C Programming — Creating a String Data Type in C

For this assignment, you will use the **`struct`** mechanism in C to implement a data type that models a character string:

```
struct _String {

   char     *pData;   // dynamically-allocated array to hold the characters
   uint32_t length;   // number of characters in the string, excluding terminator
};
typedef struct _String String;
```

Since C arrays (and C strings are essentially arrays) don't automatically keep track of their dimensions or usage, it seems completely reasonable to capture all of that information inside of a **`struct`**, and use that to represent a flexible string type.

A *proper* `String` object `S` satisfies the following conditions:

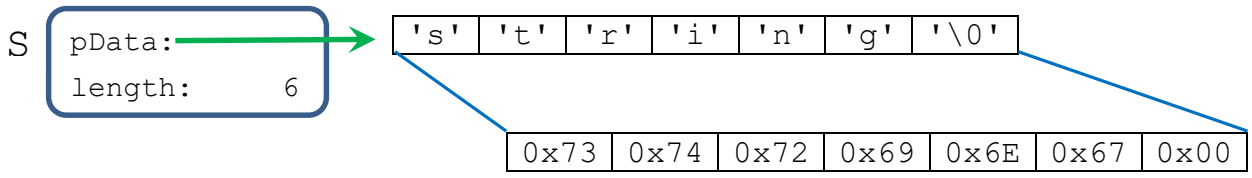
- `S.pData` points to an array of dimension `S.length + 1` and `S.pData[S.length] == '\0'`.
- If `S.length > 0`, then `S.data[0:S.length-1]` hold the character data for the string.

A *raw* String object S satisfies the following conditions:

- `S.pData` may or may not be `NULL`.
- `S.length` has no significant value.

Pay close attention to the pre- and post-conditions and the return specifications for the functions declared later in this assignment; you must make sure that you satisfy all those conditions.

A `String` object S representing "string" would have the following logical structure:



A few points should be made. First, the character array is sized precisely to the string it represents, so there is no wasted memory. Second, even though a `String` object stores the length of the character string, we still zero-terminate that array; this helps with operations like printing the contents of the `String` object.

In a proper `String` object, `pData` will never be **`NULL`**.

Also, the `String` type raises a deep-copy issue, since the character array is allocated dynamically. Since C does not provide automatic support for making a deep copy of structured variables, the functions we will implement are designed to receive pointers to `String` objects. This is discussed further on page 4.

The use of pointers as parameters provides an excuse to make good use of the **`const`** qualifier, applied to the pointer and/or its target, as appropriate.

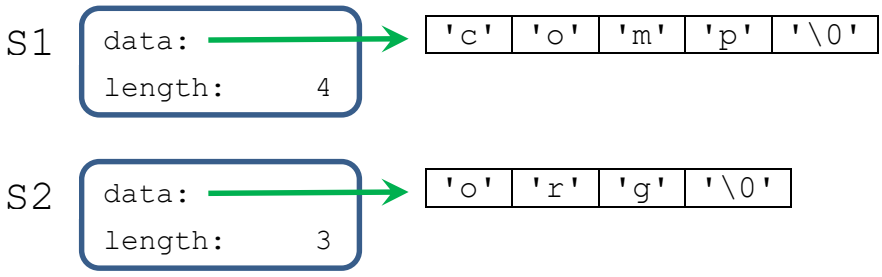
`String` Operations

A *data type* consists of a collection of values and a set of operations that may be performed on those values. For a string type, it would make sense to provide the common string operations; for example:

