

Evil Hangman

This semester we will be building a project that plays the game of Hangman in a strange way. In order to begin this process we need to build a working string type for ourselves using opaque objects. Your task for this week is to set up the project makefile area and write a few initialization functions, the destroy function, and a working string compare function as well as writing some test code to test all of these operations. The lab assumes that you have completed all steps involved in lab-1.

Switch to your HANGMAN directory: `cd /Summer2016/COMP1020/HANGMAN`

Clean up your directory by typing `make clean`

Modify your Makefile so that it makes an executable named `string_driver` instead of one named `hello`. You will have to also modify the rule for `clean` so that it removes the correct file instead of the executable `hello`.

We need some additional files for our string type. Create a `my_string.h` interface file and a `my_string.c` implementation file. Create rules in your Makefile for the following:

1. You need a rule for a `my_string.o` object file which depends on `my_string.h` and `my_string.c`
2. Add `my_string.o` to the list of objects in the `OBJECTS` variable.

You should be able to build the project using `make` even though the new files are completely empty. Your directory should contain the following files after you run `make`:

```
dbadams@cs3:~/Summer2016/COMP1020/HANGMAN$ ls  
main.c main.o Makefile my_string.c my_string.h my_string.o string_driver
```

Verify that your “`make clean`” still cleans up all of the object files.

Make the project again and then type `make` once more so that you get the message:

```
make: `string_driver' is up to date.
```

Type “`touch my_string.h`” and hit enter. Then type `make` again. This should force a recompile of `my_string.o` and, because of the dependency on `my_string.o`, `string_driver`. Notice how `main.o` did not need to be recompiled because we have not yet added a dependency on `my_string.h` for `main.o`.

We are now ready to begin writing our own string library. We will begin by making a handle type and the bookends, an `init_default` function for `my_string` and a `destroy` function.

Create a new handle type for our string type by simply adding a line to our `my_string.h` file that renames `void*` to be `MY_STRING`. `MY_STRING` will then become the type for any handle holding on to one of our `my_string` opaque objects.

Next, add the following declarations to your header file.

```
//Precondition: None
//Postcondition: Allocate space for a string object that represents the empty
// string. The string will have capacity 7 and size 0 by default. A copy of
// the address of the opaque object will be returned on success and NULL on
// failure.
MY_STRING my_string_init_default(void);

//Precondition: phMy_string holds the address of a valid handle to a MY_STRING
// object.
//Postcondition: The memory used for the MY_STRING object has been reclaimed by
// the system and the handle referred to by the pointer phMy_string has been
// set to NULL.
void my_string_destroy(MY_STRING* phMy_string);
```

Attempt to build this code using stub functions in the implementation file and then continue.

Write the corresponding definitions for these functions along with a definition for the internal structure of the `My_string` object in your `my_string` implementation file so that the pre and postconditions are satisfied. The internal object, much like `vector`, has three parts: An integer, `size`, representing the number of characters the string currently holds, an integer, `capacity`, representing the number of characters the string CAN hold and a character pointer, `data`, that will hold the dynamic array of characters representing a given string.

Test your code with the `valgrind` and the following main program:

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

int main(int argc, char* argv[])
{
    MY_STRING hMy_string = NULL;

    hMy_string = my_string_init_default();

    my_string_destroy(&hMy_string);

    return 0;
}
```

TA CHECKPOINT 1: Demonstrate to your TA that you can build your code using make and that the output of your valgrind matches the following

```
==22618== HEAP SUMMARY:
==22618==    in use at exit: 0 bytes in 0 blocks
==22618== total heap usage: 2 allocs, 2 frees, 23 bytes allocated
==22618==
==22618== All heap blocks were freed -- no leaks are possible
```

The memory usage should list 4 bytes per integer (8 bytes total), 8 bytes for the character pointer and 7 bytes for the character storage making 23 bytes. Any other numbers here will indicate a problem with your code.

Add another initialization function to your code that will allow your users to initialize a string object so that it has the same value as any given c-string. This function will have the following declaration:

```
//Precondition: c_string is a valid null terminated c-string.
//Postcondition: Allocate space for a string object that represents a string
// with the same value as the given c-string. The capacity of the string
// object will be set to be one greater than is required to hold the string.
// As an example, the string "the" would set capacity at 4 instead of 3. A
// copy of the address of the opaque object will be returned on success and
// NULL on failure.
MY_STRING my_string_init_c_string(const char* c_string);
```

Implement this function and test it using the following main program:

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

int main(int argc, char* argv[])
{
    MY_STRING hMy_string = NULL;

    hMy_string = my_string_init_c_string("hi");

    my_string_destroy(&hMy_string);

    return 0;
}
```

The output of your valgrind test should be:

```
==1856== HEAP SUMMARY:
==1856==    in use at exit: 0 bytes in 0 blocks
==1856==   total heap usage: 2 allocs, 2 frees, 19 bytes allocated
==1856==
==1856== All heap blocks were freed -- no leaks are possible
```

Here it should only allocate 3 bytes for the character array, two characters for the 'h' and the 'i' and one for some mysterious purpose...

Add declarations for the following three functions to your header file:

```
//Precondition: hMy_string is the handle of a valid My_string object.
//Postcondition: Returns a copy of the integer value of the object's capacity.
int my_string_get_capacity(MY_STRING hMy_string);

//Precondition: hMy_string is the handle of a valid My_string object.
//Postcondition: Returns a copy of the integer value of the object's size.
int my_string_get_size(MY_STRING hMy_string);

//Precondition: hLeft_string and hRight_string are valid My_string objects.
//Postcondition: returns an integer less than zero if the string represented
// by hLeft_string is lexicographically smaller than hRight_string. If
// one string is a prefix of the other string then the shorter string is
// considered to be the smaller one. (app is less than apple). Returns
// 0 if the strings are the same and returns a number greater than zero
// if the string represented by hLeft_string is bigger than hRight_string.
int my_string_compare(MY_STRING hLeft_string, MY_STRING hRight_string);
```

Write implementations for the three functions above and write your own main program to test them.

TA CHECKPOINT 2: Demonstrate your test program for the size, capacity, and compare functions along with confirmation that there are no memory leaks in your code.