COMSM1201 : Exercises in C

Neill Campbell

Department of Computer Science, University of Bristol

Contents

```
#include <stdlib.h>
#include <stdio.h>

int main (void) {
   printf("Hello, World!\n");
   exit(0);
}
```

1. Hello World (Week 0 & 1)

The exercises in this Chapter are largely taken from the book "C by Dissection".

1.1 Lecture Notes Chapter B

Exercise 1.1 Once you've studied Chapter *B* of the lecture notes (*Hello World*), compile and run the examples given in the handout. ■

1.2 Twice the Sum

Here is part of a program that begins by asking the user to input three integers:

```
#include <stdio.h>

int main(void)
{
   int a, b, c;

   printf("Input three integers: ");
```

...

Exercise 1.2 Complete the program so that when the user executes it and types in 2, 3, and 7, this is what appears on the screen:

```
Input three integers: 2 3 7
Twice the sum of integers plus 7 is 31 !
```

■

1.3 Letter C

Execute this program so you understand the output:

```
#include <stdio.h>
```

```
#define HEIGHT 17

int main(void)
{

    int i = 0;

    printf("\n\nIIIIIII\n");
    while(i < HEIGHT){
        printf(" III\n");
        i = i + 1;
    }
    printf("IIIIIII\n\n\n");
    return 0;

}
```

Exercise 1.3 Write a similar program that prints a large letter C on the screen (it doesn't need to be curved!). ■

1.4 Lecture Notes Chapter C

Exercise 1.4 Once you've studied Chapter *C* of the lecture notes (*Grammar*), compile and run the examples given in the handout. ■

1.5 ++a++

Study the following code and write down what you think it prints.

```
int a, b = 0, c = 0;
a = ++b + ++c;
printf("%d %d %d\n", a, b, c);
a = b++ + c++;
printf("%d %d %d\n", a, b, c);
a = ++b + c++;
printf("%d %d %d\n", a, b, c);
a = b−− + −−c;
printf("%d %d %d\n", a, b, c);
```

Exercise 1.5 Then write a test program to check your answers. ■

1.6 Randomness

The function `rand()` returns values in the interval [0, RAND_MAX]. If we declare the variable `median` and initialise it to have the value `RAND_MAX/2`, then `rand()` will return a value that is sometimes larger than `median` and sometimes smaller.

Exercise 1.6 Write a program that calls `rand()`, say 500 times, inside a `for` loop, increments the variable `minus_cnt` every time `rand()` returns a value less than `median`. Each time through the `for` loop, print out the value of the difference of `plus_cnt` and `minus_cnt`. You might think that this difference should oscillate near zero. Does it ? ■

1.7 Lecture Notes Chapter D

Exercise 1.7 Once you've studied Chapter *D* of the lecture notes (*Flow Control*), compile and run the examples given in the handout. ■

1.8 find_max

Exercise 1.8 Write a program that finds the largest number entered by the user. Executing the program will produce something like:

```
How many numbers do you wish to enter ? 5
Enter 5 real numbers: 1.01 −3 2.2 7.0700 5
Maximum value: 7.07
```

■

1.9 Loving Oddness

Suppose that you detest even integers but love odd ones.

Exercise 1.9 Modify the `find_max` program so that all variables are of type `int` and that only odd integers are processed. Explain all this to the user via appropriate `printf()` statements. ■

1.10 Lecture Notes Chapter E

Exercise 1.10 Once you've studied Chapter *E* of the lecture notes (*Functions*), compile and run the examples given in the handout. ■

1.11 Hailstone

The next number in a hailstone sequence is $n/2$ if the current number n is even, or $3n+1$ if the current number is odd. If the initial number is 77, then the following sequence is produced:

```
77
232
116
58
29
88
44
22
11
34
```

Exercise 1.11 Write a program that, given a number typed by the user, prints out the sequence of *hailstone* numbers. The sequence terminates when it gets to 1. ■

1.12 Primes

A prime number can only be exactly divided by itself or 1. The number 17 is prime, but 16 is not because the numbers 2, 4 and 8 can divide it exactly. (Hint $16\%4 == 0$).

Exercise 1.12 Write a program that prints out the first n primes, where n is input by the user. The first 8 primes are:

```
2
3
5
7
11
13
7
19
```

What is the 3000th prime ? ■

1.13 Triangles

A triangle can be equilateral (all three sides have the same length), isosceles (has two equal length sides), scalene (all the sides have a different length), or right angled where if the three sides are a, b and c, and c is the longest, then : $c = \sqrt{a^2 + b^2}$

Exercise 1.13 Write a program so that you can process a number of triples of side lengths in a single run of your program using a suitable unlikely input value for the first integer in order to terminate the program. e.g. -999.

