ulm university universität
uulm

Institut für Numerische Mathematik

Dr. Andreas F. Borchert

14. Juni 2018
Blatt 6

Objektorientierte Programmierung mit C++ (SS 2018)

Abgabe bis zum 21. Juni 2018, 16:00 Uhr

Lernziele:

- Zusammenspiel von Funktions- und Klassen-Templates
- Anwendung der SFINAE-Technik

Aufgabe 8: Virtuelle Container

Auf der Shell-Ebene ist das *cat*-Kommando bekannt, mit dem die Inhalte mehrerer Dateien aneinandergehängt werden können. Dabei können die Dateien sehr unterschiedlich sein, aber durch eine einheitliche I/O-Schnittstelle ist es möglich, aus sehr vielen verschiedenen Quellen einen zusammenhängenden Strom von Bytes zu erzeugen.

So etwas könnte auch in C++ auf der Ebene von Containern gelegentlich hilfreich sein. Wenn wir mehrere Container haben mit dem gleichen Elementtyp, warum sollten wir dann diese nicht in einer aneinandergehängten Weise betrachten können?

Gegeben seien beispielsweise zwei Container mit dem Elementtyp **int**:

```
std::vector<int> a = {1, 2, 3};
std::list<int> b = {4, 5, 6};
```

Wenn wir ein geeignetes Funktionstemplate *concatenate* haben, könnten wir die Elemente der beiden Container in einer **for**-Schleife hintereinander durchlaufen:

```
for (int i: concatenate(a, b)) {
    std::cout << i << std::endl;
}
```

Damit das funktioniert, müsste *concatenate* ein Objekt einer Klasse erzeugen, das die Methoden *begin* und *end* anbietet mit einer dazu passenden Forward-Iterator-Klasse, die zuerst alle Objekte des ersten Containers durchläuft und dann die Objekte des zweiten Containers. Wenn die Elementtypen übereinstimmen, lässt sich das machen.

Aus Effizienzgründen sollten dabei aber keine Container mit deren vollständigen Inhalten kopiert werden in der Annahme, dass die von *concatenate* erzeugte Abbildung nicht länger zur Verfügung steht als die zugrundeliegenden Container.

Erstrebenswert ist es dabei, wenn das Resultat von *concatenate* wiederum bei *concatenate* verwendet werden kann:

```
std::set<int> c = {7, 8, 9};
for (int i: concatenate(concatenate(a, b), c)) {
    std::cout << i << std::endl;
}
```

Damit dies gelingen soll, ist zu beachten, dass das innere *concatenate*(*a*, *b*) ein temporäres Objekt erzeugt, das nach der Übergabe an den äußeren *concatenate*-Aufruf nicht mehr existiert. Da weder die vollständigen Container kopiert werden dürfen, noch Referenzen auf nicht mehr existierende temporäre Objekte sinnvoll sind, sollte konsequent mit Iteratoren gearbeitet werden. D.h. die von *concatenate* erzeugten Objekte sollten weder den Inhalt der übergebenen Container, noch Referenzen darauf haben, sondern nur Iteratoren, da sich Iteratoren-Objekte problemlos kopieren lassen.

Abrunden lässt sich die Implementierung durch die Definition eines passenden Operators (z.B. „&“):

```
for (int i: a & b & c) {
    std::cout << i << std::endl;
}
```

Überlegen Sie dabei, wie Sie unter Anwendung des SFINAE-Prinzips die Anwendung der Funktion *concatenate* und dieses Operators auf Objekte beschränken könnten, bei denen

- die Methoden *begin* und *end* zur Verfügung stehen und
- bei denen die Elementtypen übereinstimmen

Wenn Sie dann feststellen, dass Sie zweimal das gleiche umfängliche auf *std*::*enable_if* basierende Konstrukt einsetzen, lohnt es sich zu überlegen, ob dies nicht herausfaktorisiert werden kann. Dies könnte mit einem Alias-Template gelingen, dem Sie passenderweise den Namen *ConcatenationResult* geben könnten:

```
template<typename C1, typename C2>
using ConcatenationResult = typename std::enable_if<
    /* ... */
>::type;
```

Dann könnten Sie den Datentyp *ConcatenationResult*<*C1*, *C2*> bei der Funktion *concatenate* und dem Operator anstelle von *ConcatenatedContainerView*<*C1*, *C2*> einsetzen.

Alternativ steht es Ihnen auch frei, eine elegantere Lösung auf Basis der Concepts TS zu implementieren. Denken Sie dann daran, dass Sie hierzu bei *g++* die Übersetzungsoption „-fconcepts“ benötigen.

Hinweise: Es bietet sich eine Umsetzung mit zwei Template-Klassen an. Eine für das Resultat von *concatenate* (etwa *ConcatenatedContainerView*) und einen dazu passenden Iterator (etwa *ConcatenatedIterator*). Letzterer könnte mit zwei Iteratoren-Paaren erzeugt werden und

dann das Durchlaufen der beiden Iteratoren-Bereiche unterstützen. Zu realisieren wäre nur ein einfacher vorwärts laufender Iterator (*forward iterator*), der Vergleiche unterstützt. Ein Beispiel für Iteratoren aus den Übungen finden Sie in der Beispiellösung zum 5. Übungsblatt.

Sie werden an mehreren Stellen von den Template-Parametern abgeleitete Typen benötigen wie beispielsweise bei dem **operator*** der Klasse *ConcatenatedIterator*, der eine Referenz auf das Element zurückgeben sollte. Die STL unterstützt dies bei ihren Container- und Iteratorentypen immer mit entsprechenden Typdefinitionen. Wenn Sie einen Iterator-Typ *IT* haben, dann ist beispielsweise *IT*::*value_type* der referenzierte Typ und *IT*::*reference* der zugehörige Referenztyp. (Bedenken Sie bitte, dass Sie dann jeweils **typename** davor benötigen.) Alternativ kann auch **decltype** benutzt werden. (Ihre eigene Iterator-Klasse kann die gleichen Typdefinitionen anbieten, indem Sie ihn von der Template-Basis-Klasse *std*::*iterator* mit entsprechenden Parametern ableiten.)

Wie üblich kann die Lösung wieder eingereicht werden:

```
thales$ submit cpp 8 concatenate.hpp testit.cpp
```

Viel Erfolg!