Introduction to Mathematical Software 2nd Exercise Sheet

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Mathematics
PD Dr. Ulf Lorenz
Christian Brandenburg

winter term 2009/2010
02/11/2009

Exercise 2.1 Greatest Common Divisor

In this exercise, we implement Euclid's algorithm for finding the greatest common divisor of two natural numbers that you have come across in your linear algebra class. This will be done in a series of small steps so that you will get familiar with various aspects of the C-programming language.

a) *Reading data from the command line.*
First of all, we want to read two natural numbers from the command line. Reading data from the command line is done with the `scanf` command. In order to read integers, we use the following form of the `scanf` command:

```
int i;
scanf("%d", &i);
```

The first argument tells scanf which format of characters to expect, in this case `%d` denotes integers. The second argument is the *address* of the variable which we assign the value to. Therefore, we have to place the *referencing operator* `&` in front of the variable name. You will learn more on addresses and references in **Exercise 2.2**.

Now, write a program that asks the user for two integers, assigns them to variables of type `int` and prints a statement of the following form:

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

where `a` and `b` are the names of the variables of type `int`.

Compile and run your program. What happens if you input characters that are not decimal numbers?

Solution:

```
#include <stdio.h>

int main(void)
{

  int a, b;

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


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

  return 0;
}
```

Characters that are not decimal numbers result in an undefined behavior.

b) *Conditional program flow control and logical statements.*
As you may have noticed, it is possible for the user to input negative integers. However, Euclid's algorithm is only defined for positive integers. Therefore, we have to check whether the numbers provided by the user are less than or equal to zero and in that case finish the program with a warning message.

To do this, we need two ingredients: *(conditional) control structures*(`if-else`) and *relational and logical operators*

`if-else`
In many situations, we might want to execute a part of the program code only if a certain condition is satisfied. If it is not satisfied, we might want to execute some alternative code or just continue with the rest of the program. In these situations, we use the `if-else`-statment, which looks like

```
if (expression)
{
  statement1;
}
else
{
  statement2;
}
```

If `expression` is true, `statement1` is executed, otherwise `statement2` is executed. An `expression` is defined to be true if its value is not equal to zero and false if it equals zero. Note that the `else` part is optional and can be omitted.

`Relational and Logical Operators:`
In the `if-else`-statment introduced above we often don't want to use any type of `expressions`, but so called `logical expressions`. These are expressions that evaluate to either 0 or 1, denoting false or true.

Simple logical expressions consist of a relational operator comparing two operands. Relational operators in C are

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

Note that the equality operator is given by ==, not just =, which is the assignement operator! For example, to test two interger variables `a` and `b` for equality, we use the logical expression

```
a == b
```

Logical expressions can be combined to arbitrarily complex logical expressions with the following *logical operators*

&&	and
||	or
!	not

For example, to test whether an integer `a` is greater than both the integers `b` and `c`, we can use the statement

```
(a > b) && (a > c)
```

To find out more about flow control and logical expressions, have a look at
http://publications.gbdirect.co.uk/c_book/chapter3/, in particular the sections 3.1, 3.2.1, and 3.3.

Modify your program so that it detects if at least one of the numbers is not greater than zero and in that case terminates the program with a warning message.

Solution:

```
#include <stdio.h>

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);

  return 0;
}
```

c) *Program flow control: iterations.*
In mathematical algorithms, very often a certain sequence of arithmetic operations has to be executed iteratively until a certain condition is satisfied (or no longer satisfied). In `C`, this is implemented by means of iteration control structures, called *loops*.

At the moment, we will restrict ourselves to so called `while`-loops. Other types of loops in `C` are `for`- and `do-while`-loops.

The syntax of the `while`-loop is as follows:

```
while (expression)
{
  statements;
}
```

What it does is the following: The loop is only entered if `expression` evaluates to true, otherwise the code block is skipped. If the loop is entered, `statements` is executed. Afterwards, `expression` is evaluated again, and if it still evaluates to true, `statements` is executed again and so on until `expression` evaluates to false.

More about `while`-loops can again be found at the link above, section 3.2.2.

d) *Euclid's algorithm.*
Now, it is time to implement Euclid's algorithm. There are several versions of it, either interative or recursive, which all yield the same result.

We want to implement the following iterative version, given in pseudocode:

if $a < b$ **then**
  swap a and b
end if
while $b > 0$ **do**
  `r = a % b;`
  `a = b;`
  `b = r;`
end while
a contains the gcd

Here, `r` is a temporary variable of type `int` containing the remainder of the division.

Add this algorithm to your program and test it with a few examples.

