Programming Assignment #2: String Muncher

COP 3502, Spring 2023

Due: <u>Wednesday</u>, February 15, *before* 11:59 PM

Figure 1. An image of a particularly epic string muncher soaring through the sky. String munchers are magical beasts that feast on strings. The above is an AI-generated image created from the prompt, “magical caterpillar floating through a sky surrounded by flying books, wisps of smoke, and burning embers.” There do not appear to be any flying books in the image, though, which I find very sad.

Abstract

In this programming assignment, you will implement the string muncher data structure – a data structure that contains an array of strings that expands automatically whenever it gets too full. This is an immensely powerful and awesome data structure, and it will ameliorate several problems we often encounter with arrays in C (see Section 1 of this PDF).

By completing this assignment, you will gain advanced experience working with dynamic memory management, pointers, structs, and strings in C. You will also learn how to use *valgrind* to test your programs for memory leaks, and you will gain additional experience managing programs that use custom header files and multiple source files. In the end, you will have an awesome and useful data structure that you can use to solve all sorts of interesting problems.

Deliverables

StringMuncher.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

Note! A friendly guide for the overwhelmed is included on pg. 19.

1. Overview

In this assignment, you will implement a new data structure called a "string muncher." A string muncher is a magical beast that feasts on strings. Its digestive tract is a corridor of multiple stomachs, each of which is designed to hold a single string pointer. If a string muncher's digestive tract ever becomes too full, it can grow new stomachs so that it can continue eating all the strings in its path. *Om nom nom*.

We will represent a string muncher's corridor of stomachs as an array of string pointers tucked inside a *StringMuncher* struct, alongside a few other fields – including variables that keep track of how many stomachs a given string muncher has and how many of those stomachs are already occupied by delicious strings. This string muncher data structure will have several key advantages over a normal array of strings in C:

1. We do not need to specify how many strings a string muncher will hold when it is created. Instead, it will expand automatically when it gets full.[1] This is great when we don't know ahead of time just how many strings our string muncher will consume over the course of a given program (which is a very common constraint when using arrays to store data in real-world programs).

2. We will use a variety of functions (described in detail below) to access the contents of individual stomachs within a string muncher's digestive tract, and those functions will check to make sure we aren't attempting to access stomachs that lie outside the bounds of the creature's digestive tract and therefore don't even exist. (Recall that with normal arrays, C doesn't check whether an array index is out of bounds before accessing it during program execution. That can lead to all kinds of wacky trouble, from segmentation faults to pointer corruption. In coding up string munchers, you will guard us against such issues.) Namely, the stomachs in a string muncher will be numbered 0 through $n - 1$, and the functions we use to interact with those stomachs will guard us against array-index-out-of-bounds issues in the event that we attempt to refer to a stomach whose index falls outside that range.

3. If the array inside a string muncher ends up having wasted space (i.e., it isn't full), we can trim it down to size to eliminate unnecessary bloat. Reducing bloat not only makes the string muncher feel better, but also helps minimize the unnecessary strain our program is placing on system resources.

4. In C, if we pass an array to a function, we typically find ourselves passing its length to that function as a second parameter, as well. Any time we pass a string muncher to a function, however, we automatically get both the array of strings that makes up that string muncher's digestive tract *and* the length of that array, as they'll both be packaged together in a struct. Neat!

While some languages offer built-in support for arrays that expand automatically as they get full (as with Java's super awesome *ArrayList* class), C does not. That's where you come in. By coding up string munchers in this assignment, you will implement robust auto-expanding array functionality in C, including:

1. automatically expanding the capacity of a string muncher's digestive tract (string array) when it gets full;

2. adding new strings at arbitrary positions in the string muncher's digestive tract, or at the next open index in the digestive tract;

---

1 "Automatically" is perhaps a strong word here. The data structure will have the *appearance* of expanding automatically once you have implemented the necessary code to make that happen. ;)

3. providing safe access to specific strings stored in a string muncher's digestive tract;
4. gracefully signaling to the user (i.e., any programmer calling the functions you've written) when they attempt to access an index in the digestive tract that is out of bounds (instead of allowing segmentation faults to occur);
5. … and more!

1.1 The *StringMuncher* Struct

The basic struct you will use for this string muncher data structure (which is already defined for you in the *StringMuncher.h* file included with this assignment – more on that below) is as follows:

```
typedef struct StringMuncher
{
   // The name of this string muncher.
   char *name;

   // How many stomachs this string muncher currently has.
   int num_stomachs;

   // How many of the stomachs are occupied with (non-NULL) string pointers.
   int num_stomachs_occupied;

   // The string muncher's stomachs. Okay, so, it's just an array of string
   // pointers. Each cell represents a stomach, and each stomach/cell holds
   // a single string pointer.
   char **stomachs;
} StringMuncher;
```

To begin with, every *StringMuncher* has a name. We use a *char* * for that variable so that it can be dynamically allocated to the perfect size depending on what name we give to each individual string muncher. The *StringMuncher* also contains a *char* ** pointer that can be used to set up a 2D *char* array (which is just an array of *char* arrays). That array represents its digestive tract. Each cell in the array represents a single stomach and can hold a single *char* * that points to a string that the string muncher has gobbled up via the *omNomNom()* function (described below in Section 3, "Function Requirements"). That *stomachs* array will have to be allocated dynamically any time you create a new *StringMuncher* struct. The *num_stomachs* field tells us how many cells are in the *stomachs* array (i.e., the length of that array), and the *num_stomachs_occupied* field tells us how many of those contain valid, non-`NULL` string pointers. Initially, upon creation of a new *StringMuncher* struct, all of its stomachs will contain `NULL`, and *num_stomachs_occupied* will be initialized to zero.

