**Programmierkurs IPK SS23**
Andreas Naumann
IWR, Universität Heidelberg

**Übung 9**
Abgabedatum —
ipk-exercises:2023-ss-6-g39ecbe3

# Übung 9

## Aufgabe 1: Wiederverwendbare Wortfrequenzen

Auf einem früheren Blatt haben Sie ein Programm geschrieben, um die Häufigkeit von Buchstaben bzw. Wörtern auszuwerten. Im folgenden werden wir dieses Programm in eine wiederverwendbare Klasse umwandeln.

Verwenden Sie hierfür folgendes Interface:

```
template<typename Map>
class LetterFrequencies {

  public:

  // Processes all letters obtained from source
  template<typename Source>
  void readData(Source& source);

  // does the statistics and prints to stdout
  void printStatistics();

};
```

`Source` soll hierbei das Konzept einer speziellen Klasse erfüllen, die einfach Buchstaben oder Wörter aus einem Eingabe-Stream einliest (z.B. der Standardeingabe oder einer Datei) und diese dann jeweils einzeln zurückliefert. Konkret funktioniert folgendes:

```
auto source = streamWordSource(std::cin);
while (true)
{
  auto data = source.next();
  // check if data valid
  if (not data.second)
  break;

  // work with actual data
  ..
  _map[data.first] += 1;
}
```

Der Rückgabewert von `Source::next()` ist je nach Variante ein `std::pair<char,bool>` oder ein `std::pair<std::string,bool>`. In der Vorlage auf der Vorlesungshomepage finde Sie eine fertige Implementierung solcher `Source` Klassen in der Datei `frequencysource.hh`.

(a) Portieren Sie Ihren Code aus der vorherigen Aufgabe in die Klasse `LetterFrequencies` und schreiben Sie ein passendes Testprogramm. Schreiben Sie ausserdem ein zweites Testprogramm, dem man auf der Kommandozeile mehrere Dateinamen mitgeben kann und das dann eine gemeinsame Statistik für den Inhalt aller Dateien ausgibt. Verwenden Sie hierfür die Source mit einem `std::ifstream` als unterliegendem Stream. Um über die Argumente der Kommandozeile zu iterieren, verwenden Sie folgenden Code:

```
for (int i = 1 ; i < argc ; ++i)
{
  std::ifstream f(argv[i]);
  auto source = streamLetterSource(f);
  ...
}
```

Testen Sie Ihre Klasse sowohl mit `std::map` als auch mit `std::unordered_map`!

**Hinweis:** Sie müssen dem `public` Interface der Klasse von weiter oben noch `private` Member-Variablen und nach Bedarf weitere Member-Funktionen hinzufügen!

(b) Kopieren Sie Ihre Klasse und erstellen eine allgemeinere Klasse `Frequencies`. Diese soll den zu analysierenden C++-Type (`char` oder `std::string`) aus der Map extrahieren (verfügbar als `typename Map::key_type`) und als `typedef` exportieren. Dieser Typ soll überall in der Klasse verwendet werden, so dass man mit der Klasse sowohl Buchstaben als auch Wörter analysieren kann. Hierfür muss allerdings das Postprocessing (Gross-Kleinschreibung, Filtern etc.) austauschbar sein. Fügen Sie hierzu Ihrer Klasse einen zweiten Template-Parameter `Filter` hinzu. `Filter` soll eine Klasse sein, die folgendes Konzept erfüllt:

```
class Filter {
  public:
  // takes the input data and returns a transformed
  // version (e.g. capitalizes all letters)
  Data transform(const Data& data);

  // decides whether the input letter or word should be
  // removed from the statistics (not added to the map)
  bool remove(const Data& data);
};
```

`Data` ist hier entweder `char` oder `std::string`, je nachdem, wofür der Filter geschrieben wurde. Schreiben Sie für beide Varianten jeweils einen passenden Filter, der den Code enthält, der in der vorherigen Aufgabe in den verschiedenen Versionen von `get_frequencies()` für das Verarbeiten der Daten verantwortlich war.

