



PROGRAMMING **C**

Lesson No. 7

Programming

C

Contents

1. Random number generator	3
2. Use of random number generator	10
3. Two-dimensional arrays as a specific case of multi-dimension arrays.	12
4. Use case	16
5. Home assignment.	18

1. Random number generator

In previous lesson we have acquainted with such a concept as an array and learned to fill it with values. But all our values we input ourselves, meaning we knew in advance what they will be. This method of filling variables and arrays limits the program functionality. You cannot create something with an artificial intelligence, if there is no way to get the data irrespective of the user. What to do if we need the randomly values selected?

Use of rand function.

In language C there is an opportunity to randomly generate a number. For this operation we use the function with the name `rand()`. This function is within the library file — `stdlib.h`, consequently for its work this file should be connected by means of the directive `#include`. To the place of call **`rand()`** within the program there will be put a random number in the range from 0 to 32767. Let's review a simple example:

```

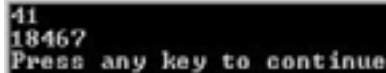
#include<iostream>
#include<stdlib.h>// this file contains the function rand

using namespace std;
void main()
{
    int a;
    //generating a random number and its record into a variable a
    a=rand();
    cout<<a<<"\n";

    /* generation of a random number and its record into the variable a */
    a=rand();
    cout<<a<<"\n";
}

```

If you created a project, typed in and run our sample you will find out that the program sequentially generates two so called random numbers. These ones:



```

41
18467
Press any key to continue

```

Run again and you will see the same picture. It turns out that random numbers are not random, besides they repeat from start to start. In spite of wrongness of the assertion it is clear why it happens so. If you come up a person in the street and ask him to say a random integer number, this person will definitely say exactly a random number. But this is far from certain. Probably a person will look at the poster or watch and gets the number out of the noticed. Computer, unlike a human being, cannot think associatively that is why the function `rand()` does not obtain a number from nowhere, but works in a capacity of the initial point — a point defined upon writing an algorithm of random number generator, i.e. a constant number. In other words, basing on this point upon

different program requests this function generates the same number and we have already got proof thereof. We can make the conclusion: with this purpose in order the `rand()` gave different numbers during different program invocations, we should change the initial generation point.

Use of the `srand` function

Position of a function — library `stdlib.h`.

Function **`srand`** installs an initial point for random numbers generation and has the following syntax:

```
void srand(unsigned int start)
```

Parameter `start` a function takes on is a new point for random number generation. Let's think what number we should write in the place of this parameter. An integer literal or variable, which value is defined at the moment of program writing, is not appropriate. Having put them to the place of the starting point we will certainly change it but make dynamic at the same time. And other numbers will be generated but the same at each startup.

Example with the literal in a capacity of a starting point:

```
#include<iostream>
//this file contains functions rand and srand
#include<stdlib.h>

using namespace std;
void main()
{
    srand(5);
    int a;
    //generating a random number and record it within the variable
    a=rand();
    cout<<a<<"\n";
}
```

Example of variable in a capacity of a starting point:

```
#include<iostream>
// this file contains functions rand and srand
#include<stdlib.h>

using namespace std;
void main()
{
    int start=25;
    srand(start);
    int a;
    //generating a random number and writing it within the variable
    a=rand();
    cout<<a<<"\n";
}
```

We need something that continuously changes regardless any external factors. Agree that time is that kind of the value. And time is exactly what we use as a starting point.

Use of the time function.

Position of the function is library time.h.

Time function has several purposes and we are not going to study it in details. Let's deal with what we need for work. In

particular in case we request the time function with the parameter NULL, this function will return a number of milliseconds since January 1st 1970 to the place of its request within the program. You will agree that this value will be different each time. And this is what we have been looking for. Let's «collect» obtained information into an integral whole and we get:

```
srand(time(NULL));
```

Srand function installs a number representing a number of millisecond since January 1st 1970 in a capacity of the starting point. Let's try:

```

#include<iostream>
#include<stdlib.h> // this file contains rand and srand
#include<time.h> // this file contains the time function

using namespace std;
void main()
{
    srand(time(NULL));
    int a;
    //generating a random number and record it into a variable
    a=rand();
    cout<<a<<"\n";
}

```

Having input a corrected number you might have ascertained that now different numbers are generated upon various program startup, but this is not all functions of the generator.

