Ranges V3 (Chapter 7)

An important part of Funtional Programming is the composition of functions. By writing small and generic functions, and reutilising them with composition, it's possible to quickly write complete programs with fewer bugs than if everything was written from scratch. In Function Programming Languages composition is used everywhere. In a language like C++, composition everywhere is not very convenient, mainly because of syntax and the semantics of passing vriables by value or reference. There's a place, however, where composition doesn't have to be pointwise: working with collections. When there's a set of operations to do on a collection, be it the whole collections or a subset of it, expressing these operations as some sort of pipeline is very intuitive, human-readable, and cheap in the number of characters typed. This idea is not new, and has been adopted even by large mainstream languages like Java (in the form of Streams). In Functional Programming Languages this concept is usually implemented in the form of lists in lazy-by-default languages, like Haskell, or lazy-lists or streams in eager-by-default languages, like Scheme.

There are many operations over collections that can be mapped to a pipeline, and many of these operations are very common. Coding them each time as a loop is tedious and most likely harder to read than simply using the abstractions. Some of these common operations include summing, multiplying, filtering, mapping, and the swiss-army knife, with which many of the other operations can be coded, the fold (also commonly known as reduce, but with slightly different semantics).

The C++ STL already has some of these operations implemented. For the most common and simplest of use cases they could be enough. Certainly beats writing a for-loop by hand. These could, however, be improved, and the Ranges library does so in two very important aspects: usage and performance.

As an example, given a vector (or a list in Haskell), filtering the elements given some predicate, applying a mapping function to each of them, and then multiplying the results could be done like so in Haskell:

```haskell
product . map mapper . filter pred $ xs
```

A handwritten for-loop in C++ could look something like this:

```cpp
for (auto x : xs) {
    if (pred(x)) {
        ret *= mapper(x);
    }
}
```

But one doesn't have to handwrite for-loops most of the time! It would be nice if it was possible to do the following:

```
xs | filter(pred) | transform(mapper) | product
```

And with the Ranges library it is! Learning about its inner workings a bit could

help understand why it's performant in comparison to the counterparts in the STL.

In the STL, the functions have as parameters two iterators, the beginning and end of the input collection, an iterator to the beginning of the output collection, and an inserter, i.e., how to insert elements into the output collection. Right away it's easy to see some points for improvement:

- Why do we have to pass the beginning and end iterators to the input collection? (I know why, but it doesn't have to be that way) It's possible to pass iterators of different collections by mistake. It's tedious to pass two arguments instead of one.
- Passing an iterator and inserter to the output collection is also tedious work; but worse, it means a new collection is always created as a result!

Ranges improves both of these aspects simply by abstracting collections and the result of operations as "ranges", which could be very simply be thought of as a pair iterators: the beginning and end.

Now it's possible to give an operation the collection itself, which will be converted into a range automatically, or the result of an operation. This last bit is very important, because it means it's possible to compose operations pointfree.

Usability is explaing. Now for performance. The pair of iterators mentioned above in a range structure are just pointers. For operations that don't mutate the original collection, there's no need to copy data around. For example, to implement `filter`, all is needed is to implement the `++` ("next") operator of the beginning iterator as searching for the next element in the input range (note *range*) that passes the predicate. If there's no element passing the predicate, the end of the resulting range has been reached.

Where change is needed there are two options: copying the input range, or mutating the original collection in-place.

All this happens *"on-demand"* (i.e., lazily), like in Haskell. If the result is not collected, nothing is done.