C++ FOR (Tuesday Morning)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

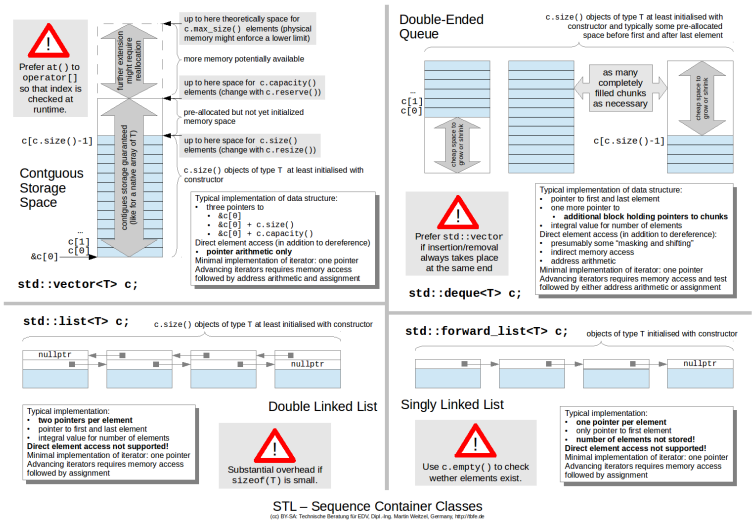
Die Besprechung der Musterlösung(en) erfolgt im direkten Anschluss an die Mittagspause.

Sequenzielle Container

- Zusammenhängender Speicher

- Doppelt verkettete Liste und ...
- ... einfach verkettete*

- Double-Ended Queue (Deque)



*: Kein Bestandteil von C++98 sondern erst mit C++11 verfügbar.

Zusammenhängender Speicher

Die grundlegende Datenstruktur ist hier ein Stück zusammenhängender Speicher.

Dieser wird – typischerweise zuzüglich etwas Reserve – auf dem Heap angelegt.

Ein Objekt der Klasse `std::vector` besitzt in der typischen Implementierung drei (private) Daten-Member:*

- Anfangsadresse im Heap-Speicher
- Größe des benutzten Bereichs
- Größe der Gesamt-Reservierung

Links zur ausführlichen Online-Dokumentation:

- http://en.cppreference.com/w/cpp/container/vector
- http://www.cplusplus.com/reference/vector/

*: As the implementation is not dictated by the C++ standard, it is free to chose whatever it sees fit, but usually an `std::vector` uses three pointers, holding the address of the whole storage area, the begin of free space after elements contained in the vector, and the next address after the allocated space.

Vorteile eines `std::vector`

Effizient unterstützt werden:

- Elemente einfügen und entnehmen **am Ende**
- Wahlfreier Zugriff (mit oder ohne Indexprüfung)
- Move-Konstruktor und -Assignment
- Vertauschung (*swap*) zweier `std::vector`-Inhalte

Die Speicheranforderung auf Vorrat macht den inkrementellen Aufbau eines `std::vector` durch vielfaches Anfügen ebenfalls relativ effizient.*

Einschränkungen eines `std::vector`

- Kein (effizientes) Einfügen und Entnehmen an verschiedenen Enden
- Kein effizientes Einfügen und Entnehmen in der Mitte
- Aufwändige Weitergabe als Wert (bzw. Wertkopie)

*: Crucial here is to increase space proportionally, i.e. if the pre-allocated size is filled completely, the next element to store will extend memory allocation by some constant factor, not a constant number of elements. In the latter case runtime behaviour would have $O(N^2)$ performance, while the former gives amortized O(1).

Doppelt verkettete Listen

Die grundlegende Datenstruktur sind einzelne Abschnitte im Heap-Speicher (typisch einer pro Element), welche untereinander durch Vor- und Rückwärts-Zeiger verbunden sind.

Ein Objekt der Klasse `std::list` besitzt in der typischen Implementierung drei (private) Daten-Member:

- je ein Zeiger auf das erste und …
- … das letzte enthaltene Element,* sowie
- eine Ganzzahl welche die Anzahl der Elemente angibt.

Links zur ausführlichen Online-Dokumentation:

- http://en.cppreference.com/w/cpp/container/list
- http://www.cplusplus.com/reference/list/

*: An empty `std::list` can simply use `nullptr`-s for both (or might leave the pointers uninitialised and check the list's size member before each access).

