Laboratory 12 – Dynamic memory allocation 22.12.2020

1. [5points] Write a program that will allocate memory to an array of the size specified by the user at runtime. Use `malloc` and `free`. Fill the array, print the elements and calculate the average value.

A. In the `main` function:

a) Ask the user for the size of the array.

b) Using the `malloc` function allocate the `double` array of the size specified by the user.

c) Check if the allocation was successful.

- If the address returned by `malloc` is not `NULL`, use the `rand` function in a `for` loop and assign pseudorandom values to the array elements. Then call the function `averagevalue.` Print the result. Free up memory with the `free` function.

- If the allocation failed and the address returned by `malloc` is `NULL`, print the message and exit the program.

B. Define the function `averagevalue` and then call it in `main`.

The function calculates the average value of the elements of the array passed as an argument and prints the array elements to the screen. The function returns the average value.

Test data:

```
Enter the size of the array.
2
x[0] = 0.097718
x[1] = 0.202734
avg = 0.150226

Enter the size of the array.
8
x[0] = 0.125083
x[1] = 0.879350
x[2] = 0.586159
x[3] = 0.045694
x[4] = 0.203307
x[5] = 0.889044
x[6] = 0.632176
x[7] = 0.823807
avg=0.523078
```

2a **[9 points]** Print the Pascal's triangle with the `realloc` function.
https://en.wikipedia.org/wiki/Pascal%27s_triangle
https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle

Test data:

```
Enter the height of Pascal's triangle:1
 1
1 1
```

```
Enter the height of Pascal's triangle:7
       1
      1 1
     1 0 1
    1 1 1 1
   1 0 0 0 1
  1 1 0 0 1 1
 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1
```

```
Enter the height of Pascal's triangle:15
              1
             1 1
            1 0 1
           1 1 1 1
          1 0 0 0 1
         1 1 0 0 1 1
        1 0 1 0 1 0 1
       1 1 1 1 1 1 1 1
      1 0 0 0 0 0 0 0 1
     1 1 0 0 0 0 0 0 1 1
    1 0 1 0 0 0 0 0 1 0 1
   1 1 1 1 0 0 0 0 1 1 1 1
  1 0 0 0 1 0 0 0 1 0 0 0 1
 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Pascal’s triangle is a triangular array of the binomial coefficients.

> The entry in the *n*th row and *k*th column of Pascal's triangle is denoted $\binom{n}{k}$. For example, the unique nonzero entry in the topmost row is $\binom{0}{0} = 1$. With this notation, the construction of the previous paragraph may be written as follows:
>
> $$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$
>
> for any non-negative integer *n* and any integer *k* between 0 and *n*, inclusive.[4] This recurrence for the binomial coefficients is known as Pascal's rule.

The first 6 rows of Pascal’s Triangle.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

How Pascal's rule works?

4+1=5

4+6 =10

1+2 = 3

We will use Pascal's rule in the program. We're going to use a **one-dimensional array** that will hold **one row** of Pascal's triangle.

As the size of the row of the Pascal triangle increases, we will use the `realloc` function to change the length of the array.

Follow the instructions step by step:

In the `main` function

a) Enter the height of Pascal's triangle. Use `scanf` function.

b) Declare a pointer to `int` and set it to `NULL`.

c) The outer `for` loop calculates and prints rows of Pascal’s triangle.

- Inside the loop, allocate an array whose size depends on the loop control variable. Use the `realloc` function.

- Check if the allocation was successful, otherwise exit the program.

- Insert the value 1 into the last cell of the array.

Loop iteration	The length of the array	The last cell of the array
0	1	0
1	2	1

2	3	2
3	4	3
4	5	4

- The inner `for` loop number 1. Using the inner loop, we move through the array from end to start, updating the values in the array using Pascal's rule.

Outer loop iteration	Before inner loop	After inner loop	Inner loop iteration
0	1	1	0
1	1 1	1 1	0
2	1 1 1	1 2 1	1
3	1 2 1 1	1 3 3 1	2
4	1 3 3 1 1	1 4 6 4 1	3
5	1 4 6 4 1 1	1 5 10 10 5 1	4

How Pascal's rule works?

1	4	6	4	1	
1	4	6	4	1	1
1	4	6	4	5	1
1	4	6	10	5	1
1	4	10	10	5	1
1	5	10	10	5	1

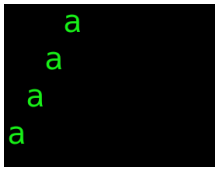
- The inner `for` loop number 2**.** In the inner loop, check whether the value in the array is even or not. If it is odd, then print "`1`". If it is even, then print "`0`".

- Add `printf("\n");`

d) Free up memory with the `free` function.

2b [1 point] If the correct values are printed on the screen, we can center them using the format "`%*`". Examine the example and modify the way the Pascal's triangle is printed.

