CYBR 401
Lab 3 – Strings, Malloc, free, #define, and IO
Adam Spanier

TASK 1 – Brains On!

In this lab, we're going to dive deeply into the whole wide world of C Strings. While programming languages like Python and Java have pre-made String structures that can easily be facilitated in programming environments, C does NOT. Instead, we have to build out our Strings by hand. To do this, we have to literally think of Strings as *arrays of characters*.

Let's explore!

Strings

As stated above, in C, strings are *arrays of characters*. This means that when we want to make a string, we first have to declare a starting address. In Python and Java the distinction between address and data is done for us. In C we have to do it ourselves. To do this we use *pointers.*

For a String, we are going to build an array of Characters (char). To allocate a char array, we need to define it as a pointer and not a char. This is done in the following way:

```
char * myString;
```

What the code above does is creates a memory slot that will hold a memory address that will point to a list of characters. The * designator on the left side of the = sign *always* denotes a pointer. Thus:

```
int * myInt;
float * myFloat;
double * myDouble;
```

all indicate pointers to a list of either ints, floats, or doubles. In this way, you can *think* of the * on the left side of the = sign as meaning "array".

An important element of arrays in C is that they ALL must be NULL TERMINATED. This means the *last* memory slot in the array must have the value '\0'. I'll beat this dead horse just so you are forewarned.

AGAIN, SAY IT WITH ME:

ALL C Arrays must be NULL TERMINATED. THIS INCLUDES STRINGS.

Function: malloc()

Now that we understand that a String in C is nothing more than an array of characters, and we understand how to build a pointer that will point to that array, we now need to tell the system to allocate some memory for us. We do this using *malloc().* This function stands for "memory allocation" and will take in a size as an argument.

Generally, we call malloc with a function to find size. This function consists of two primary elements: 1) the number of slots for the array and 2) the size of each slot.

For a character array we'll do a call like this:

```
char * myString = malloc(10 * sizeof(char));
```

In the code above we create a pointer to an array of characters called *myString.* We call *malloc* with a function: 10 * sizeof(char). The 10 is the number of slots we'll need, the *sizeof()* function determines the size of that data type; a char is 1 byte, and the multiplication takes that size and multiplies it times the number of slots to give *malloc* the right size. In this case, the size would be 10.

Now, as I mentioned above, ***WE HAVE TO NULL TERMINATE!*** Right now, there is no termination, just an open memory slot. To terminate the string, we'll do this:

```
myString[10] = '\0';
```

This sets the highest index of the memory allocation to Null. Now, when the string is iterated, the iterating mechanism like *printf()* will see the Null and know when to stop. If there is not Null termination, you'll get the dreaded *segmentation fault.*

YOU MUST NULL TERMINATE ALL STRINGS.

Function: scanf()

You already know a bit about *scanf().* With what we've done thus far with our char * variable *myString,* we can now use that variable name to push into *scanf()* and take user input. To do that we use the following code:

```
scanf("%9s", myString);
```

In the code above, the format string designates the format specifier: %s, gives it a size: 9, leaves the Null: 10 – 1 = 9, and pushes the *myString* variable into the stack to save the user input. We use the number 9 here because at 10 we have a Null stored and we need to leave that alone.

It's important to note that the user CANNOT type more than 10 characters at this point. If they do, guess what… they get a BUFFER OVERFLOW. That's a horrible security risk and why we won't use *scanf()* for actual code.

Function: free()

Now that we have memory allocated for a String, some data stored in it, and we've used it, we now want to be sure we free up the resources we've used. If we don't, we risk creating what is called a *memory leak.* To do this, we use the function *free().*

A common rule of thumb: for every *malloc()* there is one and only one *free().*We code it in the following manner:

```
free(myString);
```

Syntax: #define

Before we go further, C allows us to designate what are called macros. We do this by using the #define keywords at the very top of our C source files. These literally map an *identifier* to a *token string.*

For instance:

```
#define STR_SIZE 20
```

will define the identifier STR_SIZE as 20. This identifier can then be used *throughout* the source file. For instance we can now use STR_SIZE in our *malloc()* call.

```
char * myString = malloc(STR_SIZE * sizeof(char));
```

Now, the *sizeof(char)* will me multiplied by *20* to create the size of your String array.

We can also use #define to identify Strings or Format Strings that are often used. For instance, we can create the following format string using #define:

```
#define FORMAT_STRING "%19s"
```

This will allow us to use this format string for all strings of the size STR_SIZE.

Alrighty! I believe we now know enough to be dangerous! Let's try this out.

TASK 2 – Make a source file

Let's start this off by making a new .c source file called lab3.c. Include stdio.h and stdlib.h, outline a *main()* method, print something, and test that the source code compiles and executes correctly.

Great! You're and old pro now!

TASK 3 – Define our String size

Next up, we want to use the *#define* functionality to setup a size for our strings. Make a macro called *STR_SIZE* and set it to *20*.

TASK 4 – Function Prototypes

Now it's time to go ahead and define our function prototypes. Place them just below the *#define* call. This time we'll do them all at once.