Vorteile einer `std::list`

Effizient unterstützt werden:

- Elemente einfügen und entnehmen **am Anfang und am Ende**
- Einfügen und Entnehmen auch in der Mitte*
- Move-Konstruktor und -Assignment
- Vertauschung (*swap*) zweier `std::list`-Inhalte

Einschränkungen einer `std::list`

- Kein wahlfreier Zugriff
- Aufwändige Weitergabe als Wert (bzw. Wertkopie)

Die doppelt verkettete Liste tendiert zu einem hohen Overhead für den **Speicherbedarf pro Element** – um so mehr, je weniger Platz die Nutzdaten benötigen (wegen typischerweise notwendigem Alignment bis hin zum 24-fachen bei `char`-Elementen und einer 64-Bit-Hardware).

*: Vorausgesetzt wird dabei, dass per Iterator bereits die Position bestimmt wurde, in deren näherer Umgebung Elemente oder komplette Listen eingefügt oder entnommen werden sollen.

Einfach verkettete Listen

Ein im Rahmen von C++11 hinzugefügter, auf minimalen Overhead ausgelegter Container mit der grundlegenden Datenstruktur einer einfach verketteten Liste, deren Elemente wiederum auf dem Heap angelegt werden.

Ein Objekt der Klasse `std::forward_list` besitzt in der typischen Implementierung genau ein (privates) Daten-Member:

- Zeiger auf das erste enthaltene Element*

> Die Anzahl der Elemente einer `std::forward_list` wird nicht gezählt (anders als bei der `std::list`).

Links zur ausführlichen Online-Dokumentation:

- http://en.cppreference.com/w/cpp/container/forward_list
- http://www.cplusplus.com/reference/forward_list/

*: An empty `std::forward_list` needs to use a `nullptr` (or maybe some other address that can be unmistakeably recognized as not being a list member address).

Vorteile einer `std::forward_list`

Effizient unterstützt werden:

- Elemente einfügen und entnehmen **am Anfang**
- Einfügen und Entnehmen auch in der Mitte*
- Move-Konstruktor und -Assignment
- Vertauschung (*swap*) zweier `std::forward_list`-Inhalte

> Das Einfügen und Entnehmen in der Mitte hat eine im Vergleich mit den anderen sequenziellen Containern ungewöhnliche Semantik, die am Namen der Operationen deutlich wird: `insert_after` und `erase_after`

Einschränkungen einer `std::forward_list`

- Kein wahlfreier Zugriff
- Kein (effizientes) Einfügen und Entnehmen an verschiedenen Enden
- Aufwändige Weitergabe als Wert (bzw. Wertkopie)

---

*: Vorausgesetzt wird dabei, dass per Iterator bereits die Position bestimmt wurde, hinter der (direkt oder in näherer Umgebung) Elemente eingefügt oder entnommen werden sollen.

Deque

Hierbei handelt es sich quasi um einen "in Abschnitte zerstückelten" `std::vector`, welcher grob zusammengefasst sich wie folgt vom `std::vector` unterscheidet:

- Keine Garantie, dass **alle** Elemente in zusammenhängendem Speicher aufeinander folgen.*
- Elemente können an **beiden** Enden eingefügt und entnommen werden.
- Wahlfreier Zugriff ist möglich und nur geringfügig weniger performant.

Anders als ein `std::vector` kann eine `std::deque` Basis für eine FIFO-Speicherung (Warteschlange) sein, ist aber nicht als Schnittstelle zu Legacy-Code nutzbar, der ein klassisches (eingebautes) Array erwartet.

*: Sofern `std::deque::operator[]()` und `std::deque::at()` eine Referenz auf den Elementtyp liefern, ist die anschließende Anwendung des Adress-Operator nicht auszuschließen, um so die Speicheradresse des Elements zu bestimmen. Damit in Schnittstellen zu Legacy-Code die Verwendung von `std::vector` sichergestellt ist, empfiehlt sich die Benutzung von dessen Member-Funktion `vector::data()`.