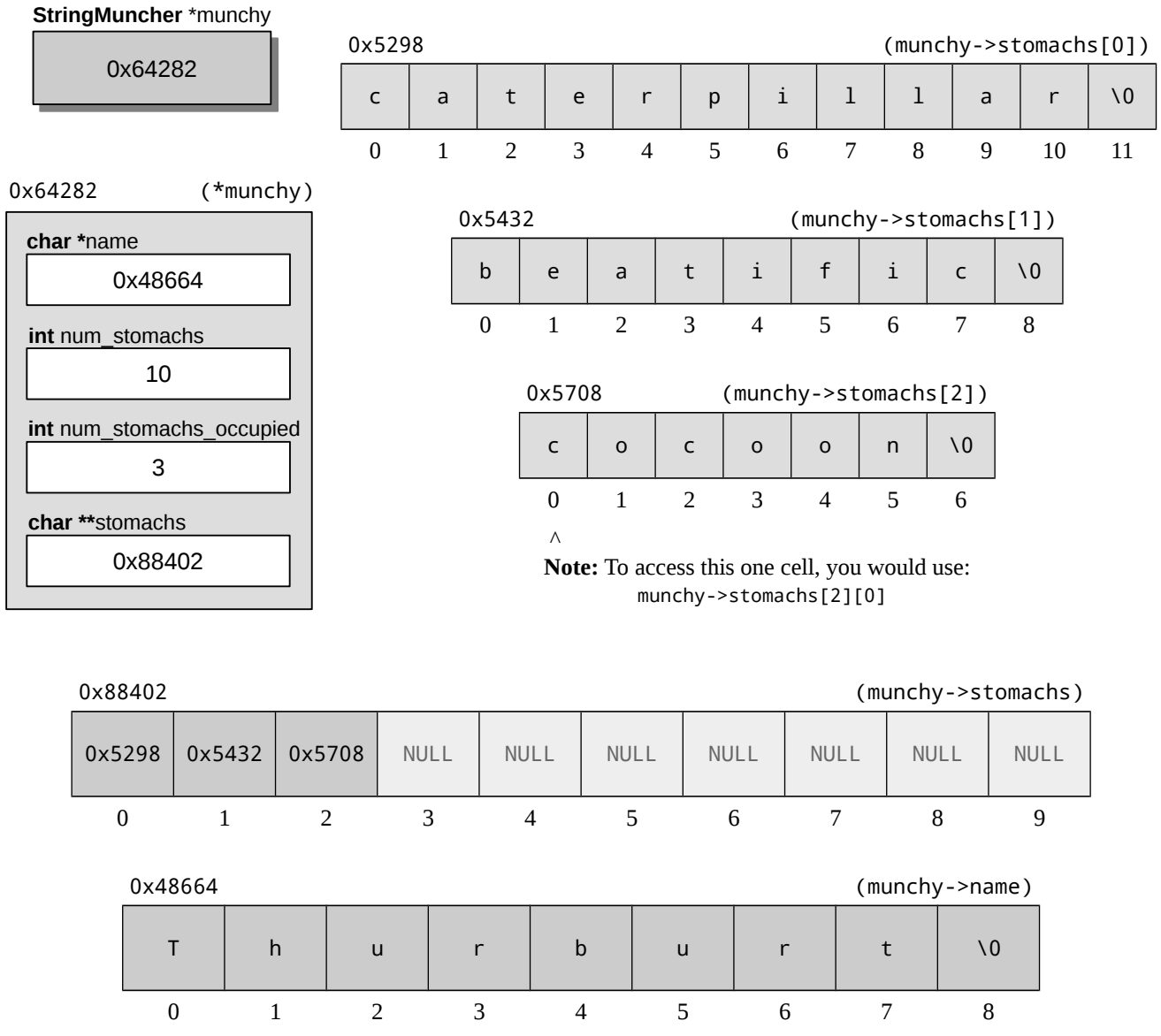
The stomachs within a string muncher are indexed 0 through (*num_stomachs* – 1). Throughout this assignment, we refer to the stomach at index 0 as the "bottom" of the string muncher's digestive tract. This should be considered the deepest stomach in its corridor of stomachs, farthest from its mouth. The stomach at index (*num_stomachs* – 1) is referred to as the "top" of the string muncher's digestive tract and is the stomach closest to its mouth.

(*StringMuncher.h* also contains a definition for `DEFAULT_DIGESTIVE_TRACT_CAPACITY`, which you will use in one of the required functions described below. Please do **not** re-define that constant in *StringMuncher.c*.)

1.2 The Anatomy of a *StringMuncher*

The following diagram shows a complete *StringMuncher* struct, with all its constituent members. In this example, we have a *StringMuncher* pointer variable, *munchy*, that holds the address of a dynamically allocated *StringMuncher* struct (0x64282). At that address in memory (0x64282) resides the actual struct with all of its fields. This particular string muncher is named "Thurburt". That names has been copied into a dynamically allocated array (seen at address 0x48664 in memory), and that pointer is stored in the struct's *name* field.

Thurburt's digestive tract is a dynamically allocated array of 10 "stomachs," which – as explained above – are just string pointers. (Each cell contains a single *char* *.) The *stomachs* field of the struct contains the address of that array (0x88402), and at address 0x88402 in memory, we see the array of length 10. The *num_stomachs* field holds the length of the *stomachs* array (10), and *num_stomachs_occupied* conveys that exactly three stomachs (i.e., the first three cells in the *stomachs* array) are occupied by non-NULL pointers. Those pointers refer us to the addresses of dynamically allocated copies of the strings Thurburt has eaten (seen at addresses 0x5298, 0x5432, and 0x5708). We see there is no wasted space in terms of how those strings have been allocated.



