Assignment 7

C++ Programming Course, Summer Term 2018

> "Because I'm hard, you will not like me.
> But the more you hate me, the more you will learn."
> – Gunnery Sgt. Hartman
> (gunned down by one of his recruits a few weeks later, so ... because of a private)

7-1 Fixing the Google Tech Interview

Google has an official YouTube channel for recruiting software engineers where they published this video of a staged tech interview, presumably to demonstrate what Google would expect from candidates:

https://www.youtube.com/watch?v=XKu_SEDAykw

For reasons one can only speculate about, comments have been disabled for this video.

Wipe the floor with the C++ implementation that is showcased as an acceptable solution
(at 15:29 in the video):

1. Shake your head in disbelief and disgust

```
(°°
 ( -)
 (`´)
```

2. Provide an improved variant of the `HasPairWithSum` function
   (feel free to also improve its name)
3. Provide use cases to explain in which numerous ways the original implementation is lacking and why your improvements are essential
   (add insult to injury: discuss parallelization aspects in particular)

UseCases:

- Numbers are not int
- Numbers are not stored in a Vector
- No parallelization is implemented -> large number of items take very long :(

Improvements:

- see source code

4. Become aware that you got pretty good at C++

7-2 Act Like You Got a Pair

The utility template `std::pair<A,B>` is a useful companion due to its utmost simplicity and predictability: just a struct of two values `first` and `second` with independent types.

Experiment with minimal examples to answer the following questions:

- How do comparison operators of `pair<A,B>` depend on types `A` and `B`?

  `A/B` needs to be comparable. Pair compares `lhs.first` with `rhs.first` (same for `second`).

- How can we specify comparison of `pair<A0,B0>` and `pair<A1,B1>`?

  Overload comparision function of lvalue (`A0/B0`) in std???

- What is the benefit of `std::get` over `pair.first` / `pair.second`
  (use compiler explorer)

  – Always use STL provides methods if possible

  – Get Element by value types

  – "For rvalue pair objects (2), the function returns an rvalue reference (as if forward was used)." (cplusplus.com)

  – After compile time there is no difference.

  – The only advantage auf `std::get` is that all parameter need to be known at compile time and therefore a code does not compile if one paramter is unknown.

- Why don't we just always use `std::get`?

Since there is no difference after compile time it does not matter which method we use.

The most traditional ways to represent a range are:

- a pair of iterators
- a pair of an iterator and an offset

Implement as minimal proof-of-concepts:

- Specialize `std::make_pair` for iterator and offset, that is:
  Define a variant of `std::make_pair(Iter,int)` which only matches parameter types `Iter` that provide a type definition `iterator_category`
- Define
  - `std::begin(std::pair<Iter,int> p)`
    returning `std::get<0>(p)`
  - `std::end(std::pair<Iter,int> p)`
    returning `std::advance(std::get<0>(p), std::get<1>(p))`

Use compiler explorer to check that your specializations have no overhead.

A usage example:

```
std::vector<std::string> v {
   "Monday", "Tuesday", "Wednesday",
   "Thursday", "Brainfryday", "Saturday", "Sunday" };

auto range = std::make_pair(v.begin(), 6);

for (const auto & value : range) {
   std::cout << value << '\n';
}
```

7-3 Iterate, Evaluate, Destroy

References:

- http://en.cppreference.com/w/cpp/algorithm

7-3-1 Algorithm Categories

Like container categories (sequential, associative, wrapper), STL algorithms are categorized into conceptual groups.

Two of those are *Modifying Sequence Operation* and *Non-Modifying Sequence Operation*.

- In the overview of STL algorithms, algorithms like `std::sort` and `std::partition` are not in die *Modifying* category. Why?

  `std::sort` and `std::partition` are in the `Partitioning / Sorting operations`.

- Are there algorithms that return their result as a new sequence? Why? Discuss a minimal use-case to illustrate this.

  Some algorithm have a copy and algorithm task. These one often return their results with new Iterators since the old and new sequence need to valid after the algorithm.
  Use Case:

  - There is a sequence of persons for example students.
  - Some of them are working within the eFormula team and should get a special award at the offical graduation event.
  - To acomplish this the organizer uses the students sequence and does a `std::remove_copy` to filter all students which are attending.
  - Now he has the original students for the graduation and an extra sequence for the eFormula attendencies.

7-3-2 Iterator Invalidation

- Which algorithms allow to add or remove elements from their input ranges?

- remove/remove_if
- remove_copy/remove_copy_if
- replace/replace_if
- replace_copy/replace_copy_if
- generate_n
- merge/inplace_merge
- push_heap/pop_heap
- iota

- Explain how `std::list`, `std::deque`, `std::vector` and `std::map` differ in iterator invalidation rules.
  Also discuss the differences between iterator invalidation rules for erasure (removing container elements) and insertion (adding elements).

`std::list` invalidates on deleted list elements

`std::deque` invalidates all iterators

`std::vector` no invalidation of iterators

`std::map`

7-4 Runtimes / They are a-Changin'

7-4-0 Prerequisites

Clone *Celero* from https://github.com/DigitalInBlue/Celero and experiment with the examples in the distribution.

7-4-1 Shaming Virtual

Let's assume a colleague of yours uses virtual for virtually everything.

- Implement micro-benchmarks using Celero that demonstrate the disadvantages of `virtual` (runtime polymorphism) in the most drastic way you can.
- Evaluate performance of CRTP vs. virtual in a micro-benchmark.
  You can use the CRTP iterator base classes from your solution to assignment 6 if you want.