C++ FOR
(Dienstagnachmittag)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

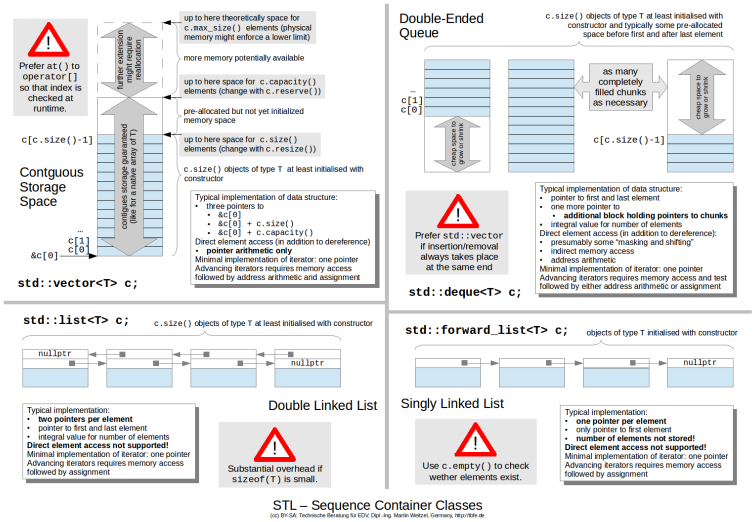
Die Besprechung der Musterlösung(en) erfolgt zu Beginn des folgenden Vormittags.

Sequenzielle Container der STL

- 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 - typischerweise zuzüglich etwas Reserve - wird auf dem Heap angelegt*

Ein Object der Klasse `std::vector`` besteht in der typischen Implementierung aus drei Zeigern.

Links zur ausführlichen Online-Dokumentation:
http://www.cplusplus.com/reference/vector/
http://en.cppreference.com/w/cpp/container/vector

Vorteile eines `std::vector`

Effizient unterstützt werden: *Elemente einfügen und entnehmen* am selben *Ende* Wahlfreier Zugriff (mit oder ohne Indexprüfung) * Move-Konstruktor und -Assignment

Durch die Speicheranforderung auf Vorrat ist auch der inkrementelle Aufbau eines `std:vector` durch wiederholtes Anfügen neuer Elemente eine relativ effiziente Operation.

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)

Doppelt verkettete Listen

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

Ein Object der Klasse `std::list`` besteht in der typischen Implementierung aus zwei Zeigern (erstes und letztes Element) und einer Ganzzahl (Anzahl der Elemente).

Links zur ausführlichen Online-Dokumentation:
http://www.cplusplus.com/reference/list/
http://en.cppreference.com/w/cpp/container/list

Vorteile einer `std::list``

Effizient unterstützt werden: *Elemente einfügen und entnehmen* auch an unterschiedlichen *Enden* Einfügen und entnehmen von Elementen in der Mitte. * Move-Konstruktor und -Assignment

Einschränkungen einer `std::list``

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

Durch die beiden pro Element notwendigen Zeiger hat eine Liste einen vergleichsweisen hohen Overhead in Bezug auf den Speicherbedarfs, wenn der Nutzdatentyp sehr klein ist (Extremfall `char`, prinzipiell aber auch alle anderen von C++ unterstützten ganzzahligen Typen).

Einfach verkettete Listen

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

Ein Object der Klasse `std::forward_list`` besteht in der typischen Implementierung aus gemau einem Zeiger - **die Anzahl der Elemente wird nicht gezählt** (anders als bei derstd::list`).

Links zur ausführlichen Online-Dokumentation:
http://www.cplusplus.com/reference/forward_list/
http://en.cppreference.com/w/cpp/container/forward_listector

Vorteile einer `std::forward_list`

Effizient unterstützt werden: *Elemente einfügen und entnehmen* am selben *Ende* Move-Konstruktor und -Assignment

Sofern die gewünschte Position durch einen Iterator bereits bestimmt wurde, ist auch das Einfügen und Entnehmen in der Mitte performant, besitzt allerdings eine - im Vergleich mit den anderen sequenziellen Containern - ungewöhnliche Semantik, die am Namen der betreffenden Operation deutlich wird: `insert_after`*

Einschränkungen einer `std::forward_list`

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

Deque

Hierbei handelt es sich quasi um einen "in Abschnitte zerstückelten `std::vector`", welcher grob zusammengefasst sich wie folgt von `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 nur geringfügig weniger performant.

