Introduction to Mathematical Software 1st Exercise Sheet

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Mathematics
PD Dr. Ulf Lorenz
Christian Brandenburg

winter term 2009/2010
26/10/2009

Exercise 1.1 The Linux Terminal

Terminal (or shell) windows allow you to directly work with the file system of your computer. This is done by typing commands to perform specific tasks. An overview of the most important Unix/Linux commands are summerized on the following command reference. A more detailed introduction is given by the UNIX TOOLBOX. These sources should offer you most of what you need to know about Linux commands.

Now, get familiar with the Linux terminal by opening a terminal emulator (e.g. `Konsole`, which can be found in the *K*-menu under `System -> Konsole` and performing the following tasks. Perform all tasks inside the terminal window without using other tools like browsers etc!

a) List the contents of your `home` directory (the directory in which all your personal data is stored). If you are not actually there, change to that directory.

b) Create a subdirectory named `exercise1` .

c) Download the file `textfile`, which is located at `http://www3.mathematik.tu-darmstadt.de/fileadmin/` into your home directory (without using a browser!). Check that the file has correctly been downloaded by listing the contents of your home directory.

Solution:

```
wget http://www3.mathematik.tu-darmstadt.de/fileadmin/textfile
```

d) Find out what the command `less` is good for. (Hint: What does the command `man` do?) Look into the file `textfile` with `less`.

Solution:

`less` is a program to view text files on the command line allowing forward and backward movement.

e) Copy the file `textfile` to the directory `exercise1`.

f) Change to the directory `exercise1` and check that the file `textfile` is actually there.

g) Move the file `textfile` to the file `textfile.txt`.

h) Change back to your home directory and delete the file `textfile`.

i) Find out how much disk space you can use at most. How much disk space does your home directory and all its subdirectories use? *Hint:* What does the option `-h` do?

Solution:

The amount of disc space you are allowed to use at most can be found with the command `quota`. `du` (disc usage) shows you the disc usage of your home directory and its subdirectories. The option `-h` tells `du` to print the output in human readable form.

j) Determine the absolute path of your working directory. List the whole contents of your home directory and find out which access rights the files in your home directory have. Does the file `.bash_history` exist? Find out which subdirectories there are in your home directory. List all files in your home directory, sorted by date.

Solution

The absolute path of the current working directory can be found with the command `pwd` (present working directory). The whole contents of a directory is shown with the `-a` option of `ls`. To see the access rights, you can use the `-l` option. (To combine these options, type `ls -al`.) The first column of the output shows the access rights for the owner of the file (listed in the third column), his/her group (fourth column), and all other users (the access rights are **r**ead, **w**rite, e**x**ecute). A leading `d` in the first column shows that the entry is a **d**irectory. To sort the files by (last modification) date, use the `-t` option.

If you have further questions or problems concerning the use of Linux you can also ask your tutors in their office hours. Furhtermore, there is help available in the Mathematics department. See the webpage http://www3.mathematik.tu-darmstadt.de/index.php?id=350 for more information.

Exercise 1.2 Text Editors

The task of this exercise is to get familiar with a text editor that is suitable for programming. The editor we recommend for you to use is called *Kate*, which is part of the *KDE desktop environment* (of course, you can use any other text editor you like, but you should make sure that it at least supports *syntax highlighting*).

To start *Kate*, select `Debian -> Applications -> Editors -> Kate` in the *K*-menu.

a) Type the `Hello World` program (Example 2 on page 18 of part 1 of the Lecture Slides) into the editor window and save the program as `hello.c` in the `exercise1` folder that you have created in **G1**. After saving your program observe how different statements in your program are highlighted by color and/or bold-face. Kate recognizes from the ending `.c`, that your file contains a C-program and highlights the syntax accordingly.

b) Kate contains its own terminal window. If it is not open, open it by pressing the `Terminal` button at the bottom of the Kate window or by selecting `Window -> Tool Views -> Show Terminal` from the menu bar. Make sure that the terminal is in the correct directory by listing the contents of the directory and checking for your `hello.c` program.

c) Now, compile your program by typing `gcc hello.c -o hello` in the terminal window. If there don't appear any error messages in the terminal window, run your program by typing `./hello` in the terminal window. The text `Hello World` should appear. If there are error messages, check your program source code in the editor window to make sure you have no errors in your code.

