Name: SATYAJIT GHANA **Roll Number**: 17ETCS002159

Laboratory 1

Title of the Laboratory Exercise: Data types, local variables and Random number generation

1. Introduction and Purpose of Experiment

   Students get familiar with the data types and local variables and random number generation. Basic concepts such as data types and local variables are part and parcel of almost all the c programs. Hence sound knowledge is most essential in this regard. Also, the random number generation essential for many applications, for ex. rolling a dice for many in gaming applications such as backgammon which requires a random number generation from 1 to 6.

2. Aim and Objectives

   Aim

   To design and develop a C programs using Data types, local variables and Random number generation to demonstrate the use and significate of the same in programming.

   Objectives

   At the end of this lab, the student will be able to

   - Use variables of the basic data types with proper declarations
   - Read and validate the input data
   - Generate random numbers for any application

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Questions

Demonstrate the use of data types, local variables and Random numbers by designing appropriate algorithms for the below problems. Tabulate the output for various inputs and verify against expected values. Analyse the efficiency of the algorithm. Describe your learning along with the limitations of overall approach if any. Suggest how these can be overcome.

i. Write a C program to illustrate random number generation. Modify the program to generate a random number between 75 to 85

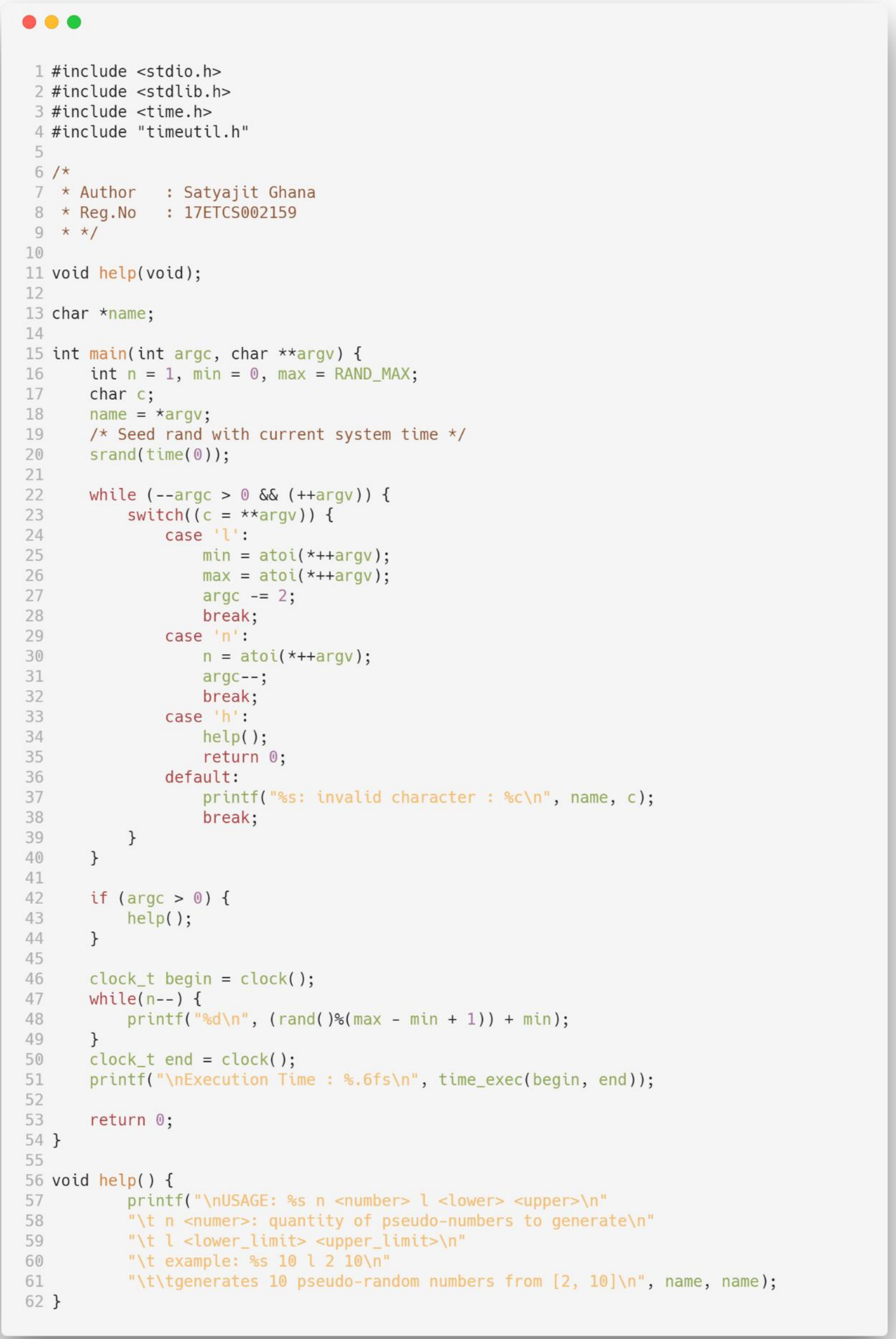
ii. Write a C program to find sum of n elements, allocate memory dynamically using malloc() and calloc() function. Modify the program include both the allocation strategies in a single program.

iii. Combine both random number generation and memory allocation in a single program to demonstrate the allocation of random number of memory blocks.

