In this function and all the remaining functions, the first argument must always be a utstring. In the case of utstrlen, we knew str is a utstring and therefore we can extract the length directly from the bytes in the front of the string.

char* utstrcpy(char* dest, char* source); -- This function is a mirror of the standard C library function called *strcpy*. This function should replace the characters in dest with the characters from source. NOTE: dest will actually be a String struct (i.e., the address stored in dest will be one of the addresses you returned from utstrdup). When you perform the copy, you must not overflow the capacity of this String.

- If the source string is longer than the capacity of dest, then copy only as many characters as will fit (i.e., copy capacity characters).
- If the capacity of dest is equal to or larger than the length of source, then copy all the characters from source.
- Be sure to set the length of the dest correctly.
- By convention (i.e., for no particularly good reason), this function should return dest when it is over. The return value of the function is not actually useful.

char* utstrcat(char* dest, char* suffix); -- This function is a mirror of the standard C library function called *strcat*. Similar to utstrcpy, utstrcat requires that the first parameter be a String struct (well the char array that is part of a String struct). The function should append characters from suffix, but must not exceed the total capacity of dest.

char* utstrrealloc(char* str, uint32_t new_capacity); -- This function is similar to the standard C library function realloc. The first parameter is a String struct. We must examine this struct and compare the current capacity to the new_capacity parameter.

- If the current capacity is equal to or larger than the new_capacity, then do nothing and return str.
- If the current capacity is smaller than the new_capacity, then create a new String struct with new_capacity. Copy all the characters from str to this new string. Deallocate the chunk where str was stored, and return the newly created string.

Test program: The stage 1 and stage 2 tests are pretty weak. Hopefully they get the idea across on how our strings will work. The tests might even find some errors in your program. However, please understand that you need to test your own programs. Just because your program happens to pass the (weak) tests provided in main.cpp, please do not assume that your program is correct. Write some additional tests of your own (in particular, utstrcpy is not thoroughly tested).

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. If it takes a long time to run (e.g., a minute or more), then you've probably done something silly.

You will be also tested on stage 4 tests.

Memory Leaks
We will use valgrind to make sure that your program has no memory leaks.

Hints

- You could write your own functions within Project2.cpp to avoid repeating code, for modularity, or for debugging (for example, a method to print out a String).
- Look for off-by-1 errors.
- Re-use functions that you have written. For example, use your own utstrfree to free String types. Use the 'raw' free also when you need to.
- Avoid using structs as much as possible, and do most operations with int* and char* pointers. You will be better prepared for the exams.

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.

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 char* that we know is part of a UTString, how do we get access to the data in the struct?
A: Use the macro defined at the top of the file.

Example, where chararray is a char* that is part of a UTString
String* s = STRING(chararray);

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.

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 UTStrings for functions that operate on UTStrings?
A: Yes. You should #include <assert.h> and use assert() to check if you are passed a char* that is part of a UTString.

Example:
assert((*my_string).check == SIGNATURE);
or
assert(isOurs(my_string));
where my_string is a char* that may be part of a UTString.
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.
You can also check for null pointers by asserting your pointer is not equal to NULL, but it is not required.

Q: What about Stage 4?

A: Stage 4 will be tested.

Stage 4 checks if you used assert() properly. our program should crash if given invalid arguments.

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.

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 String.h to work?
- ☐ Make sure that your program passes all our testcases?
- ☐ Seal all memory leaks?
- ☐ Make up your own testcases?
- ☐ 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. Please consult with us if you have multiple files.
- ☐ Download your uploaded solution into a fresh directory on the ECE server and re-run all testcases?