C++ FOR (Mittwochvormittag)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

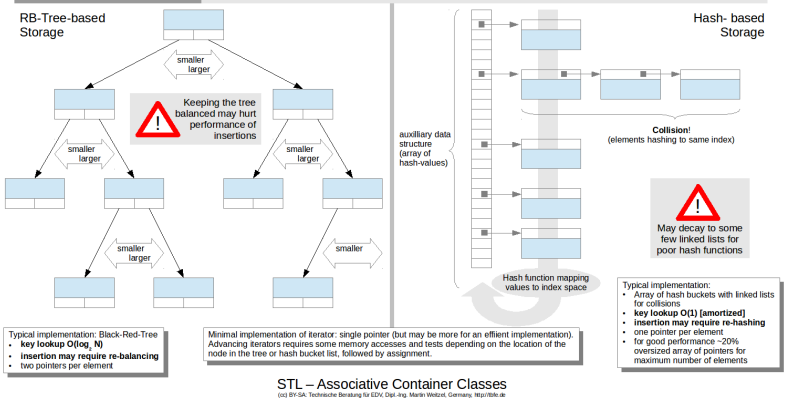
Die Besprechung der Musterlösung(en) erfolgt im direkten Anschluss an die Mittagspause.

Assoziative Container der STL

- Kombination der Varianten

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

Contained elements	*STL Class Name*		*Restrictions*
objects of type T	`std::set`	`std::unordered_set`	unique elements guaranteed
	`std::multiset`	`std::unordered_multiset`	multiple elements possible (comparing equal to each other)
pairs of objects of type T1 (key) and type T2 (associated value)	`std::map`	`std::unordered_map`	unique keys guaranteed
	`std::multimap`	`std::unordered_multimap`	multiple keys possible (comparing equal to each other)



STL – Associative Container Classes

*: 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.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

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 Vergleiche, 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 unbalancierter Binärbaum entspricht im Extremfall einer einfach verketteten Liste.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

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 Obergrenze des Zeitbedarfs eines Algorithmus in Abhängigkeit von der bearbeiteten Datenmenge (N) anzugeben.

Die folgende Tabelle soll nur eine grobe Orientierung geben:

Größenordnung	**und typisches Beispiel**
$O(1)$	Konstante Zeit unabhängig von der Datenmenge
$O(\log_2(N))$	Suchzeiten in (nicht entartetem) 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)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Prinzip des Hashing

Diese Verfahren verläuft nach folgendem Grundprinzip:

1. Ausgehend von 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 nun als Index, welcher die Position zur Abspeicherung innerhalb eines Arrays festlegt, dessen Gesamtgröße wiederum dem Streubereich der Hash-Funktion entspricht.

> Abhängig von der Anzahl der Datenwerte und der Größe des Streubereichs gibt es meist weder theoretisch noch praktisch eine eindeutige Abbildung des Schlüssels auf den Streubereichn.

*: Bei den verschiedenen Varianten des `std::set` gibt es keinen dem Schlüssel zugeordneten Datenwert - der Wert ist der Schlüssel.

Entstehung von Überläufern

Gut streuende Hashfunktionen liefern bis zu einer ca. 80%-igen Belegung des zur Verfügung stehenden Index-Raumes nur selten Überläufer.

> Bildet die Hash-Funktion verschiedene Schlüssel auf ein und denselben Index ab, spricht man von einem Überläufer.

Prinzipiell können Überläufer aber ab dem zweiten Eintrag immer auftreten.

- Eine Strategie zur Behandlung von Überläufern muss Bestandteil jedes hash-basierten Datenspeicherungsverfahrens sein.
- Übliche ist, für *jeden* per Hash-Funktion errechneten Index die Möglichkeit einer verketteten Liste vorzusehen, die den ersten aufgetretenen Wert zusammen mit allen späteren Überläufern aufnimmt.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Re-Hashing

Ist der Umfang der Datenmenge unbekannt, muss der Indexraum möglichwerweise 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.
- Dazu muss die Hash-Funktion angepasst werden, so dass diese künftig in einen dem vergrößerten Datenbereich entsprechend Indexbereich streut.