1. Our first function will be called *getUserInput()* and it will take in *two char* * variables and return a *char *.*
2. Our second function will be called *printWelcome()* and will take in *one char* * and return a *char *.*
3. The final function we'll use is called *printUserInput().* It will take in *three char* * variables and return a single *char *.*

For a bit of help, here's a good way to define the *getUserInput()* function prototype:

```
char* getUserInput(char *, char *);
```

YOU NEED TO FIGURE THE OTHERS OUT YOURSELF.

TASK 5 – getUserInput()

Alrighty. Now that we have our function prototypes defined, let's go ahead and compile the program and make sure everything is working. If you don't get any compiler errors and you can run the program without any output, you're good!

Now it's time to dig in a bit deeper. Let's start with the *getUserInput()* function. This will be placed *JUST BELOW THE MAIN METHOD.* As we can see by the function prototype, it will take in two *char* * variables and return a single *char *.* What this means is that we will take in *two Strings,* and return *one String.*

The two input strings will be a *prompt* that will tell the user what to do. For instance, when we want the user to enter their first name, we'll send a prompt like, "Please enter your first name:" When we want them to enter their last name, we'll need to send a prompt like, "Please enter your last name:"

The output string will be *whatever the user entered.* Since we're going to have the user enter Strings, we're going to treat the input as such (by returning a *char* *). If we wanted them to enter an integer, we would return and integer.

So, here's the algorithm for you (you write the code):

1. Define the function header
    1. Return type: *char **
    2. Name: *getUserInput*
    3. Args: *char * prompt, char * format*

2. Allocate memory for an input buffer
    1. Create a *char ** variable named *userInput*
    2. Call *malloc()* using *STR_SIZE* and the size of *char – see example above*
    3. Null terminate the new *char ** variable

3. Print the prompt from the input parameters: *(printf("%s", prompt);*

4. Use *scanf()* to grab user input.
    1. Argument 1: *format – From the input parameters*
    2. Argument 2: *userInput – Results of your malloc() call*
    3. *Ex: scanf(format, userInput);*

5. Return *userInput*

Beautiful! Compile the program and check for errors. Call the new method from the *main()* using the following test code:

```
char * test = getUserInput("Enter a word: ", "%19s");
printf("%s\n", test);
```

If you get the input you entered printed to the console, you're good! If not, try to troubleshoot your code by running one line of the function at a time. See if you can figure out where it broke.

TASK 6 – printWelcome()

Now that we have the hardest function out of the way, let's build a couple helpers. The *printWelcom()* function will take in a welcome message and print it to the console.

Here's how you build it:

1. Define the function header:
    1. Return type: *void*
    2. Name: *printWelcome*
    3. Args: *char * welcome*

2. Use *printf()* to print the prompt
    1. Format String: *\n%s\n\n*
    2. Variable: *welcome*

Compile the program and make sure there are no errors. Call the function from the *main()* method with the following code:

```
printWelcome("Welcome");
```

You should simply see “Welcome” printed to the console.

TASK 7 – printUserInput()

Next up, we’ll create a function that will print out the user input we’re going to take for this program. The program will take in a *first name*, a *last name*, and a *format string* as arguments and will return a *void* when done.

Here’s how you do it:

1. Define the function header:
    1. Return type: *void*
    2. Name: *printUserInput*
    3. Args: *char * fName, char * lName, char * format*

2. Call the *printf()* function using the data passed into the function
    1. Call *printf()*
    2. First Arg: *fName*
    3. Second Arg: *lName*
    4. Third Arg: *format*

Compile the program and check for errors. Call the function from the *main()* method using the following test code:

printUserInput("FirstName", "LastName", "%s %s\n");

TASK 8 – main()

Now let’s put it all together. For the sake of time, I’m going to give you all the *prompt* and *format strings*. ***Enter these all at the top of your main() method as shown below:***

```
int main() {
  // Define Strings
  char * welcomePrompt = "Welcome to C Lab 2!";
  char * firstNamePrompt = "Please enter your first name: ";
  char * lastNamePrompt = "Please enter your last name: ";
  char * outputFormat = "\n%s %s, welcome to CYBR 401!\n\n";
  char * inputFormat = "%19s";
```

For the next section, let’s start calling our functions.

1. Call *printWelcome()* with the *welcomePrompt* variable.

2. Create a *char * variable* called *firstName* and call *getUserInput* with the following args:
    1. Arg 1: *firstNamePrompt*
    2. Arg 2: *inputFormat*

3. Create a *char * variable* called *lastName* and call *getUserInput* with the following args:
    1. Arg 1: *lastNamePrompt*
    2. Arg 2: *inputFormat*

4. Call the *printUserInput()* function with the following args:
    1. Arg 1: *firstName*
    2. Arg 2: *lastName*
    3. Arg 3: *outputFormat*

5. Finally, let's free our two *malloc()* calls using the following code:
    1. *free(firstName);*
    2. *free(lastName);*

Great!

Compile the code, check for errors, and run the program. If you did it correctly, you should get the following output:

```
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 4/Lab 3$ ./lab3

Welcome to C Lab 2!

Please enter your first name: Adam
Please enter your last name: Spanier

Adam Spanier, welcome to CYBR 401!

aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 4/Lab 3$
```

TASK 9 – Turn it In!

If you have any trouble, get ahold of me and I'll guide you thorugh. When you're done, save your work and submit the .c source file on Canvas.