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     *data;      // 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.data points to an array of dimension S.length + 1 and S.data[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.data 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:
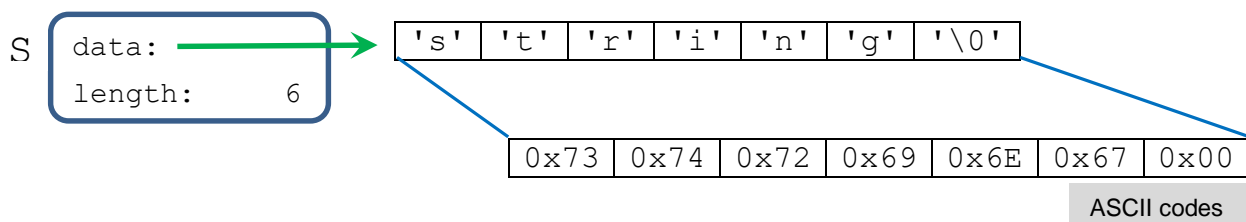
A few points should be made. First, the character array is sized precisely to the string it represents, so there is no wasted memory. Second, even though a String object stores the length of the character string, we still zero-terminate that array; this helps with operations like printing the contents of the String object.

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 <u>pointers</u> to String objects.

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

In the case of the String_Cat() function that follows, there is a logical reason the function needs to modify the contents of the String object *dest, 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.

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

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:

```c
/** Appends the String *src to the String *dest.
 *
 *  Pre:
 *    *dest is a proper String object
 *    *src is is a proper String object
 *    src != dest
 *  Post on success:
 *    *src is appended to the String *dest
 *    *dest is a proper String object
 *  Post on failure:
 *    dest->data == NULL, dest->length == 0
 *
 *  Returns:
 *    the length of dest->data, if nothing goes wrong;
 *    a negative value, if some error occurs
 */
int32_t String_Cat(String const *dest, const String* const src);
```

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

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.

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

```c
/** The String is initialized to hold the values in *src.
 *
 *  Pre:
 *    *src 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 != src
 *        Up to slength characters in *src are copied into dest->data
 *          (after dynamic allocation) and the new string is terminated
 *          with a '\0'
 *        Slength is set to the number of characters copied from *src;
 *          this is no more than slength, but will be less if a '\0' is
 *          encountered in *src 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 src, 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 third parameter 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 `*src` to `slength`.

The next required function is a memory-management tool:

```
/**  Deallocates a String object and all its content.
 *
 *   Pre:
 *      **str is a proper String object
 *      **str was allocated dynamically
 *   Post:
 *      (**str).data has been deallocated
 *      **str has been deallocated
 *      *str == NULL
 */
void String_Dispose(String** str);
```

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

```
String *pStr = malloc( sizeof(String) );
. . .
// 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.

The next required function is for comparisons:

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

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.

The last required function supplies a deep copy tool. Suppose we have a `String` object S representing "string", and assign it to another `String` object T:
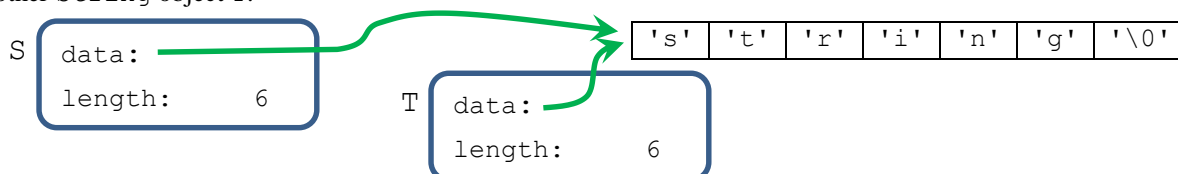
The two `String` objects will "share" the same character array, which is probably not what we want when we write an assignment statement. Instead, we probably want:
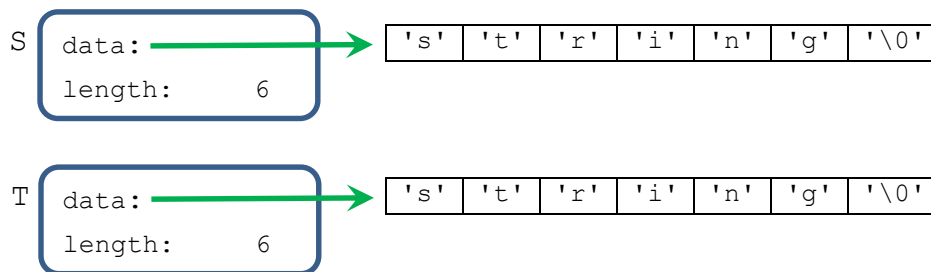
This function creates such a copy operation:

```
/** 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 S
 *
 * Returns:
 *    pointer to a String object which holds a copy of *src;
 *    NULL if failure occurs
 */
String* String_Copy(const String* const  src);
```

There are a number of other useful operations, such as extracting substrings, modifying characters, 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 use the posted header file as your starting point, and place your function definitions in a file named `String.c`.

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

See the posted `String.h` file for complete descriptions of the functions that are required for this version of the assignment.

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.

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

You <u>may</u> use functions from the C Standard Library `string.h` in any <u>testing</u> code you write.

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:

`driver.c`	test driver
`String.h`	declarations for specified function; do not modify this file!
`testString.h`	declarations for checking/grading functions
`testString.o`	64-bit Linux binary for checking/grading code

Create `String.c` and implement your version of it, then compile it with the files above. Read the header comments in `driver.c` for instructions on using it.

The testing/grading code won't automatically deduct points for using standard C string functions, but the course staff will manually examine your code to make sure you have followed these requirements.

You can, and should, check for memory-related errors this by running the tests of your solution on Valgrind. For this assignment, we expect a completely "clean" Valgrind run: no memory leaks, no invalid writes or reads, no issues with uninitialized variables or decisions depending on uninitialized data.

During grading, we will also examine the behavior of your solution on Valgrind, and we will assess penalties of up to 20% if Valgrind detects any unacceptable behavior.

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 runtime problems is most likely inside of your code, although runtime errors may trace directly to testing code that is broken by errors in the character arrays created by 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 <u>improper</u> `String` object that then causes a crash in the testing code.

To expand on those points, where the code crashes is often different than where the logical error 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 newly tested function broke something that is only now being triggered inside of `String_Dispose()`. So, long story short, you need to examine stack backtraces in gdb, look at the function calls that preceded the crash, 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 result you want to see when you run the test code on Valgrind:

```
==14713== Memcheck, a memory error detector
==14713== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14713== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==14713== Command: ./driver
==14713== Parent PID: 14712
==14713==
==14713==
```

```
==14713== HEAP SUMMARY:
==14713==     in use at exit: 0 bytes in 0 blocks
==14713==   total heap usage: 636 allocs, 636 frees, 25,953 bytes allocated
==14713==
==14713== All heap blocks were freed -- no leaks are possible
==14713==
==14713== For counts of detected and suppressed errors, rerun with: -v
==14713== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

No leaks, no invalid reads or writes, and no reports of use of uninitialized values. Of course, 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 C10. 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	Nov 6		Base document.