STL - Überblick und Grundbegriffe

Markus Scheidgen

1. Mai 2002

Überblick

- Wie kann ein Algorithmus auf verschiedensten Datenmengen operieren?
- STL Grundkonzepte
  - ⋆ Container
  - ⋆ Iteratoren
  - ⋆ Algorithmen
- Beispiele

Das Problem

- gegeben:
  - ⋆ verschiedene Mengen von Daten: *set*, *list*, *tree*, *bag*, . . .
  - ⋆ verschiedene Algorithmen: *sort*, *find*, . . .
- gesucht: Implementiere Algorithmen nur einmal, so dass sie auf allen Mengen arbeiten können.
- OOP nicht sehr Hilfreich: Daten und Algorithmen durch *Objekte* verbunden. Können nicht getrennt abstrahiert werden.

Das Problem

- gegeben:
  - ⋆ verschiedene Mengen von Daten: *set*, *list*, *tree*, *bag*, . . .
  - ⋆ verschiedene Algorithmen: *sort*, *find*, . . .
- gesucht: Implementiere Algorithmen nur einmal, so dass sie auf allen Mengen arbeiten können.
- OOP nicht sehr Hilfreich: Daten und Algorithmen durch *Objekte* verbunden. Können nicht getrennt abstrahiert werden.

$$\Longrightarrow \text{entkopple Algorithmen und Daten}$$

Entkopplung von Daten und Algorithmen

- Algorithmen und Daten in unabhängigen programmiersprachlichen Einheiten (*Objekten, globale Funktionen, ...*) realisiert
- können so unabhängig entwickelt werden
- keine 1:1 Beziehung mehr
- Problem: Algorithmen sollen aber auf Daten arbeiten, sie brauchen also eine *Schnittstelle* zu den Daten
- Lösung: abstrakte Interfaces die für verschiedene Daten *einheitliche* Schnittstellen definieren
  - ⋆ Wichtigstes: *Iterator*

Realisierung in STL

Die STL basiert auf dem Zusammenspiel von mehreren wohlstrukturierten Komponenten.

Realisierung in STL

Die STL basiert auf dem Zusammenspiel von mehreren wohlstrukturierten Komponenten.

- Container
  - ⋆ verwalten eine Menge von Objekten eines bestimmten Types
  - ⋆ unterschiedliche Containerklassen spiegeln unterschiedliche Techniken wieder
  - ⋆ haben demnach ihre Vor- und Nachteile

Realisierung in STL

Die STL basiert auf dem Zusammenspiel von mehreren wohlstrukturierten Komponenten.

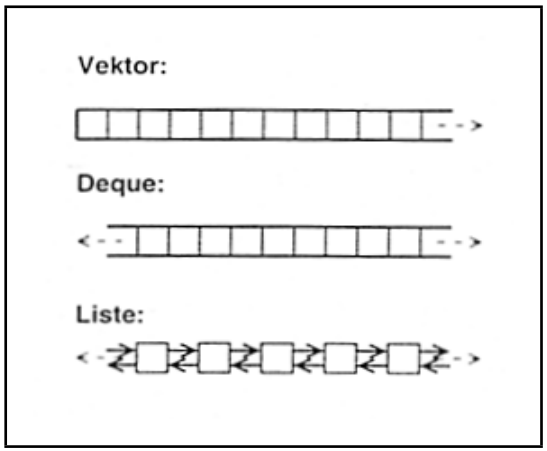
- Container
  - ⋆ verwalten eine Menge von Objekten eines bestimmten Types
  - ⋆ unterschiedliche Containerklassen spiegeln unterschiedliche Techniken wieder
  - ⋆ haben demnach ihre Vor- und Nachteile

- Iteratoren
  - ⋆ sollen über eine Menge von Objekten (Elemente eines Containers) wandern (*iterieren*)
  - ⋆ so dass sie für alle Container die gleiche Schnittstelle zum Zugriff auf Elemente bieten

- Algorithmen
  - ⋆ sollen Mengen als ganzes und Elemente in den Menegen in irgendeiner Form bearbeiten
  - ⋆ verwenden dabei Iteratoren, um nicht für jede Containerklasse neu implementiert werden zu müssen.

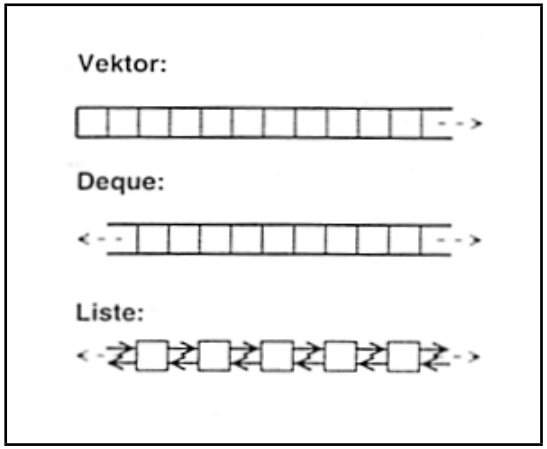
Container

- Sequenzielle Container
  - ⋆ *vector*, *deque*, *list*, ...

