CPP-07 – Standard Template Library (STL)

CPVR Vertiefungsmodul BTI-7281

Urs Künzler (urs.kuenzler@bfh.ch)

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CPP-07 Table of Contents – Course Introduction

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

07.01 Standard Template Library Overview

- **Die Standard Template Library bildet einen zentralen Bestandteil der C++ Standard Library (entwickelt von der Firma HP).**
- **Das Design der STL entspricht nicht dem typisch objektorientierten Lösungsansatz, weil Daten und die dazugehörigen Funktionen absichtlich voneinander getrennt implementiert sind.**
- **Der Vorteil besteht in der Entkoppelung von Operationen und Daten, sodass diese beliebig miteinander kombiniert werden können, was die Standard Template Library sehr universell einsetzbar macht.**
- **Durch die Implementation mittels Template-Klassen kann die STL mit beliebigen (auch eigenen) Datentypen verwendet werden und die Performance bleibt trotzdem sehr gut.**

STL - Allgemeines Konzept

- **Die STL setzt sich aus den folgenden Bestandteilen zusammen:**
  - **Container** werden verwendet um Mengen von Daten eines bestimmten Datentyps zu organisieren und zu verwalten.
  - **Iteratoren** werden verwendet um eine Menge von Daten zu durchlaufen (iterieren).
  - **Algorithmen** werden verwendet um Mengen als Ganzes oder nur bestimmte Daten in diesen Mengen zu bearbeiten.
- **Die Funktionsweise dieser STL Bestandteile besteht im wesent-lichen in der Verwaltung von Daten durch Container, wobei deren Bearbeitung durch Algorithmen erfolgt und die Iteratoren als Bindeglied zwischen Containern und Algorithmen eingesetzt werden.**

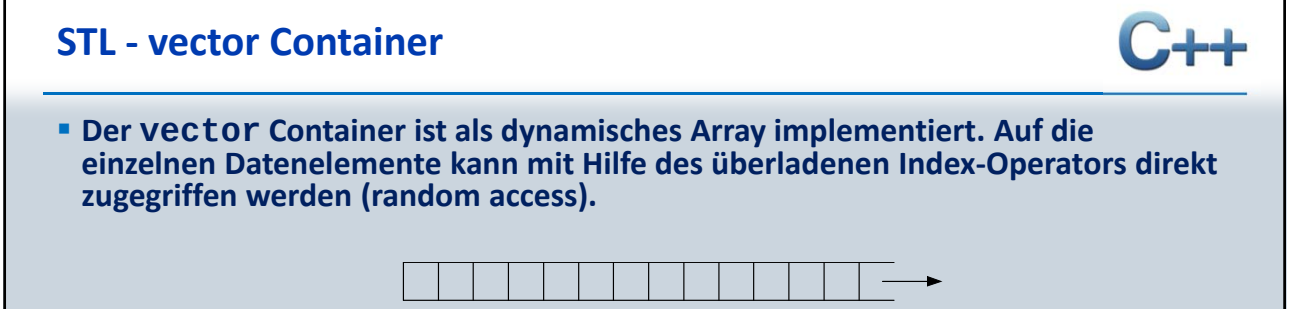
Standard Template Library WWW Resourcen

- **Online Reference Manuals und weitere Resourcen:**
  - www.cppreference.com (STL Reference)
  - www.mathcs.sjsu.edu/faculty/horstman/safestl.html ("Safe" STL Implementation)
  - www.cs.brown.edu/people/jak/proglang/cpp/stltut/tut.html (STL Tutorial)

07.02 STL Containers

- **Container werden verwendet um Mengen von Objekten oder Werte vordefinierter Datentypen zu verwalten, wobei je nach Anwendung verschiedene Container verwendet werden können:**
  - Bei **sequentiellen Containern** besitzt jedes Datenelement eine bestimmte Position, welche durch den Ort des Einfügens definiert wird. Die STL kennt **`vector`**, **`deque`** und **`list`** als sequentielle Container.
  - **Assoziative Container** sortieren ihre Datenelemente automatisch anhand einer Vergleichsfunktion, die von aussen definiert werden kann. Da diese Container als Binärbäume implementierten sind, ist insbesondere die Suche von Elementen sehr schnell. Die STL kennt **`set`** bzw. **`multiset`** und **`map`** bzw. **`multimap`** als assoziative Container.
- **Die Container Klassen können auch rekursiv verwendet werden (z.B. Liste von Vektoren).**

STL - vector Container



- Der `vector` Container ist als dynamisches Array implementiert. Auf die einzelnen Datenelemente kann mit Hilfe des überladenen Index-Operators direkt zugegriffen werden (random access).