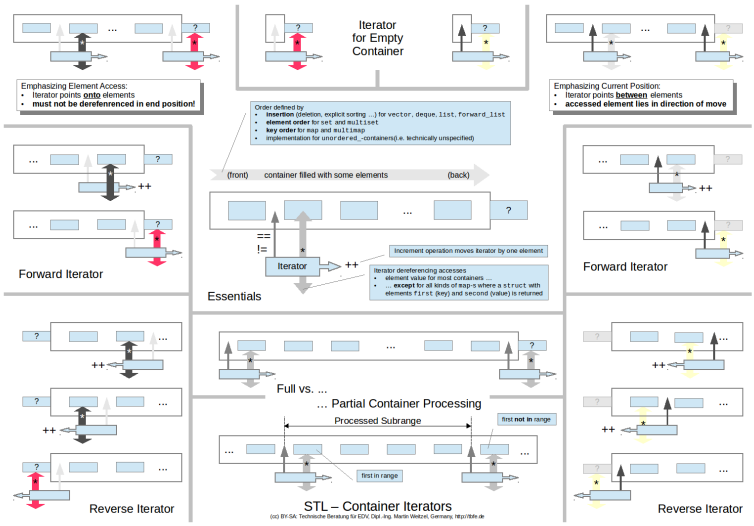
Iterator-Grundlagen

- Prinzip des Iterators ...
- ... auf Elemente oder ...
- ... dazwischen zeigend

- Vorwärts-Iterator auf ...
- ... oder zwischen Elemente zeigend

- Rückwärts-Iterator auf ...
- ... oder zwischen Elemente zeigend

- Gesamter Container ...
- ... und Beschränkung auf Teilbereich



Prinzip des Iterators

Iteratoren sind spezifisch für ihren jeweiligen Container-Typ als geschachtelte (Helfer-) Klassen definiert.

Sie kommen vorwiegend dann zur Anwendung,

- wenn auf die Elemente eines Containers oder eines Teilbereichs sequenziell nacheinander zugegriffen werden soll,
- werden aber auch verwendet, um Grenzen anzugeben (gesamter Container oder Teilbereich),
- oder weisen auf spezifische Elemente, z.B. das Ergebnis einer Suche.

Iterator *auf* Elemente zeigend

Man kann Iteratoren grafisch als *auf Elemente zeigend* veranschaulichen.

Dann bekommt man allerdings ein Erklärungsproblem mit der Iterator-Endposition und auch mit der Erklärung, wohin der Iterator im Falle eines leeren Containers zeigt.

Als Ausweg wird häufig ein – nicht wirklich vorhandenes – Element hinter dem letzten Container-Element angenommen, zusammen mit der Einschränkungen, dass ein Iterator in dieser Position **nicht mehr dereferenziert** werden darf.

Iterator *zwischen* Elemente zeigend

Mit dieser grafischen Veranschaulichung werden die zuvor beschriebenen Probleme vermieden.

Dafür stellt sich die Frage, was ein Iterator bei der Dereferenzierung liefert – er zeigt ja nicht auf Elemente sondern dazwischen.

> Veranschaulicht man Iteratoren als zwischen Elemente zeigend, ist die Bewegungsrichtung des Iterators wichtig, also in wohin er bei Anwendung des `operator++` verschoben wird.
>
> Genau das Element, über das hinweg der Iterator damit weiter gesetzt würde, wird bei der Dereferenzierung geliefert.

Damit wird zugleich klar, dass ein Iterator in End-Position nicht dereferenziert werden darf, denn es gibt ja auch keine Position, zu der er weiter gesetzt werden könnte.

Vorwärts-Iterator *auf* Elemente zeigend

Bei Dereferenzierung liefern Vorwärts-Iteratoren eine Referenz auf dasjenige Element im Container, auf das sie gerade zeigen.

- Mit ++ bewegen sich Vorwärts-Iteratoren *vom Container-Anfang zu dessen Ende*.*
- Dort ist ein – technisch gesehen schon außerhalb des Containers liegendes – Element anzunehmen, auf das ein Vorwärts-Iterator in seiner letzten gültigen Position am Container-Ende noch zeigen kann.

[!] Einen Vorwärts-Iterator in dieser Position zu *dereferenzieren* oder *weiterzubewegen*, liefert undefiniertes Verhalten.