1. Calculations/Computations/Algorithms

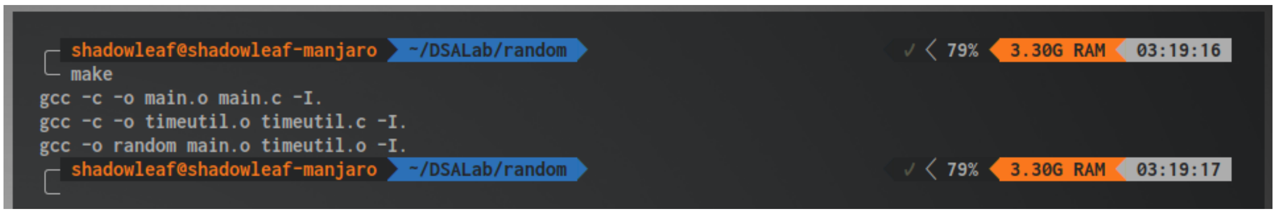
```
Step 1: Start

Step 2: Declare variables min, max, n

Step 3: seed random with current system time

Step 4: Read min, max and n from user

Step 5: Declare i <- 0

Step 6: Repeat steps until i = n

   6.1 print rand()%(max - min +1 ) + min
   6.2 i <- i + 1
Step 7: Stop
```

2. Presentation of Results



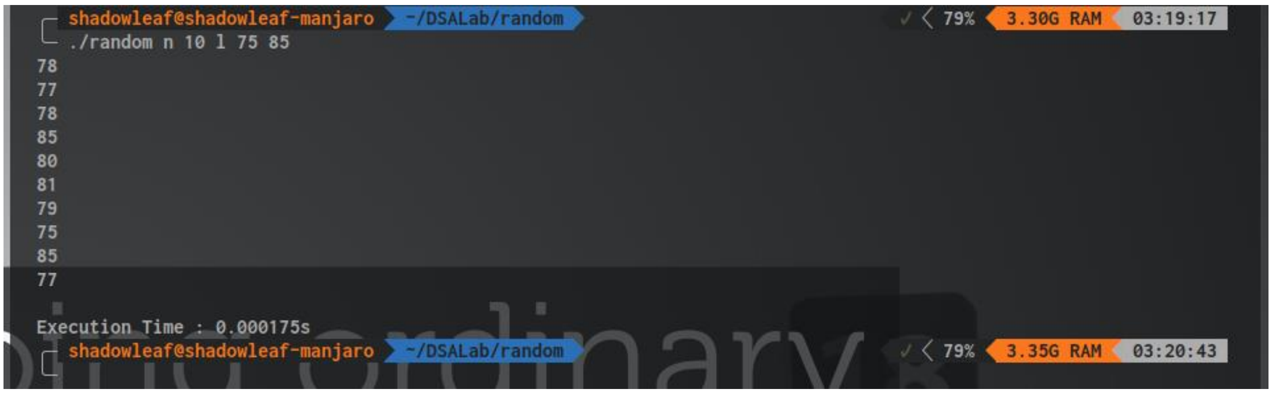
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "timeutil.h"

/*
 * Author   : Satyajit Ghana
 * Reg.No   : 17ETCS002159
 * */

void help(void);

char *name;

int main(int argc, char **argv) {
    int n = 1, min = 0, max = RAND_MAX;
    char c;
    name = *argv;
    /* Seed rand with current system time */
    srand(time(0));

    while (--argc > 0 && (++argv)) {
        switch((c = **argv)) {
            case 'l':
                min = atoi(*++argv);
                max = atoi(*++argv);
                argc -= 2;
                break;
            case 'n':
                n = atoi(*++argv);
                argc--;
                break;
            case 'h':
                help();
                return 0;
            default:
                printf("%s: invalid character : %c\n", name, c);
                break;
        }
    }

    if (argc > 0) {
        help();
    }

    clock_t begin = clock();
    while(n--) {
        printf("%d\n", (rand()%(max - min + 1)) + min);
    }
    clock_t end = clock();
    printf("\nExecution Time : %.6fs\n", time_exec(begin, end));

    return 0;
}

void help() {
        printf("\nUSAGE: %s n <number> l <lower> <upper>\n"
        "\t n <numer>: quantity of pseudo-numbers to generate\n"
        "\t l <lower_limit> <upper_limit>\n"
        "\t example: %s 10 l 2 10\n"
        "\t\tgenerates 10 pseudo-random numbers from [2, 10]\n", name, name);
}
```

Figure 1.1: Source Code



