Programming Languages 1
Lesson 8 - Guide

Random numbers

Personal computers cannot generate real random numbers since they are deterministic devices. In this section we are talking about only pseudo-random numbers, however usually the 'pseudo' prefix will be ignored.
The generation of random numbers means that starting from an initial value (seed) the computer calculates another value which will be the base of further generations, and so forth. Always the elements of the same very long cyclic number sequence are determined by a mathematical method (e.g. linear congruential generator), but these elements of the sequence seem random for human because the relationship between the neighboring number seems stochastic, thus we suppose they are random values.
A parameterless function of the `stdlih.h` header, called `rand`, is used to do this. Each call of the function results in an integer number between `0` and `RAND_MAX` (based on the actual seed value). The `RAND_MAX` is a symbolic name, also defined in the `stdlib.h`, referring to the possible greatest random number. It is important to know that this value in Linux systems is usually 2 147 483 647, while in Windows system it is only 32 767, which is a huge difference. (For instance, during the generation of one-million random number in windows the same sequence is repeated circa 30 times.) All the integer numbers between zero and `RAND_MAX` has the same probability
After each call of `rand`, the seed value of the system changes, but after each start of the given program, it uses the same initial seed value. Thus a program should always produce the same numbers. In order to avoid this the seed of the random generator must change at the beginning of the execution. It is the aim of the `srand` procedure, its parameter will be set as a new seed value. So it is enough to call the srand procedure only once at the beginning of the program and then call the rand function as many times as it is necessary. It worth to initialize the random seed to a value which changes in case of any execution. In this case, all the runs provide different random sequence. That is why the recent system time or the process ID (PID) is usually used. (E.g.: `srand(time(NULL));`)
Up to the present, we generated random integers within the [`0 ; RAND_MAX`] closed interval with equal probability. Everything else is available through mathematical transformations. Here are some general situations including but not limited to all the possibilities:
Integer random number within [A ; B] closed interval with equal probability: `int x = rand()%(B-A+1)+A;`
Real random number within [A ; B] closed interval with equal probability: `double y = (B-A)*(double)rand()/RAND_MAX+A;`
Value of `X` with probability P (0%≤P≤100%), and the value of `Y` with probability 1-P: `float Q = ((double)rand()/RAND_MAX <= P) ? X : Y;`

Dynamic memory management

The memory space of an array is automatically allocated when its program unit is activated and it is deallocated when we leave the program unit. Between these two time moments (in runtime) the size of the allocated area cannot be changed. The size of the array can be defined by only a constant expression according to the ANSI standard. Since all the local variables, so the arrays also stored in the stack, which size is limited. Thus we cannot allocate a large size array even if there enough RAM. Arrays are used for continuously represented data structures.
The above facts result in limitation for the programmer, that is why the C provides library functions for programmer controlled memory management. These are defined in `stdlib.h` (or in `malloc.h`). The allocation takes place not in the stack but in the head which is not so limited.

One of the more important tools is the `malloc` function. It needs an integer parameter, which tells the size of the contiguous memory space in bytes to be allocated in the RAM. Its return value is a pointer to the beginning of the allocated area (or `NULL` in case of error). The type of the return value is `void*`, so it should be converted to a required pointer type. If we need space for 100 integers, but we are not sure the size of a given type, we can use the `sizeof` operator in this way:

```
int* p=(int*)malloc(100*sizeof(int));
```

The allocated space is not initialized, so it contains memory waste. Since the indexing in C is a pointer arithmetic operation, the allocated space can be used as a normal array: `p[7]=-93;`

When we need initialized memory space we can use the `calloc` function, which sets '0' bits to the allocated area. (It can be interpreted as fixed-point 0 or floating-point 0.0 or an empty string.) The first parameter is the number of elements to allocate and the second one is the size of each element in bytes. In case of success the return value is a `void*` memeory address. So in a similar situation as previously but with initialization looks like this:

```
p=(int*)calloc(100,sizeof(int));
```

It can happen that we allocate given size memory but later (in runtime) we realize that it is too large or too small. We have the opportunity to change the size of these memory space by reallocation. It can be done by the `realloc` function. Its first parameter is a pointer to the previously allocated memory area and the second one is the new required size in bytes. If the original area is lagrer than the new needs, then the "unwanted" closing part of the allocated area is freed (so it will be available for later allocation). If we need more space than earlier, then there are two possible scenarios. If there are enough empty spaces beyond the allocation then the system just allocates necessary space from here and concatenates it to the earlier allocation. So the old locations remain the same, but the new ones will be uninitialized. On the other hand, if there is no enough space immediately after the end of the array, then the system looks for enough large free memory area, then copies the old content of the array to the new addresses and the new part will be uninitialized. Finally, the original array will be deallocated. After the reallocation, the function returns with the address of the new `void*` starting address of the allocation (so sometimes the address remains the same). If we need a double size area than in the above example, then we write this:

```
p=(int*)realloc(p,2*100*sizeof(int));
```

Take care of the runtime, is the size is increased maybe we need a lot of data movement (copy of old values) in the background, which takes a long time.

The allocated memory does not deallocated automatically, even if we leave the program unit. In order to avoid running out of the memory programmer must deallocate each area if it is not necessary any more. This is the goal of the `free` procedure. Its only one parameter is the starting address of the allocated area: `free(p);`

(Uninitialized) memory areas can be filled with certain values by the `memset` procedure of the `string.h` header. Its first parameter is the starting address of the preallocated area. The least significant byte of the second parameter will be copied to all the bytes of the area. The third parameter is the number of bytes to be overwritten by the second parameter byte. In this way the

```
p=(int*)calloc(100,sizeof(int));
```

instruction is equivalent to the two below instructions

```
p=(int*)malloc(100*sizeof(int));
memset(p,0,100*sizeof(int));
```

Distributed data structures are unimaginable without pointers and the above dynamic memory management. Now the storage unit is a complex type (record, structure). It generally contains one or more data fields and one or more pointers to other structures allocated in runtime. For example, the code below means a balanced binary tree of three values (where the payload of leaf elements are 0s):

```
typdef struct element{
  int data;
  struct element *left, *right;
  } TreeElement;
TreeElement *root = (TreeElement *)calloc(1,sizeof(TreeElement));
```

```
root.data  = 24;
root.left  = (FaElem*)calloc(1,sizeof(TreeElement));
root.right = (FaElem*)calloc(1,sizeof(TreeElement));
```

References:

- Brian W. Kernighan, Dennis M. Ritchie: *The C programming language*, Prentece Hall (2012)
- https://www.programiz.com/c-programming/c-dynamic-memory-allocation
- https://www.tutorialspoint.com/c_standard_library/c_function_rand.htm