As you can see from the diagram above, there are several details to keep track of in each *StringMuncher* struct. This might seem overwhelming or even mystifying at first, but Section 3 of this document, "Function Requirements," will walk you through a number of functions that you will use to create and manage these string munchers. As you work through those functions and continue to refer back to the diagram on the previous page – and possibly draw new diagrams of your own to reflect what is happening in each of the test cases included with this assignment – all of these pieces will start to click together.

1.3 Overview of What You'll Submit

A complete list of the functions you must implement for this assignment, including their functional prototypes, is given below in Section 3, "Function Requirements."

You will submit a single source file, named *StringMuncher.c*, that contains all required function definitions, as well as any auxiliary functions you deem necessary. In *StringMuncher.c*, you should *#include* any header files necessary for your functions to work, including *StringMuncher.h* (see Section 2, "StringMuncher.h (Super Important!)").

Note that you will <u>not</u> write a *main()* function in the source file you submit! Rather, we will compile your source file with our own *main()* function(s) in order to test your code. We have included example source files that have *main()* functions, which you can use to test your code. You should also write your own *main()* functions for testing purposes, but your code must not have a *main()* function when you submit it. We realize this is still fairly new territory for most of you, so don't panic. We've included instructions for compiling multiple source files into a single executable (i.e., mixing your *StringMuncher.c* with our *StringMuncher.h* and *testcaseXX.c* files) in Section 5 (pg. 14) of this PDF.

Although we have included sample *main()* functions to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

2. StringMuncher.h (*Super Important!*)

Your code for this assignment will go in a file named *StringMuncher.c*. At the very top of that file, write a comment with your name, the course number, the current semester, your NID, and perhaps a *very* brief overview of the data structure you're implementing. Directly below that, you **<u>must</u>** include the following line of code:

```
#include "StringMuncher.h"
```

If you do not *#include* that file properly, your program will not compile on our end, and it will not receive credit. Note that you should also *#include* any other standard libraries your code relies upon (e.g, *stdio.h* and *stdlib.h*).

Think of *StringMuncher.h* as a bridge between source files. It contains struct definitions and functional prototypes for all the functions you'll be defining in *StringMuncher.c*, but which will be called from different source files (such as *testcase01.c*). Because all the files in this project *#include "StringMuncher.h"*, they all have awareness of those struct definitions and functional prototypes and can compile happily.

If you write auxiliary functions ("helper functions") in *StringMuncher.c* (which is strongly encouraged!), you should **<u>not</u>** add those functional prototypes to *StringMuncher.h*. Our test case programs will not call your helper functions directly, so they do not need any awareness of the fact that your helper functions even exist. (We only

list functional prototypes in a header file if we want multiple source files to be able to call those functions.) So, just put the functional prototypes for any helper functions you write at the top of your *StringMuncher.c* file. Please do not modify *StringMuncher.h* in any way, and do not send *StringMuncher.h* when you submit your assignment. We will use our own copy of *StringMuncher.h* when compiling your program.

For your reference, the contents of *StringMuncher.h* are described above in Section 1.1, "The StringMuncher Struct."

3. Function Requirements

In the source file you submit, *StringMuncher.c*, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly. In this section, I often refer to *malloc()*, but you're welcome to use *calloc()* or *realloc()* instead, if you're familiar with those functions.

```
StringMuncher *summonStringMuncher(char *name, int num_stomachs);
```

Description: This function will create a *StringMuncher* (SM) with the given name and number of stomachs. Start by dynamically allocating space for a new *StringMuncher* struct. Initialize the struct's *name* field to be a copy of the name passed to this function. In doing so, be sure to dynamically allocate exactly the amount of space needed to hold the given name – no more, no less. Dynamically allocate the *stomachs* array to be an array of *num_stomachs* char pointers, and initialize all those pointers to `NULL`. (Eventually, that array will store pointers to dynamically allocated copies of any strings the string muncher consumes.) Finally, initialize the *num_stomachs* and *num_stomachs_occupied* fields as appropriate.

If *num_stomachs* is less than `DEFAULT_DIGESTIVE_TRACT_CAPACITY`, use `DEFAULT_DIGESTIVE_TRACT_CAPACITY` for the length of the *stomachs* array instead. Note that `DEFAULT_DIGESTIVE_TRACT_CAPACITY` is defined in *StringMuncher.h*. You should refer to that constant rather than hard-coding its value in this function (in case we change that constant in our header file when grading this assignment). Also, do not *#define* that constant in your source file. You should be able to refer to `DEFAULT_DIGESTIVE_TRACT_CAPACITY` simply by #including *StringMuncher.h*.

If any of your calls to *malloc()* fail, you should free any memory you have dynamically allocated in this function call up until that point (to avoid memory leaks), and then return `NULL`.

If the *name* passed to this function is either `NULL` or an empty string (""), you should immediately return `NULL` without setting aside any dynamically allocated memory or printing anything to the screen.

Output: If the function successfully creates a new SM, print the following: "`-> <NAME>, a string muncher with <N> stomachs, has emerged from the void!`" Do not print the quotes, and do not print <brackets> around the variables. Terminate the line with a newline character, '`\n`'. `<NAME>` is the name of this string muncher. `<N>` is simply the number of stomachs the new string muncher has. If `<N>` is 1, print "stomach" (singular) in place of "stomachs" (plural).