*: Sofern es sich um Iteratoren aus der Kategorie der Bidirektional-Iteratoren handelt, bewegen sie sich bei -- natürlich Richtung Container-Anfang.

Vorwärts-Iterator *zwischen* Elemente zeigend

Bei Dereferenzierung liefern Vorwärts-Iteratoren eine Referenz auf dasjenige Element im Container, über das sie mit der nächsten ++-Operation weiter geschaltet würden.

- Mit ++ bewegen sich Vorwärts-Iteratoren vom Container-Anfang zu dessen Ende.*
- In ihrer Endposition zeigen sie hinter das letzte Element im Container.

[!] Einen Vorwärts-Iterator in dieser (End-) Position *weiterzubewegen* oder zu *dereferenzieren*, liefert undefiniertes Verhalten.

*: Sofern es sich um Iteratoren aus der Kategorie der Bidirektional-Iteratoren handelt, bewegen sie sich bei -- natürlich Richtung Container-Anfang.

Rückwärts-Iterator *auf* Elemente zeigend

Bei Dereferenzierung liefern Rückwärts-Iteratoren eine Referenz auch dasjenige Element des Containers, auf das sie gerade zeigen.

- Mit ++ bewegen sich Rückwärts-Iteratoren *vom Container-Ende zu dessen Anfang*.*
- Dort ist ein – technisch gesehen schon außerhalb des Containers liegendes – Element anzunehmen, **auf** das ein Rückwärts-Iterator in seiner letzten gültigen Position **am Container-Anfang** noch zeigen kann.

[!] Einen Rückwärts-Iterator in dieser (End-) Position zu *dereferenzieren* oder *weiterzubewegen*, liefert undefiniertes Verhalten.

*: Sofern es sich um Iteratoren aus der Kategorie der Bidirektional-Iteratoren handelt, bewegen sie sich bei -- natürlich Richtung Container-Ende.

Rückwärts-Iterator *zwischen* Elemente zeigend

Bei Dereferenzierung liefern Rückwärts-Iteratoren eine Referenz auf dasjenige Element im Container, über das sie mit der nächsten ++-Operation weiter geschaltet würden.

- Mit ++ bewegen sich Rückwärts-Iteratoren vom Container-Ende zu dessen Anfang.*
- In ihrer Endposition zeigen sie vor das erste Element im Container.

[!] Einen Rückwärts-Iterator in dieser (End-) Position *weiterzubewegen* oder zu *dereferenzieren*, liefert undefiniertes Verhalten.

*: Sofern es sich um Iteratoren aus der Kategorie der Bidirektional-Iteratoren handelt, bewegen sie sich bei -- natürlich Richtung Container-Ende.

Alle Elemente im Container

Um **über alle Elemente** eines Containers zu iterieren, wird oft folgende Konstruktion verwendet (hier demonstriert am Beispiel einer `std::list`):

```
std::list<int> li;
…
for (auto it = li.begin(); it != li.end(); ++it)
    … *it …
```

Mit C++98 muss statt `auto` der exakte Iterator-Typ angegeben werden, wozu sich der Übersichtlichkeit und Wartungsfreundlichkeit wegen eine Typ-Definition für den Container anbietet:*

```
typedef std::list<int> INTLIST;
```

Der Iterator ergibt sich damit als:

```
for (INTLIST::iterator it = li.begin(); it != li.end(); ++li)
   … *it …
```

*: Die neue C++11-Syntax zur Typdefinition funktioniert natürlich auch: `using INTLIST = std::list<int>;`

Teilbereich eines Containers

Um über die **Elemente in einem Teilbereich** eines Container zu iterieren, werden die Grenzen mit zwei Iteratoren angegeben, die

