In-Class Work: C++

1. Copy & Compile & Run

This problem guides you through the first steps of writing and running a program:

Log into linuxremote1 (redo the steps 1.-3. of the Linux Exercise). Get into your capstone directory. Next make a directory for C++ sample files, so
`mkdir C++samples.dir`
Get into this directory (using `cd`) and copy all C++-sample programs by typing
cp ~kvollmay/classes.dir/capstone_s2011.dir/unix_C++_intro.dir/C++sample*.cc .
Also copy the data-file
cp ~kvollmay/classes.dir/capstone_s2011.dir/unix_C++_intro.dir/birthdays.data .
Look at C++sample_inout.cc with
`cat C++sample_inout.cc` or with `nedit C++sample_inout.cc &` (or use any other editor.)
This is the text file ("source code") of a program. This textfile needs to be converted to a format which the computer will understand. To do so you "compile" the source code (text file):
`g++ -o C++sample_inout.out C++sample_inout.cc`
This creates a file C++sample_inout.out (the "executable file") which you can now use to run with
`C++sample_inout.out`
You will be asked to type in your first and last name separated by blanks, do so and hit enter and then answer the next question. Check in the source code that you understand what exactly the program was doing.

2. Input/Output

2a. Write a program that prints on screen "Hello, Hallo, Hi" To do this, first copy `C++sample_inout.cc` to another file, `C++2a.cc`, and modify the program accordingly.

2b. Write a program that reads in (from screen) the name of a fruit (e.g. banana) and its price (e.g. 0.5), and prints out a sentence which says `One` *fruitname* costs $ *cost*. For this example: `One banana costs $0.5`.

3. Repetitions & Input/Output

3a. Write a program which prints ten times "Hello" on the screen. (Yes, this is not (yet) very useful.:-)

3b. Write a program which prints again ten times "Hello", but this time into a file named "Hellofile." (Hint: Use `ofstream`) Check after running the program that the Hellofile was written correctly.

3c. Copy ~ kvollmay/classes.dir/capstone_s2011.dir/unix_C++_intro.dir/groceries.data
into your working directory. Write a program which reads in 10 groceries and their prices from the file groceries.data using fstream and prints on screen a sentence for each item as in 2b.

3d. Next we want to enable to share our programs with each other. Let's assume you had called your program for 3c "C++3c.cc." Make this file readable (accessible to be copied) for others by typing
`chmod a+r C++3c.cc`
Whenever I will ask you to make your programs available for others (e.g. at the next homework assignment) please copy your programs into your ~/share.dir and make the corresponding program readable with `chmod a+r` *filename.cc*

In-Class Work continued (if enough time)

4. Repetitions Use your program from 3c, i.e. the program reads in the 10 groceries (item and price) from the file groceries.data. Change the program such that it determines the total cost (adding up the prices of each item) and print this total cost on the screen.

5. Factorial Write a program which calculates $N! = 1 * 2 * 3 * \ldots * N$ and prints out N and $N!$ for $N = 1, 2, \ldots, 20$. When using integers what happens for $N \geq 14$?

In-Class Work: C++

4. Repetitions Use your program from 3c or the solution program
`~kvollmay/classes.dir/capstone_s2011.dir/unix_C++_intro.dir/C++3c.cc`
Change the program such that it still reads in the items and prices from the file `groceries.data` but instead of printing for each line a sentence, the program determines the total cost (adding up the prices of each item) and prints this total cost on the screen.

5. Integer Range (Decisions)

Read in an integer i in the range $0 \leq i \leq 30$. For the case of $0 \leq i < 10$ print on screen "A", if $10 \leq i < 20$ print "B" and otherwise print "C."

6. Spring Break (Decisions)

Read in a date in 2011 (as two integers) and check if it is during spring break (March 12 - March 20).

7. Traffic Model (Decisions & Repetitions

Copy the file
~kvollmay/classes.dir/capstone_s2011.dir/unix_C++_intro.dir/road.data
into your working directory. The file contains 100 integers describing a road of 100 lattice sites at a certain time. -1 means an empty site and a number ≥ 0 means a car with the corresponding speed in mi/h.

7a. How many cars are on this road?

7b.(if time) What is the average velocity

$$\langle v \rangle = \frac{1}{N} \sum_{i=1}^{N} v_i$$

where v_i is the velocity of car i and there are N cars.

