Institute of Computer Science

CMSC 21: Fundamentals of Programming

Module JOURNAL

Instructions: Accomplish this every week so we can monitor your progress and improve everyone's learning experience. Completely and honestly answer the following questions in relation to this module's lesson/topic.

Submission:
Filename: "<surname><FirstnameInitial><MiddleInitial>_module0#_journal.pdf"
Example: "delaCruzJC_week01_journal.pdf"
Submit to Google Classroom on Module 01

***Accomplish this journal when you are done (or almost done) with the requirements for the module.*

Name:	**Keith Ginoel S. Gabinete**	Module No:	**4**
Student Number:	**2020-03670**	Date:	**08/05/2021**

- Choose at least one of the things discussed that you understood the most. Imagine explaining it to a classmate. **Explain it in your own words**.

I learned that an array, in C programming language, is simply a variable that can hold multiple values. For example, if you want to store 100 float numbers or 200 integers or 300 integers and use them at some point in your program, then you can declare an array for them. However, unlike in Python wherein an array can store data of different types, an array in C can only hold values sharing a single data type, and that data type must also be specified upon that array's declaration. Also note that when an array with n components is defined in a program, data are being stored in memory locations consecutively and that a computer labels the whole array with its array name.

There are two types of arrays: static arrays and dynamic arrays. The concept behind these types of arrays are likewise comparable to the concepts of static variables and dynamic variables as well (notice that I've mentioned earlier that arrays are simply variables as well). So an array whose size cannot be altered (as its size is already determined when its created) is called a static array. Static arrays, just like other variable types, can be declared with initialization or not. Declaring it with no initialization, one must follow the convention

```
<data_type> <array_name> [<array_size>] ;
```

Here is one example:

```
int number [8];
```

Whereas having it initialized follows the convention:

```
<data_type> <array_name> [] = {'<element_0>', '<element_1>',...} ;
```

Here is an example as well:

```
char red [] = {'r', 'e', 'd', '\0'};
```

Here, you can notice that the size of the array isn't defined beside its array name since the number of elements in the initialization part would automatically specify that array's size.

When accessing each element in an array, indices are used. The lowest index is zero and the highest index is <size_of_array> - 1. It follows the syntax:

<array_namr>[index];

Additionally, when assigning a value to an array element, one must follow the format:

<array_name>[index] = <expression/value>;

Whenever we use an array name without indexing, it 'magically' turns into a pointer - returning the address of the array's first element. At times like this, we can actually use pointer operators for that 'magically' turned pointer as well. And since it somehow returns the address of the first element of the array, we can actually assign its value to other variable pointers as well. Without indexing, elements of the array can also be accessed using this trick:

*(<array_name> + i)

where i is an integer equivalent to its position when called in the normal indexing method.

Having these kinds of information, one must still remember that array name is still not actually a pointer. It is an array identifier with a constant value. Thus, its value cannot be changed.

Moreover, when we use arrays as function parameters, it is actually the address of their first element that is being passed.

The second type of arrays are called the dynamic arrays. Here, size of arrays doesn't need to be determined ahead of time. It also allows elements to be added or removed. When dealing with C programs of much complexity, it is truly wise to use dynamic variables in one's code as it saves less memory consumption since arrays are only defined during runtime and can be released after its use. Just like creating normal dynamic variables, malloc function defined in the stdlib.h library also plays a vital role in creating dynamic arrays in C. A malloc function allocates a block of memory with a given size and returns a pointer that points to the first byte of the allocated space.

When using the malloc function, we use the statement format:

And when the computer has already made use of the defined dynamic array, we can then use the free function (also defined in the stdlib.h library) to free the used block of memory so we can use it up again for other program's processes.

And that's it, that is all what I've learned about one-dimensional array in the Module 4 of CMSC 21. Note that a multidimensional array exists and it bears much complexity than the one being discussed here. And that there is an array of characters in C enclosed within double quotes (" ") called a string and is actually popular in any programming language.

