CpSc 1111 Lab 12
Strings & Char Pointers

Overview

In C, a string is simply an array of characters. We have used *literal* strings all along – the stuff in between the quotes in a printf is a literal string. A literal string can be stored in a character array for later use, just as you would store the value 12 in an `int` variable. The string library `string.h` provides several functions for manipulation strings.

In this week's lab, you will see ***static*** and ***dynamic*** ways of storing strings, and write your own versions of the string library function `strcpy()`, which copies one string into another.

Initializing Character Arrays

To declare storage for a string, declare a character array big enough to hold that string.

```
char myString[80];
```

You can now put a string into `myString`.

1. INTIALIZING WHEN DECLARING

In C, a string should always be terminated with the *null character* `'\0'`. One way to load a string is to initialize it when declaring, as with the following:

```
char myString[] = "quit ";
```

2. USING `scanf()`

Another way of supplying a string value to a character array is with `scanf` for user input, such as:

```
char myString[80];
printf("Enter a string: ");
scanf("%s", myString);      // no '&' needed with arrays
```

Note that the `scanf()` function reads up until a space, tab, or newline character, and places a null character at the end of the string. So, if the user enters `Hello World` at the keyboard, `scanf` will put only `Hello` into the character array, along with the null character right after the `o`. If you want the user to enter two words and want to capture both of the words, then the following, using two character arrays, will work:

```
char myString1[80], myString2[80];
printf("Enter two strings: ");
scanf("%s%s", myString1, mystring2);
```

3. USING `gets()`

The function `gets()` can be used instead of `scanf()` to read in a string from user input. It will read everything up to the end of the line, so if the user entered `Hello World`, `gets()` would put that entire string into the array and automatically replace the newline character with a null character.

```
char myString[80];
printf("Enter a string: ");
gets(myString);
```

But – the use of `gets()` is frowned upon because there is no checking for overflow, i.e. there is no way to catch whether the user enters a string that overflows the size of the array, which could cause all sorts of unpredictable results.

4. USING `fgets()`

Another (perhaps better) option is to use `fgets()` such as the following:

```
char myString[80];
printf("Enter a string: ");
fgets(myString, 80, stdin);
```

The problem with `fgets()` is that the newline character IS read into the array, and then the null character is placed after that. So you can add the following line of code right after the `fgets()` above to get rid of the newline character and replace it with the null character:

```
myString[strlen(myString) - 1] = '\0';
```

Dynamic Memory

When declaring an array the following way,

```
char myString[80];
```

this array is said to be a ***static*** array. By "***static***" we mean that the compiler knows at compile-time how big the array will be. In other words, memory big enough to hold 80 characters is reserved at compile time for that array declared above.

We can request memory ***dynamically***, which means that the memory will be reserved at run-time instead of at compile-time.

To do this requires the use of ***pointers***, along with a function that will allocate memory and assign your pointer to point to that area of allocated memory.

Remember that an array name is essentially a "pointer" – it references the area of memory where the elements of the array reside (which is why the '`&`' is not needed with a `scanf()` because the name of the array already refers to the address location in memory of those items in the array).

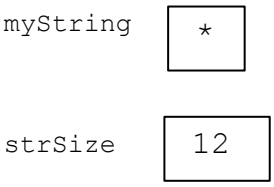
However, when ***dynamically*** allocating memory for a character array, a *pointer type* must be used. The pointer for a string must be a character pointer (because the pointer will be pointing to an individual item in the array). When declaring the character pointer, space is reserved in memory for just the pointer. It's after the call to the function to allocate the memory (at run-time) that the pointer will be assigned to point to the beginning of a chunk of memory.

```
char *myString;   // character pointer
const int strSize = 12;

myString = malloc(strSize  * sizeof(char)); // malloc() function is in  <stdlib.h>
```

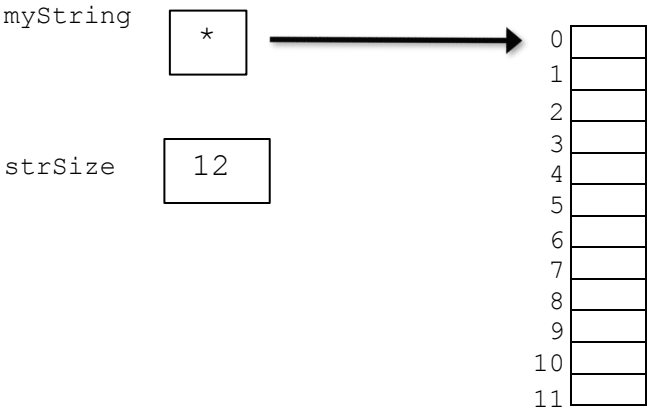
Using the `calloc()` function would do the same thing. The difference is that `calloc()` has two arguments whereas `malloc()` has one:

```
char *myString;   // character pointer
const int strSize = 12;

myString = calloc(strSize, sizeof(char));  // calloc() function is in  <stdlib.h>
```

