COMSM1201 : Exercises in C

Neill Campbell

Department of Computer Science, University of Bristol

Contents

1 Hello World (Week 0 & 1) 5

1.1 Lecture Notes Chapter B 5

1.2 Twice the Sum 5

1.3 Letter C 5

1.4 Lecture Notes Chapter C 6

1.5 ++a++ 6

1.6 Randomness 6

1.7 Lecture Notes Chapter D 7

1.8 find_max 7

1.9 Loving Oddness 7

1.10 Lecture Notes Chapter E 7

1.11 Hailstone 7

1.12 Primes 7

1.13 Triangles 8

1.14 Linear Congruent Generator 8

1.15 Lecture Notes Chapter F 9

1.16 Unit Circle 9

1.17 Time Flies 9

1.18 Lecture Notes Chapter G 9

1.19 Roulette 9

2 *1D* Arrays (Week 2) 11
2.1 Lecture Notes Chapter H 11
2.2 Neill's Microwave 11
2.3 Music Playlisters 12
2.4 Rule 110 12
2.5 Lecture Notes Chapter I 13
2.6 Vowelness 13
2.7 Planet Trium 14
2.8 Planet Bob 14
2.9 Secret Codes 15
2.10 Cash Machine (ATM) 15
2.11 Hailstone Sequence 16
2.12 Monte Carlo Π 16
2.13 Leibniz Π 16
2.14 Dirichlet Distribution 17
2.15 Triangle Numbers 17
2.16 Higher-Lower 18
2.17 Irrational numbers 18

3 Strings 19
3.1 Yahtzee 19
3.2 Palindromes 21
3.3 Int to String 22
3.4 Merging Strings 22
3.5 Roman Numerals 23
3.6 Soundex Coding 24

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

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

Lecture Notes Chapter B
Twice the Sum
Letter C
Lecture Notes Chapter C
++a++
Randomness
Lecture Notes Chapter D
find_max
Loving Oddness
Lecture Notes Chapter E
Hailstone
Primes
Triangles
Linear Congruent Generator
Lecture Notes Chapter F
Unit Circle
Time Flies
Lecture Notes Chapter G

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. ■

Lecture Notes Chapter H
Neill's Microwave
Music Playlisters
Rule 110
Lecture Notes Chapter I
Vowelness
Planet Trium
Planet Bob
Secret Codes
Cash Machine (ATM)
Hailstone Sequence
Monte Carlo Π
Leibniz Π
Dirichlet Distribution
Triangle Numbers
Higher-Lower
Irrational numbers

2. *1D* Arrays (Week 2)

Note that these exercises are in **NO** particular order - try the ones you find easy before attempting more complex ones.

2.1 Lecture Notes Chapter H

Exercise 2.1 Once you've studied Chapter H of the lecture notes, compile and run the examples given in the handout. ■