Think hard about the test data for your program to ensure that all possible cases are covered and all invalid data results in a sensible error message. Such cases can include sides of negative length, and impossible triangles (e.g. one side is longer than the sum of the other two). ■

1.14 Linear Congruent Generator

One simple way to generate ‘random’ numbers is via a Linear Congruent Generator which might look something like this:

```
int seed = 0;
/* Linear Congruential Generator */
for(i=0; i<LOOPS; i++){
   seed = (A*seed + C) % M;
   /* Seed now contains your new random number */
}
```

here, A, C and M are constants defined in the code. All of these pseudo-generators have a period of repetition that is shorter than M. For instance, with A set to 9, C set to 5 and M set to 11 the sequence of numbers is :

```
5
6
4
8
0
5
```

and so repeats after 5 numbers (period equals 5).

Exercise 1.14 Adapt the above program so that it prints the period of the LCG, where you've `#defined` the constants A, C and M and `seed` always begins at zero. For the constants described above, the program would output 5. For $A = 7$, $C = 5$ and $M = 11$ it will output 10. ■

1.15 Lecture Notes Chapter F

Exercise 1.15 Once you've studied Chapter *F* of the lecture notes (*Data Storage*), compile and run the examples given in the handout. ■

1.16 Unit Circle

In mathematics, for all real x, it is true that:

$$sin^2(x) + cos^2(x) = 1$$

i.e. $sin(x) * sin(x) + cos(x) * cos(x) = 1$.

Exercise 1.16 Write a program to demonstrate this for values of x input by the user. ■

1.17 Time Flies

Exercise 1.17 Write a program which allows the user to enter two times in 24-hour clock format, and computes the length of time between the two, e.g.:

```
Enter two times : 23:00 04:15
Difference is : 5:15
```

or,

```
Enter two times : 23:40 22:50
Difference is : 23:10
```

■

1.18 Lecture Notes Chapter G

Exercise 1.18 Once you've studied Chapter *G* of the lecture notes, compile and run the examples given in the handout. ■

1.19 Roulette

This is an chance for you to practice self-document techniques such as sensible identifier naming, commenting, `typedefs` and enumeration.

Exercise 1.19 Write a roulette program. The roulette machine will select a number between `0` and 35 at random. The player can place an odd/even bet, or a bet on a particular number. A winning odd/even bet is paid off at 2 to 1, except that all odd/even bets lose if the roulette selects `0`. If the player places a bet on a particular number, and the roulette selects it, the player is paid off a 35 to 1. ■

1.20 Lecture Notes Chapter H

Exercise 1.20 Once you've studied Chapter *H* of the lecture notes, compile and run the examples given in the handout. ■

1.21 Neill's Microwave

Last week I purchased a new, state-of-the-art microwave oven. To select how long you wish to cook food for, there are three buttons: one marked "10 minutes", one marked "1 minute" and one marked "10 seconds". To cook something for 90 seconds requires you to press the "1 minute" button, and the "10 seconds" button three times. This is four button presses in total. To cook something for 25 seconds requires three button presses; the "10 second" button needs to be pressed three times and we have to accept a minor overcooking of the food.

Exercise 1.21 Using an array to store the cooking times for the buttons, write a program that, given a required cooking time in seconds, allows the minimum number of button presses to be determined.

Example executions of the program will look like :

```
Type the time required
25
Number of button presses = 3
Type the time required
705
Number of button presses = 7
```

■

1.22 Music Playlisters

Most MP3 players have a "random" or "shuffle" feature. The problem with these is that they can sometimes be **too** random; a particular song could be played twice in succession if the new song to play is truly chosen randomly each time without taking into account what has already been played.

To solve this, many of them randomly order the entire playlist so that each song appears in a random place, but once only. The output might look something this:

```
How many songs required ? 5
4 3 5 1 2
```

or :

```
How many songs required ? 10
1 9 10 2 4 7 3 6 5 8
```

Exercise 1.22 Write a program that gets a number from the user (to represent the number of songs required) and outputs a randomised list. ■

Exercise 1.23 Rewrite Exercise 1.22 so that the program passes an array of integers (e.g. `[1,2,3,4,5,6,7,8,9,10]`) to a function which shuffles them **in-place** (no other arrays are used) and with an algorithm having complexity $O(n)$. ■

1.23 Rule 110

Rather interesting patterns can be created using *Cellular Automata*. Here we will use a simple example, one known as *Rule 110* : The idea is that in a 1D array, cells can be either on 🟧 or off ⬜ (perhaps represented by the integer values 1 and 0). A new 1D array is created in which we decide upon the state of each cell in the array based on the cell above and its two immediate neighbours.

