Goal

This is a follow-up exercice to selection_sort_file. Here we push the idea of generic programming one step further by using ideas from meta-programming (template specialization).

As before, you are given a filename as input and you need to read its content and then output a sorted version of this content. To do so, we will load the file into a container, then use my_selection_sort to sort the container. As before, the type of container that has to be used is defined within the input file. This input file has the following format:

```
[iufdcs] value type
[lvd] container type
value0
value1
value2
...
```

The first line contains a char corresponding to the type of the values:

i	integer
u	unsigned integer
f	float
d	double
c	char
s	std::string

The second line corresponds to the type of container that HAS to be used for sorting:

l	std::list
v	std::vector
d	std::deque

All other lines correspond to values (one value per line).

You program must then create a new file with the name "sorted_values.txt" containing the value type, the container type and the sorted values:

```
[iufdcs] (copied) value type
[lvd] (copied) container type
sorted value0
sorted value1
sorted value2
...
```

Extension for this exercise

Optimising read_file_to_cont for chars

Note that when reading a file containing chars (that is the type identifier is c), all lines in the file only contain two characters: the char itself, followed by a newline. We want to use this knowledge to optimize the function void read_file_to_cont(XXX& f, XXX c) such that is does not use the generic *operator*>> but takes advantage of this specific knowledge.

Modifying the sorting itself

Instead of sorting all types in ascending order, we want them to behave differently. In particular we want

- All integral types to be sorted: first all even values, then all odd ones. Within even/odd values in ascending order
- All other types in descending order

Restrictions

You may not include *algorithm* or use any other third party lib. You must use your selection_sort algorithm implemented in the last exercise. You must use generic implementations, you can stick to the proposed prototypes are design your own layout.

Challenge

Implementations faster than our (suboptimal but not bad) reference implementation get extra points.

Bonus

Another bonus is given to an implementation if it is at least 10% faster than any of the other implementations (including our reference implem)