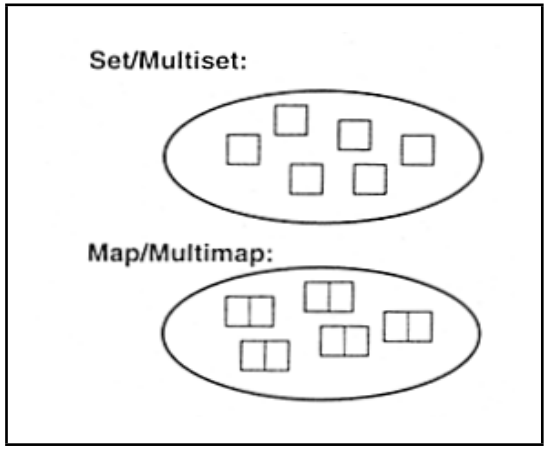


Container

- Sequenzielle Container
  - ⋆ *vector*, *deque*, *list*, ...

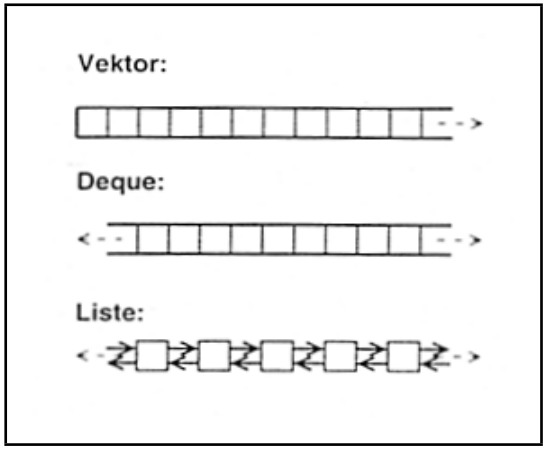


- Assoziative Container
  - ⋆ *set/multiset*, *map/multimap*

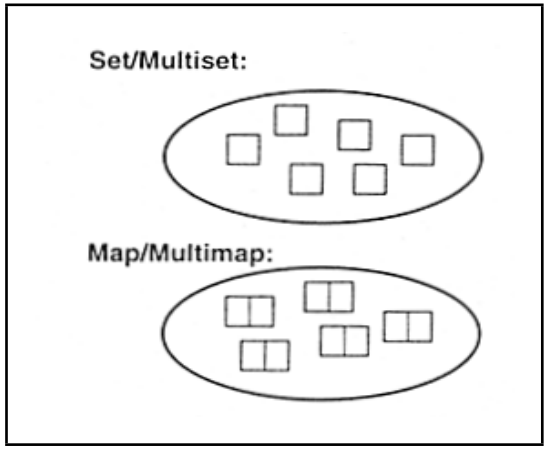


Container

- Sequenzielle Container
  - ⋆ *vector*, *deque*, *list*, ...



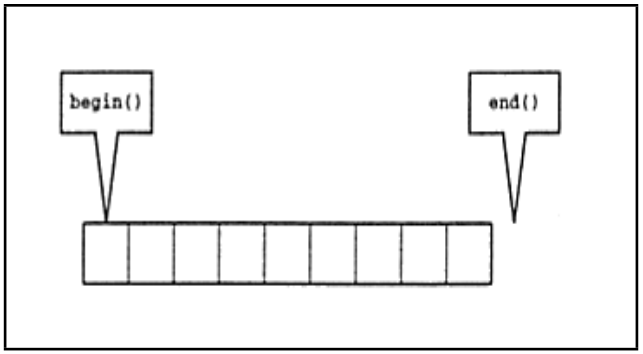
- Assoziative Container
  - ⋆ *set/multiset*, *map/multimap*

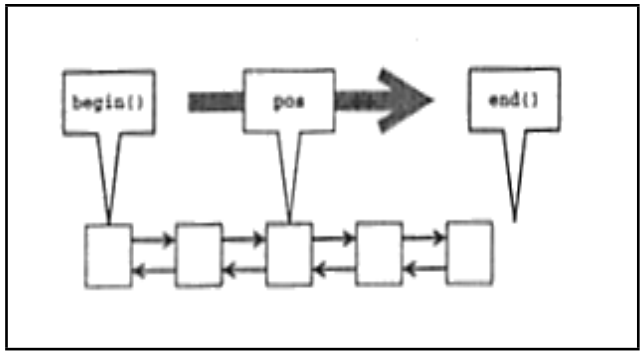


- Container Adapter
  - ⋆ Adapter bilden fundamentale Container auf spezielle Anforderungen ab
  - ⋆ *stack*, *queue*, *Priority-queue*