Returns: A pointer to the new *StringMuncher*, or `NULL` if appropriate (as indicated above).

`StringMuncher *banishStringMuncher(StringMuncher *munchy);`

Description: Free all dynamically allocated memory associated with this *StringMuncher* struct, and return `NULL`. Be careful to avoid segfaulting in the event that *munchy* is `NULL`.

Output: "`-> <NAME> has returned to the void.`" (Output should not include the quotes or <brackets> around the variable. `<NAME>` is the name of this string muncher. Terminate the line with a newline character, '`\n`'.) If *munchy* is `NULL`, this function should not print anything to the screen.

Returns: This function should always return `NULL`.

`StringMuncher *expandDigestiveTract(StringMuncher *munchy, int num_stomachs);`

Description: This function grows *munchy*'s digestive tract. Dynamically allocate a new array of *num_stomachs* string pointers. Copy the contents of *munchy*'s old *stomachs* array into the newly allocated array, initialize any remaining pointers in the new array to `NULL`, free *munchy*'s old *stomachs* array, and set up *munchy* so that its *stomachs* field points to the newly allocated array. Be sure to update the *num_stomachs* and *num_stomachs_occupied* fields of the *StringMuncher* (if applicable).

If *num_stomachs* is less than or equal to *munchy*'s current number of stomachs, or if the *munchy* pointer is `NULL`, you should not modify the *StringMuncher* at all. In that case, simply return from the function right away without producing any output.

If any calls to *malloc()* fail within this function, you should free any memory that has been dynamically allocated since the start of this function call and then return `NULL`. In this case, you should avoid destroying or modifying anything within *munchy*. When we return `NULL` in response to a *malloc()* failure in this function, *munchy* should be in the same state it was in when we called this function.

Output: "`-> <NAME>'s digestive tract has expanded to consist of <N> stomachs!`" Output should not include the quotes or <brackets> around the variable. `<NAME>` is the name of this string muncher, and <N> is the new number of stomachs. Terminate the line with a newline character, '`\n`'. If `<N>` is 1, print "stomach" (singular) in place of "stomachs" (plural). If *munchy* is `NULL`, or if the *stomachs* array is not expanded for any reason, this function should not print anything to the screen.

Returns: This function should return `NULL` if any calls to *malloc()* failed or if the string muncher was not modified. Otherwise, return a pointer to the *StringMuncher* that was passed to this function.

`StringMuncher *contractDigestiveTract(StringMuncher *munchy);`

Description: This function gets rid of any extra stomachs that *munchy* is not currently using. If *munchy*'s *num_stomachs* is greater than its *num_stomachs_occupied*, trim the length of the *stomachs* array to *num_stomachs_occupied*. You will probably want to *malloc()* a new array to achieve this, copy the contents of the old *stomachs* array into that new array, and update *munchy*'s *stomachs* pointer to point to that new array. Be sure to avoid memory leaks as you get rid of the old *stomachs* array, and be sure to update any other fields in *munchy* that need to be updated as a result of this action.

If *num_stomachs* is already less than or equal to *munchy*'s *num_stomachs_occupied*, or if the *munchy* pointer is `NULL`, you should not modify the *StringMuncher* at all. In that case, simply return from the function right away without producing any output.

If any calls to *malloc()* fail within this function, you should free any memory that has been dynamically allocated since the start of this function call and then return `NULL`. In this case, you should avoid destroying or modifying anything within *munchy*. When we return `NULL` in response to a *malloc()* failure in this function, *munchy* should be in the same state it was in when we called this function.

Output: "`-> <NAME>'s digestive tract has contracted to consist of only <N> stomachs!`" Output should not include the quotes or <brackets> around the variable. `<NAME>` is the name of this string muncher, and <N> is the new number of stomachs. Terminate the line with a newline character, '`\n`'. If `<N>` is 1, print "stomach" in place of "stomachs". If *munchy* is `NULL`, or if the length of the *stomachs* array is not modified for any reason, this function should not print anything to the screen.

Returns: This function should return `NULL` if any calls to *malloc()* failed or if the string muncher was not modified. Otherwise, return a pointer to the *StringMuncher* that was passed to this function.

```
char *omNomNom(StringMuncher *munchy, char *snack);
```

Description: This function causes the *StringMuncher* (*munchy*) to consume a new string (*snack*). Insert a dynamically allocated copy of *snack* into the next available stomach in the *stomachs* array. The string muncher wants to have as much room as possible at the top of its digestive tract at all times, and so it always moves the first thing it eats to the very bottom of its digestive tract, at stomach 0. From there, the digestive tracts fills up at stomachs 1, 2, 3, and so on, all the way up to the top of its digestive tract, at the stomach that is numbered (*num_stomachs* – 1), without ever leaving any gaps.

When copying *snack* into the *stomachs* array, be sure to allocate exactly the amount of space necessary to store that string – no more, no less. Be sure to update any other fields within *munchy* as appropriate.

If the *stomachs* array is already full when this function is called, call *expandDigestiveTract()* to grow the array to length (*num_stomachs* * 2 + 1) before inserting the new string.

Output: This function should not print anything to the screen.

Returns: Return a pointer to the copy of the new string that was inserted into the array, or `NULL` if the string could not be added to the array (e.g., if *malloc()* failed or if *munchy* or *str* was `NULL`).

```
char *endoscopy(StringMuncher *munchy, int which_stomach);
```

Description: This function allows us to examine the contents of one of *munchy*'s stomachs – namely, the stomach at the given index (*which_stomach*) in *munchy*'s *stomachs* array. This is one of the functions where you protect the user from going out-of-bounds in the *stomachs* array.

