Introduction to Mathematical Software 3rd Exercise Sheet

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Mathematics
PD Dr. Ulf Lorenz
Christian Brandenburg

winter term 2009/2010
16/11/2009

Important Notice

The understanding of exercises **2.1** and **2.2** on the last exercise sheet are of utmost importance for mastering the C programming language. So, if you have not finished them yet, we advise you to finish these exercises before starting exercise sheet number 3.

If you still have difficulties with or questions about the material covered in these exercises, don't hesitate to ask questions or join the office hours!

Exercise 3.1 Function Calls – Greatest Common Divisor

Modify the greatest common divisor program that you have implemented in Exercise 2.1 such that the actual computation of the gcd is done in a function.

The function should accept two integers as input parameters and return their greatest common divisor:

```
int gcd(int a, int b)
{

  // compute gcd

  return gcd;
}
```

Your main function should therefore only be responsible for accepting the user input and checking that the provided data are in the admissible range.

Remark: Remind that function declarations have to appear in your source file *before* the function is called for the first time! The definition of the function can be placed anywhere after its declaration.

Solution:

```
#include <stdio.h>

int gcd(int a, int b);

int main(void)
{

  int a, b;

  printf("Enter first number: ");
  scanf("%d", &a);
  printf("Enter second number: ");
  scanf("%d", &b);

  if (a <= 0 || b <= 0)
  {
    printf("Both numbers must be positive integers!\nAborting\n");
```

```
    return(-1);
  }

  printf("Computing the greatest common divisor of %d and %d.\n", a, b);

  a = gcd(a, b);

  printf("The greatest common divisor is %d.\n", a);

  return 0;
}


int gcd(int a, int b)
{

  int tmp;  // temporary variable
  // if b is greater than a, swap variables
  // note that the gcd algorithm also works corretly
  // if we do not swap here
  // (the first iteration of the while-loop does
  //  the swapping if b > a)
  // however, we want to learn how to swap variables here
  if (b > a)
  {
    tmp = b;
    b = a;
    a = tmp;
  }

  int r;  // temporary variable containing the remainder
  while (b > 0)
  {
    r = a % b;
    a = b;
    b = r;
  }

  return a;

}
```

Exercise 3.2 Recursion – Tower of Hanoi

In the towers of Hanoi puzzle there are three pegs where on the leftmost one there are n discs of decreasing radius (from bottom to top). The problem is to move all discs to the rightmost peg, subject to the following rules:

- Only one disc can be moved at a time, which consequently has to be the topmost disc on its peg.
- A disc can only be placed on an empty peg or on top of a disc that is larger.

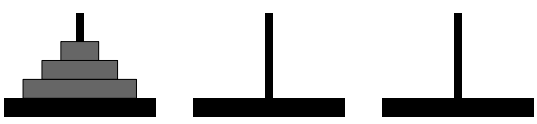

Figure 1: The Towers of Hanoi with $n = 3$

While it is very hard to write an iterative algorithm that solves this problem for n arbitrary, the recursive solution is rather simple and consists of three steps:

For moving n discs from peg A to peg C via peg B, we

- move $n-1$ discs recursively from A to B
- move one disc from A to C
- move the remaining $n-1$ discs from B to C (via A)

In the base case $n = 0$, we simply do nothing.

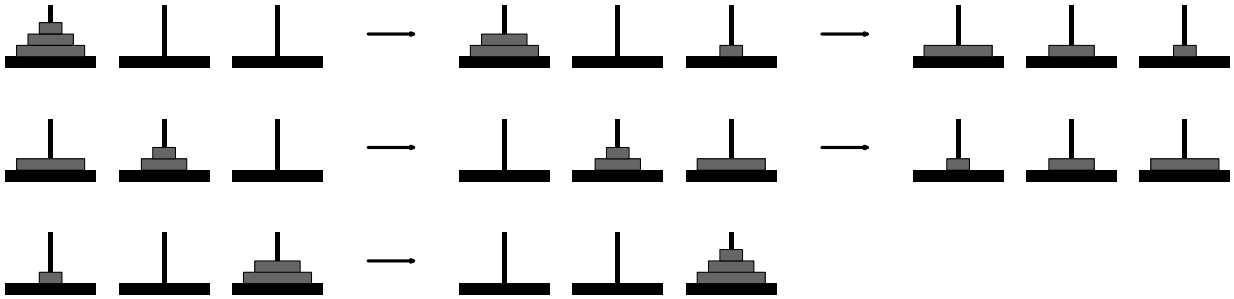

Figure 2: Solution to the Towers of Hanoi with $n = 3$

a) Implement the recursive algorithm that solves the *Towers of Hanoi* puzzle. Your recursive function should have the declaration

```
void hanoi(int n, char from, char via, char to);
```

For moving a disc from one peg to another, just write a statement of the form

```
printf("%c -> %c\n", from, to);
```

b) Validate your program for the cases $n = 3$ and $n = 4$ with pen and paper.

