STL - `std::multiset`

Markus Scheidgen

31. Mai 2002

Überblick

- das *Interface* zu `multiset`
  - ⋆ *Konzepte, Modelle* - Darstellung von Eigenschaften
  - ⋆ Überblick
  - ⋆ Besonderheiten von `multiset`
- Möglichkeiten eigene Ordnungen zu verwenden
  - ⋆ `operator<` überladen
  - ⋆ `less` überladen
  - ⋆ eigener *functor*
  - ⋆ Smartpointer
  - ⋆ gegebene Ordnung definiert auch Equivalenz

Wie werden gemeinsame Eigenschaften von stl-Typen beschrieben?

- duch Abstraktionshierarchie von *Konzepten*
- Konzepte beschreiben Eigenschaften
  - ⋆ Member- Daten, -Funktionen, -Typen
  - ⋆ ihre jeweilige Semantik
  - ⋆ Konzepte können von anderen Konzepten erben
- Konzepte sind reines Beschreibungsmittel
- werden nicht nur für Container verwand: Auch *Iteratoren*, *Ordnungen*, etc.
- Konzepte können durch Modelle realisiert werden

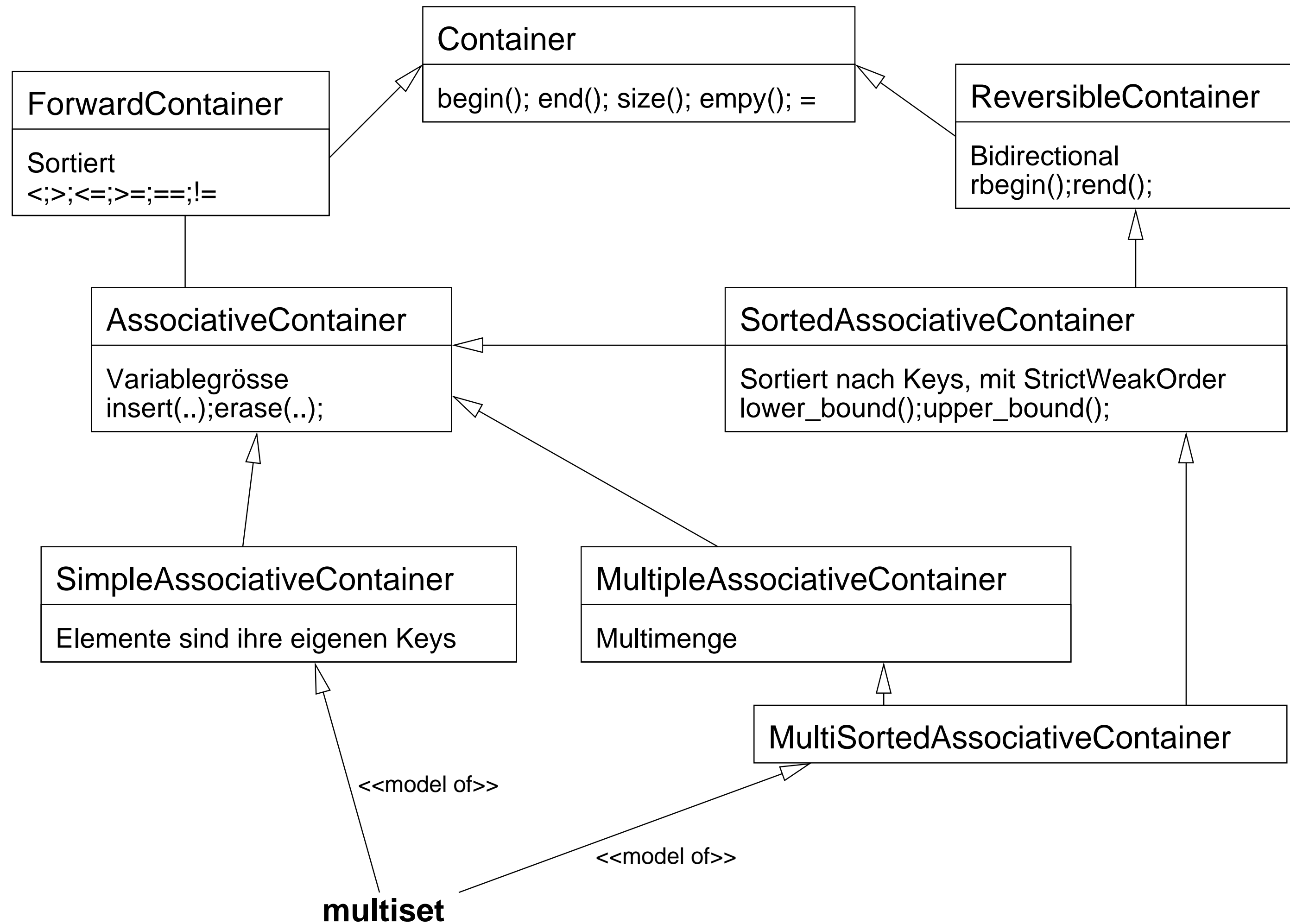
Warum gerade so?

