Institut für Informatik — TUM
Scientific **C**omputing in **C**omputer **S**cience
R. Wittmann, G. Buse

WT 2013

Tutorial
(Advanced Programming)
Worksheet 8:

This worksheet shall introduce the main library around the C++ programming language, the *Standard Template Library (STL).* "Templates" therein refer to the idea of *type generic programming.* At this point this simply means for us, that we can implement algorithms (such as sorting), which will work on integer data as well as floating point numbers or even complex composite types. Templates will be discussed in more detail later in the course of the lecture. For the remainder of this worksheet, we will primarily focus on the basic concepts behind the STL and give examples on how to use some of its features.

Background Knowledge 1: Containers and Iterators

Familiarize yourself with the concept of *Containers and Iterators* implemented in the STL:

- What is the purpose of a container / iterator?
- How are iterators related to containers?
- What containers are provided by the STL, and what are their specific properties?
- What operations do iterators offer?
- Are there different kinds of iterators?

Background Knowledge 2: Timing

The STL now provides a package for timing. To use it, include the `chrono` header. The package offers platform-independent access to the system's clock. Extract the helpful method `printTiming` from the following code snippet and use it in the exercises of this worksheet.

Make sure to specify the `-std=c++0x` (GCC 4.6) or `-std=c++11` (GCC 4.7 and newer) compiler switch to compile the code.

```
#include <chrono>
#include <iostream>

typedef
  std::chrono::time_point<std::chrono::system_clock> TimeStamp;

void
printTiming(const TimeStamp& start, std::string operation)
{
  TimeStamp end = std::chrono::system_clock::now();
  double t = std::chrono::duration<double>(end-start).count();
  std::cout << "Time: " << t << " s (Operation: "
    << operation << ")" << std::endl;
}

...

// start timing
TimeStamp start = std::chrono::system_clock::now();
// TODO do sth. meaningful...
for (size_t i = 0lu; i < (1lu<<32); ++i);
// end timing
printTiming(start, "SOMETHING");
```

Assignment 1

In this exercise, we will generate random double numbers and sort them using functionality offered by the STL. First, create an STL Vector container for double values of arbitrary size as follows:

```
size_t size = 10lu; // make this arbitrary / dynamic
std::vector<double> v(size);
```

Next, use iterators to fill this vector with random values which are generated as follows:

```
#include <random>

std::default_random_engine generator;
std::uniform_real_distribution<double> distribution(0.0,1.0);
double random_val = distribution(generator);
```

Use iterators to move from one element to the next. Also, be aware that the support for random numbers is a C++11 feature, which requires the respective compiler flag (see background information for the timing methods). Then use `std::sort` to sort the array. The default invocation of `sort` takes two arguments: The first argument is the start iterator and the second argument is the end iterator. The third argument denotes a function with the following signature:

```
bool compare(const double& left, const double& right);
```

This function shall return `true` if the left item and the right item are already in their right positions (e.g. the left one is smaller than the right).

- Define an *anonymous* compare function for `std::sort`, which implements the "larger" relation.
- Use your compare function with `std::sort` to ensure a descending order on a vector.

Assignment 2

Implement your own version of *merge sort* on an `std::vector`. Time its performance and compare it against the implementation of `sort` in the STL.

- Sort numerical values (such as doubles).
- Create a vector of `Name`s by reading the names from the file "namelist" which is downloadable from Moodle. Sort the vector using both algorithms.

```
class Name {
  Name(std::string first, std::string last)
  : firstName(first), lastName(last) {
  }

  std::string getFirstName() const {
    return firstName; };
  std::string getLastName() const {
    return lastName; };

private:
  std::string firstName;
  std::string lastName;
 };
```

Hint: Reading the names from the file can be done using the idea from the following code snippet:

```
#include <iostream>
#include <fstream>

...
```

```
std::ifstream file;
file.open("namelist");

while (!file.eof()) {
  std::string firstName, lastName;
  file >> firstName;
  file >> lastName;
  Name n(firstName, lastName);
  // TODO do sth. with this name!
}
file.close();
```

- Create another vector for type `Name*`. Sort it, too, and compare to the previously obtained results. Do the timings deviate? Can you find explanations?

Homework Assignment 1: Aligning `std::vector`

In order to take advantage of the convenience of the `std::vector` class and exploit the full power of vectorization at the same time, it would be best if the array allocated internally in the class was properly aligned in memory.

- `std::vector` offers a corresponding hook. Implement a solution. Check the address of the array by examining the pointer returned by the `data()` member.
- C++ offers other, more generic ways to ensure aligned memory. How would you try to consistently achieve this in your programs?
  Hint: Check on the `new` operator.