- Das Zeitverhalten des Vektors ist für die Mutation von Daten am Ende optimal, für das Einfügen und Löschen von Elementen am Anfang und im Bereich der Mitte jedoch sehr langsam.
- Der Speicherplatz eines Vektors wird dynamisch erweitert, indem eine Reallozierung durchgeführt wird. Dabei werden die Elemente mittels Copy-Konstruktor in den neuen Speicher kopiert und die alten Elemente mit dem Destruktor gelöscht (Zeitverhalten!).

STL - vector Container Beispiel

```
#include <iostream>
#include <vector>
...
// int vector definieren
vector<int> iMenge;

// int vector füllen
for ( int iElement = 10; iElement <= 100; iElement += 10 ) {
    iMenge.push_back( iElement );
}

// int vector auslesen
for ( int iCounter = 0; iCounter < iMenge.size(); iCounter++ ) {
    cout << iCounter << ". Element: "<< iMenge[ iCounter ] << endl;
}

// Ausgabe: 0. Element: 10
//          1. Element: 20
//          2. Element: 30
...
```

STL - deque Container

- Der deque Container ("double ended queue") ist als dynamisches Array implementiert, welches in beide Richtungen wachsen kann. Der Zugriff auf die einzelnen Datenelemente erfolgt mittels dem Index-Operators (random access).

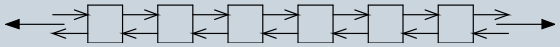
- Das Zeitverhalten ist hier für Änderungen am Anfang und am Ende optimal, hingegen ist die Deque für Einschieben und Löschen von Datenelementen im Bereich der Mitte nicht ideal. Durch die aufwendigere Speicherorganisation ist dieser Container etwas langsamer als der Vektor.

STL - deque Container Beispiel

```
#include <iostream>
#include <deque>
...
// int deque definiern
deque<int> iDeque;
int iCounter = 0;

//int deque füllen
for ( iCounter = 1; iCounter <= 10; iCounter++ ) {
     iDeque.push_back( iCounter );
     iDeque.push_front( iCounter );
}

// int deque auslesen
for ( iCounter = 0; iCounter < iDeque.size(); iCounter++ ) {
     cout << iDeque[ iCounter ] << " ";
}
cout << endl;

// Ausgabe: 10 9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8 9 10
```

STL - list Container

- Der `list` Container ist als doppelt verkettete Liste implementiert.
  Der direkte Zugriff auf Datenelemente ist damit nicht möglich, wes-halb eine sequentielle Suche mit linearem Zeitverhalten notwendig ist. (Die Liste besitzt dem entsprechend keinen Index-Operator.)



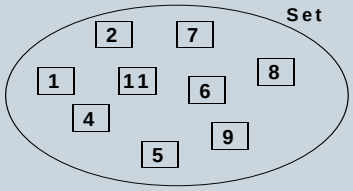
- Das Zeitverhalten der Liste ist in Bezug auf Einfügen von Elementen am Anfang und am Ende sehr gut. Das Einfügen und Löschen von Datenelementen im Bereich der Mitte der Liste ist sehr schnell, da aufgrund der Pointer-Verknüpfung nur sehr wenige Elemente bearbeitet werden müssen.