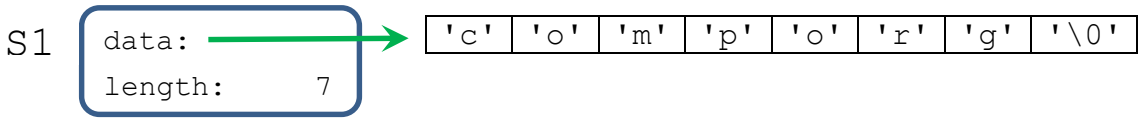
```
/** Appends the String *pSrc to the String *pDest.
 *
 *  Pre:
 *    *pDest is a proper String object
 *    *pSrc is is a proper String object
 *    pSrc != pDest (i.e., the source and destination are different String objects)
 *  Post on success:
 *    pSrc->data is appended to the String pDest->data
 *    *pDest is a proper String object
 *  Post on failure:
 *    *pDest is unchanged
 *
 *  Returns:
 *    the length of pDest->data, if nothing goes wrong;
 *    a negative value, if some error occurs
 */
int32_t String_Cat(String const *pDest, const String* const pSrc);
```

The design of `String_Cat()` follows the expected semantics of concatenating two strings. The function will append a `String *pSrc` to the `String *pDest`, adjusting `pDest->data` and `pDest->length` as required. Note that `*pDest` must be proper (as defined earlier) both before and after the function is called. And, there must be no memory leaks.

For example, suppose we have the following `String` objects:



Then, the call `String_Cat(&S1, &S2)` should leave `S1` as shown below, return the value 7, and leave `S2` unchanged:



In the case of the `String_Cat()` function, there is a logical reason the function needs to modify the contents of the `String` object `*pDest`, but not the pointer to it which is passed into the function, so the pointer is qualified as **`const`**, but the target of the pointer is not[1].

On the other hand, the same function has no reason to modify the `String` object pointed to by `pSrc`, nor to modify where `pSrc` points, so both the pointer and its target are qualified as **`const`**[1].

There is also the question of whether stated preconditions should be checked within the function. The need for efficiency would argue against; after all, the preconditions have been stated, so it's the caller's fault if they are not satisfied, and checking them would require extra steps at runtime. And, some preconditions are essentially impossible to check.

On the other hand, the need for robustness would argue in favor of checking (checkable) preconditions, if violations of them could result in serious runtime errors, especially if those errors could occur much later than the call itself.

You should consider these points carefully when designing your solution to this assignment. The testing/grading code will always honor the stated preconditions, unless there are errors in your own code (e.g., creating a `String` object that is not proper).

We will copy one aspect of an OO design; it's useful to provide a function that will create and initialize a new `String` object:

```
/** The String is initialized to hold the values in *pSrc.
 *
 *  Pre:
 *    *pSrc is C string with length up to slength (excludes null char)
 *  Post on success:
 *    A new, proper String object S is created such that:
 *       S.data != pSrc->data
 *       Up to slength characters in *pSrc are copied into S.data
 *         (after dynamic allocation) and the new string is terminated
 *         with a '\0'
 *       S.length is set to the number of characters copied from *pSrc;
 *         this is no more than slength, but will be less if a '\0' is
 *         encountered in *pSrc before slength chars have occurred
 * Post on failure:
 *    NULL is returned
 *
 * Returns:
 *    pointer to the new String object;
 *    NULL value if some error occurs
 */
String* String_Create(const char* const pSrc, uint32_t slength);
```

To some degree, this plays the roles of an allocator and of a constructor in an OO implementation. In Java, the constructor does not allocate memory for the object itself (**`new`** does that); but the constructor may allocate memory for dynamic content in the object. This function has both responsibilities.

The parameter `slength` allows us to initialize a `String` object from an unterminated **`char`** array, or using a selected part of an existing C-string. It also allows something of a safety net, in that the function will limit the number of characters read from `*pSrc` to `slength`[2].

The second required function is for comparisons:

```
/** Compares two Strings.
 *
 *  Pre:
 *    *pLeft is a proper String object
 *    *pRight is is a proper String object
 *
 *  Returns:
 *    < 0 if *pLeft precedes *pRight, lexically
 *      0 if *pLeft equals *pRight
 *    > 0 if *pLeft follows *pRight, lexically
 */
int32_t String_Compare(const String* const pLeft, const String* const pRight);
```

The interface is adapted from **`strcmp`**`()` in the C Standard Library. Note that the return specification does not imply that the values -1, 0 and 1 are the only possible results. That's entirely up to your design, and your solution is not expected to return the same integers as the reference solution.