Iteratoren

- drei fundamentale Operatoren definieren das Verhalten von Iteratoren
  - ⋆ `iterator::operator*()`
  - ⋆ `iterator::operator++()`
  - ⋆ `iterator::operator==()`
- zum Arbeiten mit Iteratoren stellen Container Elementarfunktionen bereit
  - ⋆ `container::begin()`
  - ⋆ `container::end()`



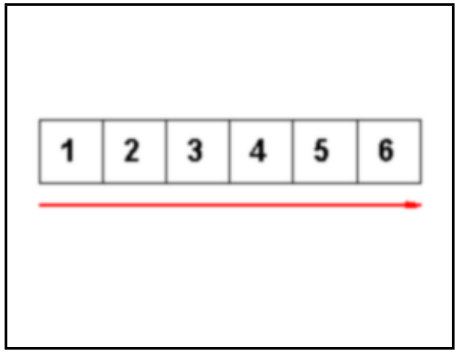


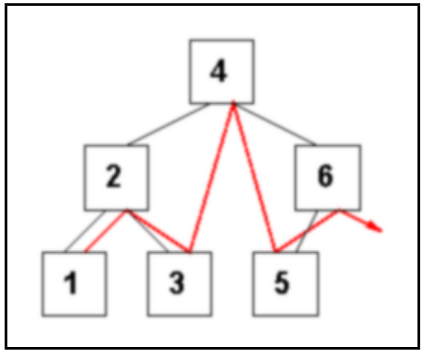
verschiedene Container aber gleicher Algorithmus

```
typedef ...<int> Set;
Set menge;

menge.insert(1);
.
.
menge.insert(6);

Set::iterator pos;
for (pos = menge.begin(); pos != menge.end(); ++pos) {
    cout << *pos << ' ';
}
```





Kategorisierung von Iteratoren

Neben den fundamentalen Operatoren können Iteratoren noch weitere Fähigkeiten besitzen. Diese hängen dann aber vom verwendeten Container ab:

- Bidirectional-Iteratoren
  - ⋆ Iteration in zwei Richtungen: vorwärts mit ++ und rückwärts mit --
  - ⋆ *list*, *set*, *multiset*, *map*, *multimaps*

- Random-Access-Iteratoren
  - ⋆ alle Fähigkeiten von Bidirectional-Iteratoren
  - ⋆ ermöglichen wahlfreien Zugriff: 'Adreß-Arithmetik'
  - ⋆ *vector*, *deque*

- Beispiel Schleifenbedingungen

```
for (pos = menge.begin(); pos != menge.end(); ++pos) {
    ...
}

for (pos = menge.begin(); pos < menge.end(); ++pos) {
    ...
}
```

Algorithmen

- zahlreiche Standard-Algorithmen
  - ⋆ Finden, Vertauschen, Sortieren, Kopieren, Aufaddieren und Modifizieren

- keine Funktionen der Containerklassen, sondern globale Funktionen, die mit Iteratoren arbeiten

- können verschiedene Mengen verknüpfen

- können auf selbstdefinierten Mengenklassen arbeiten

$\Longrightarrow$reduzierter Umfang und erhöhte Mächtigkeit

Bereiche in Algorithmen

- Algorithmen bearbeiten einen oder mehrere *Bereiche*
- Bereiche werden durch Anfang und Ende gegeben
- durch die Breiche wird iteriert
- Bereiche werden immer *halboffen* ausgewertet: $[a, b)$ oder $[a, b[$

find

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main() {
    list<int> menge;

    for(int i=1; i<=20; i++)
        menge.push_back(i);

    list<int>::iterator pos5, pos13;
    pos5 = find(menge.begin(), menge.end(), 5);
    pos13 = find(menge.begin(), menge.end(), 13);

    cout << "max: " << *max_element(pos5, pos13) << endl;       // -> 12

    cout << "max: " << *max_element(pos5, ++pos13) << endl;     // -> 13
}
```

copy

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;

int main() {
    list<int> menge1;
    vector<int> menge2;

    for(int i=1; i<=20; i++)
        menge1.push_back(i);


   menge2.resize(menge1.size()); //ohne resize -> Laufzeitfehler beim Kopieren

    copy(menge1.begin(), menge1.end(),  //erster Bereich
         menge2.begin());               //zweiter Bereich
}
```

in a nutshell

- grundlegendes OO-Konzept: Daten und Algorithmen bilden Einheit, ist unbrauchbar
- Algorithmen und Mengen entkoppeln
- Iteratoren als Schnittstelle zwischen Algorithmen und Mengen

$\Longrightarrow$ ein Algorithmus für n verschiedene Mengentypen