2.2 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 2.2 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
```

■

2.3 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 2.3 Write a program that gets a number from the user (to represent the number of songs required) and outputs a randomised list.

Exercise 2.4 Rewrite Exercise 2.3 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 $O(n)$.

2.4 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’ ($111 \to 0$). If the three cells above are ‘on’, ‘on’, ‘off’ then the new cell is set to ‘on’ ($110 \to 1$). The rules, in full, are:
$111 \to 0$
$110 \to 1$
$101 \to 1$
$100 \to 0$
$011 \to 1$
$010 \to 1$
$001 \to 1$
$000 \to 0$

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 2.5 Write a program that outputs something similar to Figure 2.1, 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.

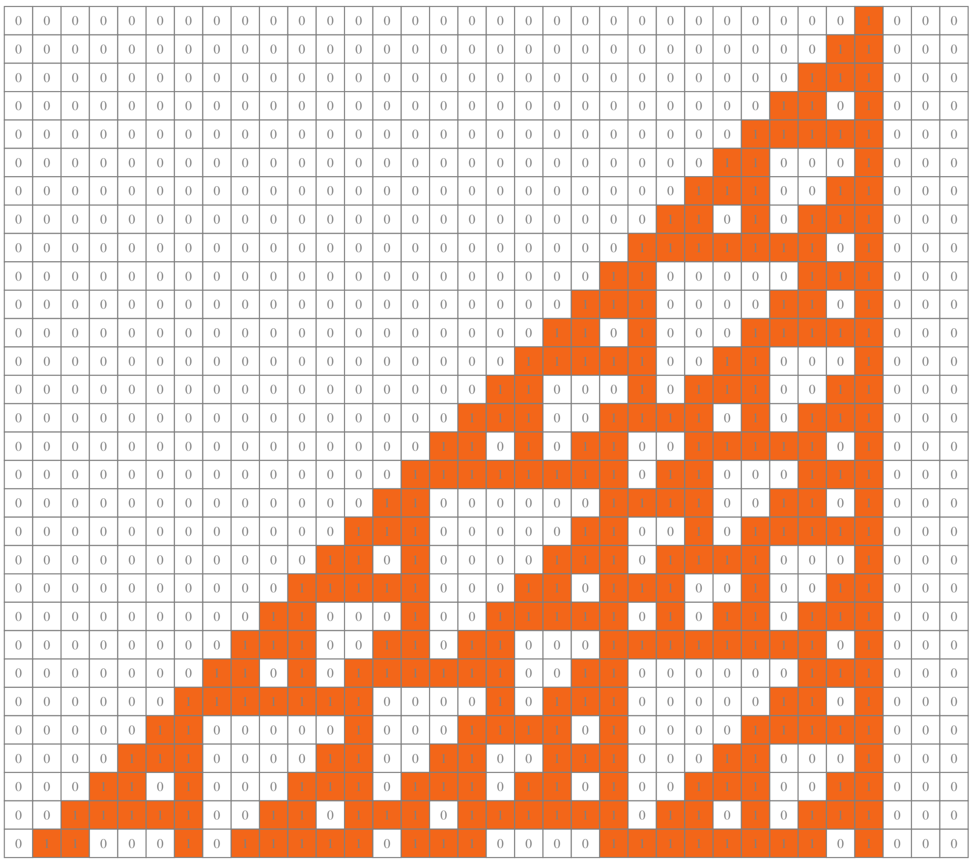


Figure 2.1: 1D cellular automaton using Rule 110. Top line shows initial state, each subsequent line is produced from the line above it. Each cell has a rule to switch it ‘on’ or ’off’ based on the state of the three cells above it in the diagram.

Exercise 2.6 Rewrite the program above to allow other rules to be displayed - for instance 124, 30 and 90.

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

■

2.5 Lecture Notes Chapter I

Exercise 2.7 Once you’ve studied Chapter *I* of the lecture notes, compile and run the examples given in the handout. ■

2.6 Vowelness

Vowels are the letters *a*, *e*, *i*, *o* and *u*.

Exercise 2.8 Write a program that reads characters from the keyboard and writes to the screen. Write all vowels as uppercase letters, and all non-vowels as lowercase letters. Do this by using writing a function `isvowel()` that tests whether or not a character is a vowel. ■

2.7 Planet Trium

On the planet Trium, everyone's name has three letters. Not only this, all the names take the form of non-vowel, vowel, non-vowel.

Exercise 2.9 Write a program that outputs all the valid names and numbers them. The first few should look like :

```
 1 bab
 2 bac
 3 bad
 4 baf
 5 bag
 6 bah
 7 baj
 8 bak
 9 bal
10 bam
11 ban
12 bap
13 baq
14 bar
15 bas
16 bat
17 bav
18 baw
19 bax
20 bay
21 baz
22 beb
23 bec
24 bed
25 bef
26 beg
```

■

2.8 Planet Bob

On the planet Bob, everyone's name has three letters. These names either take the form of consonant-vowel-consonant or else vowel-consonant-vowel. For the purposes here, vowels are the letters `{a,e,i,o,u}` and consonants are all other letters. There are two other rules :

1. The first letter and third letters of the name must always be the same.
2. The name is only 'valid' if, when you sum up the values of the three letters ($a = 1, b = 2$ etc.), the sum is prime.

The name "bob" is a valid name: it has the form consonant-vowel-consonant, the first letter and third letters are the same ('b') and the three letters sum to $19(2+15+2)$, which is prime. The name "aba' is **not** valid, since the sum of the three letters is $4(1+2+1)$ which is **not** prime.

Exercise 2.10 Write a program that outputs all the valid names and numbers them. The first few names should look like :