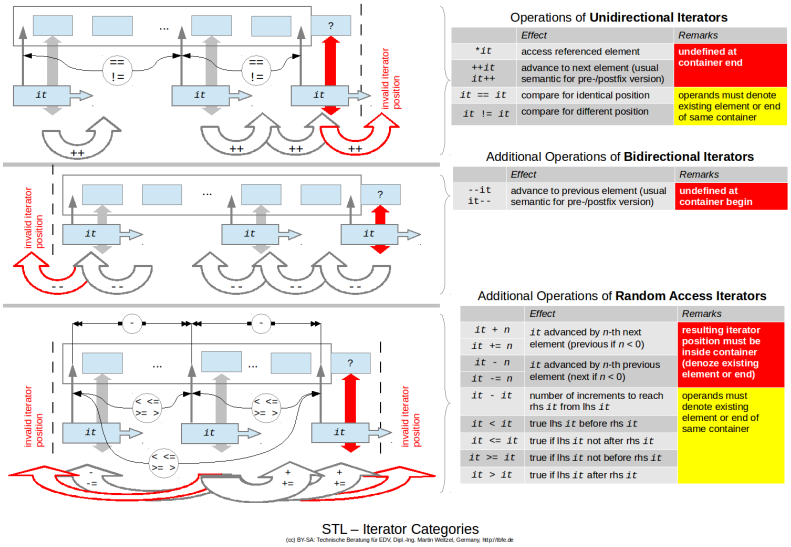
> Anschließend müssen alle mit der vorher gültigen Hash-Funktion erzeugten Zuordnungen von Schlüssel zu (Streubereichs-) Index mit der neuen Hash-Funktion neu berechnet werden.*

*: Mit Hilfe spezieller Techniken, welche die Neuberechnung auf die Hälfte der vorhandenen Einträge reduziert, lässt sich der Aufwand für das Re-Hashing halbieren.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Iteratoren-Kategorien

- Unidirektional-Iteratoren
- Bidirektional-Iteratoren
- Iteratoren mit wahlfreiem Zugriff



Unidirektional-Iteratoren

Iteratoren, welche der Kategorie der Unidirektional-Iteratoren angehören, können sich - ihrem Namen entsprechend - nur in eine Richtung bewegen, nämlich vom Anfang eines Containers zu dessen Ende hin.

Um pure Unidirektional-Iteratoren handelt es sich bei

- `std::forward_list<T>::iterator`

die `const_`-Variante eingeschlossen.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Bidirektional-Iteratoren

Iteratoren, die der Kategorie der Bidirektional-Iteratoren angehören, können sich - ihrem Namen entsprechend - in zwei Richtung bewegen, nämlich vom Anfang eines Containers zu dessen Ende hin und umgekehrt.

Um Bidirektional-Iteratoren handelt es sich bei*

- `std::list<T>::iterator`
- sowie den Iteratoren der assoziativen Container,

die jeweiligen `const_`, `reverse_` und `const_reverse_`-Varianten eingeschlossen.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Iteratoren mit wahlfreiem Zugriff

Iteratoren, die der Kategorie der Random-Access-Iteratoren angehören, können – ihrem Namen entsprechend - innerhalb des zugehörigen Containers effizient wahlfrei zugreifen.

Neben der Addition und Subtraktion von Ganzzahlen (womit der Iterator über größere Distanzen weitergschaltet wird) kann über Differenzbildung auch die Anzahl der Elemente bestimmt werden, die sich zwischen zwei Iteratorpositionen befinden.

