

1. [3points] 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.

C. Use `valgrind` to check memory usage.

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`

c. 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)
```

2. [7 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
```

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 0 0 0 0 0 0 1
1 1 0 0 0 0 0 1 1
1 0 1 0 0 0 0 1 0 1
1 1 1 1 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

```

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 will 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.

Program structure:

```
int main (void){
    //Declare a variable that will hold the height of the Pascal triangle.
    //Assign its value using the scanf function.
    //Declare a pointer to int and set it to NULL.

    //The outer for loop.
        //Allocate an array whose size depends on the loop control variable. Use the realloc function.
        //In the first iteration of the loop, the size of the array is one.
        //The array size increases by one for each iteration.
        //Check if the allocation was successful, otherwise exit the program.

        //In each iteration of the loop, put 1 at the end of the array.

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

            //The inner for loop number 2.
                //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");

        //Free up memory with the free function.

        return 0;
}
```

How Pascal's rule works?

The contents of the array after the fifth iteration of the outer loop:

1	4	6	4	1
---	---	---	---	---

Changing the contents of the array during iteration of the inner loop.

1	4	6	4	1	1
---	---	---	---	---	---

1	4	6	4	5	1
---	---	---	---	---	---

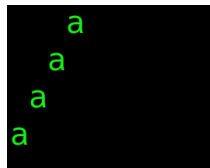
1	4	6	10	5	1
---	---	---	----	---	---

1	4	10	10	5	1
---	---	----	----	---	---

1	5	10	10	5	1
---	---	----	----	---	---

If the correct values are printed, we can center them using the format "%*s". 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');
```



B. Use valgrind to check memory usage.

```
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)
```

Next lab 11 – Strings and multidimensional arrays.