- What problem/confusion did you encounter about the module lessons? **Explain the problem.**

1. Since the problem indicated in the individual exercise for Module 4 is mainly about a palindrome (a word, phrase, or sequence that reads the same backward as forward), I had spent much time thinking as to how I can reverse the input string from the user and neatly compare it to its forward counterpart.

2. When I practice using string functions in C, I noticed that the strlen function behaves differently from the len function I learned in Python. Thus, this brought a little confusion to me while going over the topics discussed in Module 4 (this is more of a realization than an actual probrlem).

- How did you solve it? **Explain the fix or solution found.**

1. I browsed online for possible solutions about this little problem of mine about reversing string components. I then found this solution https://forgetcode.com/c/1913-reverse-the-string-using-for-loop. Here, the method is to use the classical method of manipulating array data – creating for loop statements that run over each element of an array paired with a basic string function in C called strlen.
By having a for loop statement that decrements the value of the condition variable i in its every iteration, and initializing value of i with the length of the given array minus 1, then we'll be able to store the reversed version of the string into an empty string with size equivalent to the given array.
Here is the code:

```c
#include <stdio.h>
#include <string.h>

int main () {
    char green [] = "green";
    int size_green = (int)strlen(green);
    char green_reversed [size_green];


    for (int i= (size_green -1); i>=0; i--) {
        green_reversed[size_green - i - 1] = green[i];
    }

    printf("Reversed version of the word green: %s\n", green_reversed);

    return 0;
}
```

Here is the output:

```
teioh@DESKTOP-HRVA4VU:/mnt/c/Users/ASUS/Desktop/CMSC 21 Codes/Laboratory Exercises$ gcc test.c
teioh@DESKTOP-HRVA4VU:/mnt/c/Users/ASUS/Desktop/CMSC 21 Codes/Laboratory Exercises$ ./a.out
Reversed version of the word green: neerg
teioh@DESKTOP-HRVA4VU:/mnt/c/Users/ASUS/Desktop/CMSC 21 Codes/Laboratory Exercises$
```

2. I noticed that when I have an initialized character array like this

```
teioh@DESKTOP-HRVA4VU:/mnt/c/Users/ASUS/Desktop/CMSC 21 Codes/Laboratory Exercises$ ./a.out
Length of the array: 5
```

and tries to run the strlen function over it, the function does not count that '\0' character defined before the closing curly bracket (although I defined it as an element of the array).

```
teioh@DESKTOP-HRVA4VU:/mnt/c/Users/ASUS/Desktop/CMSC 21 Codes/Laboratory Exercises$ ./a.out
Length of the array: 5
```

This made me speculate that the strlen function does not actually count all the elements present in the array but counts all the elements in the array before it encounters the '\0' element. Still had doubts about my generalization, I browsed through the internet to find out how this strlen function actually behaves and I actually found a site that confirmed this little hypothesis of mine. Unlike in Python, where len function actually returns the number of characters/elements in the string <https://www.w3schools.com/python/ref_func_len.asp>, the site I've stumbled upon <https://fresh2refresh.com/c-programming/c-strings/c-strlen-function/> states that stlen function stops counting the character when null character is found because, null character indicates the end of the string in C. Thus, if I try something like this:

```
char green [] = {'g', 'r', 'e', 'e', 'n', '\0', 'y', 'e', 'l', 'l', 'o', 'w'};
```

The output would still be like the one I've got before.

```
teioh@DESKTOP-HRVA4VU:/mnt/c/Users/ASUS/Desktop/CMSC 21 Codes/Laboratory Exercises$ gcc test.c
teioh@DESKTOP-HRVA4VU:/mnt/c/Users/ASUS/Desktop/CMSC 21 Codes/Laboratory Exercises$ ./a.out
Length of the array: 5
```

- Comments/Suggestions (Optional but we will appreciate it if you tell us one you have in mind)