Solution:

```
#include <stdio.h>

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);


  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;
  }

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

  return 0;
}
```

Exercise 2.2 Pointers and Arrays

a) Obtain the file `pointer.c` from the Course Webpage.

b) Open the file in your text editor and try to understand what happens.

c) Compile and run the program. Is the result what you expected?

d) To learn more about pointers, see
http://www.cplusplus.com/doc/tutorial/pointers.html.

Solution:

```
/* Pointers
        Last modified date 22/01/2009
*/

#include <stdio.h>


int main(void)
{

  int       i1 = 66;      // any integer value
  int*      ip1;
  int       i2 = 20;
  int*      ip2 = &i2;    // Pointer ip2 is initialized with the address of i2
  float     f = 96.666;
  float*  fp = &f;
  double  d;
  double* dp;
  int **ipp;

  ip1 = &i1;

  printf("i1 is %d and ip1 is %d\n",i1,*ip1);
  printf("i2 is %d and ip2 is %d\n",i2,*ip2);
  printf("f is %f and *fp is %f\n",f,*fp);

  *fp = 966.66;

  printf(" After manipulation of f through fp \n");
  printf(" is f %f and *fp is %f \n",f,*fp);

  // and another pointer manipulation :

  dp  = &d;
  *dp = 96666666.666666;
  printf("d is %lf and *dp is also %lf\n",d,*dp);

  ipp = &ip1;
  *ipp = &i2;
  **ipp = 71;
  printf("i1 is %d and i2 is %d\n",i1,i2);

  return(0);
}
```

Homework 2.3 Binary Representation of Integers

Preliminary Remark: In this exercise, you are supposed to develop an understanding of how integers are represented in computers and how computations with them are performed. Most of this exercise is a tutorial, but there are also some small exercises within each section for you to check if you are familiar with the material.
You should not spend much more than half of the lab session on this exercise. If you don't get through with this exercise and still have questions, come and see us in our office hours.

Binary Representation

For computers the least logical entity is a *bit*, which can have two states – typically written as 0 and 1. Thus, in most programming languages the natural numbers are represented binary, i.e. they are written in base 2.
For example, the pattern $1101_{(2)}$ means

$$\begin{aligned} 1101_{(2)} &= 1 \cdot 2^3_{(10)} + 1 \cdot 2^2_{(10)} + 0 \cdot 2^1_{(10)} + 1 \cdot 2^0_{(10)} \\ &= 8_{(10)} + 4_{(10)} + 1_{(10)} \\ &= 13_{(10)} \end{aligned}$$

In fact, this representation works in exactly the same way as we are used to write numbers in the decimal system, where we have base 10:

$$\begin{aligned} 423_{(10)} &= 4 \cdot 10^2_{(10)} + 2 \cdot 10^1_{(10)} + 3 \cdot 10^0_{(10)} \\ &= 400_{(10)} + 20_{(10)} + 3_{(10)} \end{aligned}$$

Here, we have the letters $0, 1, 2, \ldots, 9 = 10 - 1$ and each position has 10 times the value of the position one further to the right.

In the same manner, in base 2 we have the letters 0 and 1 and each position is 2 times the value of the position one further to the right.

Conversion

As you have seen, conversion from base 2 to base 10 is straightforward; we just add the values of the positions with non-zero digits. Unfortunately, the other way round is more involved.

For converting from base 10 to base 2 there are two approaches, the naïve and the systematic one. In the naïve approach we iteratively find the largest power of 2 which is less than or equal to the number we want to convert, place a 1 at the corresponding position of the binary representation and substract this power of two from our decimal number.
For example, if we want to convert $817_{(10)}$ to the binary representation, we find that $2^9 = 512$ is the largest power of 2 less than or equal to $817_{(10)}$. Thus, we place a 1 at the 9th position from the right (note that we start counting positions with 0!) After substracting, we have $817 - 512 = 305$, we find that $2^8 = 256$, place a 1 at the 8th position and substract $305 - 256 = 49$. Continuing, we place ones at positions 5,4 and 0. Thus, the binary representation of $817_{(10)}$ is given by $1100110001_{(2)}$.

In the systematic approach, we make use of the following properties:

- if we add a zero at the end of a binary number, we multiply it by 2
- if we erase the rightmost digit, we divide by two, neglecting the remainder
- every number can be written in the form $2n$ or $2n + 1$
- we can read of the last digit of a binary number by checking if it is even or odd

Combining these properties, we arrive at an algorithm for converting numbers from base 10 (and in fact from any base) to base 2:

- decide whether the number is even or odd; if it is even, we place a 0, otherwise we append a 1 to the left of the binary representation (in the first iteration, we just place the digit).
- divide by 2 without remainder and iterate until you arrive at 0.