- *hinter den Anfang* und/oder
- *vor das Ende* zeigen,

hier demonstriert am Beispiel eines `std::vector<std::string>`:*

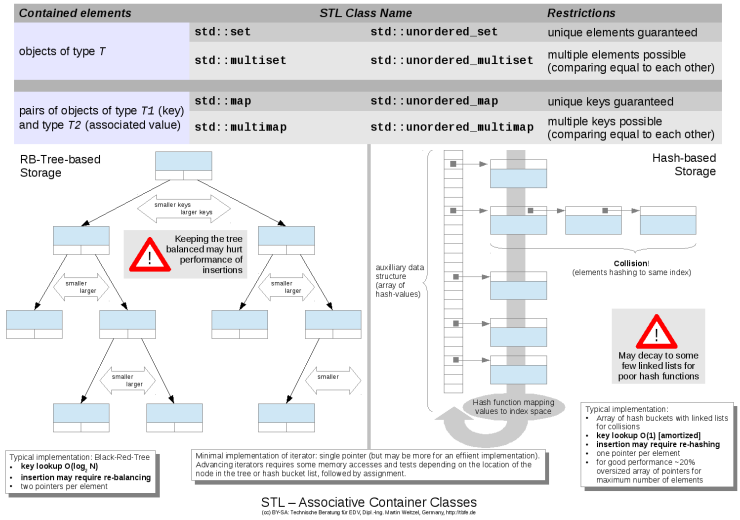
```
std::vector<std::string> sv;
…
assert(sv.size() >= 5);
for (auto it = sv.begin()+3; it != sv.end()-2; ++it)
    … *it …
```

*: Die Korrektheit des Beispiel beruht wesentlich auf der Tatsache, dass die Iteratoren eines `std::vector` zur Kategorie der Random-Access-Iteratoren gehören. Andernfalls liefert die Addition oder Subtraktion einer Ganzzahl bei Anwendung auf einen Iterator ein Compile-Fehler.

Assoziative Container

- Kombination der Varianten

- Speicherverfahren: Red-Black-Tree*
- Speicherverfahren: Hashing



*: Eine stets balancierte Variante des Binärbaums.

Kombination der Varianten

Die verfügbaren assoziativen Container unterscheiden sich in folgenden Aspekten:

- Zur Abspeicherung verwendete Datenstruktur:
  - Red-Black-Tree (Binärbaum) vs. Hashing (neu in C++11)
- Nur Schlüssel oder Schlüssel mit zugeordnetem Wert:
  - `set` vs. `map`
- Eindeutige Schlüssel oder Mehrfachschlüssel erlaubt:
  - normale vs. `multi`-Version

Da die drei Kriterien unabhängig voneinander sind, ergeben sich insgesamt acht (2^3) Kombinationen.

Speicherverfahren Red-Black-Tree

Hierbei handelt es sich um eine permanent balancierte* Variante des Binärbaums.

Die folgende Tabelle verdeutlicht den Zusammenhang zwischen der Tiefe eines balancierten Binärbaums und der Anzahl enthaltener Einträge. Die Tiefe bestimmt vor allem auch die (maximal) nötige Anzahl von Vergleichen, mit der ein gegebener Schlüssel gefunden werden kann.

Tiefe	und minimale .. maximale Zahl der Einträge
1	1 .. 1
2	2 .. 3
5	16 .. 31
7	64 .. 127
10	512 .. 1023
20	524 288 .. 1 048 575 (ca. eine Million)
30	536 870 912 .. 1 073 741 823 (ca. eine Milliarde)
40	549 755 813 888 .. 1 099 511 627 775 (ca. eine Billion)

*: Ein Binärbaum ist balanciert, wenn alle Zweige eine möglichst gleichmäßige Tiefe haben. Ein nicht balancierter Binärbaum entspricht im Extremfall einer einfach verketteten Liste.

Speicherverfahren Hashing

Unter dem Begriff *Hashing* wird (u.a.) ein Abspeicherungsverfahren verstanden, welches – eine gut streuende Hash-Funktion vorausgesetzt – Suchzeiten in der Größenordnung O(1) erlaubt.

> Die Big O Notation wird in der Informatik häufig verwendet, um die (theoretische) Effizienz eines Algorithmus anzugeben.

Es ist die Obergrenze des Zeitbedarfs abhängig von der Datenmenge (N):

Größenordnung	**und typisches Beispiel**
O(1)	Suchzeiten bei Hashing
$O(\log_2(N))$	Suchzeiten in (balanciertem) Binärbaum
O(N)	Suchzeiten in linearer Liste
$O(N \times \log_2(N))$	optimaler Sortieralgorithmus (z.B. Quicksort)
$O(N^2)$	primitiver Sortieralgorithmus (z.B. Bubblesort)
O(n!)	Lösung des "Handlungsreisenden Problems"*

*: The TSP or "Traveling Salesman Problem" is infamous for its combinatoric explosion of possibilities to check, when trying a brute force solution.