Output: This function should not print anything to the screen. This function merely returns a pointer to a string (or `NULL`); it is up to whoever receives that return value to determine whether or how that string should be printed to the screen.

Returns: Return the pointer at position *which_stomach* of the *StringMuncher*'s *stomachs* array, or `NULL` if *which_stomach* is out of bounds or if *munchy* itself is `NULL`. If *munchy* is not `NULL`, you may assume none of its internal fields will lie to you. So, if *munchy*'s *num_stomachs* field is set to 10, you may assume *munchy* has a non-`NULL` *stomachs* array with 10 cells, indexed 0 through 9 (although some of those cells may contain `NULL` pointers).

```
char *transmogrify(StringMuncher *munchy, int which_stomach, char *snack);
```

Description: This is one of the string muncher's more magical abilities, which it evolved in order to satisfy intense cravings for specific strings: this function transmogrifies an existing string in a specified stomach (*which_stomach*) into a new string (*snack*), but only if there is already a valid string at the given index. If so, dynamically allocate a new copy of the *snack* passed to this function, and store the pointer to that string at the given stomach within the string muncher. In doing so, be sure to allocate exactly the amount of space needed for the new string – no more, no less – and in getting rid of the old string at the given index, be sure to avoid any memory leaks.

If there is no string in pointer in the stomach specified by *which_stomach*, then the string muncher has no raw material to work with there, and the transmogrification fails. In that case, or if this function fails for any other reason, simply return `NULL` without making any changes to the string muncher struct.

Output: This function should not print anything to the screen.

Returns: Return a pointer to the newly allocated string placed in *munchy*'s digestive tract, or `NULL` if the operation failed for any reason (e.g., if *which_stomach* was an invalid index, if *munchy* or *snack* were `NULL`, or if any calls to *malloc()* failed).

```
char *gobbleGulp(StringMuncher *munchy, int which_stomach, char *snack);
```

Description: Have you ever accidentally swallowed something so hard that it hurt going down? That's what this function is doing. The string muncher swallows the given string *hard*, sending it down to the stomach specified by the second function parameter (*which_stomach*). On its way, the snack displaces all the strings it passes along the way, since each stomach can only hold a single pointer.

More specifically, this function should insert a dynamically allocated copy of the *snack* at the index specified by *which_stomach*. Any strings above the stomach where this new string is inserted should be shifted one space upward, toward the top of the digestive tract.[2] If the specified index is greater than the SM's *num_stomachs_occupied*, the string should simply be placed in the lowest empty position in the digestive tract. Thus, there should never be any gaps in the digestive tract; the occupied stomachs should always be those at indices 0 through (*num_stomachs_occupied* – 1). As with the *omNomNom()* function, if the digestive tract is already full when this function is called, call *expandDigestiveTract()* to expand the number of stomachs to (*num_stomachs* * 2 + 1) before inserting the new string. When dynamically allocating space to hold a copy of the given *snack*, be sure to allocate exactly the amount of space needed for the string – no more, no less. Also, be sure to update *num_stomachs_occupied* and any other fields within *munchy* as appropriate after adding this *snack* to its digestive tract.

If *munchy* or *snack* are `NULL`, or if *which_stomach* is negative, this function should return `NULL` right away without modifying the string muncher or making any calls to *expandDigestiveTract()*.

Output: This function should not print anything to the screen.

Returns: Return a pointer to the newly allocated string placed in *munchy*'s digestive tract, or `NULL` if the operation failed for any reason (e.g., if *munchy* or *snack* were `NULL`, or if any calls to *malloc()* failed).

---

2 Recall that the "top" of the digestive tract is the end toward the string muncher's mouth – closer to index (*num_stomachs* – 1).

```
int digest(StringMuncher *munchy, int which_stomach);
```

Description: This function allows the string muncher to fully absorb the nutrients of whatever string resides in the stomach at the given index (*which_stomach*). Remove that string from the SM's digestive tract, avoiding memory leaks. Any strings in stomachs above the one specified in the function call must be shifted one space lower in the digestive tract, so as not to leave any gaps at the bottom of the stomach corridor, and you should place a `NULL` pointer the last stomach whose contents are moved down. The SM's *num_stomachs_occupied* field should be updated accordingly. If *which_stomach* is an invalid index or refers to an unoccupied stomach, nothing should be removed from the digestive tract.

Output: This function should not print anything to the screen.

Returns: Return 1 if a string was successfully removed from the digestive tract, 0 otherwise (including the case where the *munchy* pointer is `NULL`).

```
int heave(StringMuncher *munchy);
```

Description: Sometimes, a string muncher will find that it wishes to expel a string it has consumed. This can happen for a variety of reasons, none of which should really concern us here. The important things is that this function is one of the mechanisms by which a string muncher can expel a string. Specifically, this function always removes whatever string is currently uppermost in *munchy*'s digestive tract. When removing that string, be sure to place a `NULL` pointer in the newly emptied stomach, and be sure to avoid memory leaks. Do not to forget to update any other members of the *StringMuncher* struct as appropriate to reflect the removal of the string in question.

If *munchy* is `NULL`, or if there are no strings currently in the SM's digestive tract, simply return 0.

Output: This function should not print anything to the screen.