Setting up a range for generator.

Numbers we get by means of rand function request are within the range from 0 to 32767. But we don't always need such a great data spread. What should we do if we need to generate the numbers from 0 to 10 or from 0 to 100 and so on?! In this case division modulo would prove helpful.

Let's take a random number 23 for example. You agree that whatever number you divide by 23 modulo you will get either 0 (if there is no excess), or an excess in the range from 1 to 22. This very characteristic we are going to use and divide a generated random number by a random number modulo:

```
int a=rand()%23;
```

Basing on this rule we can conclude a formula:

NUMBER IN THE RANGE FROM ZERO TO X:

```
rand() % X
```

But the range not always starts from zero. Assume we need the range from 11 to 16. Everything is simple. We should generate numbers from 0 to 5 (difference between 16 and 11), and then to «move» an obtained result to 11 digits.

```
int a=rand()%5+11;
```

Basing on the modified rule we can conclude the formula:

NUMBER IN THE RANGE FROM Y TO X:

```
rand() % (X-Y) + Y
```

So, we have got acquainted with the random numbers generator and now we can make our work with the arrays simple. But how? And the next lesson unit will tell us how.

2. Use of random number generator

Let's review an example of random number generator use; in particular we will talk about filling of an array with random numbers:

```
#include<iostream>
// this file contains functions rand() and srand()
#include<stdlib.h>
#include<time.h> // this file contains time() function
using namespace std;

void main()
{
    srand(time(NULL));
    int array[10];
    for (int i=0;i<10;i++)
    {
        // generating a random number and its record in the current array element
        array[i]=rand()%100;

        // display of element value
        cout<<array[i]<<"\n";
    }
}
```

1. In the described above example an array of 10 elements will be displayed and it will be filled with random numbers.
2. At each loop iteration a new random number is generated.
3. Upon each program startup an array is filled in a different way due to the string `srand(time(NULL))`;
4. Numbers within the array vary from zero to 100 because generation result can be divided by 100 modulo.

As you can see a random number generator is easy to work with and now we have a tool that allows not only testing the programs without inputting data from the keyboard, but creating a primitive artificial intellect.

3. Two-dimensional arrays as a specific case of multi-dimension arrays

We already know what the arrays are and in previous lesson we have reviewed that a so called one-dimension array. One-dimension array — is a data array where each value has the only characteristic — index number (index). That is why we address the particular element by this index.

Now we are going to talk about multi-dimension arrays, i.e. arrays where each element is described by several characteristics. Example of multi-dimension array value could be practically anything:

1. Chess board — each square has two dimensions E2 (letter and digit)
2. KVN MARK — three dimensions JURY_MEMBER, CONTEST, COMMAND.

Maximum array dimension acceptable for C is we will dwell on a question of two-dimension array which is also called matrix.

Two-dimension array

Declaration and positioning within the memory.

Two-dimension array is an aggregate of lines and columns with the particular value on crossing thereof. It is not difficult to declare a two-dimension array, with this purpose you have to indicate a number of lines and columns. Wherein, we base on the same rules as when we declare a one-dimension array. I.e. we cannot indicate non-constant or non-integer values to denote the number of lines and columns.

