Chapter 8 – Data structures

1 Introducing data structures in C

All the programs we have written work with data of some predefined base type, like `int`, `double`, `char`, etc. These are known as **scalar data types**. A scalar variable holds a single value of the corresponding type. However, there are many situations where we need to group several values together in a single variable, forming a so-called **composite data type**. For example, think of a *contact list* program, where you would want to store a person's name together with their phone number and email address. In C, such groupings are known as **structured variables** and declared using the keyword `struct` (cf K&R chap. 6).

Our running example today will be that of fractions, i.e. numbers of the form $\frac{a}{b}$ where a and b are both integers, with b being non-zero, referred to as the *numerator* and *denominator*, respectively. The C syntax for declaring such a data type is illustrated below. Values grouped together in a structure are known as its **members** or **fields**.

```
struct fraction {
    int numerator;
    int denominator;
};
```

After this new type is defined, we can declare a variable `r` of this type with `struct fraction r`. To access the fields of a structure, we use the **structure member operator** "`.`" (dot). For example, we can write `printf("%d/%d", r.numerator, r.denominator);`

In C, structures behave exactly like other data types: you may use structures as function arguments (cf K&R §6.2) or return values, allocate arrays of structures (cf K&R §6.3), or even declare a variable as pointer to a structure (cf K&R §6.4), e.g. `struct fraction *pr`. To access the fields of an object pointed to by a pointer `pr`, you may either write e.g. `(*pr).numerator` or use the equivalent shorter syntax `pr->numerator`.

2 Fraction multiplication

We provide you with the `fraction.c` file (copied below) that includes:

- the type declaration of `struct fraction`
- an implementation of function `multiply()`, which for $\frac{a}{b}$ and $\frac{c}{d}$ returns a fraction $\frac{a \times c}{b \times d}$
- some code in `main()` to parse command-line arguments into fractions, then apply various operations to them.

Exercise 1 Read the code, compile it then explore its behavior with different arguments. Ask us questions about anything confusing.

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

struct fraction{
    int numerator;
    int denominator;
};

struct fraction multiply(struct fraction x, struct fraction y)
{
    struct fraction r = {x.numerator*y.numerator, x.denominator*y.denominator};
    return r;
}

double eval(struct fraction x)
{
  // IMPLEMENTATION NEEDED
  return 0;
}

int main(int argc, char ** argv)
{
    if(argc==6 && !strcmp(argv[1],"multiply"))
    {
        int n = atoi(argv[2]);
        int d = atoi(argv[3]);
        assert(d!=0);
        struct fraction f = {n, d};

        n = atoi(argv[4]);
        d = atoi(argv[5]);
        assert(d!=0);
        struct fraction g = {n, d};

        struct fraction res = multiply(f,g);

        printf("%d/%d * %d/%d = %d/%d\n",
               f.numerator, f.denominator,
               g.numerator, g.denominator,
               res.numerator, res.denominator);
    }
    else if(argc==4 &&  !strcmp(argv[1],"eval"))
    {
          // IMPLEMENTATION NEEDED
          printf("eval: implementation needed!\n");
    }
    else
    {
        printf("usage: %s eval A B\n",argv[0]);
        printf("   or: %s multiply A B C D\n",argv[0]);
        printf("   or: %s reduce A B\n",argv[0]);
        printf("   or: %s add A B C D\n",argv[0]);
        exit(1);
    }