Anders als der `std::vector` ist die `std::deque` damit auch eine gute Basis für eine FIFO-Speicherung (Warteschlange).

*: Ein diesbezüglicher Anwendungsfehler wäre es, die Adresse des ersten Elements zu bestimmen. Da `std::deque::operator[]()` und `std::deque::at()` technisch gesehen eine Referenz liefern ist die Anwendung des Adress-Operators syntaktisch zulässig und leider nicht durch eine entsprechende Fehlermeldung des COmpilers erkennbar. auszuschließen.

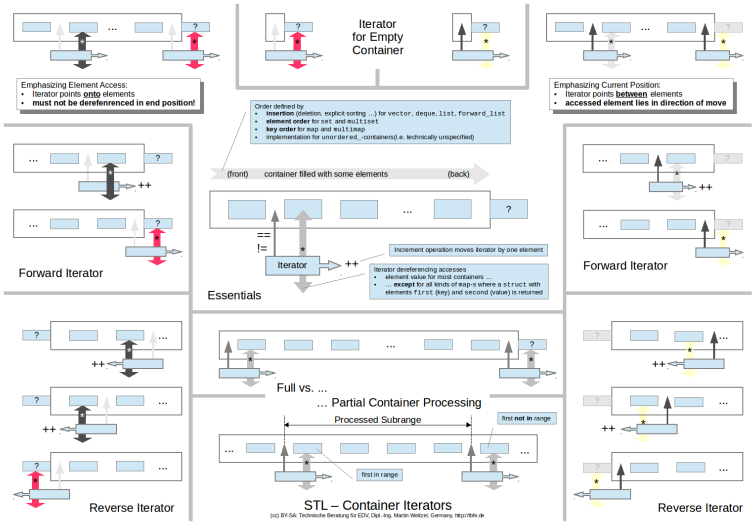
Grundlegendes zu Iteratoren

- 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, die vorwiegend dann zur Anwendung kommen, wenn auf die Elemente eines Containers oder eines Teilbereichs sequenziell nacheinander zugegriffen werden soll.

Iterator *auf* Elemente zeigend

Man kann Iteratoren grafisch als *auf Elemente zeigend* veranschaulichen, bekommt dann ellerdings ein Erklärungsproblem mit der Iterator-Endposition und auch 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 dann **nicht mehr derefenziert** werden darf.

Iterator *zwischen* Elemente zeigend

In dieser grafischen Veranschaulichung werden die zuvor beschriebenen Probleme zwar vermieden, dafür taucht dann die Frage auf, was die Iteratoren bei der Dereferenzierung liefern.

> 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 er dann weitergesetzt würde, wird bei der Derefenzierung geliefert.

Damit wird zugleich klar, dass ein Iterator in End-Position nicht derefenziert werden darf, denn es gibt ja auch keine Position, zu der er mit ++ weitergesetzt werden könnte.

Vorwärts-Iterator *auf* Elemente zeigend

Bei Dereferenzierung liefern Vorwärts-Iteratoren eine Refererenz 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 jenseits der Container-Endes 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 -- dann natürlich Richtung Container-Anfang.

Vorwärts-Iterator *zwischen* Elemente zeigend

Bei Dereferenzierung liefern Vorwärts-Iteratoren eine Refererenz auf dasjenige Element im Container, über das sie mit der nächsten ++-Operation weitergeschaltet 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 -- dann 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 **jenseits des Container-Anfangs** noch zeigen kann.

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

Rückwärts-Iteratoren bewegen sich mit ++ vom Container-Ende zu dessen Anfang.*

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

Rückwärts-Iterator *zwischen* Elemente zeigend

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

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

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

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

Alle Elemente im Container

Um über alle Elemente in einem Container zu iterieren, wird typischerweise folgende Konstruktion verwendet (hier hezeight am Beispiel einer `std::list`):

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

Mit C++98 statt `auto` der exakte Iterator-Typ angegeben werden, wozu sich der Übersichtlichkeit und Wartungsfreundlichkeit halber eine Typdefinition 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 …
```

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 gezeigt am Beispiel eines `std::vector<std::string>`:*