- man muss Anforderungen an einen konkreten Templateparameter spezifizieren können
  - ⋆ allgemeines Template-Problem
  - ⋆ gibt kein C++ Sprachmittel
  - ⋆ Konzepte beschreiben Eigenschaften von Modellen (stl-Typen) und können so *requirements* für Templateparameter spezifizieren

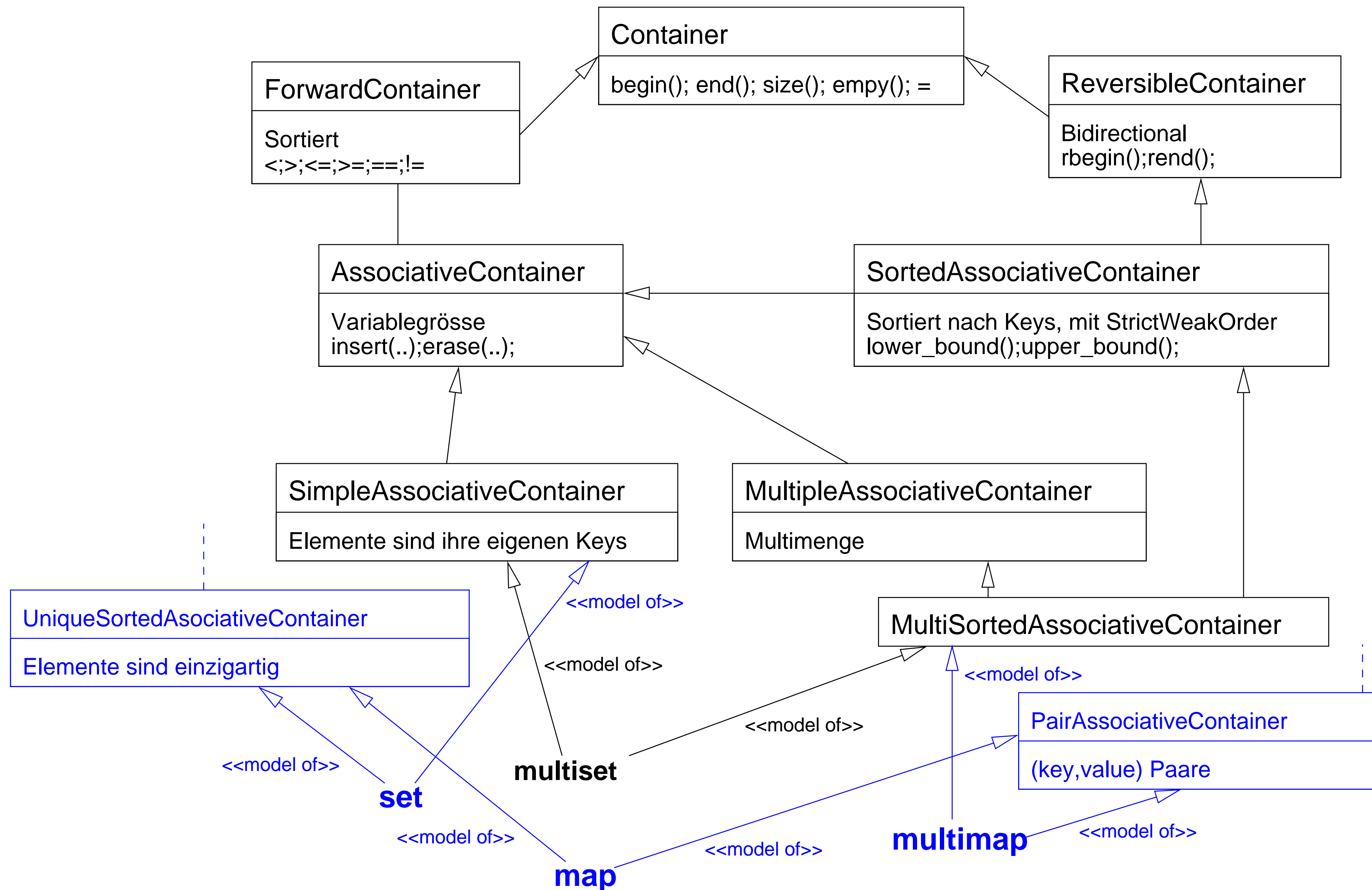
- unverzichtbar zur Beschreibung möglicher Templateparameter eigener generischer Funktionen und Klassen