With our example, this algorithm goes as follows:

decimal	binary
817	
408	1
204	01
102	001
51	0001
25	10001

decimal	binary
12	110001
6	0110001
3	00110001
1	100110001
0	1100110001

Exercises:

a) Convert the following decimals to binaries: $4_{(10)}$, $5_{(10)}$, $78_{(10)}$, $127_{(10)}$

b) Convert the following binaries to decimals: $111_{(2)}$, $10101_{(2)}$, $10111_{(2)}$, $1110111_{(2)}$

Solution:

a) $4_{(10)} = 100_{(2)}$, $5_{(10)} = 101_{(2)}$, $78_{(10)} = 1001110_{(2)}$, $127_{(10)} = 1111111_{(2)}$

b) $111_{(2)} = 7_{(10)}$, $10101_{(2)} = 21_{(10)}$, $10111_{(2)} = 23_{(10)}$, $1110111_{(2)} = 119_{(10)}$

Negative Numbers

To also represent negative integers, somehow the "$-$"-sign has to be represented. In the "sign and magnitude"-approach, this is achieved by first fixing the number of bits the representation of an integer may use (e.g. 8 bits) and then using the leftmost bit for the sign. However, this approach has several disadvantages: Firstly, there exist two represenations for "zero", i.e. $+0$ and -0, and secondly, for all arithmatic operations one needs special cases to check for signs.

Two's Complement

Today, negative integers are typically represented by their "Two's Complement". The Two's Complement of a negative integer n is built by inverting all bits of the representation of $|n|$ and then adding 1 to it. To represent e.g. $-13_{(10)}$ in an 8 bits wide representation, you first represent $13_{(10)} = 00001101_{(2)}$, then invert all its bits to $11110010_{(2)}$ and then add 1 to it, thus $-13_{(10)} = 11110010_{(2)} + 1_{(2)} = 11110011_{(2)}$.
Whether a signed binary number is positive or negative can be determined by looking at the highest bit, which is 0 for nonnegative and 1 for negative numbers.

Exercises:

a) Find the 8 bits wide representations for $-27_{(10)}$ and $-78_{(10)}$.

b) Which integers may be represented at all with a

1. 8 bits wide representation?
2. 32 bits wide representation?

c) How could you build the Two's Complement using decimal calculations?

Solution:

a) $27_{(10)} = 00011011_{(2)}$, hence $-27_{(10)} = 11100100_{(2)} + 1_{(2)} = 11100101_{(2)}$
$78_{(10)} = 01001110_{(2)}$, hence $-78_{(10)} = 10110001_{(2)} + 1_{(2)} = 10110010_{(2)}$

b)
1. With unsigned integers, we can represent the numbers $0 \ldots 255$, with signed integers in Two's Complement we can represent the numbers $-128 \ldots 127$.
2. With unsigned integers, we can represent the numbers $0 \ldots 2^{32} - 1 = 4294967295$, with signed integers in Two's Complement we can represent the numbers $-2^{31} = -2147483648 \ldots 2^{31} - 1 = 2147483647$.

c) One way to compute the Two's Complement of a negative decimal number a using decimal calculations is to compute the number $2n - |a| - 1$ and then converting this number to binary representation.

Arithmetic

Arithmetic for the binary system works exactly as for the decimal system, which means that you can reuse all the algorithms you learned in school. In fact, the correctness of these algorithms is independent from the base.

Addition

Computing $15_{(10)} + 5_{(10)} = 20_{(10)}$ in a binary 8-bit representation:

$$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0_1 & 0_1 & 1_1 & 0_1 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

Note that, in this example, we produce a carry in each bit.

Substraction

For substracting two binary numbers, we can again use the algorithm taught in school, or we can reduce the problem to addition by negating the second argument (using the two's complement).

For computing $5_{(10)} - 15_{(10)} = -10_{(10)}$ in a binary 8-bit representation, we first negate the second argument using two's complement: $15_{(10)}$ is in binary representation given by $00001111_{(2)}$, so its two's complement is $11110000_{(2)} + 1_{(2)} = 11110001_{(2)}$. Thus, we can perform an addition:

$$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0_1 & 1 \\ \hline 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \end{array}$$

Multiplication and Division

They also work as taught in school. Check this for some small examples.

Exercises:

Do the following calculations like a computer using 8 bits:

a) $13 + 8$

b) $27 - 78$

c) $-27 - 78$

d) $-78 - 78$

e) $7 \cdot 6$

What happens in d)?

Solution:

You can verify your results by performing the computation in decimal representation and converting the result to binary representation. In d) we get an *underflow*, i.e. the resulting number is too small to be represented with 8 bits.