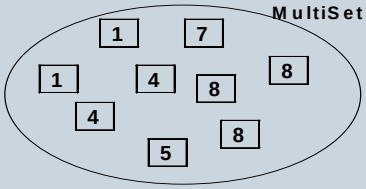
STL - list Container Beispiel

```
#include <iostream>
#include <list>
...
// int liste definieren
list<int> iListe;

// int liste füllen
for ( int iCounter = 1; iCounter <=10; iCounter++ ) {
     iListe.push_back( iCounter );
}

// int liste auslesen
while ( ! iListe.empty()) {
     cout << iListe.front() << ' ';
     iListe.pop_front();
}
cout << endl;

// Ausgabe:   1 2 3 4 5 6 7 8 9 10
```

STL - set und multiset Container

- Ein set ist ein assoziativer Container in dem die Datenelemente jeweils nur einmal vorkommen dürfen und die automatisch nach ihrem Wert sortiert werden. In einem multiset können die Elemente auch mehrfach vorkommen.

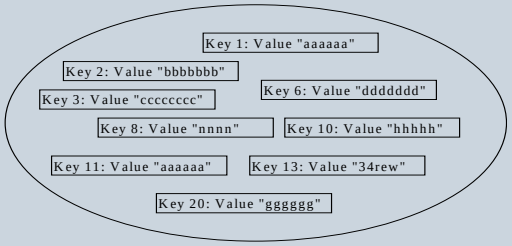


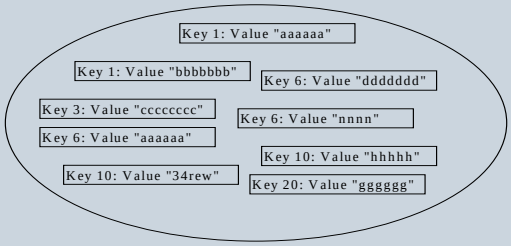


- Aufgrund der Sortierung und der Implementation als binäre Baum-struktur erfolgt vor allem das Suchen von Elementen sehr schnell. Das Zeitverhalten für Einfügen und Löschen von Datenelementen ist ähnlich wie bei einer Liste.
- Aufgrund der Sortierung der Datenelemente nach ihrem Wert, ist es nicht möglich die Werte der verwalteten Daten zu verändern.

STL - map und multimap Container

- `map` bzw. `multimap` Container verwalten Schlüssel-Wert-Paare als Datenelemente. Die Elemente werden nach dem Schlüssel sortiert, wobei in einem Map jeder Schlüssel nur einmal, bei einem Multimap mehrmals vorkommen kann.





- Da Map und Multimap Container als binäre Baumstruktur implementiert sind, ist das Zeitverhalten analog wie für Set Container.
- Im Gegensatz zu den Datenelementen eines Set Containers, dürfen die Werte von Elementen eines Map Containers modifiziert werden, nur die Schlüssel können nicht verändert werden.

STL - Container Adapter

- **Mit Container-Adaptoren werden die Standard Container an spezi-fische Bedürfnisse angepasst und das Interface entsprechend vereinfacht. Die STL definiert die folgenden Adapter-Klassen:**
  - **Queue** wird zur Implementation einer FIFO-Queue ("First In First Out") verwendet, wobei intern ein Deque-Container verwendet wird.
  - **Stack** wird zur Implementation eines LIFO-Speichers ("Last In First Out") verwendet. Für die interne Implementation wird intern ein Deque-Container verwendet.
  - **Priority-Queue** definiert eine Queue deren Datenelemente nach ihrer Priorität wieder ausgelesen werden. Für die Implementation wird intern ein Vector-Container verwendet.

STL - Container Eigenschaften

- **Gemeinsame Eigenschaften aller Container-Klassen (Auszug):**
  - `ContTyp c` Erzeugt einen leeren Container ohne Datenelemente
  - `ContTyp c1(c2)` Erzeugt einen Container c1 als Kopie des Containers c2
  - `~ContTyp` Destruktor, löscht alle Datenelemente und gibt den Speicher frei
  - `c.size()` liefert die aktuelle Anzahl Datenelemente im Container
  - `c.max_size()` liefert die maximal mögliche Anzahl Datenelemente
  - `c.empty()` prüft ob der Container leer ist
  - `c1 ==` bzw. `!= c2` prüft ob c1 gleich bzw. ungleich c2 ist
  - `c1 <` bzw. `> c2` vergleicht ob c1 kleiner bzw. grösser c2 ist
  - `c1 <=` bzw. `>= c2` vergleicht ob c1 kleiner od. gleich bzw. grösser od. gleich c2 ist
  - `c1 = c2` weist c1 die Datenelemente von c2 zu
  - `c.begin()` liefert einen Iterator für das erste Datenelement
  - `c.end()` liefert einen Iterator für Position hinter dem letzten Datenelement
  - `c.insert(pos,e)` fügt an der Position (pos) das Datenelement (e) in c ein
  - `c.erase(pos)` löscht das Datenelement an der Position (pos)
  - `c.clear()` löscht alle Datenelemente (leert den Container)