7c.(if time) What is the largest velocity ?

Solutions:
~kvollmay/classes.dir/capstone_s2011.dir/unix_C++_intro.dir/C++2a.cc etc.

In-Class Work: C++

8. Intro to Arrays

8a. Define a 1d-array (vector) `A` and assign the values of `A` to be (2,1,0). Then print A on the screen: `2 1 0`.

8b. Define a 2d-array (matrix) `B` and assign the values of `B` to be

```
 3  5  8
 1  0  2
-1  3  1
```

Print `B` on the screen as matrix as shown here.

9. Game of Life

The following tasks will be the first steps for programming our first in-class model, the game of life.

9a. Write a program that defines a 5x5 array. Set all values of this array zero:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

9b. Now add lines to your program such that the values of this 5x5 array are as follows:

```
0 0 0 0 0
0 1 1 1 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 0
```

9c. Add to your program that it prints the lattice on the screen in the format as shown above.

9d. Define the length of the array, i.e. 5, as constant in the header of your program, i.e. before main. This will allow us later to change the size of our lattice easily. (Hint: `C++sample_arraysII.cc` gives you an example for this task.)

9e. (Tool for Fun) You can look at this array graphically with

```
C++9d.out > arraydata        (this prints instead of on screen into file arraydata)
 DynamicLattice -nx 5 -ny 5 -matrix < arraydata
```

or these two commands can be replaced with

```
C++9d.out | DynamicLattice -nx 5 -ny 5 -matrix
```

9f. Now let's use `DynamicLattice` to make a movie. Use your program of 9d and add to it, that after writing the matrix you change each "0" to "1" and each "1" to "0". Print the matrix on screen, but ensure that there is exactly one empty line between the old array and the updated array. Then swap again all "0" and "1"s and print again the array on screen (also separated with an empty line). Do this for 50 swaps and then `C++9f.out | DynamicLattice -nx 5 -ny 5 -matrix`

10. Contrast Filter (2d-Array) (if time)

Copy into your working directory
~ kvollmay/classes.dir/capstone_s2011.dir/unix_C++_intro.dir/pic1.data
This file contains a 10x10 matrix. Have a look at this matrix (picture) with `cat` and with `DynamicLattice`. Hidden in this picture is some pattern, which you can see if you make a "filter", i.e. if you change the picture to a picture which gives you more contrast: Write a program which reads in this 10x10 matrix and prints out into the file ("pic1contrast.data") a different 10x10 matrix which replaces every number < 0.5 with -1.0 and every number ≥ 0.5 with $+1.0$. Look at piccontrast.data with DynamicLattice.

11. Population Dynamics (see Jan.27 class notes)

One of the simplest models for population growth of one species (e.g. humans) is the so called "logistic map." You describe the population with one number n and say from year t to year $t+1$ that n changes:

$$n_{t+1} = an_t(1 - n_t) \tag{1}$$

where a is a constant and $0 \geq n \geq 1$ (that means the number of people N has been devided by the maximum number of people $n = N/N_{\max}$). If you rewrite the equation to $n_{t+1} = an_t - an_t * n_t$ you see that from year t to $t+1$ you get an_t more people (because they are born) and you get $an_t * n_t$ fewer people (because they die). Write a program which starts with $n(t=0) = 0.5$ and then you do 40 timesteps $t = 1, \ldots, 40$, so a loop which does Eq.(1) again and again. Print on the screen for each time step t and n_t (and use the `endl`).

Run your program for $a = 2.5$ (let's say the executable of your program is "logmap.out" and look at the output with
`logmap.out | xgraph -m`
Then run your program with $a = 2.7, a = 3.2, a = 3.5$ and $a = 3.7$. Describe your results.

12. Mandelbrotset (If Time: For Ryan)

In this set of programs you work on making a picture of the Mandelbrot set. (I put a link to the wikipedia webpage on our webpage.)

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.

12a. Write $z_{\text{new}} = z_{\text{old}}^2 + c$ on paper explicitly what that means for Re(z_{new}) and Im(z_{new}) as function of Re(z_{old}),Im(z_{old}),Re(c), and Im(c). Write a program which initializes $z = 0$ and $c = -1.7 - 1.0i$ and prints out Re(z_{new}) and Im(z_{new}).