Solution:

What the command `gcc <source file name> -o <executable name>` does is the following: `gcc` is the GNU C-Compiler for creating exacutable programs from C source code. `<source file name>` is the name of the file that contains the C source code, in this case `hello.c`. Your C source files should always end with `.c` and should not contain spaces. The option `-o <executable name>` tells `gcc` how to name the created executable file, in this case `hello`.

When executing the program, the leading `./` in front if the program name `hello` tells your terminal to execute the file `hello` in the current working directory.

d) Modify your program by replacing the space between `Hello` and `World` in turn with the characters `\n` , `\t` , `\b` , `\f` , `\\` , `\"` , and `\'` .

In each case, recompile the program. What happens?

Solution:

Combinations of the backslash \, followed by a letter or a combination of digits are called *escape sequences*. To find out about their meaning, see slide 22 of the first part of the lecture notes or, e.g., the Escape Sequences page in the Microsoft MSDN library.

Exercise 1.3 Integer Arithmetic

Type in the following program:

```
#include <stdio.h>

int main(void)
{
  short i1 = 11, i2 = 22, i3;
  i3 = i1 + i2;
  printf("i1, i2, i3 = %d  %d  %d \n", i1, i2, i3);
  return(0);
}
```

Save the program into the directory `exercise1`. You can choose a name for the program, but mind the ending `.c`.

a) Compile and run the program. What happens? Is this the result that you expected?

Solution:

The result is 33, which is what we expected.

b) Change the line `i3 = i1 + i2;` to `i3 = i1 * i2;` and recompile your program. What happens? Is this the result that you expected?

Solution:

The result is 242, which is what we expected.

c) Change the line `i3 = i1 + i2;` to `i3 = i1 / i2;` and recompile your program. What happens? Is this the result that you expected?

Solution:

The result is 0, which might be surprising in the first moment. However, this result is correct as we are performing integer arithmetic, so `i1 / i2` is `0` plus some remainder, which can be computed by `i3 = i1 % i2;` (`%` is the modulo operator in integer arithmetic in C).

d) Change the line `i3 = i1 + i2;` to `i3 = i2 / i1;` and recompile your program. What happens? Is this the result that you expected?

Solution:

The result is 2, which is what we expected.

e) Change the line `short i1 = 11, i2 = 22, i3;` to `short i1 = 11111, i2 = 22222, i3;` and repeat the steps a) to d). What happens? Is this the result that you expected?

Solution:

The addition results in an *overflow*. This means that the number that is computed is too large to be stored in a variable of type `short`. The leading digit(s) are cut off, which in this case results in a negative number. Overflows (or underflows) are a serious problem in computer arithmetic. But be warned that they may often be harder to spot than in this case!

The multiplication also results in an overflow, yielding a negative result.

Both divisions work as expected.

Exercise 1.4 Floating Point Arithmetic

Type in the following program (you can use your program of **Exercise 1.3** as a starting point):

```
#include <stdio.h>

int main(void)
{
  float x1 = 1.0e5, x2 = 3.1415926, x3;
  x3 = x1 + x2;
  printf("i1, i2, i3 = %f  %f  %f \n", x1, x2, x3);
  return(0);
}
```

Save the program into the directory `exercise1`.

a) Compile and run the program. What happens? Is this the result that you expected? How accurately is `x3` computed?

   Solution:

   The computed result is 100003.140625, which differs from the exact result 100003.1415926 by a relative error of $\approx 1 \cdot 10^{-8}$.
   Remark: The relative error is the difference of the two values divided by the correct value.