Prinzip des Hashing

Diese Verfahren verläuft nach folgendem Grundprinzip:

1. Zu jedem abzulegenden Schlüssel* wird mittels einer Hash-Funktion ein ganzzahliger Wert berechnet.
2. Die Formel zu dieser Berechnung ist so ausgelegt, dass der Wert in einem vorgegebenen Bereich streut.
3. Dieser Wert dient als Index, welcher (für Schlüssel und Datenwert)
    - die Position zur Abspeicherung innerhalb eines Arrays festlegt,
    - dessen Größe dem Streubereich der Hash-Funktion entspricht.

Abhängig von der Anzahl der Schlüssel und Datenwerte (im Verhältnis zur Größe des Arrays) wird es dabei zu Mehrdeutigkeiten kommen: **Früher oder später werden verschiedene Schlüssel auf ein und denselben Index abgebildet.**

*: Bei den verschiedenen Varianten des `std::set` gibt es keinen dem Schlüssel zugeordneten Datenwert – oder anders gesagt: der Datenwert ist in diesem Fall auch der Schlüssel.

Behandlung von Kollisionen

Bildet die Hash-Funktion verschiedene Schlüssel auf ein und denselben Index ab, spricht man von einer Kollission und das auslösende Element wird nun auch *Überläufer* genannt.

> Gut streuende Hashfunktionen liefern bis zu einer ca. 70 .. 80%-igen Belegung des zur Verfügung stehenden Indexraums nur relativ selten Kollisionen.

Prinzipiell können Kollisionen aber ab dem zweiten Eintrag immer auftreten!

- Eine Technik zur Speicherung von Überläufern muss Bestandteil jedes hash-basierten Speicherverfahrens sein.
- Übliche ist, für *jeden* per Hash-Funktion errechneten Index die Möglichkeit einer verketteten Liste vorzusehen, die nach dem ersten aufgetretenen Wert auch alle späteren Überläufer aufnimmt.*

---

*: There are variations, like simply using the next (available) empty slot in the array. Though this simplifies data structures, it may lead to complications (or at least bad performance) if the array gets close to full, especially if many entries may be inserted and removed over time.

Re-Hashing

Ist der Umfang der Datenmenge unbekannt, muss der Indexraum möglicherweise zu einem späteren Zeitpunkt vergrößert werden, um stets gute Performance zu bieten.

- Eine universell verwendbare Klasse wie `unordered_set` oder `unordered_map` (die jeweiligen `multi`-Varianten eingeschlossen) kann dies vollautomatisch tun.
- Zusammen mit der Erweiterung des Indexraums muss die Hash-Funktion angepasst werden, so dass sie die Schlüssel auf einen entsprechend größeren Bereich verteilt.*

> Anschließend müssen alle mit der vorher gültigen Hash-Funktion im alten, kleineren Indexraum verteilten Schlüssel mit der neuen Hash-Funktion wieder im neuen, größeren Indexraum verteilt werden.

*: Für diesen auch Re-Hashing genannten Vorgang gibt es spezielle Techniken, welche die erforderlichen Neuberechnungen hinauszögern (und damit besser über die Zeit verteilen) oder von der Anzahl her reduzieren (und damit den Rechenaufwand insgesamt vermindern).