12b. Do the calculation of z_{new} in a function. That means your function has as input variables Re(c), Im(c), Re(z_{old}), and Im(z_{old}), and output variables are Re(z_{new}) and Im(z_{new}). Test your program by using the same initial values as in 5a. and by calling the program five times.

12c. 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 $z_{\rm real}$, $z_{\rm im}$, $|z|$ within the while-loop.

12d. Now we are ready to program the Mandelbrot set. Build in two other loops: one loop over $-1.7 \leq \mathrm{Re}(c) \leq 0.8$ in steps of 0.01 and another loop over $-1.0 \leq \mathrm{Im}(c) \leq 1.0$ in steps of 0.01. Print $\mathrm{Re}(c)$ and $\mathrm{Im}(c)$ only if your while-loop reached maximum iteration i.e. z remained finite. Look at your result with
`C++12d.out | xgraph -m -nl`

In-Class Work: C++

13. Hello World Function

Write a program which calls a function, which writes on screen "Hello World." (Yes, you would never write a function for this task, this is just to get used to functions.)

14. Pass On Variable into Function

Write a program which reads in from screen an integer `i`, then uses a function to determine `5 + 3*i` and then prints the result on the screen (in `main`). (Warning: The function name should start with a letter, not with a number.)

15. Line Function

Write a program which reads in from screen two double variables, an intercept `intercept` and a slope `slope`. Write a function which let's you determine for any value `x` and for any values of `intercept` and `slope`, `y= intercept + slope * x`. Within `main` print on screen two columns `x` and `y` for `x=0,0.1,0.2,...,1.0`.

16. Function Which Passes Back Multiple Variables

Write a program which reads in from screen two integers `i1` and `i2`. Write a function which determines not only `i3=i1+i2` and `i4=3*i1+i2` but also changes `i1` to `5*i1`. Check your function by printing on screen `i1,i2, i3` and `i4` after the function call.

17. Pass On Array into Function

Please get me, when you get to this task. I will give some short explanations for how to work with arrays in functions. Use your program of the in-class work 9d or the solution
~ kvollmay/classes.dir/capstone_s2011.dir/unix_C++_intro.dir/C++9d.cc
Add to the program a function which determines the "trace" of a matrix, that is the sum of the diagonal elements. For this example this means `A[0][0]+A[1][1]+...+A[4][4]`. Call the function in `main` and print the result on the screen. (Hint: The sample program
~ kvollmay/classes.dir/capstone_s2011.dir/unix_C++_intro.dir/C++sample_arraysII.cc
gives you examples for arrays in functions.)

18. Change Array in Function

Now add to your program of 17. a function which changes the matrix elements: every 1 is changed to 0 and every 0 is changed to 1. Check your function by calling the function before printing the matrix.

12. Mandelbrotset (If Time: For Ryan)

In this set of programs you work on making a picture of the Mandelbrot set. (I put a link to the wikipedia webpage on our webpage.)

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.

12a. Write $z_{\rm new} = z_{\rm old}^2 + c$ on paper explicitly what that means for Re($z_{\rm new}$) and Im($z_{\rm new}$) as function of Re($z_{\rm old}$),Im($z_{\rm old}$),Re(c), and Im(c). Write a program which initializes $z = 0$ and $c = -1.7 - 1.0i$ and prints out Re($z_{\rm new}$) and Im($z_{\rm new}$).

12b. Do the calculation of $z_{\rm new}$ in a function. That means your function has as input variables Re(c), Im(c), Re($z_{\rm old}$), and Im($z_{\rm old}$), and output variables are Re($z_{\rm new}$) and Im($z_{\rm new}$). Test your program by using the same initial values as in 5a. and by calling the program five times.

12c. 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 $z_{\rm real}$, $z_{\rm im}$, $|z|$ within the while-loop.

12d. Now we are ready to program the Mandelbrot set. Build in two other loops: one loop over $-1.7 \leq$ Re(c) ≤ 0.8 in steps of 0.01 and another loop over $-1.0 \leq$ Im(c) ≤ 1.0 in steps of 0.01. Print Re(c) and Im(c) only if your while-loop reached maximum iteration i.e. z remained finite. Look at your result with
`C++12d.out | xgraph -m -nl`