Returns: Return 1 if a string was successfully removed from the digestive tract, 0 otherwise (including the case where the *munchy* pointer is `NULL`).

```
int evacuate(StringMuncher *munchy);
```

Description: This another function that allows a string muncher to expel a string it has consumed. It behaves similarly to the *heave()* function, except that it expels the bottom-most string from the SM's digestive tract. Any strings in stomachs above the one being evacuated in this function call must all be shifted one space lower in the digestive tract, so as not to leave any gaps at the bottom of the stomach corridor, and you should place a `NULL` pointer the last stomach whose contents are moved down. When removing the string in question, be sure to avoid memory leaks, and do not to forget to update any other members of the *StringMuncher* struct as appropriate to reflect the removal of the string in question.

If *munchy* is `NULL`, or if there are no strings currently in the SM's digestive tract, simply return 0.

Output: This function should not print anything to the screen.

Returns: Return 1 if a string was successfully removed from the digestive tract, 0 otherwise (including the case where the *munchy* pointer is `NULL`).

```
int getDigestiveTractCapacity(StringMuncher *munchy);
```

Description: This function should simply return the maximum number of strings *munchy*'s digestive tract can hold currently, based on the value of its *num_stomachs* field. For example, for the string muncher shown on pg. 5 of this PDF, *getDigestiveTractCapacity()* would return 10, because it has a total of 10 stomachs, each of which can hold exactly one string.

This function may seem oddly simple once you have it implemented. One of the primary reasons it's here (in conjunction with the *getDigestiveTractSize()* function below) is to introduce you to a common distinction people often make between the terms "size" and "capacity" when referring to data structures.[3]

Output: This function should not print anything to the screen.

Returns: Return the capacity of the SM's digestive tract (as just described), or -1 if *munchy* is `NULL`.

```
int getDigestiveTractSize(StringMuncher *munchy);
```

Description: This function should simply return the number of non-`NULL` string pointers currently in *munchy*'s digestive tract. For example, for the string muncher shown on pg. 5 of this PDF, *getDigestiveTractSize()* would return 3, because the digestive tract currently holds 3 valid (non-`NULL`) string pointers.

This function may seem oddly simple once you have it implemented. One of the primary reasons it's here (in conjunction with the *getDigestiveTractCapacity()* function) is to introduce you to a common distinction people often make between the terms "size" and "capacity" when referring to data structures.[3]

Runtime Restriction: You must find a way to return this value in O(1) time, *without* looping through the stomachs array.

Output: This function should not print anything to the screen.

Returns: Return the size of the SM's digestive tract (as just described), or -1 if *munchy* is `NULL`.

```
void printStomachContents(StringMuncher *munchy);
```

Description: This function prints all the strings currently in the string muncher's digestive tract.

Output: Print all non-`NULL` strings currently in the string muncher's digestive tract, from bottom to top. Print a newline character, '\n', after each string. If *munchy* is `NULL`, simply print "(???)" to the screen (with the parentheses, but without the quotes), followed by a newline character, '\n'. If *munchy* is non-`NULL` but the string muncher's digestive tract is empty (e.g., it only contains `NULL` pointers), simply print "`(<NAME>'s digestive tract is empty.)`" (with the parentheses, but without the quotes, and with `<NAME>` being replaced with the given string muncher's name), followed by a newline character, '\n'.

Returns: Nothing. This is a *void* function.

---

3 It is common to use the term "size" to refer to the number of elements that have been inserted into a data structure, while "length" or "capacity" often refer to the number of cells in an array, whether those cells have been used or not. (I.e., "length" and "capacity" refer to the maximum capacity of an array, in terms of the number of elements it can hold.)

```
double difficultyRating(void);
```

Description: Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult). This function should not print anything to the screen. You do not need to call this function anywhere in your code. I will call this function myself to gather data on how difficult students found this assignment.

Output: This function should not print anything to the screen.

```
double hoursSpent(void);
```

Description: Return an estimate (greater than zero) of the number of hours you spent on this assignment. Your return value must be a realistic and reasonable estimate. Unreasonably large values will result in loss of credit. This function should not print anything to the screen.

Output: This function should not print anything to the screen.

3.1 Bonus Function

This function is optional. Any credit awarded for this function will be given as bonus points.

```
StringMuncher *cloneStringMuncher(StringMuncher *munchy);
```

Description: Dynamically allocate a new *StringMuncher* struct and set it up to be a clone of *munchy*. The clone should have entirely new, separate copies of all the data contained within *munchy*. (For example, the clone should not simply contain copies of the string pointers in *munchy*'s digestive tract. Instead, it should have entirely new copies of those strings.)

If any calls to *malloc()* fail, free any memory that this function dynamically allocated up until that point, and then return `NULL`.

Output: If *munchy* is non-`NULL`, print the following: "`-> Successfully cloned <NAME>, a string muncher with <N> stomachs, <M> of which <IS/ARE> full.`" This output should not include the quotes or angled brackets. Terminate the line with a newline character, '`\n`'. `<NAME>` is the name of the string muncher being cloned. `<N>` is the number of stomachs the SM has in its digestive tract. `<M>` is the number of stomachs that are occupied. If `<N>` is 1, print "stomach" (singular) in place of "stomachs" (plural). Similarly, if `<M>` is 1, print the word "is" in place of `<IS/ARE>`. Otherwise, print the word "are" in place of `<IS/ARE>`.

If *munchy* is `NULL`, or if any calls to *malloc()* fail, this function should not print anything to the screen.

Returns: If *munchy* is `NULL`, or if any calls to *malloc()* fail, simply return `NULL`. Otherwise, return a pointer to the newly allocated string muncher.

Continued on the following page...

4. Running All Test Cases on Eustis (*test-all.sh*)

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, *test-all.sh*, that will compile and run all test cases for you.