   The reason for this inaccuracy is that the data type `float`, which implements the single precision format of the *IEEE Standard for Floating-Point Arithmetic (IEEE 754)*, can only represent finitely many values. This means that values which cannot be represented correctly are rounded.

   The IEEE 754 standard guarantees that for the single precision format the relative error between a real number and its floating-point representation is at most $2^{-24} \approx 5.9604645E-8$. This value is reffered to as *machine epsilon*.

   So, the computed result is within the guaranteed accuracy, but not much better.

b) Change the line `x3 = x1 + x2;` to `x3 = x1 * x2;` and recompile your program. What happens? Is this the result that you expected? How accurately is `x3` computed?

   Solution:

   The computed result is 314159.250000, which differs from the exact result 314159.260000 by a relative error of $\approx 3 \cdot 10^{-8}$.

c) Change the line `x3 = x1 + x2;` to `x3 = x1 / x2;` and recompile your program. What happens? Is this the result that you expected? How accurately is `x3` computed?

   Solution:

   The computed result is 31830.990234, which differs from the exact result 31830.989 by a relative error of $\approx 4 \cdot 10^{-8}$.

d) Change the line `x3 = x1 + x2;` to `x3 = x2 / x1;` and recompile your program. What happens? Is this the result that you expected? How accurately is `x3` computed?

   Solution:

   The computed result is 0.00003141592605970800, which differs from the exact result 0.000031415926 by a relative error of $\approx 2 \cdot 10^{-9}$.

e) Change the line `x3 = x1 + x2;` to `x3 = x1 % x2;` and recompile your program. What happens? Is this the result that you expected? How accurately is `x3` computed? (`%` is the modulo operator in C).

Solution:

We receive a compile time error of the form:

```
floating.c: In function 'main':
floating.c:6: error: invalid operands to binary % (have 'float' and 'float')
```

This means that the binary operator `%` (which is the modulo operator for integer arithmetic) is not defined for variables of type float.

f) Change the line `float x1 = 1.0e5, x2 = 3.1415926, x3;` to `float x1 = 1.0e25, x2 = 1.0e10, x3;` and repeat the steps a) to e). What happens? Is this the result that you expected?

Solution:

First of all `x1 = 1.0e25` is not represented exactly, but as `9999999562023526247432192.0`, thus we have a relative error in the representation of $\approx 4 \cdot 10^{-8}$.

Addition yields the same value as `x1`, which means we have a relative error of $\approx -4 \cdot 10^{-8}$. So the floating point representation of `x1` is the representable number closest to the sum of `x1` and `x2`.

Multiplication yields `99999994188327561679933120182222848`, which has a relative error of $\approx 6 \cdot 10^{-8}$ from the exact solution $1 \cdot 10^{35}$. This error is larger than the machine epsilon, which means that we have an accumulation of errors here, firstly from the inexact floating point representation of `x1`, and secondly from an additional inexactness in the multiplication.

The first division yields `999999986991104.0`, differing from $1 \cdot 10^{15}$ by a relative error of $\approx 1 \cdot 10^{-8}$.

The second division yields `0.0000000000000010000000036274937255387218`, which differs from $1 \cdot 10^{-15}$ by a relative error of $\approx 4 \cdot 10^{-9}$.

Remark: In order to increase the number of digits that are printed to the screen, you can use, e.g., `printf("i1, i2, i3 = %f  %f  %40.40f \n", x1, x2, x3);`

As we have seen, floating point arithmetic is not exact in general. In larger computations, the errors can accumulate so much that the result is just plain wrong! (This can even happen with floating point representations of higher precision like `double`s.)

Some computations are particularly prone to the accumulation of errors:

- substraction of numbers that are very close to each other (in extreme cases, **all** digits of the computed result are wrong)
- addition of (relatively) very small numbers to large numbers (the small numbers are "swallowed"). For exapmle, if you start with 1.0 and add the number $1 \cdot 10^{-8}$ as many times, the result will be 1.0 instead of 2.0 when using `float`s!