STL - Container Eigenschaften II

- **Anforderungen damit ein (eigener) Datentyp als Datenelement eines Containers verwendet werden kann:**
  - Da Container intern Kopien der Datenelemente anlegen und auch Kopien zurückge-geben werden, muss jeder Datentyp einen **Copy-Konstruktor** zur Verfügung stellen.
  - Da Container und Algorithmen Zuweisungen verwenden, muss jeder Datentyp einen **Zuweisungsoperator** bereitstellen.
  - Da beim Löschen der Container auch die internen Kopien der Datenelemente gelöscht werden, muss ein entsprechend implementierter **Destruktor** aufgerufen werden können.
  - Für einige Funktionen sequentieller Container muss zusätzlich ein **Default-Konstruktor** implementiert sein.
  - Bei assoziativen Containern muss für die automatische Sortierung der Datenelemente grundsätzlich der **Operator <** definiert sein.
  - Für zahlreiche Funktionen (hauptsächlich zum Suchen von Datenelementen) muss der **Vergleichsoperator ==** definiert sein.

STL - Container Eigenschaften III

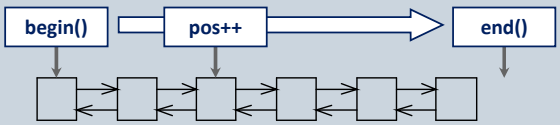
- **Da die STL Container die Datenelemente default-mässig "by value" verwalten, d.h. eine lokale Kopie der Elemente erstellen, ist dies bei Verwendung der Container mit Pointer-Elementen besonders zu berücksichtigen:**
  - Vergleichsoperationen werden mit Pointer-Adressen durchgeführt, was meist nicht der gewünschten Funktionalität entspricht (--> eigene Vergleichsfunktion implementieren).
  - Externe Verweise auf Datenelemente können ungültig werden, weil bei verschiedenen Operationen die Elemente intern verschoben oder umkopiert werden.
- **Bei Verwendung der STL Container mit "by value" Datenelementen können ebenfalls Probleme entstehen, wenn z.B. Objekte einer abstrakten Basisklasse verwaltet werden sollen (Compiler-Error).**
- **Im weiteren ist zu beachten, dass die Funktionen und Operatoren der STL keine Fehlerprüfungen durchführen und somit keine Exceptions ausgelöst werden! (vgl. "Safe" STL Implementation)**

07.03 STL Iterators I

- Iteratoren sind Objekte, die verwendet werden um Container zu durchlaufen (iterieren). Sie ermöglichen eine allgemeingültige Verbindung zwischen Algorithmen und Containern herzustellen.
- Jede Container-Klasse definiert eine Container spezifische Iterator-Klasse mit einem einheitlichen Interface, welches den Algorithmen erlaubt verschiedene Container einheitlich zu verwenden.
  (vgl. Iterator Design Pattern)
- Folgende Operatoren bilden die Grundfunktionalität aller Iteratoren:
  - iterator::operator*() liefert das Element an der aktuellen Position des Iterators
  - iterator::operator++() setzt den Iterator zum nächsten Datenelement
  - iterator::operator==() prüft ob zwei Iteratoren auf das gleiche Datenelement zeigen
  - iterator::operator!=() prüft ob zwei Iteratoren auf versch. Datenelemente zeigen

STL Iterators II

- Iteratoren werden für die Schleifensteuerungen bei der iterativen Bearbeitung von Container-Elementen verwendet. Dabei wird ein Iterator-Objekt als Pointer auf das aktuelle Datenelement benutzt.