- Vererbungs- und Modellrelationen und die daraus resultierende Hierarchie visualisieren Gemeinsamkeiten und Unterschiede in stl-Typen

Konkret: multiset in stl



Konkret: `multiset` in stl



Member

`value_type`	Container
`key_type`	Associative Container
`key_compare`	Sorted Associative Container
`value_compare`	Sorted Associative Container
`pointer`	Container
`reference`	Container
`const_reference`	Container
`size_type`	Container
`difference_type`	Container
`iterator`	Container
`const_iterator`	Container
`reverse_iterator`	Reversible Container
`const_reverse_iterator`	Reversible Container
`iterator begin() const`	Container
`iterator end() const`	Container
`reverse_iterator rbegin() const`	Reversible Container
`reverse_iterator rend() const`	Reversible Container
`size_type size() const`	Container
`size_type max_size() const`	Container
`bool empty() const`	Container
`key_compare key_comp() const`	Sorted Associative Container
`value_compare value_comp() const`	Sorted Associative Container
`multiset()`	Container
`multiset(const key_compare& comp)`	Sorted Associative Container
`template <class InputIterator>` `multiset(InputIterator f, InputIterator l)`	Multiple Sorted Associative Container

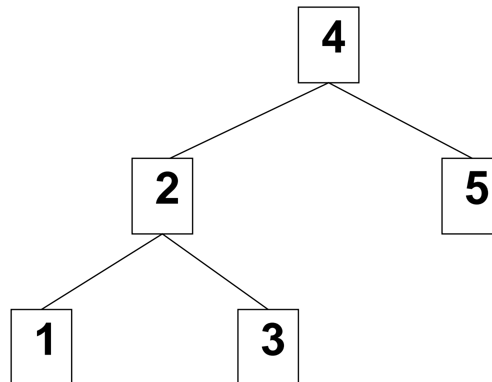
Member

Member	Concept
template <class InputIterator> multiset(InputIterator f, InputIterator l, const key_compare& comp)	Multiple Sorted Associative Container
multiset(const multiset&)	Container
multiset& operator=(const multiset&)	Container
void swap(multiset&)	Container
iterator insert(const value_type& x)	Multiple Associative Container
iterator insert(iterator pos, const value_type& x)	Multiple Sorted Associative Container
template <class InputIterator> void insert(InputIterator, InputIterator)	Multiple Sorted Associative Container
void erase(iterator pos)	Associative Container
size_type erase(const key_type& k)	Associative Container
void erase(iterator first, iterator last)	Associative Container
void clear()	Associative Container
iterator find(const key_type& k) const	Associative Container
size_type count(const key_type& k) const	Associative Container
iterator lower_bound(const key_type& k) const	Sorted Associative Container
iterator upper_bound(const key_type& k) const	Sorted Associative Container
pair<iterator, iterator> equal_range(const key_type& k) const	Sorted Associative Container
bool operator==(const multiset&, const multiset&)	Forward Container
bool operator<(const multiset&, const multiset&)	Forward Container

Wie sind `multisets` implementiert?

- `multisets` sind wie alle Modelle von `AssociativeContainer` sortierte Mengen variabler grösse.

- Implementiert als binäre Bäume



- logarithmische Komplexitität in Suchen und Einfügen, sowie konstante Komplexitität in Sortieren

Definition und Konstruktion

Templateparameter:

```
template <class Key, class StrictWeakComparable=less<Key>, class Alloc=alloc>
class multiset;

typedef Key key_type;
typedef Key value_type;
StrictWeakComparable key_compare, value_compare;
```

Konstruktoren:

```
multiset(const key_compare& comp);

template <class InputIterator>
multiset(InputIterator f, InputIterator l);

template <class InputIterator>
multiset(InputIterator f, InputIterator l, const key_compare& comp);
```

Einfügen

```
iterator insert(const value_type& x);

iterator insert(iterator pos, const value_type& x)

template <class InputIterator>
void insert(InputIterator, InputIterator);
```

Beispiel:

```
int *numbers = new int[N];
for(int i = 0; i < N; i++) numbers[i] = i;

multiset<int,greater<int> > ms;
multiset<int>::iterator hint = ms.begin();

for(int* pos = numbers; pos != numbers + N; pos++)
    hint = ms.insert(hint, (*pos));
```

Löschen

```
void erase(iterator pos);

size_type erase(const key_type& k);

void erase(iterator first, iterator last);

void clear();
```

Finden

iterator find(**const** key_type& k) **const;**

size_type count(**const** key_type& k) **const;**

iterator lower_bound(**const** key_type& k) **const;**

iterator upper_bound(**const** key_type& k) **const;**

pair<iterator, iterator> equal_range(**const** key_type& k) **const**;

Vergleichen

bool operator==(**const** multiset&, **const** multiset&);

bool operator<(**const** multiset&, **const** multiset&);