The third required function allows a user to append one `String` to the end of another `String`:

```
/** Appends the String *pSrc to the String *pDest.
 *
 *  Pre:
 *    *pDest is a proper String object
 *    *pSrc is is a proper String object
 *    pSrc != pDest (i.e., the source and destination are different String
 *        objects)
 *  Post on success:
 *    pSrc->pData is appended to the String pDest->data
 *    *pDest is a proper String object
 *  Post on failure:
 *    *pDest is unchanged
 *
 *  Returns:
 *    the length of pDest->pData, if nothing goes wrong;
 *    a negative value, if some error occurs
 */
int32_t String_Cat(String* const pDest, const String* const pSrc);
```

Of course, this will normally require making sure that the destination `String` has a sufficiently large array to hold the necessary data, and doing so without creating any memory leaks[3].

The fourth required function allows a user to extract a (copy of) a specified substring of a given `String`:

```
/** Makes an exact, full copy of a substring.
 *
 * Pre:
 *   *pSrc is a proper String object
 *   startIdx + length <= pSrc->length
 * Post:
 *    no memory leaks have occurred
 *   A new, proper string object S has been created such that S holds
 *      the specified substring of *pSrc
 *
 * Returns:
 *   pointer to a proper String object which holds a copy of the
 *      specified substring; NULL if failure occurs
 */
String* String_subString(const String* const pSrc, uint32_t start, uint32_t length);
```

The fifth required function allows a user to remove a specified substring from a given `String`:

```
/** Erases a specified sequence of characters from a String.
 *
 * Pre:
 *   *pSrc is a proper String object
 *   startIdx + length <= src->length
 * Post:
 *    no memory leaks have occurred
 *    the specified range of characters have been removed
 *   *pSrc is proper
 *
 * Returns:
 *   if successful, pSrc
 *    NULL if failure occurs
 */
String* String_Erase(String* const pSrc, uint32_t start, uint32_t length);
```
[4]

The final required function is a memory-management tool:

```
/**  Deallocates a String object and all its content.
 *
 *   Pre:
 *     *ppStr is a pointer to a proper String object, so
 *     **ppStr is a proper String object
 *     **ppStr was allocated dynamically
 *   Post:
 *     (**ppStr).data has been deallocated
 *     **ppStr has been deallocated
 *     *ppStr == NULL
 */
void String_Dispose(String** ppStr)[5];
```

A call to `String_Dispose()` would look something like this:

```
String *pStr = String_Create("And as we wind on down the road...", 34);
. . .
// Initialize the String and use it until we're done with it.
. . .
String_Dispose(&pStr);
// At this point, every trace of the String object is gone and pStr == NULL.
```

The `String` object `String_Dispose()` is working on must have been allocated dynamically, because `String_Dispose()` will attempt to deallocate that object. In addition, `String_Dispose()` will reset your pointer to **`NULL`**, which is why we use a pointer-to-pointer in the interface.

For each of these required functions, you should think carefully about edge cases in testing. The supplied test code is fairly thorough, but there is no guarantee that a small number of runs will manage to create every possible scenario.

There are a number of other useful operations, such as reporting the position of a character in a string, modifying characters in a string, inserting one string into another, etc, which we will not support, in order to keep this assignment reasonably small. A practical implementation would have many more features.

Pay attention to the comments in the header file. All the stated pre- and post-conditions are part of the assignment. Pay particular attention to avoiding memory leaks.

You should consider implementing additional "helper" functions. Those should be private to your C file, so make them **`static`**; they will be invisible to the testing code and never called directly from outside your file.

Your solution will be compiled with the supplied testing/grading code, so if you do not conform to the specified interfaces there will very likely be compilation and/or linking errors.

Requirements

While you implement your `String` type, you **may not use any of the C standard library string functions** we discussed in class. **You may not include `<string.h>` in String.c, nor use any function declared within.** You must implement any needed functionality yourself as part of the assignment. You may write your own helper functions.