If the three cells above are all 'on', then the cell is set to 'off' ($111 \rightarrow 0$). If the three cells above are 'on', 'on', 'off' then the new cell is set to 'on' ($110 \rightarrow 1$). The rules, in full, are:
$111 \rightarrow 0$
$110 \rightarrow 1$
$101 \rightarrow 1$
$100 \rightarrow 0$
$011 \rightarrow 1$
$010 \rightarrow 1$
$001 \rightarrow 1$
$000 \rightarrow 0$

You take a 1D array, filled with zeroes or ones, and based on these, you create a new 1D array of zeroes and ones. Any particular cell uses the three cells 'above' it to make the decision about its value. If the first line has all zeroes and a single one in the middle, then the automata evolves as:

Exercise 1.24 Write a program that outputs something similar to Figure 1.1, but using plain text, giving the user the option to start with a randomised first line, or a line with a single 'on' in the central location. Do not use $2D$ arrays for this - a couple of $1D$ arrays is sufficient. ■

Exercise 1.25 Rewrite the program above to allow other rules to be displayed - for instance $124, 30$ and 90.

WWW `http://en.wikipedia.org/wiki/Rule_110`

■

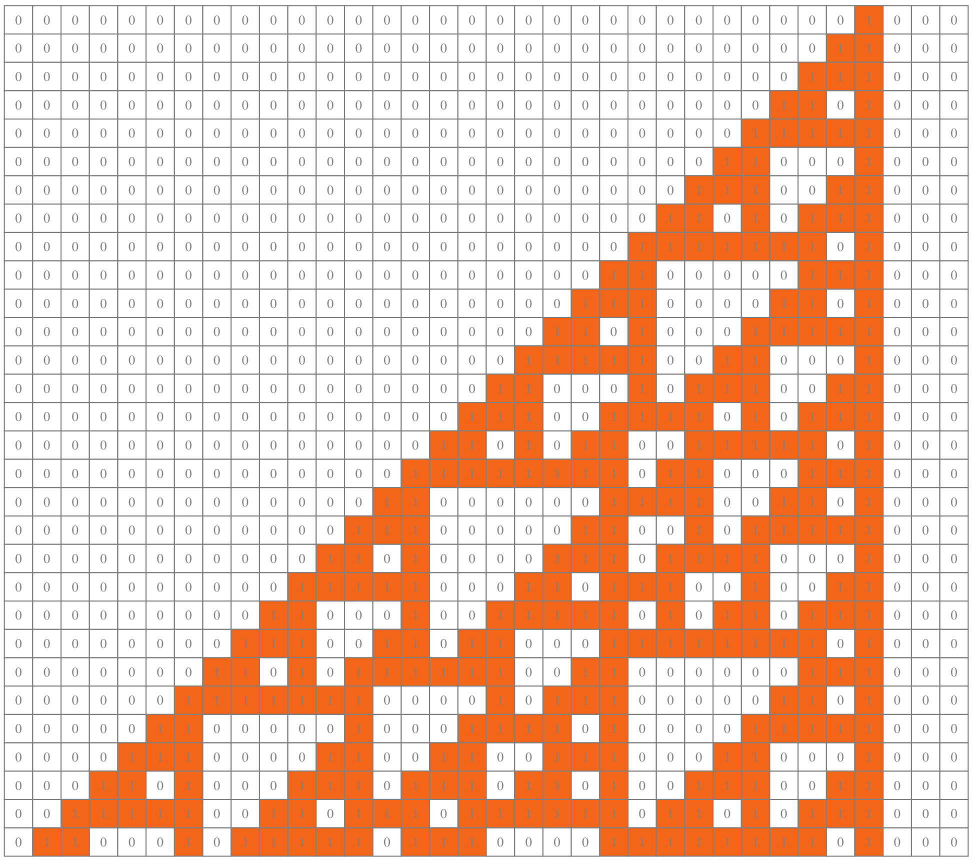

Figure 1.1: 1D cellular automaton using Rule 110. Top line shows initial state, each subsequent line is produced from the line above it. Each cell has a rule to switch it ‘on’ or ’off’ based on the state of the three cells above it in the diagram.