```
 1 aca
 2 aka
 3 aqa
 4 bab
 5 bib
 6 bob
 7 cac
 8 cec
 9 ded
10 did
11 dod
12 dud
13 ece
14 ege
15 eme
16 ese
17 faf
```

2.9 Secret Codes

Write a program that converts a stream of text typed by the user into a ‘secret’ code. This is achieved by turning every letter ’a’ into a ’z’, every letter ’b’ into a ’y’, every letter ’c’ into and ’x’ and so on.

Exercise 2.11 Write a function whose ‘top-line’ is :

int scode(**int** a)

that takes a character, and returns the secret code for this character. Note that the function **does** need to preserve the case of the letter, and that non-letters are returned unaffected.

When the program is run, the following input:

```
The Quick Brown Fox Jumps Over the Lazy Dog !
```

produces the following output :

```
Gsv Jfrxp Yildm Ulc Qfnkh Levi gsv Ozab Wlt !
```

2.10 Cash Machine (ATM)

Some cash dispensers only contain £20 notes. When a user types in how much money they’d like to be given, you need to check that the amount requested can be dispensed exactly using only £20 notes. If not, a choice of the two closest (one lower, one higher) amounts is presented.

Exercise 2.12 Write a program that inputs a number from the user and then prompts them for a better choice if it is not correct. For example :

```
How much money would you like ? 175
I can give you 160 or 180, try again.
How much money would you like ? 180
```

```
OK, dispensing ...
```

or :