A score of 0 will be assigned if you violate this restriction.

Further, objects of your string type must be able grow and shrink as necessary, and will require you to dynamically allocate and free memory. For example, appending a `String` to the end of another `String` type will cause dynamic allocation to occur, and when you no longer need a `String` object, or its array, you must free the associated memory (in the appropriate function).

As usual, the tar file that is posted for the assignment contains testing/grading code. In particular, the following files are supplied:

`c05driver.c`	test driver
`String.h`	declarations for specified functions
`String.c`	empty shells for implementing the specified functions
`testString.h`	declarations for checking/grading functions
`testString.o`	64-bit Linux binary for checking/grading code

Edit `String.c` and implement the functions, then compile it with the files above. Compile `c05driver.c`, and execute it as:

```
./c05driver <-mode> [-repeat]
```

The `mode` switch selects which functions will be tested, and can be be any of the following:

```
-all, -create, -compare, -cat, -substring, -erase
```

The `-repeat` switch is optional, and can only be used if you have previously run the testing code without it. This switch causes the tests to be performed with previously-used test cases.

The testing/grading code won't automatically deduct points for improper memory management, or using standard C string functions, but the course staff will manually examine your code to make sure you have followed these requirements. You can check for memory leaks by running the tests of your solution on Valgrind.

Helpful Hints and Information

As you start working on the String project, here are some helpful hints and information:

The test suite is a wrapper that calls your functions and checks the results. It doesn't do anything else, so the ultimate source of any problems is most likely inside of your code. Given the nature of dynamic memory errors your program can crash in unexpected (unrelated) places and also run fine on one machine and then crash on another. And, a bug in your code may create an improper `String` object that then causes a crash in the testing code.

To expand on those points, where the code crashes is sometimes different than where the problem is actually located. For example if your `String_Dispose()` function worked previously but sometimes crashes under testing, chances are there is something wrong with an **another** function. Most likely, the previously tested function broke something that is only now being triggered inside of `String_Dispose()`. So long story short, you need to backtrack, look at the previous function calls etc.

Further, these errors may happen sporadically making the problems difficult to find. Look closely at your `String.c` file for the issues described above, `gdb` may also be helpful. And, Valgrind is invaluable. Here's the sort of thing you want to see when you run the test code on Valgrind:

```
1022 wmcquain@centosvm in ~/2505/C05/code> gcc -o c05 -std=c11 -Wall -ggdb3 c04river.c String.c
testString.o
1023 wmcquain@centosvm in ~/2505/C05/code> valgrind --leak-check=yes --show-leak-kinds=all --track-
origins=yes -v c05 -all
. . .
==38972==
==38972== HEAP SUMMARY:
==38972==     in use at exit: 0 bytes in 0 blocks
==38972==   total heap usage: 639 allocs, 639 frees, 58,624 bytes allocated
==38972==
==38972== All heap blocks were freed -- no leaks are possible
==38972==
==38972== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
. . .
```

There should be no bytes in use at exit. The number of allocations and frees may be different, as may the total number of bytes allocated. But you want to see no leaks, no invalid reads or writes, and no reports of use of uninitialized values. The number of bytes that are allocated will vary with different tests; the number of allocations and frees will vary as well.

A quick start guide is here:

http://valgrind.org/docs/manual/quick-start.html

You should also consider using the allocation functions `calloc()` and `realloc()` in your solution.

What to Submit

You will submit your file `String.c` to the Curator, via the collection point c05. That file must include any helper functions you have written and called from your version of the specified functions; any such functions must be declared (as **`static`**) in the file you submit. You must not include any extraneous code (such as an implementation of `main()` in that file).