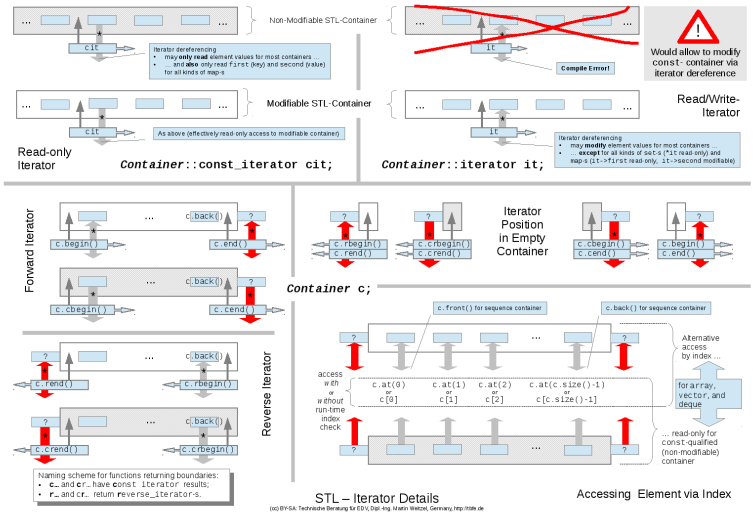
Iterator-Details

- Iterator nur für lesenden vs. …
- … modifizierenden Zugriff

- Vorwärts laufender vs. …
- … rückwärts laufender Iterator

- Iterator-Position im leeren Container

- Index-basierter Zugriff



Iterator für nur lesenden Zugriff

Diese Seite zeigt einige typische Schleifenkonstrukte mit einem Iterator für den nicht-modifizierenden Zugriff, am Beispiel einer Container-Klasse `C` und eines entsprechenden Container-Objekts `c`:

In C++11 mit neuer Bedeutung von `auto` und neuer Initialisierungssyntax:

```
for (auto cit{c.cbegin()}; cit != c.cend(); ++cit)
    … *cit …
```

In C++98 mit expliziter Angabe des Iterator-Typs:

```
for (C::const_iterator cit = c.begin(); cit != c.end(); ++it)
    … *cit …
```

[!] Den dereferenzierten Iterator zur Modifikation eines Container-Elements zu verwenden – also in der Form `*cit = …` – ist ein Compilezeit-Fehler.

Iterator für modifizierenden Zugriff

Diese Seite zeigt einige typische Schleifenkonstrukte mit einem Iterator für den potenziell modifizierenden Zugriff, am Beispiel einer Container-Klasse `C` und eines entsprechenden Container-Objekts `c`:

In C++11 mit neuer Bedeutung von `auto` und neuer Initialisierungssyntax:

```
for (auto it{c.begin()}; it != c.end(); ++cit) {
    … *it … // Zugriff sowohl lesend als
    *it = … // auch modifizierend erlaubt
}
```

In C++98 mit expliziter Angabe des Iterator-Typs:

```
for (C::iterator it = c.begin(); it != c.end(); ++it) {
    … *it … // Zugriff sowohl lesend als
    *it = … // auch modifizierend erlaubt
}
```

Vorwärts laufender Iterator

Diese Seite zeigt weitere typische Schleifenkonstrukte für den potenziell modifizierenden Zugriff am Beispiel einer Container-Klasse `C` und eines entsprechenden Container-Objekts `c`:

In C++98 mit expliziter Angabe des Iterator-Typs:

```
for (C::iterator it = c.begin(); it != c.end(); ++it) {
    … *it … // Zugriff sowohl lesend als
    *it = … // auch modifizierend erlaubt
}
```

In C++11 konkurriert dies auch mit der bereichsbasierten Schleife:*

```
for (auto &v : c) {
    … v … // Zugriff sowohl lesend als
    v = … // auch modifizierend erlaubt
}
```

*: Auch wenn es hier keinen explizit sichtbaren Iterator gibt, verwendet die bereichsbasierte Schleife in ihrer internen Implementierung die Iterator-Schnittstelle der Klasse des beteiligten Container-Objekts.

Rückwärts laufender Iterator

Diese Seite zeigt das typische Schleifenkonstrukt für den potenziell modifizierenden Zugriff am Beispiel einer Container-Klasse `C` und eines entsprechenden Container-Objekts `c`:

In C++98 mit expliziter Angabe des Iterator-Typs:

```
for (C::reverse_iterator it = c.rbegin(); it != c.rend(); ++it) {
    … *it … // Zugriff sowohl lesend als
    *it = … // auch modifizierend erlaubt
}
```

Die einzige, in C++11 mögliche Vereinfachung ist hier die Verwendung von `auto` für den Typ des Iterators.

```
for (auto it = c.rbegin(); it != c.rend(); ++it) …
```