Um Random-Access-Iteratoren handelt es sich bei*

- `std::array<T, N>::iterator`,
- `std::vector<T>::iterator` und
- `std::deque<T>::iterator`

die jeweiligen `const_`, `reverse_` und `const_reverse_`-Varianten eingeschlossen.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Grenzprüfung bei Random-Access-Iteratoren

Hier ist zu beachten, dass die Minimal-Anforderungen des C++-Standards so gefasst sind, dass eine hoch-effiziente Implementierung möglich ist, auch wenn dies auf Kosten der Sicherheit geht.

> **[!]** Wenn Operationen, an denen Iteratoren mit wahlfreiem Zugriff beteiligt sind, eine (neue) Iterator-Position außerhalb des Containers liefern, ist das Ergebnis undefiniert.

Gültige Positionen sind dabei auch die möglichen Endstellungen (am Container-Ende für normale Iteratoren bzw. an Container-Anfang für Reverse-Iteratoren).

> **[!]** Für ein definiertes Ergebnis bei der Differenzbildung müssen die beteiligten Iteratoren gültige Positionen (inklusive der Endpositionen) im selben Container haben.

Sichergestellt ist aber, dass zwei Iteratoren, die in verschiedene Container zeigen, niemals als "gleich" angesehen werden.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

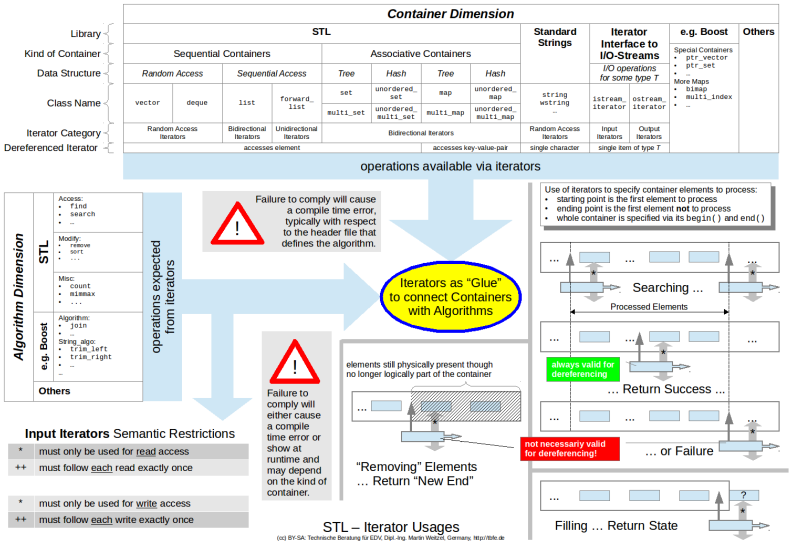
Iterator-Konventionen

- Iteratoren verbinden ...
- ... Container mit ...
- ... Algorithmen

- Input-Iteratoren
- Output-Iteratoren

- Erfolgreiche und ...
- ... gescheiterte Suche

- Füllstands- und ...
- ... Löschenanzeige



Iteratoren als verbindendes Element

Mit Iteratoren als Bindeglied zwischen Containern und Algorithmen sind bei

- ca. 40 Algorithmen (-Familien) und
- 12 Containern im ISO-Standard

keine 480 (= 12 × 40) Implementierungen notwendig.*

*: Unter Berücksichtigung der Tatsache, dass nicht alle Algorithmen für alle Container sinnvoll sind, wird die Zahl der erforderlichen Implementierungen um die hundert sein.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Container- und Algorithmen-Achse

In C++98 wurden drei sequenzielle und vier assoiative Container standardisiert.

Mit C++11 sind es nun insgesamt vier sequenzielle und acht assoziative Container.

(C++14 hat keine weiteren Container hinzugefügt.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Algorithmen-Achse

Hier gibt es - je nach zählweise - im Rahmen des ISO&ANSI-Standards gut drei Dutzend Einträge.

Viele Algorithmen kommen in "Familien" wie etwa:

- Grundlegende Variante
- Variante endend mit `_if` für flexible Bedingungsprüfung
- Variante endend mit `_copy` für Ergebnisablage in neuem Container
- Soweit sinnvoll die Kombination von Obigem

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Input-Iterator-Kategorie

Input-Iteratoren sind *abwechselnd*

- für den *Lese-Zugriff* zu dereferenzieren (*)
- und *weiterzuschalten* (++).