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

To run *test-all.sh* on Eustis, first transfer it to Eustis in a folder with *StringMuncher.c*, *StringMuncher.h*, all the test case files, and the *sample_output* directory. Transferring all your files to Eustis with MobaXTerm is fairly straightforward, but if you want to transfer them from a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*StringMuncher .c*, *StringMuncher.h*, all the test case files, and the *sample_output* folder).
2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

   ```
   scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
   ```

   Warning: Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

Once you have all your files on Eustis, you can run *test-all.sh* by connecting to Eustis and typing the following:

```
bash test-all.sh
```

If you put those files in their own folder on Eustis, you will first have to *cd* into that directory. For example:

```
cd StringMuncherProject
```

That command (*bash test-all.sh*) will also work on Linux systems and with the bash shell for Windows. It will not work at the Windows Command Prompt, and it might have limited functionality in Mac OS.

Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type (`cd project\ 2`), or the entire name needs to be wrapped in double quotes.

5. Running the Provided Test Cases Individually

If the *test-all.sh* script is telling you that some of your test cases are failing, you'll want to compile and run those test cases individually to inspect their output. Here's how to do that:

1. Place all the test case files released with this assignment in one folder, along with your *StringMuncher.c* file.

2. At the command line, *cd* to the directory with all your files for this assignment, and compile your source file with one of our test cases (such as *testcase01.c*) like so:

```
gcc StringMuncher.c testcase01.c
```

3. To run your program and redirect the output to *output.txt*, execute the following command:

```
./a.out > output.txt
```

4. Use *diff* to compare your output to the expected (correct) output for the program:

```
diff output.txt sample_output/testcase01-output.txt
```

   If the files differ, *diff* will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
1c1
< fail whale :(
---
> Hooray!
seansz@eustis:~$ _
```

   If the contents of *output.txt* and *testcase01-output.txt* are exactly the same, *diff* won't have any output:

```
seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
seansz@eustis:~$ _
```

Super Important: *Remember, using the test-all.sh script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.*

6. Testing for Memory Leaks with Valgrind

Part of the credit for this assignment will be awarded based on your ability to implement the program without any memory leaks. To test for memory leaks, you can use a program called *valgrind*, which is installed on Eustis.

Valgrind will **not** guarantee that your code is completely free of memory leaks. It will only detect whether any memory leaks occur when you run your program. So, if you have a function called *foo()* that has a nasty memory leak, but you run your program in such a way that *foo()* never gets called, *valgrind* won't be able to find that potential memory leak.

The *test-all.sh* script will automatically run your program through all test cases and use *valgrind* to check whether any of them result in memory leaks. If you want to run *valgrind* manually, simply compile your program with the *-g* flag, and then run it through *valgrind*, like so:

```
gcc StringMuncher.c testcase01.c -g
valgrind --leak-check=yes ./a.out
```

In the output of *valgrind*, the magic phrase you're looking for to indicate that no memory leaks were detected is:

```
All heap blocks were freed -- no leaks are possible
```

For more information about *valgrind*'s output, see: http://valgrind.org/docs/manual/quick-start.html

7. Special Restrictions (*Super Important!*)

You must abide by the following restrictions in the *StringMuncher.c* file you submit. Failure to abide by any one of these restrictions could result in a catastrophic loss of points.

- ★ Do not read or write to files (using, e.g., C's *fopen()*, *fprintf()*, or *fscanf()* functions). Also, please do not use *scanf()* to read input from the keyboard.
- ★ Do not declare new variables part way through a function. All variable declarations should occur at the *top* of a function, and all variables must be declared inside your functions or declared as function parameters.
- ★ Do not use *goto* statements in your code.
- ★ Do not make calls to C's *system()* function.
- ★ Do not write malicious code, including code that attempts to open files it shouldn't be opening, whether for reading or writing. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

8. Style Restrictions (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */

- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed ***above*** your *#include* statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed ***above*** the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be indented with two tabs.
- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ When defining a function that doesn't take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.
- ★ When defining or calling a function, do not leave a space before its opening parenthesis. For example: use *int main(int argc, char **argv)* instead of *int main (int argc, char **argv)*. Similarly, use *printf("...")* instead of *printf ("...")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do ***not*** have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ Use meaningful variable names that convey the purpose of your variables. It's fine to use single-letter variable names for short functions (e.g., a simple *max* function, such as *int max(int a, int b)*, where it would be silly to try to come up with more meaningful variable names for those two input parameters), for control variables in your *for* loops (where *i*, *j*, and *k* are common variable name choices), or for sizes and lengths of certain inputs (e.g., using *n* for the length of an array). Otherwise, please try to use variable names that convey the intended use of your variables. Names like *cheeseburger* and *pizza* are not good choices for this particular program.

9. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *StringMuncher.c*, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work. Be sure to include your name and NID in a header comment at the top of your source file. Also, don't forget to *#include "StringMuncher.h"* in your source code (with correct capitalization).

Do not submit additional source files, do not submit a modified *StringMuncher.h* header file, and **do not** include a *main()* function in your *StringMuncher.c* source file. Your source file must work with the *test-all.sh* script, and it must also compile on Eustis in both of the following ways:

```
gcc -c StringMuncher.c
gcc StringMuncher.c testcase01.c
```

10. Grading

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "How could these functions be called in ways that don't violate the function descriptions, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

60%	Passes test cases with 100% correct output formatting.
10%	Passes *valgrind* test cases (no memory leaks).
10%	Implementation details and adherence to the special restrictions imposed on this assignment. This will likely involve some manual inspection of your code.
20%	Follows all style restrictions; has adequate comments and whitespace; source file is named correctly and includes student name and NID (*not* your UCF ID) in a header comment. We will likely impose huge penalties for small deviations from style restrictions because we really want you to develop good style habits in this class.

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly. Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Start early. Work hard. Good luck!

Appendix A:
Getting Started: A Guide for the Overwhelmed

Okay, so, this might all be overwhelming, and you might be thinking, “Where do I even start with this assignment?! I’m in way over my head!” First and foremost:

DON’T PANIC

There are oodles of TA office hours where you can get help, and here’s my general advice on starting the assignment:

1. First and foremost, sit down and read Section 1, “Overview,” on pg. 3. Read it carefully and thoroughly, and make sure you understand the diagram on pg. 5. This might take some time. It might help to print out this PDF and go read it somewhere quiet, with your phone turned off and no other distractions.

2. Glance through Sections 2 (“StringMuncher.h (Super Important!)”) and 3 (“Function Requirements”). Also check out Appendix B (“Type Casting and the *long long unsigned int* Data Type”) on pg. 21. Again, take some time to figure out what’s going on.

3. Start by creating a skeleton *StringMuncher.c* file. Add a header comment, add some standard *#include* directives, and be sure to *#include "StringMuncher.h"* from your source file. Then copy and paste each functional prototype from *StringMuncher.h* into *StringMuncher.c*, and set up all those functions to return dummy values (zero, `NULL`, etc.). For example:

```
#include <stdio.h>
#include <stdlib.h>
#include "StringMuncher.h"

StringMuncher *summonStringMuncher(char *name, int num_stomachs);
{
   return NULL;
}

StringMuncher *banishStringMuncher(StringMuncher *munchy);
{
   return NULL;
}

StringMuncher *expandDigestiveTract(StringMuncher *munchy, int num_stomachs);
{
   return NULL;
}

// ... and so on.
```

4. Test that your *StringMuncher.c* source file compiles. If you're at the command line, your source file will need to be in the same folder as *StringMuncher.h*, and you can test compilation like so:

   ```
   gcc -c StringMuncher.c
   ```

   Alternatively, you can try compiling it with one of the test case source files, like so:

   ```
   gcc StringMuncher.c testcase01.c
   ```

   For more details, see Section 5, "Running the Provided Test Cases Individually."

5. Once you have your project compiling, go back to the list of required functions (Section 3, "Function Requirements"), and try to implement one function at a time. Always stop to compile and test your code before moving on to another function!

6. You'll probably want to start with the *summonStringMuncher()* function. As you work on *summonStringMuncher()*, write your own *main()* function that calls *summonStringMuncher()* and then checks the results. For example, you'll want to ensure that *summonStringMuncher()* is returning a non-`NULL` pointer to begin with, and that the fields inside the *StringMuncher* struct that it creates are properly initialized when you examine them back in *main()*. If you're uncertain about how to call certain functions, read through my sample test case files for examples.

7. After writing *summonStringMuncher()*, I would probably work on the *omNomNom()*, *endoscopy()*, and *printStomachContents()* functions, because they will be immensely useful in debugging your code as you work. Here's how I'd test these functions at first: In your own *main()* function, call *summonStringMuncher()*. Then, from *main()*, call *omNomNom()* to insert one or two strings into the SM you just created, and then call *endoscopy()* on a few different stomach indices (some valid, some invalid) to make sure everything is working as intended. If you get some unexpected output, trace carefully through your code to see what went wrong.

8. If you get stuck, draw diagrams. Make boxes for all the variables in your program. If you're using pointers and dynamically allocated memory, diagram everything out and make up addresses for all your variables. Trace through your code carefully using these diagrams.

9. With so many pointers, you're bound to encounter errors in your code at some point. Use *printf()* statements liberally to verify that your code is producing the results you think it should be producing (rather than making assumptions that certain components are working as intended). You should get in the habit of being immensely skeptical of your own code and using *printf()* to provide yourself with evidence that your code does what you think it does.

10. When looking for the cause of a segmentation fault, you should always be able to use *printf()* and *fflush()* to track down the *exact* line you're crashing on. Alternatively, you can use *valgrind* to help debug your code.

Appendix B:
Gallery of Various String Munchers

The remainder of this document contains various AI-generated string muncher images and the prompts used to create them.

Figure 2. “magical caterpillar eating words”

I’m not sure why this AI image generator was so opposed to adding words to these images. No matter what I tried, I couldn’t get it to do that. Still, I thought the pink flowers were a lovely and whimsical addition.

Figure 3. “magical caterpillar with blowholes”

Initially, I considered giving string munchers the ability to spit strings out of each of their stomachs via a series of blowhole-like orifices.

Figure 4. “magical caterpillar floating through a dark night sky filled with asterisks and ampersand symbols”

This was an attempt to create an image of a string muncher with code-related elements (asterisks and ampersands). The AI did not deliver on that request, but it produced a beautiful and whimsical string muncher nonetheless.

Figure 5. “magical caterpillar with lorem ipsum text in the background”

The AI again fails to give me the text I so desire. This string muncher is absolutely wild and beautiful, though, and I appreciated the spontaneous emergence of the openings in its sides, which I imagined might be a novel evolutionary trait designed to improve its ability to gobble up large strings and actively manage its digestive tract.

Figure 6. “magical caterpillar floating through a purple night sky filled with books, stars, fireflies, and wisps of smoke”

I appreciated the sudden shift in the art style here and the fact that this string muncher seems to be infused with electric power.

Figure 7. “magical bioluminescent caterpillar floating through the sky”

This was one of my earliest attempts at generating a string muncher. This one is unexpectedly bohemian and has several brightly colored silk scarves, apparently.