Iterator-Position im leeren Container

Für einen leeren Container sind die Iteratoren in Beginn- und Endposition einander gleich. Alle der folgenden Schleifen werden daher niemals durchlaufen, wenn `c` leer ist:

```
// mit modifizierbaren Vorwaerts-Iterator:
for (auto it = c.begin(); it != c.end(); ++it) …
…
// mit nicht-modifizierbaren Vorwaerts-Iterator:
for (auto it = c.cbegin(); it != c.cend(); ++it) …
…
// mit modifizierbaren Rueckwaerts-Iterator:
for (auto it = c.rbegin(); it != c.rend(); ++it) …
…
// mit nicht-modifizierbaren Rueckwaerts-Iterator:
for (auto it = c.crbegin(); it != c.crend(); ++it) …
```

Da die von den verschiedenen Varianten der `begin()`-Funktion gelieferten Iterator-Typen unterschiedlich sind, muss der Vergleich auf die Endposition mit derjenigen Variante der `end()`-Funktion erfolgen, die den selben Iterator-Typ liefert.

Index-basierter Zugriff

Für die Container

- `std::array`,
- `std::vector` und
- `std::deque`

besteht alternativ zum Durchlaufen mittels Iterator auch die Möglichkeit des indexbasierten Zugriffs. Die Basis-Variante – wieder dargestellt am Beispiel eines Containers der Klasse `C` und einem entsprechenden Container-Objekt `c` – sieht wie folgt aus :

```
for (C::size_type i = 0; i < c.size(); ++i) {
    … c[i] …    // Zugriff ohne Index-Pruefung
    … c.at(i) … // Zugriff mit Index-Pruefung
}
```

(i) Die Verwendung des in allen STL-Containern definierten Typs `size_type` für die Indexvariable vermeidet eine eventuelle Warnung beim Vergleich mit dem Rückgabewert der `size()`-Member-Funktion, da deren Ergebnis ebenfalls diesen Typ hat.

Iteratoren für Assoziative Container

Eine Besonderheit bei assoziativen Container ist, dass auch über normale Iteratoren – also nicht nur bei `const_iterator` – auf **Elemente** von

- `std::set`,
- `std::multiset`,
- `std::unordered_set` und
- `std::unordered_multiset`

sowie auf **Schlüssel** von

- `std::map`,
- `std::multimap`,
- `std::unordered_map` und
- `std::unordered_multimap`

ausschließlich **lesend** zugegriffen werden kann.*

*: Der Zugriff auf den **Wert** (gemäß nächster Seite) ist bei den diversen `map`-Varianten wie üblich über `…::const_iterator` nur lesend und über `…::iterator` sowohl lesend wie auch schreibend möglich.

Iteratoren für Maps

Die Iteratoren für `std::map`, `std::multimap`, `std::unordered_map` und `std::unordered_multimap` müssen den separaten Zugriff auf beides, **Schlüssel** und **Werte** ermöglichen.

Dies geschieht über ihren speziellen dereferenzierten Typ, ein `std::pair` über

- dessen erste Komponente (`first`) der Schlüssel und
- dessen zweite Komponente (`second`) der Wert erreicht wird.

Als Beispiel ein Programmfragment zum Zählen von Worthäufigkeiten:*

```
std::string word;
std::map<std::string, int> wfreq;
while (std::cin >> word)
    ++wfreq[word];
for (auto it = wfreq.cbegin(); it != wfreq.cend(); ++it)
    std::cout << it->first << ": " << it->second << std::endl;
```

*: The original "pure C" version in *K&R: The C Programming Language* has roughly 100 lines ...

Bereichsbasierte Schleifen für Maps

Die neue, bereichsbasierte `for`-Syntax von C++11 kann ebenfalls über Maps iterieren:

```
for (auto e : wfreq)
    std::cout << e.first << ": " << e.second << std::endl;
```

Eine Referenz erspart das Kopieren des jeweiligen Elements:

```
for (auto &e : wfreq)
    std::cout << e.first << ": " << e.second << std::endl;
```

Eine konstante Referenz erlaubt nur noch den Lesezugriff ...

```
for (const auto &e : wfreq)
    std::cout << e.first << ": " << e.second << std::endl;
```

... und schützt damit vor unbeabsichtigten Modifikationen, z.B. wenn statt eines Vergleichs versehentlich die Zuweisung geschrieben würde:

```
if (e.second = 100) … // ERROR
```

Praktikum