At **compile time**, either of the two text boxes above will result in memory being reserved for the following:



At **run-time**, you can imagine the following, where the `myString` pointer is assigned to point to another area of memory where 12 characters can be stored:



NOTE: Usually, when visualizing pointers, we draw an arrow to point to our pretend area of memory; but in actuality, the value of the `myString` pointer would be *the address* of that area of memory. In other words, once assigned, ***the value of a pointer is the address of where it is "pointing to"***, not really an asterisk with an arrow pointing somewhere else in memory. (Before a pointer is assigned a value (i.e. an address), it has the value of NULL.)

The following is an entire example program using a character pointer and `malloc()`:

```
#include <stdio.h>
#include <stdlib.h>  // the library that provides malloc

int main(void) {
   char *myString;
   const int strSize = 12;

   myString = malloc(strSize  * sizeof(char));

   // if malloc returns NULL, it wasn't able to allocate enough memory
   if (myString == NULL) {
      printf("malloc failed to allocate enough memory!\n");
      return 1;   // returns 1, which quits the program with a value
                  // that indicates an unsuccessful run of the program
   }

   myString = "Hello World!!";   // notice that if using a pointer,
                                 // a string value can be assigned;
                                 // could have used strcpy() instead
   printf("myString is:  %s\n", myString);

   return 0;
}
```

Lab Assignment – Part 1 – Using Static Arrays

Do this part on your own. Write a program named `lab12_a.c` containing the following function:

```
// preconditions:  src is terminated by '\0'
//                 dest is big enough to hold src
// postconditions: dest contains src and is terminated by '\0'
//
void my_strcpy(char dest[], const char src[])
```

This function will take two character arrays as parameters. The function should use a loop to copy the contents of `src` from start to `'\0'` into `dest`. It should also copy the `'\0'` into `dest`. (This function will fail if `dest` isn't big enough to hold `src`, but there's really no easy way for us to check for that, so we'll have to assume that the user uses our function properly.)

In `main()`, define two character arrays `str1` and `str2` of length 31. Prompt the user for a string and use `scanf()` to read the string into the first character array, `str1`.

Then use `my_strcpy()` to copy the string from `str1` into the other char array, `str2`. Now copy "`You entered:`" into `str1` using your `my_strcpy()` function. Use `printf("%s %s\n", str1, str2);` to print both strings.

Example Output:

```
$ ./a.out
Enter a string of length at most 30: Hello World
You entered: Hello
```

Lab Assignment – Part 2 – Using Character Pointers and Dynamically Allocated Memory

Do this part on your own. Write a program named `lab12_b.c` containing the following function:

```
// preconditions:  src is terminated by '\0'
//                 dest is big enough to hold src
// postconditions: dest contains src and is terminated by '\0'
//
void my_strcpy(char *dest, const char *src)
```

This function will take two character pointers as parameters. The function should use a loop to copy the contents of `src` from start to `'\0'` into `dest`. It should also copy the `'\0'` into `dest`. Note that incrementing a pointer makes it point to the next memory location.

In `main()`, define two character pointers `str1` and `str2` and an integer. Prompt the user for string length and use `scanf()` to store this in your `int` variable. Use `malloc()` (or `calloc()`, whichever one your prefer) to allocate memory for both character pointers with the length (plus one) read from the user. Make sure that `malloc()` (or `calloc()`) returned non-NULL addresses for both character pointers. Then prompt the user for a string using `scanf()` to read the string into the first pointer, `str1`.

Then, as in Part 1, use `my_strcpy()` to copy the string from `str1` into the other char pointer, `str2`. Now copy "`You entered:`" into `str1` using your `my_strcpy()` function. Use `printf("%s %s\n", str1, str2);` to print both strings.

Example Output 1:

```
$ ./a.out
What is the longest length of a string that you will you enter?  30
Enter a string: Hello World
You entered: Hello
```

Example Output 2:

```
$ ./a.out
What is the longest length of a string that you will you enter? 3000000000
malloc failed to allocate enough memory!
```

Turn In Work

Before submitting your files, show your ta that you completed both parts of the lab assignment. Then submit both of your files `lab12_a.c` and `lab12_b.c` using the handin page: http://handin.cs.clemson.edu