Warum less nicht immer ausreichend ist.

```
#include <iostream>
#include <set>
using namespace std;

class Car {
    public:
        int speed; int price;
        Car(int s, int p): speed(s), price(p) {}
};

ostream& operator <<(ostream & os, const Car & c) {
    os << "Speed:" << c.speed << "Prize:" << c.prize;
    return os;
}
template<typename T>
void print_set(T & s) {
    for(typename T::iterator pos = s.begin(); pos != s.end(); pos++)
        cout << (*pos) << endl;
}
```

operator< definieren oder less spezialisieren

```
bool Car::operator<(const Car& rhs) const {
    return this->speed < rhs.speed;
}

bool operator<(const Car& lhs, const Car& rhs) {
    return lhs.speed < rhs.speed;
}

template<>
class std::less<Car>: public binary_function<Car,Car,bool> {
    bool operator()(const Cat& lhs, const Car&rhs) const {
        return lhs.speed < rhs.speed;
    }
};

typedef set<Car> uCars; typedef multiset<Car> mCars;
```

Gute Idee oder schlechte Idee?

Programmierer machen Annahmen

- copy-Konstruktoren kopieren, Operator $+$ addiert und `less` vergleicht auf $<$
- man sollte die *natürliche* Bedeutung von programmiersprachlichen Entitäten nicht ändern.
- Doch was ist ein *natürliches* "$<$" für Autos?

Programmierer machen Annahmen

- copy-Konstruktoren kopieren, Operator $+$ addiert und `less` vergleicht auf $<$
- man sollte die *natürliche* Bedeutung von programmiersprachlichen Entitäten nicht ändern.
- Doch was ist ein *natürliches* "$<$" für Autos?
- Es gibt keins! Daherr eigene Ordnung definieren.

Programmierer machen Annahmen

- copy-Konstruktoren kopieren, Operator $+$ addiert und `less` vergleicht auf $<$
- man sollte die *natürliche* Bedeutung von programmiersprachlichen Entitäten nicht ändern.
- Doch was ist ein *natürliches* "$<$" für Autos?
- Es gibt keins! Daherr eigene Ordnung definieren.
- ausserdem ist das Verändern *standartisierter* Klassen und Funktionen selten eine gute Idee

Schliesslich bietet die stl die Möglichkeit eine eigene Ordnung zu definieren

```
class SpeedCompare: public binary_function<Car, Car, bool> {
    bool operator () (const Car& a1, const Car& a2) const {
        return (a1.speed > a2.speed);
    }
};

typedef set<Car, SpeedCompare> uCars;
typedef multiset<Car, SpeedCompare> mCars;
```

less spezialisieren kann aber auch Sinnvoll sein

```
template<typename T>
class std::less<boost::shared_ptr<T> >: public
        binary_function<Car,Car,bool> {
    public:
    bool operator()(const Cat& lhs, const Car&rhs) const {
        return less<T*>()(a.get(),b.get());
    }

};
```

- boost ist ergänzende Bibliothek zur stl
- bietet mehrere Implementationen von intelligenten Zeigern
- natürliche Bedeutung von less beibt erhalten, less wird lediglich auf normale builtin Zeiger abgebildet

Ordnung definiert Äquivalenz

```
main() {
    uCars s; mCars ms;
    Car c1(10, 10); Car c2(20, 10); Car c3(10, 10);

    s.insert(c1); ms.insert(c1);
    s.insert(c2); ms.insert(c2);
    s.insert(c3); ms.insert(c3);

    print_set(s);
}
```

Output:

```
Speed:10 Prize10
Speed:20 Prize10

Speed:10 Prize10
Speed:10 Prize10
Speed:20 Prize10
```

Ordnung definiert Äquivalenz

```
typedef set<int,less_equal<int> > Ints;
typedef set<int,less_equal<int> > mInts;

main() {
    int numbers[4] = {1,2,2,3};
    Ints s(numbers, numbers+4);
    mInts ms(numbers, numbers+4);
    copy(s.begin(), s.end(), ostream_iterator<int>(cout," "));
    cout << endl;
    copy(ms.begin(), ms.end(), ostream_iterator<int>(cout," "));
}
```

Output:

1 2 2 3

1 2 2 3

Zusammenfassung

- Anforderungen an Templateparameter können durch Konzepte beschrieben werden. Anhand ihrer Konzepte werden Gemeinsamkeiten und Unterschiede von stl-Typen deutlich

- `multisets` sind `sets`, welche mehrere "gleiche" Elemente enthalten können.

- Die verwendete Ordnung kann auf mehrere Arten beeinflusst werden, jedoch sollte niemals die eigentliche Bedeutung von stl-Einheiten geändert werden.

Danke