C++ Introduction

1. Input/Output

1a. Write your first program which prints on the screen: `Hello World!  Hallo Welt!`

1b. Write a program which reads in a name and prints out `Hallo` *name*`!` (Where the *name* will be the same name which has gotten read in.

2. Repetition

2a. Write a program that reads in a number N and prints on the screen $N!$.

2b. Write a program that reads in 800 real numbers from the file:
~kvollmay/classes.dir/caps_s2005/unix_C++_intro.dir/vxi.data
(copy first the file into your directory). These numbers are the
velocities in x-direction from a molecular dynamics simulation. Your program should then print into a file named "stat.data" the average

$$\overline{x} = \frac{1}{800.0} \sum x_i$$

and the standard deviation

$$\sigma = \sqrt{\frac{\sum x_i^2 - \frac{1}{800.0} \left(\sum x_i\right)^2}{799.0}}$$

3. Decisions

3a. Write a program that reads in two real numbers and prints out the smaller number.

3b. Read in the 800 velocities v_i from the file vxi.data (see 2b). Print on the screen the smallest v_i and the largest v_i.

C++ Introduction

3. Decisions

3a. Write a program that reads in two real numbers and prints out the smaller number.

3b. Read in the 800 velocities v_i from the file vxi.data (see 2b). Print on the screen the smallest v_i and the largest v_i.

4. Functions

4a. Use your program from 2a or the solution:
~kvollmay/classes.dir/caps_s2005/unix_C++_intro.dir/C++2a.cc
and modify the program such that $N!$ is determined in a function. I.e. the function has as input an integer N and returns $N!$.

4b. $N!$ gives for large N very large numbers. This leads to an "overflow" (larger number than `int` can handle). Determine for which N you obtain an overflow. To do so use your program from 4a. and change it such that it prints for $N = 1, 2, \cdots, 50$ two columns: N and $N!$.

5. Mandelbrotset

In this set of programs you work on making a picture of the Mandelbrot set. I am sure that you have seen a figure of it yet (fractal). Rob showed us for his idea of a project a figure which looked very similar to what you will get.

The Mandelbrot set is defined as follows:
For some complex number c we start with $z = 0$ and evaluate subsequent z's by the iteration

$$z = z^2 + c$$

If z remains finite, even after an infinite number of iterations, then the point c is a member of the Mandelbrot set.

5a. Write $z_{\text{new}} = z_{\text{old}}^2 + c$ on paper explicitly what that means for $\text{Re}(z_{\text{new}})$ and $\text{Im}(z_{\text{new}})$ as function of $\text{Re}(z_{\text{old}})$,$\text{Im}(z_{\text{old}})$,$\text{Re}(c)$, and $\text{Im}(c)$. Write a program which initializes $z = 0$ and $c = -1.7 - 1.0i$ and prints out $\text{Re}(z_{\text{new}})$ and $\text{Im}(z_{\text{new}})$.

5b. Do the calculation of z_{new} in a function. That means your function has as input variables $\text{Re}(c)$, $\text{Im}(c)$, $\text{Re}(z_{\text{old}})$, and $\text{Im}(z_{\text{old}})$, and output variables are $\text{Re}(z_{\text{new}})$ and $\text{Im}(z_{\text{new}})$. Test your program by using the same initial values as in 5a. and by calling the program five times.

5c. Next replace in your program the for-loop by a while-loop which repeats the Mandelbrot mapping only if you have iterated fewer than 30 iterations and if $|z| \leq 2.0$. To check your program print z_{real}, z_{im}, $|z|$ within the while-loop.

5d. Now we are ready to program the Mandelbrot set. Build in two other loops: one loop over $-1.7 \leq \text{Re}(c) \leq 0.8$ in steps of 0.01 and another loop over $-1.0 \leq \text{Im}(c) \leq 1.0$ in steps of 0.01. Print $\text{Re}(c)$ and $\text{Im}(c)$ only if your while-loop reached maximum iteration i.e. z remained finite. Look at your result with

```
executable.out | xgraph -m -nl
```

C++ Introduction (Last Part: Arrays)

6. Vectors

6a. Copy
~kvollmay/classes.dir/capstone_s2005.dir/unix_C++_intro.dir/C++sample_arrays_2.cc
into your working directory. Copy this example program for example into C++6a.cc and change it such that you have two vectors `A1`$= (1, 4, 9)$ and `A2`$= (2, 3, 4)$. Print `A1` and `A2` on the screen. Determine and print the sum of the vector entries, so for `A1` the sum is $1 + 4 + 9$, and for `A2` the sum is $2 + 3 + 4$.

6b. Now do the sum and the printing of the sum of a vector in a function (instead of in the main program). The sample program
~kvollmay/classes.dir/capstone_s2005.dir/unix_C++_intro.dir/C++sample_arrays_3.cc
shows you how to pass vectors to a function (see function `printvector`).

7. Matrices

7a. Use C++sample_arrays_2.cc to assign and print the two matrices `B1`$= \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$ and `B2`$= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$. Determine and print `C = B1 + B2`.

7b. Use C++sample_arrays_3.cc as help to modify your program from 7a. such that the matrix addition is done in one function and the printing of a matrix in another. For the printing use the function printmatrix from C++sample_arrays_3.cc.

8. Velocity Distribution (only if time)

8a. Copy the file
~kvollmay/classes.dir/capstone_s2005.dir/unix_C++_intro.dir/vxi_T.data
into your working directory. The file contains 1000 velocities v_x (x-component). When you get to this point please call me. I will give a few more explanations. Determine the distribution $P(v_x)$. Use `binmin=-0.8, binsize = 0.1,` and `Nbins = 20`. Print your result for $P(v_x)$ into the file `Pofvx.data` and look at your result with
`xgraph -m Pofvx.data`.

8b. Next we assume that the velocities were drawn from a system in equilibrium at a specific temperature T. Determine T via $\langle v_x^2 \rangle = \frac{kT}{m}$. We use units for which $k = 1$ and $m = 48.0$.

8c. Let's now compare the distribution $P(v_x)$ with the expected normal distribution $G(v_x) = \frac{1.0}{\sqrt{2\pi}\sigma} \exp\left(-v_x^2/(2\sigma^2)\right)$ where $\sigma = \sqrt{\langle v_x^2 \rangle} = \sqrt{\frac{kT}{m}}$. Write into the file `Gofvx.data` the expected distribution using the same v_x values as you had used them for $P(v_x)$. Look at the comparison with `xgraph -m Gofvx.data Pofvx.data`.