```
printf("%*c\n", 4, 'a');
printf("%*c\n", 3, 'a');
printf("%*c\n", 2, 'a');
printf("%*c\n", 1, 'a');
```



3. [5points] There are two strings `str1` and `str2`. Write a function `swapstr` which copies the contents of `str1` to `str2`, and the contents of `str2` to `str1`.

A. In the `main` function:

a) Declare two strings and set them to values `"abcdefg`" and `"ijklmno"` respectively. Don't use `malloc`.

b) Print both strings.

c) Call the `swapstr.` The first argument is the address of the first string, the second argument is the address of the second string, and the third argument is the size of the first string. The function returns nothing with `return.` Calculate the size of the string using the `strlen` function.

b) Print both strings.

B. Definition of the `swapstr` function.

a) With the `malloc` function create a temporary variable into which you copy the contents of one of the strings.

b) If the allocation was successful, call `strncpy` 3 times to swap the contents of the strings. Free up memory by using the free function.

`Test data:`

```
Before:  str1 = abcdefg, str2 = ijklmno
After:   str1 = ijklmno, str2 = abcdefg
```

Next lab 13 – Multidimensional arrays.

Bonus assignment for curious programmers.

If you have valgrind (https://valgrind.org/) installed on your computer, you can check how the program handles memory allocation and deallocation.

From the web page:
“Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.”

A. Report for exercise 1 program.

`valgrind ./a.out`

```
==54742== Memcheck, a memory error detector
==54742== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==54742== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==54742== Command: ./a.out
==54742==
Enter the size of the array.
3

x[0] = 0.943133
x[1] = 0.006100
x[2] = 0.028777
avg=0.326003
==54742==
==54742== HEAP SUMMARY:
==54742==     in use at exit: 0 bytes in 0 blocks
==54742==   total heap usage: 3 allocs, 3 frees, 2,072 bytes allocated
==54742==
==54742== All heap blocks were freed -- no leaks are possible
==54742==
==54742== For lists of detected and suppressed errors, rerun with: -s
==54742== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

B. Report for exercise 2 program.

`valgrind ./a.out`

```
==16143== Memcheck, a memory error detector
==16143== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16143== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==16143== Command: ./a.out
==16143==
Enter the height of Pascal's triangle:7
       1
      1 1
     1 0 1
    1 1 1 1
   1 0 0 0 1
  1 1 0 0 1 1
 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1
==16143==
==16143== HEAP SUMMARY:
==16143==     in use at exit: 0 bytes in 0 blocks
==16143==   total heap usage: 10 allocs, 10 frees, 2,192 bytes allocated
==16143==
==16143== All heap blocks were freed -- no leaks are possible
==16143==
==16143== For lists of detected and suppressed errors, rerun with: -s
==16143== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

C. Report for exercise 3 program.

`valgrind ./a.out`

```
==136727== Memcheck, a memory error detector
==136727== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==136727== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==136727== Command: ./a.out
==136727==
str1 = abcdefg, str2 = ijklmno
str1 = ijklmno, str2 = abcdefg
==136727==
==136727== HEAP SUMMARY:
==136727==     in use at exit: 0 bytes in 0 blocks
==136727==   total heap usage: 2 allocs, 2 frees, 1,031 bytes allocated
==136727==
==136727== All heap blocks were freed -- no leaks are possible
==136727==
==136727== For lists of detected and suppressed errors, rerun with: -s
==136727== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```