```
std::vector<std::string> sv;
…
assert(sv.size() >= 4);
for (auto it = sv.begin()+3; it != sv.end()-2; ++it)
 …
```

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

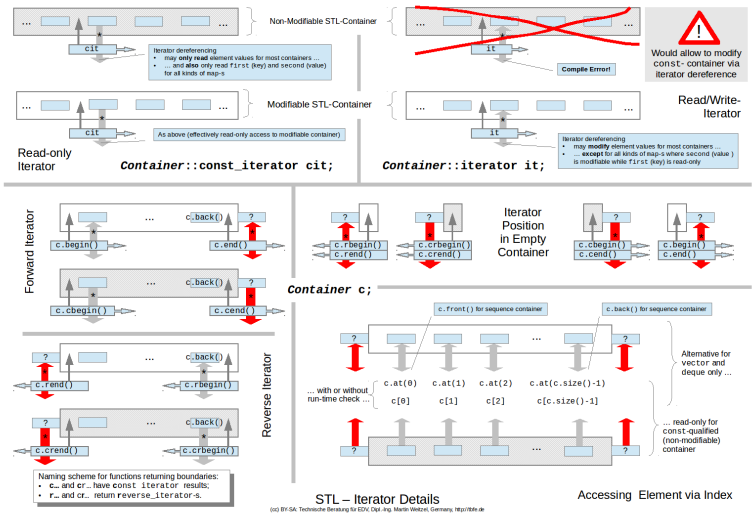
Iteratoren im Detail

- Iterator nur für lesenden vs. ...
- ... modifzierenden Zugriff

- Vorwärts laufender vs. ...
- ... rückwärts laufender Iterator

- Iterator-Position im leeren Container

- Index-basierter Zugriff



Iteratoren für nur lesenden Zugriff

Diese Seite zeigt einige typische Schleifenkonstrukte mit Iterator für nicht-modifizierenden Zugriff am Beispiel einer Container-Klasse C und eines entsprechenden 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 Iteratortyps:

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

[!] Den derefenzierten Iterator zur Modifikation eines Container-Elements zu verwenden - also in der Form `*cit = …` - wird als Compilezeit-Fehler bewertet.

Iterator für modifzierenden Zugriff

Diese Seite zeigt einige typische Schleifenkonstrukte mit Iterator für (potenziell) modifizierenden Zugriff am Beispiel einer Container-Klasse `C` und eines entsprechenden 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 modfizierend erlaubt
}
```

In C++98 mit expliziter Angabe des Iteratortyps:

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

Vorwärts laufender Iterator

Diese Seite zeigt das typische Schleifenkonstrukte mit Iterator für (potenziell) modifizierenden Zugriff am Beispiel einer Container-Klasse C und eines entsprechenden Objekts c:

In C++98 mit expliziter Angabe des Iteratortyps:

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

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

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

*: Hierbei gibt es zwar keinen explizit sichtbaren Iterator, die bereichsbasierte Schleife stützt sich jedoch in ihrer internen Implementierung auf die Iterator-Schnitstelle der Klasse des angegebenen Container-Objekts.

Rückwärts laufender Iterator

Diese Seite zeigt das typische Schleifenkonstrukte mit Iterator für (potenziell) modifizierenden Zugriff am Beispiel einer Container-Klasse `C` und eines entsprechenden Objekts `c`:

In C++98 mit expliziter Angabe des Iteratortyps:

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

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

> (i) Eine spezielle, rückwärts laufende Form der bereichsbasierten Schleife ist in C++11 nicht verfügbar.*

*: Mit Hilfe von Templates ließe sich jedoch ein Adapter schreiben, mit dem die folgende Syntax möglich wird: `for (auto &v : my::reverse_adapter(c)) …`

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 != .cend(); ++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 diversen Iterator-Typen, die von den verschiedenen Varianten der `begin()`-Funktion geliefert werden, jeweils unterschiedlichen sind, muss der Vergleich auf die Endposition mit derjenigen `end()`-Funktion erfolgen, die denselben Iterator-Typ liefert.

Index-basierter Zugriff

Für die Container

- `std::vector` und
- `std::deque`

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

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

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

Übung

Ziel der Aufgabe:

Eine kleine Serie von STL-Beispielen soll

- zunächst analysiert und
- dann in einigen Details modifiziert

werden. Dabei besteht auch die Gelegenheit, in C++11 neu hinzugekommene Sprachmechanismen zu verwenden.

Weitere Details werden vom Dozenten anhand des Aufgabenblatts sowie der vorbereiteten Eclipse-Projekte erläutert.