[!] Bei Nichtbeachtung dieser Anwendungsvorschrift betrifft das Fehlverhalten oft nicht alle Arten von Containern, so dass Verstöße nur bei ausreichenden Tests auffallen.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Output-Iterator-Kategorie

Output-Iteratoren sind *abwechselnd*

- für den *Schreib-Zugriff* zu derefenenzieren (*)
- und *weiterzuschalten* (++).

[!] Bei Nichtbeachtung dieser Anwendungsvorschrift betrifft das Fehlverhalten oft nicht alle Arten von Containern, so dass Verstöße nur bei ausreichenden Tests auffallen.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Erfolgreiche Suche

Beim Suchen (und verwandten Container-Operationen) wird der Erfolg dadurch angezeigt, dass der als Rückgabewert gelieferte Iterator

- auf eine gültige Position

im zur Bearbeitung übergebenen Daten-Bereich zeigt.

[!] Beim Suchen in einem leeren Container wird stets die Endposition geliefert.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Fehlgeschlagene Suche

Beim Suchen wird der Misserfolg damit angezeigt, dass der als Rückgabewert gelieferte Iterator

- immer auf die Endposition

des zur Bearbeitung übergebenen Bereichs zeigt.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Zustandsanzeige nach Füllen

Beim Füllen eines Containers ist es üblich, dass Algorithmen

- den (neuen) Füllstand des Containers

durch einen entsprechenden Iterator als Rückgabewert anzeigen.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Löschanzeige durch neues (logisches) Ende

Da die Algorithmen für eine große Zahl von Containern funktionieren sollen, finden gibt es einige aus pragmatischen Sicht sinnvolle aber dennoch "merkwürdige"* Besonderheiten.

Eine davon betrifft die Frage, wie mit aus einem Container gelöschten ELementen grundsätzlich zu verfahen ist, wenn der "löschende" Algorithmus den Container nicht physisch verkleinern kann.

Beim "logischen Löschen" von Elementen aus Containern

- werden die enthaltenen Elemente lediglich anders angeordnet, und
- ein Iterator, welcher das "neue Ende" bezeichnet, wird als Rückgabewert geliefert.

> In den späteren Code-Fragmenten dieses Kapitels folgen konkrete Beispiele.

*: ... des Merkens würdige ...

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Flexibilität durch Iteratoren

Die konsequente Verwendung von Iteratoren bei der Implementierung aller STL-Algorithmen verschafft eine besondere Flexibilität.

Iteratoren sind in der Regel einfache (Helfer-) Klassen, welche gemäß dem Container, für den sie jeweils zuständig sind, eine vollkommen unerschiedliche Implementierung besitzen können.

Damit können die Daten, die ein STL-Algorithmus bearbeitet, aus ganz unterschiedlichen Quellen stammen und ebenso die gelieferten Ergebnisse an ganz unterschiedlichen Zielen abgelegt werden.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Beispiel: Implementierung des copy-Algorithmus

Zum besseren Verständnis, wie mittels Iteratoren die Algorithmen eine besondere Flexibilität erzielt wird, ist ein Blick auf eine mögliche Implementierung des copy-Algorithmus hilfreich:.*

```
template<typename T1, typename T2>
T2 my_copy(T1 from, T1 upto, T2 dest) {
    while (from != upto)
        *dest++ = *from++;
    return dest;
}
```

*: Damit dieser Algorithmus ggf. per *Copy&Paste* in ein Programm übernommen und ausprobiert werden kann, wurde er vorsorglich `my_copy` genannt, so dass auch bei `using namespace std;` ein Konflikt mit `std::copy` ausgeschlossen ist.