Your submission will be graded by running the supplied test/grading code on it.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//  On my honor:
//
//  - I have not discussed the C language code in my program with
//    anyone other than my instructor or the teaching assistants
//    assigned to this course.
//
//  - I have not used C language code obtained from another student,
//    the Internet, or any other unauthorized source, either modified
//    or unmodified.
//
//  - If any C language code or documentation used in my program
//    was obtained from an authorized source, such as a text book or
//    course notes, that has been clearly noted with a proper citation
//    in the comments of my program.
//
//  - I have not designed this program in such a way as to defeat or
//    interfere with the normal operation of the Curator System.
//
//    <Student Name>
//    <Student's VT email PID>
```

We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.

Change Log

Version	Posted	Pg	Change
4.00	Oct 18		Base document.

Notes

[1] See the course notes for a full explanation of the use of **`const`** with pointer parameters. Be aware that it is possible, and sometimes necessary, to sidestep a const specification in C; in this assignment, you should not do so.

[2] It is entirely possible for this function to fail. A necessary memory allocation could fail; you can and should test for that. It's also possible the user could provide a source array that does not provide the promised number of characters, which could lead to a runtime error; you cannot test for that in all cases, but you could at least be sure that your function respects the existence of a terminator in the source array.

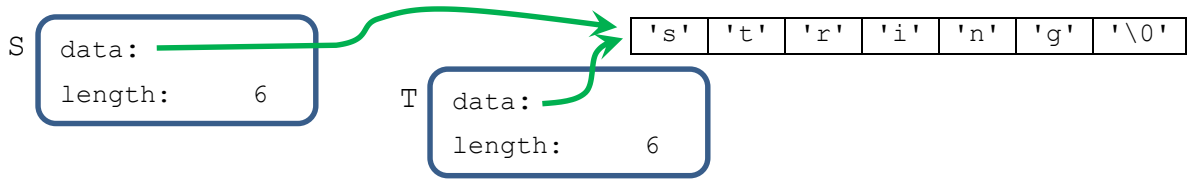
[3] The `String_Cat()` function is a good place to take advantage of the `realloc()` function. It's also a good place to be careful to avoid memory leaks.

[4] Be careful of indexing errors here.

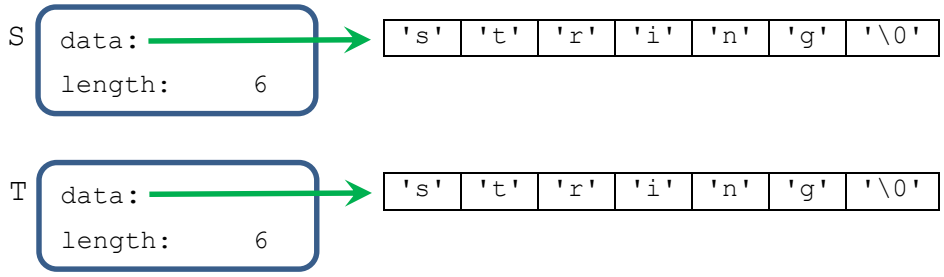
[5] This function is likely to be a source of errors, both runtime and otherwise. The supplied test code depends entirely on your implementation of `String_Dispose()` for deallocations of memory related to `String` objects that are created during testing. Since those objects will be created by calls from the testing code, Valgrind may trace the origins of some memory-related errors to the testing code, which you cannot examine in gdb, but the source of such errors will lie in an incorrect or incomplete `String_Dispose()` implementation.

Appendix Deep Copy

This assignment does not require a function that supplies a deep copy tool. Suppose we have a `String` object `S` representing "string", and assign it to another `String` object `T` (e.g., `S = T`):



The two `String` objects will "share" the same character array, which is probably not what we want when we write an assignment statement. Instead, we probably want:



A function like this could offer such a copy operation:

```
/** Makes an exact, full copy of a String.
 *
 * Pre:
 *   *src is a proper String object
 * Post:
 *    no memory leaks have occurred
 *    A new, proper string object S has been created such that S is a
 *      copy of *pSrc
 *
 * Returns:
 *   pointer to a String object which holds a copy of *src;
 *   NULL if failure occurs
 */
String* String_Copy(const String* const pSrc);
```

There is no requirement that you implement such a function, but it may be useful as a **`static`** helper. The course notes contain an example that shows the necessary details.