- Iterator-Objekte für Container-Beginn und -Ende können für jeden Container mittels folgenden Funktionen erzeugt werden:
  - `c.begin()` liefert einen Iterator für das erste Element
  - `c.end()` liefert einen Iterator für die Position hinter dem letzten Element
- Für Iteratoren müssen keine spezifischen Header-Files eingebunden werden, weil diese innerhalb der jeweiligen Container definiert sind.

STL Iterators Beispiel III

```
#include <set>

// int set definieren
set<int> iSet; int iCounter = 0;

// set füllen
for ( iCounter = 1; iCounter <= 10; iCounter++ ) {
     iSet.insert( iCounter );      // set 1. mal füllen
}
// set füllen - mit teilweise gleichen Werten
for ( iCounter = 6; iCounter <= 15; iCounter++ ) {
     iSet.insert( iCounter );      // set 2. mal füllen
}

// set mit iterator auslesen
set<int>::iterator pos;
for ( pos = iSet.begin(); pos != iSet.end(); pos++ ) {
     cout << *pos << ' ';
}
cout << endl;

// Ausgabe: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```


STL Iterators IV

- **Neben den Grundfunktionen können Iteratoren weitere Funktionen besitzen. Aufgrund dieser spezifischen Eigenschaften werden Iteratoren in verschiedene Kategorien eingeteilt:**
  - **Forward Iteratoren** - Diese Iteratoren können nur vorwärts (operator++) iteriert werden.
  - **Bidirectional Iteratoren** - Diese Iteratoren können vorwärts (operator++) und rückwärts (operator--) iteriert werden.
  - **Random Access Iteratoren** - Diese Iteratoren können ebenfalls vorwärts und rückwärts iteriert werden, haben aber zusätzlich weitere Funktionen implementiert
    um z.B. direkt auf einzelne Datenelemente zugreifen zu können (operator[ ]).
    Die Container-Klassen vector und deque erzeugen Iteratoren dieser Kategorie.
- **Forward- und Bidirectional-Iteratoren werden von sämtlichen STL Containern zur Verfügung gestellt.**

07.04 STL Algorithms

- **STL-Algorithmen bieten eine grosse Anzahl von Funktionen zum Finden, Vertauschen, Sortieren, Kopieren und Modifizieren von Containern.**
- **Die Algorithmen sind keine Funktions-Member von Container-Klassen, sondern globale Funktionen die mit Hilfe von Iteratoren gesteuert werden und damit Zugriff auf die Elemente der Container erhalten.**
- **Alle Algorithmen können auch nur auf einen bestimmten Bereich (range) von Datenelementen angewendet werden.**
- **Viele Algorithmen können durch die Übergabe einer benutzer-definierten Funktion für spezifische Aufgaben spezialisiert werden.**

STL Algorithms II

- **Alle STL Algorithmen sind im Header File `<algorithm>` definiert.**
- **Nicht modifizierende Algorithmen**
  - `find()` sucht ein bestimmtes Datenelement
  - `for_each()` ruft für jedes Datenelement eine readonly Operation auf
  - `count()` zählt Datenelemente die eine bestimmte Bedingung erfüllen.
- **Modifizierende Algorithmen**
  - `copy()` kopiert einen Bereich
  - `transform()` modifiziert jedes Element eines Bereichs
- **Löschende Algorithmen**
  - `remove()` löscht bestimmte Elemente
  - `unique()` löscht aufeinanderfolgende Duplikate

STL Algorithms III

- **Mutierende Algorithmen**
  - `reverse()` kehrt die Reihenfolge von Elementen um
  - `random_shuffle()` bringt die Reihenfolge der Elemente durcheinander
- **Sortierende Algorithmen**
  - `sort()` sortiert Elemente
  - `partial_sort()` sortiert die ersten n Elemente
- **Algorithmen für sortierte Bereiche**
  - `includes()` prüft ob alle Elemente einer Teilmenge enthalten sind
  - `merge()` fasst die Elemente zweier Bereiche zusammen
- **Numerische Algorithmen**
  - `accumulate()` verknüpft Elemente
  - `partial_sum()` verknüpft ein Element jeweils mit allen Vorgängern