Kopieren der Elemente eines `std::set` in einen `std::vector`

Die gerade gezeigte Funktion kann problemlos zwischen zwei unterschiedlichen Containern kopieren:

```
std::vector<int> v;
std::set<int> s;
…
v.resize(s.size());
std::copy(s.begin(), s.end(), v.begin());
```

Da über einen normalen Iterator - wie von `v.begin()` geliefert - nur bereits vorhandene Elemente überschrieben aber keine neuen Elemente angefügt werden können, ist es wichtig, den Vector vor dem Kopieren auf die notwendige (Mindest-) Größe zu bringen.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Anhängen an Vektoren

Hilfsklasse: `back_insert_iterator`

Ohne den aufnehmenden Vektor zuvor auf eine passende Größe zu bringen, lässt sich eine sichere Variante des Kopierens erreichen, indem man die neuen Elemente stets mit `push_back` anfügt:*

```
std::copy(s.begin(), s.end(),
          std::back_insert_iterator<std::vector<int>>(v)
);
```

*: Das Umbrechen längerer Zeilen bei der Argumentübergabe erscheint sinnvoll, um die Argumente nach ihrer Bedeutung zu gruppieren. Die dabei verwendete asymmetrische Klammernsetzung ist sicher Geschmackssache, entspricht aber dem oft auch bei geschweiften (Block-) Klammern angewandten Stil.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Hilfsfunktion: `back_inserter`

Die - redundant erscheinende - Angabe von `int` als Datentyp des Containers `v` ist technisch der Tatsache geschuldet, dass ein (temporäres) Objekt der Template-Klasse `std::back_insert_iterator` erzeugt werden muss und der Konstrukor aus dem Argumenttyp nicht auf den Instanziierungstyp schließen kann.

Eine Hilfsfunktion* vermeidet diese Redundanz:

```
std::copy(s.begin(), s.end(), std::back_inserter(v));
```

---

*: Diese Funktion ist zwar Bestandteil der Standardbibliothek, muss also nicht selbst implementiert werden, ihre Implementierung ist aber insofern interessant, als die zugrundeliegende Technik in ähnlich gelagerten Fällen zur Vermeidung redundanter Typangaben eingesetzt werden kann:

```
template<typename T>
 inline std::back_inserter_iterator<T> std::back_inserter(T &c) {
     return std::back_insert_iterator<T>(c);
 }
```

Am Anfang oder Ende sequenzieller Container einfügen

Selbstverständlich funktionert der `std::back_inserter_iterator` bzw. die Hilfsfunktion `std::back_inserter` auch für die Klassen

- `std::list` und
- `std::deque`

und es existiert auch ein `std::front_insert_iterator` sowie eine Hilfsfunktion `std::front_inserter` welche mit den Klassen

- `std::list`,
- `std::deque` und
- `std::forward_list`

verwendbar ist.*

*: Allgemeiner ausgedrückt ist die Anforderung beim `std::back_insert_iterator`, dass für den zur Instanziierung verwendeten Container eine Member-Function `push_back` existiert, und beim `std::front_insert_iterator` entsprechend, dass `push_front` existiert.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

In assoziative Container einfügen

Hilfsklasse: `insert_iterator`

Hiermit können z.B. alle Elemente einer Liste wie folgt in ein `std::set` übertragen werden (wobei Duplikate natürlich nicht übernommen werden*):

```
std::list<MyClass> li;
…
std::set<MyClass> s;
std::copy(li.begin(), li.end(),
          std::insert_iterator<std::set<MyClass>>(s)
);
```

*: Es sei denn, es handelt sich um ein `std::multiset`.

Hilfsfunktion: `inserter`

Auch hier gibt es eine Hilfsfunktion zur Vermeidung der (eigentlich redundanten) Angabe des Container-Typs:

```
std::set<MyClass> s;
std::copy(li.begin(), li.end(),
          std::inserter<std::set<MyClass>>(s, s.begin())
);
```

Etwas ungewöhnlich ist hier das zusätzliche Argument. Dessen Zweck ist es, einen Optimierungshinweis zu geben, den der Insert-Iterator ggf. verwenden würde, um die Suche nach der Einfügeposition abzukürzen. Üblicherweise wird – wenn kein besser Hinweis gegeben werden kann – der Beginn-Iterator des jeweiligen Sets verwendet.