Die Klasse `Frequencies` sollte im Konstruktor eine Kopie des Filters bekommen und diese als Member-Variable speichern. In `readData()` rufen Sie dann die jeweiligen Methoden von `Filter` auf, statt direkt etwas an den Buchstaben / Wörtern zu verändern.

Testen Sie, ob Ihre Implementierung sowohl für Buchstaben als auch für Wörter funktioniert.

**Hinweis:** Sie müssen dem `public` Interface der Klasse auch hier noch `private` Member-Variablen und nach Bedarf weitere Member-Funktionen hinzufügen!

(c) Bis jetzt haben Sie immer nur die Liste der Häufigkeiten ausgegeben. Theoretisch könnte man sich hier weitere ausdenken oder zur Laufzeit entscheiden, welche Ausgaben man haben möchte.

Derartige optionale Fähigkeiten, die alle ein gemeinsames *Interface* erfüllen, werden oft mit Hilfe sogenannter *Plugins* realisiert. Plugins sind kleine Softwarebausteine, die auf wohldefinierte Weise mit dem eigentlichen Programm interagieren und diesem nach Belieben hinzugefügt werden können. Wir werden im folgenden ein sehr einfaches Plugin verwenden:

```cpp
template<typename Map>
class AnalysisPlugin {

  public:

  using Data = typename Map::key_type;

  // always add virtual destructor
  virtual ~AnalysisPlugin()
  {}

  // returns the name of the plugin
  virtual std::string name() const = 0;

  // does some statistics on the map and prints it to stdout
  virtual void printStatistics(const Map& map)
  {
    // do nothing by default
  }

};
```

Schreiben Sie zwei Plugins `PrintFrequencies` und `PrintTotalCount`, die von `AnalysisPlugin` erben und eine Liste der Buchstaben/Wörter mit Ihrer Häufigkeit bzw. die Gesamtzahl an Buchstaben/Wörtern ausgeben.

**Hinweis:** Die Plugins sind selbst Templates und müssen von der korrekten Instantiierung der Basisklasse erben:

```cpp
template<typename Map>
class PrintTotalCount
: public AnalysisPlugin<Map>
{ ... };
```

(d) Fügen Sie Ihrer `Frequencies-Klasse` folgenden Code hinzu, um mit Plugins arbeiten zu können:

```cpp
#include <memory>

template<typename Map, typename Filter>
class Frequencies
{
  public:

  // abstract Plugin base type
  using Plugin = AnalysisPlugin<Map>;

  // add a new plugin
  void addPlugin(const std::shared_ptr<Plugin> plugin)
  {
    _plugins.push_back(plugin);
  }

  private:
  std::vector<std::shared_ptr<Plugin>> _plugins;
};
```

`std::shared_ptr` ist ein sogenannter *smart pointer*. Er verhält sich wie ein normaler Pointer, aber kümmert sich automatisch um das Allokieren und Freigeben von Speicher. Um ein Plugin anzulegen bzw. aufzurufen, verwenden Sie folgenden Code:

```
1  // define type of map to be used (at the beginning of the program)
2  using WordMap = std::unordered_map<std::string,int>;
3  // create custom word filter (exercise (b) )
4  WordFilter wordFilter;
5  // create main class
6  Frequencies<WordMap,WordFilter> frequencies(wordFilter);
7  // add plugin of type PrintTotalCount<Map>
8  frequencies.addPlugin(
9  std::make_shared<PrintTotalCount<Map>>());
10
11 ...
12
13 // inside Frequencies: plugin is a pointer, so need to dereference!
14 for (auto& plugin : _plugins)
15 plugin->printStatistics();
```

Schreiben Sie die `printStatistics()`-Funktion in `Frequencies` so um, dass Sie über alle Plugins iteriert, den Namen des Plugins ausgibt und dann die Analyse-Funktion des Plugins aufruft.