```
How much money would you like ? 25
I can give you 20 or 40, try again.
How much money would you like ? 45
I can give you 40 or 60, try again.
How much money would you like ? 80
OK, dispensing ...
```

In this assessment you may assume the input from the user is “sensible” i.e. is not a negative number etc. ■

2.11 Hailstone Sequence

Hailstones sequences are ones that seem to always return to 1. The number is halved if even, and if odd then the next becomes `3*n+1`. For instance, when we start with the number 6, we get the sequence : `6, 3, 10, 5, 16, 8, 4, 2, 1` that has nine numbers in it. When we start with the number 11, the sequence is longer, containing 15 numbers : `11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1`.

Exercise 2.13 Write a program that :

- displays which initial number (less than $50,000$) creates the **longest** hailstone sequence.
- displays which initial number (less than $50,000$) leads to the **largest** number appearing in the sequence.

■

2.12 Monte Carlo Π

At :

www `http://mathfaculty.fullerton.edu/mathews/n2003/montecarlopimod.html`

a square whose sides are of length r, and a quarter-circle, whose radius is of r are drawn.

If you throw random darts at the square, then many, but not all, also hit the circle. A dart landing at position (x,y) only hits the circle if $x^2+y^2 \leq r^2$.

The area of the circle is $\frac{\pi}{4}r^2$, and the area of the square is r^2.

Therefore, a way to approximate π, is to choose random (x,y) pairs inside the square h_a, and count the h_c ones that hit the circle. Then:

$$\pi \approx \frac{4h_c}{h_a} \tag{2.1}$$

Exercise 2.14 Write a program to run this simulation, and display the improving version of the approximation to π. ■

2.13 Leibniz Π

See:

`https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80`

The Mathematical constant π can be approximated using the formula :

$$\pi = 4 - \tfrac{4}{3} + \tfrac{4}{5} - \tfrac{4}{7} + \tfrac{4}{9} - \tfrac{4}{11} + \dots$$

Notice the pattern here of alternating $+$ and $-$ signs, and the odd divisors.

Exercise 2.15 Write a program that computes π looping through smaller and smaller fractions of the series above. How many iterations does it take to get π correctly approximated to 9 digits ? ■

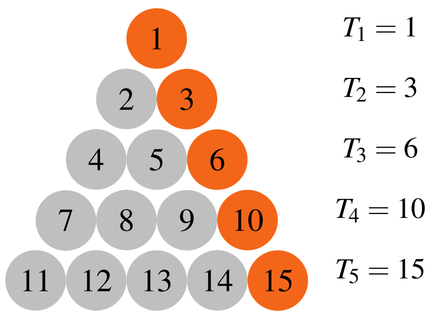
2.14 Dirichlet Distribution

When you first start thinking about prime numbers, it becomes pretty obvious that most of them end with the digit $1, 3, 7$ or 9 (they can't end with two, because they can't be even etc.)

Exercise 2.16 Write a program that computes what fraction of all primes end with a 3. You can do this by generating a large number of primes, in sequence, and keeping a running count of the number that end with 3, as compared to the total of primes generated. ■

2.15 Triangle Numbers

A Triangle number is the sum of numbers from 1 to n. The 5th Triangle number is the sum of numbers $1, 2, 3, 4, 5$, that is 15. They also relate to the number of circles you could stack up as equilateral triangles



Exercise 2.17 Write a program that prints out the sequence of Triangle numbers, using iteration, computing the next number based upon the previous.

Check these against:

`http://oeis.org/A000217`

and also by generating the nth Triangle number based on the Equation :

$$T_n = n * (n+1)/2$$

■

2.16 Higher-Lower

In the game "higher-Lower", a user has to guess a secret number chosen by another. They then repeatedly guess the number, being only told whether their guess was greater, or less than the secret one.

Exercise 2.18 Write a program that selects a random number between 1 and 1000. The user is asked to guess this number. If this guess is correct, the user is told that they have chosen the correct number and the game ends. Otherwise, they are told if their guess was too high or too low. The user has 10 goes to guess correctly before the game ends and they lose. ■

2.17 Irrational numbers

Neill's favourite number (don't ask why!) is e which has the value 2.71828182845904523536. This is an example of an irrational number - one that can only ever be *approximated* by the ratio of two integers - a bad approximation of e is 87 divided by 32, that is $\frac{87}{32}$.

Exercise 2.19 Write a program that loops through all possible denominators (that is the integer on the bottom, b in the fraction $\frac{a}{b}$) and finds which a and b pair give the best approximation to e. The output of your program might look like :

```
271801/99990 = 2.71828182818281849364
```

You need only investigate denominators $< 100,000$. #define the number being searched for. Check your code works for other famous constants such as $\pi = 3.14159265358979323846$:

```
312689/99532 = 3.14159265361893647039
```

■

Yahtzee
Palindromes
Int to String
Merging Strings
Roman Numerals
Soundex Coding

3. Strings

For some of these exercises you'll need to understand command line input to `main()` from the shell, often simply referred to as `argc/argv` in C. Please see:

www `http://www.thegeekstuff.com/2013/01/c-argc-argv/`

for more information about this.

3.1 Yahtzee

The game of Yahtzee is a game played with five dice, and you try to obtain certain 'hands'. In a similar way to poker, these hands could include a *Full House* (two dice are the same, and another three are the same), e.g.:
⚅⚅⚀⚅⚀ or ⚃⚀⚀⚀⚃
or another possible hand is *Four-of-a-Kind*, e.g.: ⚂⚂⚀⚂⚂ or ⚄⚄⚄⚁⚄ (but not ⚂⚂⚂⚂⚂ which is *Five-of-a-Kind*)

A little mathematics tells use that the probability of these two hands should be 3.85% and 1.93% respectively.

Exercise 3.1 Complete the following program which simulates, by analysing a large number of random dice rolls, the probabilty of each of these two hands. The five dice of the hand are stored in an array, and to facilitate deciding which hand you've got, a histogram is computed to say how often a ⚀ occurs in the hand, how often a ⚁ occurs and so one. A *Full-House* occurs when both a 2 and a 3 occurs in the histogram; a *Four-of-a-Kind* occurs when there is a 4 somewhere in the histogram.

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

#define MAXTHROW 6
#define NUMDICE 5
#define TESTS 10000000
```

```
enum bool {false, true};
typedef enum bool bool;

/* Fill the array d with random numbers 1..6 */
void randomthrow(int d[NUMDICE]);
/* Decide if the number n occurs anywhere in the histogram h */
bool histo_has(const int h[MAXTHROW], const int n);
/* Compute a histogram, given a dice hand */
void makehist(const int d[NUMDICE], int h[MAXTHROW]);
/* Check that the histograms h1 & h2 are the same */
bool hists_same(const int h1[MAXTHROW], const int h2[MAXTHROW]);
/* Does this hand have 2 lots of one number and 3 lots of another */
bool isfullhouse(const int d[NUMDICE]);
/* Does this hand have 4 lots of one number and 1 of another ? */
bool is4ofakind(const int d[NUMDICE]);
/* Do some testing of the functions required */
void test(void);

int main(void)
{

   int dice[NUMDICE];
   int k4 = 0;
   int fh = 0;
   int i;

   test();
   for(i=0; i<TESTS; i++){
      randomthrow(dice);
      if(isfullhouse(dice)){
         fh++;
      }
      if(is4ofakind(dice)){
         k4++;
      }
   }
   printf("FH=%.2f%% 4oK=%.2f%%\n", (double)fh*100.0/(double)TESTS,
                                     (double)k4*100.0/(double)TESTS);

   return 0;
}
```

the test function should look like:

```
void test(void)
{
   int i, j;
   int h1[MAXTHROW] = {0,0,0,0,0,5}; /* 5-of-a-kind */
   int h2[MAXTHROW] = {0,0,5,0,0,0}; /* 5-of-a-kind */
   int h3[MAXTHROW] = {0,2,0,0,3,0}; /* Full-House */
   int h4[MAXTHROW] = {2,0,3,0,0,0}; /* Full-House */
   int h5[MAXTHROW] = {3,2,0,0,0,0}; /* Histo of d1 */
   int h6[MAXTHROW] = {0,0,0,2,3,0}; /* Histo of d2 */
   int h7[MAXTHROW] = {4,1,0,0,0,0}; /* Histo of d3 */
   int h8[MAXTHROW] = {0,0,0,0,1,4}; /* Histo of d4 */
   int h9[MAXTHROW]; /* Temp */
```

```
    int d1[NUMDICE] = {1,1,1,2,2}; /* Full House */
    int d2[NUMDICE] = {5,4,5,4,5}; /* Full House */
    int d3[NUMDICE] = {2,1,1,1,1}; /* 4−of−a−kind */
    int d4[NUMDICE] = {6,6,6,6,5}; /* 4−of−a−kind */
    int d5[NUMDICE] = {6,6,6,6,5}; /* Temp */

    /* Tricky to test a random throw ... */
    for(i=0; i<100; i++){
        randomthrow(d5);
        for(j=0; j<NUMDICE; j++){
            assert((d5[j] >= 1) && (d5[j] <= MAXTHROW));
        }
        makehist(d5,h9);
        for(j=0; j<MAXTHROW; j++){
            assert((h9[j] >= 0) && (h9[j] < NUMDICE));
        }

    }
    assert(hists_same(h1,h1)==true);
    assert(hists_same(h5,h5)==true);
    assert(hists_same(h1,h2)==false);
    assert(hists_same(h4,h5)==false);
    makehist(d1,h9); assert(hists_same(h9,h5));
    makehist(d2,h9); assert(hists_same(h9,h6));
    makehist(d3,h9); assert(hists_same(h9,h7));
    makehist(d4,h9); assert(hists_same(h9,h8));
    assert(histo_has(h1,5)==true);
    assert(histo_has(h1,0)==true);
    assert(histo_has(h1,2)==false);
    assert(histo_has(h2,5)==true);
    assert(histo_has(h2,0)==true);
    assert(histo_has(h2,2)==false);
    assert(histo_has(h3,3)==true);
    assert(histo_has(h3,2)==true);
    assert(histo_has(h3,1)==false);
    assert(histo_has(h4,3)==true);
    assert(histo_has(h4,2)==true);
    assert(histo_has(h4,1)==false);
    assert(isfullhouse(d1)==true);
    assert(isfullhouse(d2)==true);
    assert(is4ofakind(d1)==false);
    assert(is4ofakind(d2)==false);
    assert(isfullhouse(d3)==false);
    assert(isfullhouse(d4)==false);
    assert(is4ofakind(d3)==true);
    assert(is4ofakind(d4)==true);
}
```