```
shadowleaf@shadowleaf-manjaro  ~/DSALab/random                    ✓ 〈 79%  3.30G RAM  03:19:16
make
gcc -c -o main.o main.c -I.
gcc -c -o timeutil.o timeutil.c -I.
gcc -o random main.o timeutil.o -I.
shadowleaf@shadowleaf-manjaro  ~/DSALab/random                    ✓ 〈 79%  3.30G RAM  03:19:17
```

Figure 1.2 : Build Successful



```
shadowleaf@shadowleaf-manjaro  ~/DSALab/random                    ✓ 〈 79%  3.30G RAM  03:19:17
./random n 10 1 75 85
78
77
78
85
80
81
79
75
85
77

Execution Time : 0.000175s
shadowleaf@shadowleaf-manjaro  ~/DSALab/random                    ✓ 〈 79%  3.35G RAM  03:20:43
```

Figure 1.3 : Run Successful

3. Analysis and Discussions

`rand()` is an library function that is defined in `rand.c` as

```
/* Return a random integer between 0 and RAND_MAX.  */
int
rand (void)
{
  return (int) __random ();
}
```

If we take a look at how `__random` works

```
/* If we are using the trivial TYPE_0 R.N.G., just do the old linear
congruential bit.  Otherwise, we do our fancy trinomial stuff, which
is the same in all the other cases due to all the global variables
that have been set up.  The basic operation is to add the number at
the rear pointer into the one at the front pointer.  Then both
pointers are advanced to the next location cyclically in the table.
The value returned is the sum generated, reduced to 31 bits by
throwing away the "least random" low bit. Note: The code takes
advantage of the fact that both the front and rear pointers can't
```

```
wrap on the same call by not testing the rear pointer if the front
one has wrapped.  Returns a 31-bit random number. */
```

As the source-code of `__random` from glibc states, it uses Linear Congruential Generator to generate the pseudo-random numbers, to generate a different set of numbers at every run event, the algorithm is seeded with a seeding value, and here we take the system time to do so.

The generator is defined by:

$$X_{n+1} = (aX_n + c) \, mod \, m$$

Where

$m, 0 < m - modulus$
$a, 0 < a < m - multiplier$
$c, 0 \leq c < m - increment$
$X_0, 0 \leq X_0 < m - \boldsymbol{the\ seed}$

4. Conclusions

A benefit of LCGs is that with appropriate choice of parameters, the period is known and long. Although not the only criterion, too short a period is a fatal flaw in a pseudorandom number generator.

While LCGs are capable of producing pseudorandom numbers which can pass formal tests for randomness, this is extremely sensitive to the choice of the parameters m and a. For example, `a = 1` and `c = 1` produces a simple modulo-m counter, which has a long period, but is obviously non–random.

C uses LCGs internally inside rand for generating these pseudorandom numbers, this can be verified by looking at the definition written in `glibc`. This cannot be used for cryptographic purposes at all, due to the fact that two systems producing generating random numbers using this algorithm may generate the same numbers arising conflicts and encryption will fail.

Nevertheless this algorithm still holds good for small applications such as in an embedded systems where memory is severely limited.

5. Comments

1. Limitations of Experiments

`rand` that used LCGs cannot be used for random number simulations in large scale operations such as a large scale Monte-Carlo Simulation. This has very high memory usage and a large amount of passes needs to be done to create close to random generator.

As the sequence of pseudo random numbers is fixed, there is a correlation among sequential random numbers. This effect can prove to be a major problem if one uses random numbers to create points in a k-dimensional space (as with Monte Carlo methods). The points will not fill up the space but will line up on n-dimensional hyperplanes. The number of hyperplanes is roughly the kth root of the constant c.

2. Limitations of Results

As the definition of `rand` states, it is a pseudo random generator and the randomness of the generated number depends on the seed value.

The random numbers generated are however limited from `0` to `RAND_MAX` and this generates only Integer values.

3. Learning happened

We learnt that `rand()` uses LCG to generate pseudo random numbers. And the generator needs to be seeded with some value to generate a new set of numbers at two different runtimes. `rand()` can only output integers and does not support generation of decimal values.

4. Recommendations

The code can be further generalized to generate random numbers, such as Floats and Doubles, and an implementation should be made to generate random characters and/or strings.