Solution:

```
#include <stdio.h>


void hanoi(int n, char from, char via, char to)
{

  if (n > 0)
  {
    hanoi(n-1, from, to, via);
    printf("%c -> %c\n", from, to);
    hanoi(n-1, via, from, to);
  }

}


int main(void)
{

  int n = 4;
  char A = 'A';
  char B = 'B';
```

```
  char C = 'C';

  hanoi(n, A, B, C);

  return 0;
}
```

Exercise 3.3 C++ Classes – Rational Numbers

In this exercise, we create a class `Rational` that implements rational numbers, similar to the way `BiggerInt` was defined in the lecture.

a) Implement the class `Rational`. Numerator and denominator should be stored as variables of type `int` and `unsigned int`, resp.
Remark: Remember that you have to use the g++ comiler: `g++ rational.cc -o rational`

To view your data type, add a class member function like

```
void Rational::print()
{
  printf("%d/%d\n", numerator, denominator);
  return;
}
```

b) Provide the operations `add`, `sub`, `mul`, and `div` as given in the lecture for `BiggerInt`.

c) Test your class with some example computations.

d) Is there an easy way to get the numerator and the denominator coprime after one of the four operations, i.e. to reduce the fraction?

Solution:

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


class Rational
{
  public:

    // Constructor
    Rational(int n, unsigned int d);

    Rational* add(Rational* a, Rational* b);
    Rational* sub(Rational* a, Rational* b);
    Rational* mul(Rational* a, Rational* b);
    Rational* div(Rational* a, Rational* b);

    void print();

  private:

    int numerator;

    unsigned int denominator;

    void reduce();
};
```

```
int gcd(int a, int b);


Rational::Rational(int n, unsigned int d)
{
  numerator = n;
  denominator = d;
  reduce();
}


Rational* Rational::add(Rational* a, Rational* b)
{
  int n = a->numerator * b->denominator + b->numerator * a->denominator;
  int d = a->denominator * b->denominator;

  // reduce() is called by the constructor, so we don't have to call it here
  return new Rational(n, d);
}


Rational* Rational::sub(Rational* a, Rational* b)
{
  int n = a->numerator * b->denominator - b->numerator * a->denominator;
  int d = a->denominator * b->denominator;

  // reduce() is called by the constructor, so we don't have to call it here
  return new Rational(n, d);
}


Rational* Rational::mul(Rational* a, Rational* b)
{
  int n = a->numerator * b->numerator;
  int d = a->denominator * b->denominator;

  // reduce() is called by the constructor, so we don't have to call it here
  return new Rational(n, d);
}


Rational* Rational::div(Rational* a, Rational* b)
{
  int n = a->numerator * b->denominator;
  int d = a->denominator * b->numerator;

  // reduce() is called by the constructor, so we don't have to call it here
  return new Rational(n, d);
}


void Rational::reduce()
{
  // use gcd to check whether n and d are coprime
  // if not, divide both by the gcd
  // if n is negative, take -n for the gcd computation
  int divisor = gcd(abs(numerator), denominator);
```

```
  numerator /= divisor;
  denominator /= divisor;
  return;
}


void Rational::print()
{
  printf("%d/%d\n", numerator, denominator);
  return;
}


int main(void)
{

  Rational* ap = new Rational(3, 6);
  printf("a = ");
  ap->print();

  Rational* bp = new Rational(2, 3);
  printf("b = ");
  bp->print();

  Rational* cp = ap->add(ap, bp);
  printf("a+b = ");
  cp->print();

  Rational* dp = ap->mul(ap, bp);
  printf("a*b = ");
  dp->print();

  return 0;
}


int gcd(int a, int b)
{

  int tmp;
  if (b > a)
  {
    tmp = b;
    b = a;
    a = tmp;
  }

  int r;
  while (b > 0)
  {
    r = a % b;
    a = b;
    b = r;
  }

  return a;
}
```