General syntax:

```
data_type array_name [number_of_lines][number_of_columns];
```

example:

```
const int row=3; // lines
const int col=4; // columns
int array[row][col]; // array with the size row by col(3x4)
```

)

	C column 0	C column 1	C column 2	C column 3
Line 0	a [0][0]	a [0][1]	a [0][2]	a [0][3]
Line 1	a [1][0]	a [1][1]	a [1][2]	a [1][3]
Line 2	a [2][0]	a [2][1]	a [2][2]	a [2][3]

column index

line index

array name

In spite the fact we represent an array in a form of matrix, in reality any two-dimension array is positioned line by line within the memory: at first there is a zero line, then the first one and so on. We should remember about that, because for overrunning an array boundaries can result in the incorrect program functioning whereupon there will be no error notification.

```
a[0][0] a[0][1] a[0][2] a[0][3] a[1][0] a[1][1] a[1][2] a[1][3] a[2][0] a[2][1] a[2][2] a[2][3]
```

Initialization.

Initialization of two-dimension array is analogical to one-dimension one:

1. Initialization upon creation

Each string is bracketed in separate curly braces:

```
int array[2][2]={{1,2},{7,8}};
```

values are indicated in sequence and line by line are written into an array:

```
int array[2][2]={7,8,10,3};
```

if a values is skipped it will be initialized by zero:

```
int array[3][3]={{7,8},{10,3,5}};
```

2. Initialization using a loop

We reveal a secret — a two-dimension array can be reviewed as an aggregate not only lines but one-dimension arrays. Meaning a one-dimension array we fill in with simple loop searching particular elements and if there is an aggregate we should also search separate arrays.

Note: addressing a particular array element is exercised by line and column number, for example — `mr[2][1]` — is a value on the crossing of the second line and first column.

Working with two-dimension array is not difficult either if compare with one-dimension and we can prove it practice.

```

#include<iostream>
#include<stdlib.h> //this file contains rand and srand
#include<time.h> // this file contains the function time
using namespace std;
void main()
{
    const int row=3; // lines
    const int col=3; // columns
    int mr[row][col]; // array of the size row by col

    /* search separate lines (one-dimension arrays cumulatively) */
    for(int i=0; i<row; i++)
    {
        // search separate elements of each line
        for(int j=0; j<col; j++)
        {
            /* initializing elements in the range from 0 to 100 */
            mr[i][j]=rand()%100;

            // display of values on a screen
            cout<<mr[i][j]<<" ";

        }

        // transition to other matrix line
        cout<<"\n\n";
    }
}

```

4. Use case

Problem statement.

Write a program that finds a maximum element of each line within a two-dimension array.

Realization code :

```
#include<iostream>
#include<stdlib.h> // this file contains rand and srand
#include<time.h> // this file contains the function time

using namespace std;

void main()
{
    // set up dimensions of an array
    const int m = 3;
    const int n = 2;
    int A[m][n]; // declare a two-dimension array

    // filling up an array with random numbers and display on a screen

    // search separate lines
    for(int i=0; i<m; i++)
    {
        // search separate elements of each line
        for(int j=0; j<n; j++)
        {
            // initializing elements by the values in the range from 0 to 100
            A[i][j]=rand()%100;

            // values display on a screen
            cout<<A[i][j]<<" ";

        }

        // transition to other line
        cout<<"\n\n";
    }
    cout << "\n\n";

    // search for a maximum element within the line

    // search separate lines
    for (int i=0; i<m; i++){
```



```

// assume that line zero element is a maximum
int max = A[i][0];

// search for a maximum element within the current line

// change of column index for the current line
for (int j=0; j<n; j++)
{
    if (A[i][j] > max)
        max = A[i][j];
}
cout << "maximum element " << i
    << "-Oh lines = " << max << endl;
}
}

```

Pay attention!

1. During each loop iteration there is a zero element of the current line is chosen as a maximum.
2. After analysis of the particular line the found maximum is displayed on a screen.

5. Home assignment

1. There is given a two-dimension array, the size 3×4 . We need to find a number of elements which values are equal to zero.

2. There is given a square matrix of n -order (n lines, n columns). We need to find the largest element value positioned within the dark-blue matrix parts.

All the arrays within this home assignment are filled in randomly.



a



b



c



d



e



f



g



h



i



j