}
```

3 Evaluation of fractions

Exercise 2 Fill in the body of function `eval()` with a proper implementation, and add the corresponding code in the `main()` function.
Note: `eval()` returns a result of type `double`, which you should `printf()` with format code *"%g"*.

Test your program with various inputs:

- `./fraction eval 1 3` should display `1/3 -> 0.333333`
- `./fraction eval 355 113` should display `355/113 -> 3.14159`

Warning ! In a struct fraction `r`, both `r.numerator` and `r.denominator` are integers. As a consequence, division `r.numerator/r.denominator` would get computed as an integer division and produce a rounded result, which is not what we want here. Instead, we should force the compiler to really work with `double`s using a **type cast** operation, e.g. `(double)r.numerator/(double)r.denominator`. For more info on type conversions, read K&R §2.7.

4 Fraction reduction

If a and b can be written $a = cd$ and $b = ce$ respectively, then fraction $\frac{a}{b}$ can be re-written $\frac{cd}{ce}$. This fraction can then be **reduced** by dividing both the numerator and denominator by c, giving $\frac{d}{e}$.
One can reduce fraction $\frac{a}{b}$ to its **lowest terms** by taking $c = gcd(a, b)$, i.e. the greatest common divisor of a and b (see below). We say that the **irreducible form** of $\frac{a}{b}$ is:

$$\frac{a/gcd(a, b)}{b/gcd(a, b)}$$

For more explanations, check `https://en.wikipedia.org/wiki/Fraction#Reduction`.

4.1 Greatest Common Divisor

The **gcd** of two integers a and b [1], is defined as the biggest integer c such that both a and b are multiples of c. An easy way to compute this number is to follow Euclid's algorithm:

> *"The method introduced by Euclid for computing greatest common divisors is based on the fact that, given two positive integers a and b such that a > b, the common divisors of a and b are the same as the common divisors of a − b and b.*
> *So, Euclid's method for computing the greatest common divisor of two positive integers consists of replacing the larger number by the difference of the numbers, and repeating this until the two numbers are equal: that is their greatest common divisor. "*

Example Let's unroll the algorithm on `a=12` and `b=8`.

- As stated above, $gcd(12, 8)$ is the same as $gcd(12 - 8, 8)$ i.e. $gcd(4, 8)$.
- We then repeat this step: $gcd(4, 8)$ is the same as $gcd(4, 8 - 4)$ i.e. $gcd(4, 4)$.
- In conclusion, we find that $gcd(12, 8) = gcd(4, 4) = 4$

Exercise 3 **(pen & paper)** With the same technique, compute $gcd(48, 18)$ and $gcd(1071, 462)$.

Exercise 4 Write a function `gcd()` that implements this algorithm.

[1] `https://en.wikipedia.org/wiki/Greatest_common_divisor`. Go look it up !

4.2 Fraction reduction

Exercise 5 Now that you have the `gcd()`, implement the `reduce()` function, and write the corresponding code in the `main()` function. For example, typing `./fraction reduce 3840 2160` should print something like `3840/2160 -> 16/9`.
Note: Your `reduce()` function should *return* the resulting fraction, not print it! Displaying the result should happen the `main()` function, like we've done so far.

5 Fraction addition

Adding two fractions $\frac{a}{b}$ and $\frac{c}{d}$ requires calculating the **least common multiple** of b and d. The lcm of two integers x and y is defined[2] as "the smallest positive integer that is divisible by both x and y." There are several methods to compute it, but one easy way is:

$$lcm(x, y) = \frac{|xy|}{gcd(x, y)}$$

If we denote $m = lcm(b, d)$, then we have:

$$\frac{a}{b} + \frac{c}{d} = \frac{a \times \frac{m}{b} + c \times \frac{m}{d}}{m}$$

Exercise 6 Implement the `lcm()` function, using your `gcd()` function from previous exercises. Test it with e.g. $lcm(21, 6) = 42$.

Exercise 7 Now implement the `add` operation in your program. Again, your `add()` function should return a `struct fraction`, and delegate the printing of the result to the `main()` function.

6 Bonus: Computing on hrs/min/sec time

Define a structure to implement hours/minutes/seconds representations of durations. In way similar to the work on fractions, propose implementations for operations on durations: unit conversions (from from hours to minutes to seconds), normalization to "human time" (e.g. convert `75s` to `1m15s`), or calculation of time differences (e.g. `1m10s - 30s = 40s`).
Fun examples: How much "human time" is 1000 seconds ? How many seconds are there in a week ?

[2] `https://en.wikipedia.org/wiki/Least_common_multiple`