■

3.2 Palindromes

From `wikipedia.org` :

> A palindrome is a word, phrase, number or other sequence of units that has the property of reading the same in either direction (the adjustment of punctuation and

spaces between words is generally permitted).

The most familiar palindromes, in English at least, are character-by-character: the written characters read the same backwards as forwards. Palindromes may consist of a single word (such as "civic" or "level"), a phrase or sentence ("Neil, a trap! Sid is part alien!", "Was it a rat I saw?") or a longer passage of text ("Sit on a potato pan, Otis."), even a fragmented sentence ("A man, a plan, a canal: Panama!", "No Roman a moron"). Spaces, punctuation and case are usually ignored, even in terms of abbreviation ("Mr. Owl ate my metal worm").

Exercise 3.2 Write a program that prompts a user for a phrase and tells them whether it is a palindrome or not. **Do not** use any of the built-in string-handling functions (`string.h`), such as `strlen()` and `strcmp()`. However, you **may** use the character functions (`ctype.h`), such as `islower()` and `isalpha()`.

Check you program with the following palindromes :

```
"kayak"
"A man, a plan, a canal: Panama!"
"Madam, in Eden I'm Adam,"
"Level, madam, level!"
```

■

3.3 Int to String

Exercise 3.3 Write a function that converts an integer to a string, so that the following code snippet works correctly:

```
int i;
char s[256];
scanf("%d", &i);
int2string(i,s);
printf("%s\n", s);
```

The integer may be signed (i.e. be positive or negative) and you may assume it is in base-10.

Avoid using any of the built-in string-handling functions to do this (e.g. `itoa()`!) including those in `string.h`. ■

3.4 Merging Strings

Exercise 3.4 Write the function `strmerge()` which concatenetes `s1` and `s2` into `s3`, discarding any overlapping characters which are common to the end (tail) of `s1` and the beginning (head) of `s2`. Ensure that the following works correctly:

```
#include <stdlib.h>
#include <string.h>
#include <assert.h>

typedef enum bool {false, true} bool;

void strmerge(const char* s1, const char* s2, char*s3);

#define LARGESTRING 1000
```

```
int main(void)
{
    char s[LARGESTRING];

    strmerge("Hello World!", "World! & Everyone.", s);
    assert(strcmp(s, "Hello World! & Everyone.")==0);

    strmerge("The cat sat", "sat on the mat.", s);
    assert(strcmp(s, "The cat sat on the mat.")==0);

    strmerge("The cat sat on the mat", "The cat sat on the mat.", s);
    assert(strcmp(s, "The cat sat on the mat.")==0);

    strmerge("One ", "Two", s);
    assert(strcmp(s, "One Two")==0);

    strmerge("", "The cat sat on the mat.", s);
    assert(strcmp(s, "The cat sat on the mat.")==0);

    strmerge("The cat sat on the mat.", "", s);
    assert(strcmp(s, "The cat sat on the mat.")==0);

    assert(strcmp(s, "123412341234")==0);

}
```

Hint : Functions such as `strncmp()` and `strcat` might be useful. ■

3.5 Roman Numerals

Adapted from:

www `http://mathworld.wolfram.com/RomanNumerals.html`

"Roman numerals are a system of numerical notations used by the Romans. They are an additive (and subtractive) system in which letters are used to denote certain "base" numbers, and arbitrary numbers are then denoted using combinations of symbols. Unfortunately, little is known about the origin of the Roman numeral system.

