Dr. Andreas Borchert
Prof. Dr. Stefan Funken
Prof. Dr. Karsten Urban
Markus Bantle, Kristina Steih

Institut für Numerische Mathematik
Universität Ulm
Sommersemester 2013

High Performance Computing – C++ Warmup

(Präsenzübung 15. April 2013)

> **NOTE:** These exercises are meant for you to refresh your C++ knowledge and/or learn some new things. Don't worry if you don't know everything – try the reference pages/ stackoverflow.com / Google / asking the Übungsleiter to obtain the missing information!

You can compile simple C++ programs consisting of one source code file, e.g. example.cpp, as follows:

```
g++ -Wall example.cpp -o example
```

This creates an executable binary example which you can run by calling

```
./example
```

Exercise 1: The very basics of I/O

a) Write a program that prints *"Hello World!"* to the command line. Do this (i) *without* and (ii) *with* the use of the line `using namespace std;`.

   - What is the difference?
   - Explain the line `#include <iostream>`.

b) Write a program that prints *"Hello ⟨name⟩!"*, where ⟨*name*⟩ stands for an arbitrary name. This name should be a string that is

   (i) specified using command line arguments. Make sure that your program prints a usage message and exits correctly if it is called with the wrong number of arguments.

   (ii) specified during runtime using the standard input stream `cin`. Make sure that you check whether reading the name was successful.

c) Write a program that prints *"Hello World!"* to a text file. The name of this file should be

   (i) specified as a command line argument. Again, remember checking the program arguments as well as the sucess of opening the text file for writing.

   (ii) specified during runtime as `testfile_<time>.txt`, where `<time>` is the return value of the function `time`.

   Keywords: ofstream, stringstream

Exercise 2:

Write a program that reads in an integer $N > 0$ and then finds the first N primes. A prime number is a number that only has two divisors: one and itself.

Exercise 3: Tic-Tac-Toe

Implement a Tic-Tac-Toe game for two players. The program should always check if the players' moves are valid and if one player has won (and then print out who that was).

a) Modify the program so that it is a one player game against the computer (with the computer making its moves randomly).

b) Modify the program so that anytime the player is about to win (aka, they have 2 of 3 `x`'s in a row, the computer will block with an `o`).

Think about a sensible design, using functions and/or class structures.

Exercise 4: The keyword `const`

a) The following snippet has bugs. Identify them and indicate how to correct them, without using a computer!

```cpp
class Point{
 private :
   int x, y;

 public :
   Point (int u, int v) : x(u),y(v){}

   int getX() { return x; }
   int getY() { return y; }

   const int* getXptr() {return &x; }

   void scale(int& a){
     x *= a;
     y *= a;
   }

};

int main (){
   const Point myPoint(5, 3);

   int s = 3;
   myPoint.scale(s);

   int* x = myPoint.getXptr();
   *x = 4;

   cout << myPoint.getX() << " " << myPoint.getY() << endl;
   return 0;
}
```

b) What are the differences between the following function declarations, respectively?

(i)

```
int getX();
```

```
int getX() const;
```

(ii)

```
const int* getXptr();
```

```
int*       getXptr();
```

```
int* const getXptr();
```

```
int const* getXptr();
```

(iii)

```
void scale(int& a);
```

```
void scale(const int& a);
```

Exercise 5: The algorithm sort and function objects

a) What's the difference between a `struct` and a `class`?

b) *Background/Explanation:*

Several C++ library functions/interfaces permit or require the user to specify a function of a given type. One important example is the STL algorithm `sort`:

```
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first, RandomAccessIterator last,
      Compare comp);
```

It sorts a C++ container using a user-specified function `comp`, which should take two elements of the container as arguments and return `true` if the first element should be sorted before the second one (and false otherwise). If no function `comp` is given, the comparison operator `<` is used.

The following example is taken from `www.cplusplus.com/reference/algorithm/sort/`:

```
// sort algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector

bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  std::vector<int> myvector (myints, myints+8);

  // using default comparison (operator <):
  std::sort (myvector.begin(), myvector.begin()+4);

  // using function as comp
  std::sort (myvector.begin()+4, myvector.end(), myfunction);

  // using object as comp
  std::sort (myvector.begin(), myvector.end(), myobject);

  return 0;
}
```

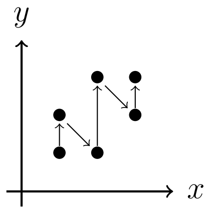
Note that as user you can either specifiy a function (here: `my_function`) or a class that has overloaded the `operator()` (here: `myclass`). Such a class is called **function object**, because you can use it like a function:

```
bool is_smaller = myobject(3,5);
```

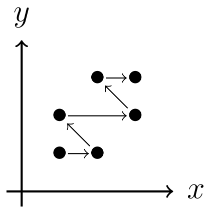
The advantage of such an object is that unlike a function, it can carry additional information (e.g. member variables) that can be used inside the `operator()`.

Task:

On the homepage you find a version of the class `Point` from Exercise 4. Of course, in 2D there are several ways to sort points. Two examples are the following:



x direction first



y direction first

Write a function object class `PointSorter` that has a member variable indicating whether points should be sorted according to their x- or their y-components and complete the missing parts in `int main()`.

Exercise 6: Bunnys

Write a program that simulates a bunny colony. Such a colony consists of bunny objects that each have

- a gender: male/female (random at creation, probability $p = 0.5$)
- a color: white/brown/black/spotted
- an age : 0-10 (years old)
- a name : randomly chosen at creation from a list of bunny names
- vampire bunny: true/false (decided at creation, probability $p = 0.02$ to be true)

Simulation rules:

- At program initialization 5 bunnies must be created and given random colors.
- Each turn afterwards the bunnies each age 1 year.
- As long as there is at least one male age 2 or older, for each female bunny in the list age 2 or older, a new bunny is created in each turn. (So if there was 1 adult male and 3 adult female bunnies, three new bunnies would be born each turn).

- New bunnies born should be the same color as their mother.
- If a bunny becomes older than 10 years, it dies.
- If a vampire bunny is born, then each turn it will change exactly one non-vampire bunny into a vampire. (If there are two vampire bunnies, two bunnies will be changed each turn and so on...)
- Vampire bunnies are excluded from regular breeding and do not count as adult bunnies.
- Vampire bunnies do not die until they reach age 50.
- If the bunny population exceeds 1000 bunnies, a food shortage occurs that kills exactly half of the bunnies (randomly chosen).

Your program should print a list of all the bunnies in the colony each turn along with all the bunnies details, sorted by age. The program should also output each turns events such as

```
    Bunny Thumper was born!
    Bunny Fufu was born!
    Vampire Bunny Darth Maul was born!
    Bunny Julius Caesar died!
```

When all the bunnies have died, the program terminates.

Modify your program so that it runs in real time, with each turn lasting 2 seconds, and a one second pause between each announement.

Again, think of a sensible design of your program.