The following table gives the Latin letters used in Roman numerals and the corresponding numerical values they represent :

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, the number 1732 would be denoted MDCCXXXII in Roman numerals. However, Roman numerals are not a purely additive number system. In particular, instead of using four symbols to represent a 4, 40, 9, 90, etc. (i.e., IIII, XXXX, VIIII, LXXXX, etc.), such numbers are instead denoted by preceding the symbol for 5, 50, 10, 100, etc., with a symbol indicating subtraction. For example, 4 is denoted IV, 9 as IX, 40 as XL, etc."

It turns out that every number between 1 and 3999 can be represented as a Roman numeral made up of the following one- and two-letter combinations:

I	1	IV	4
V	5	IX	9
X	10	XL	40
L	50	XC	90
C	100	CD	400
D	500	CM	900
M	1000		

Exercise 3.5 Write a program that reads a roman numeral (in the range `1 - 3999`) and outputs the corresponding valid arabic integer. Amongst others, check that `MCMXCIX` returns `1999`, `MCMLXVII` returns `1967` and that `MCDXCI` returns `1491`.

You should use the following template :

```
#include <stdio.h>

int romanToArabic( char *roman );

int main(int argc, char **argv)
{
    if( argc==2 ){
          printf("The roman numeral %s is equal to %d\n", \
          argv[1], romanToArabic(argv[1]));
    }else{
          printf("ERROR: Incorrect usage, try e.g. %s XXI\n", argv[0]);
    }
    return 0;
}
```

You need to add the function `romanToArabic()`. ■

3.6 Soundex Coding

First applied to the 1880 census, Soundex is a phonetic index, not a strictly alphabetical one. Its key feature is that it codes surnames (last names) based on the way a name sounds rather than on how it is spelled. For example, surnames that sound the same but are spelled differently, like Smith and Smyth, have the same code and are indexed together. The intent was to help researchers find a surname quickly even though it may have received different spellings. If a name like Cook, though, is spelled Koch or Faust is Phaust, a search for a different set of Soundex codes and cards based on the variation of the surname's first letter is necessary.

To use Soundex, researchers must first code the surname of the person or family in which they are interested. Every Soundex code consists of a letter and three numbers, such as B536, representing names such as Bender. The letter is always the first letter of the surname, whether it is a vowel or a consonant.

The detailed description of the algorithm may be found at :

www `http://www.highprogrammer.com/alan/numbers/soundex.html`

The first letter is simply the first letter in the word. The remaining numbers range from 1 to 6, indicating different categories of sounds created by consanants following the first letter. If the word is too short to generate 3 numbers, 0 is added as needed. If the generated code is longer than 3 numbers, the extra are thrown away.

Code	*Letters Description*
1	*B, F, P, V Labial*
2	*C, G, J, K, Q, S, X, Z Gutterals and sibilants*
3	*D, T Dental*
4	*L Long liquid*
5	*M, N Nasal*
6	*R Short liquid*
SKIP	*A, E, H, I, O, U, W, Y Vowels (and H, W, and Y) are skipped*

There are several special cases when calculating a soundex code:

- *Letters with the same soundex number that are immediately next to each other are discarded. So Pfizer becomes Pizer, Sack becomes Sac, Czar becomes Car, Collins becomes Colins, and Mroczak becomes Mrocak.*
- *If two letters with the same soundex number seperated by "H" or "W", only use the first letter. So Ashcroft is treated as Ashroft.*

Sample Soundex codes:

Word	*Soundex*
Washington	*W252*
Wu	*W000*
DeSmet	*D253*
Gutierrez	*G362*
Pfister	*P236*
Jackson	*J250*
Tymczak	*T522*
Ashcraft	*A261*

Exercise 3.6 Write a program that takes the name entered as `argv[1]` and prints the corresponding soundex code for it. ■