---

*: Da es sich nur um einen Hinweis handelt, ist die Angabe unkritisch, auch wenn man einen falschen Hinweis gibt (oder keinen passenden Hinweise angibt, obwohl einer existiert). Dann bleibt lediglich eine Optimierungs-Chance ungenutzt …

Stream-Iteratoren

So wie ein Back- oder Front-Insert-Iterator letzten Endes (bei der derefenzierten Zuweisung) eine `push_back`- bzw. `push_front`-Operation für den betroffenen Container ausführt, lassen sich auch andere Operationen in den (überladenen) Operatoren spezieller Iteratoren unterbringen.

In Stream schreiben mit `std::ostream_iterator`:

Das folgende Beispiel kopiert den Inhalt einer `std::forward_list` auf die Standard-Ausgabe und fügt nach jedem Wert ein Semikolon ein:

```
std::forward_list<long> fli;
…
std::copy(fli.begin(), fli.end(),
          std::ostream_iterator<long>(std::cout, ";")
);
```

Aus Stream lesen mit `std::istream_iterator`:

Das folgende Beispiel* kopiert Worte von der Standardeingabe bis EOF in eine `std::forward_list`:

```
std::forward_list<std::string> fli;
copy(std::istream_iterator<std::string>(std::cin),
     std::istream_iterator<std::string>(),
     std::front_inserter(fli)
);
```

*: Zumeist ist es zwar ausreichend, das obige Beispiel mehr oder weniger "kochrezeptartig" und ggf. sinngemäß verändert anzuwenden, dennoch taucht häufig die Frage auf, wie es denn "hinter den Kulissen" funktioniert. Daher als Hinweis das folgende:
Offensichtlich gibt es zwei Konstruktoren, von denen einer ein `std::istream`-Objekt als Argument erhält. Ein auf diese Weise erzeugter Input-Stream-Iterator liefert im Vergleich mit einem per Default-Konstrukor erzeugten Gegenstück zunächst `false`.
Die Operationen `*` und `++` werden auf eine Eingabe mit `operator>>` abgebildet, wobei die eingelesene Variable genau Typ hat, der zur Instanziierung der Template verwendet wurde. Sobald der Eingabestrom den "good"-Zustand verlässt, liefert der damit verbundene Istream-Iterator beim erneuten Vergleich mit dem per Default-Konstruktor erstellten `true`.

Zeiger als Iteratoren

Da die für Iteratoren implementierten Operationen syntaktisch wie semantisch denen für Zeiger entsprechen, können auch klassische Arrays bearbeitet werden.

Kopieren AUS klassischem Array

Mit einem klassischen Array

```
double data[100]; std::size_t ndata{0};
```

und der Annahme, dass nach dessen Befüllen die ersten `ndata` Einträge tatsächlich gültige Werte enthalten:

```
std::copy(&data[0], &data[ndata], … );
```

Oder:

```
std::copy(data, data + ndata, … );
```

Kopieren IN klassisches Array

Mit einem klassischen Array als Ziel der Kopie:

```
const auto endp = copy( … , … , &data[0]);
```

Umrechnung des Füllstatus-Iterators in Anzahl gültiger Elemente:

```
ndata = endp - data; // oder: ... endp - &data[0]
```

Hier findet allerdings keinerlei Überlaufkontrolle statt!

[!] Enthält der ausgelesene Container mehr Elemente als `data` aufnehmen kann, könnten **dahinter** (= an größeren Adressen im Speicher) liegende Variablen überschrieben werden.

Kopieren zwischen unterschiedlichen Container-Typen

Beim Kopieren zwischen ganz unterschiedlichen Containern erfolgen auch ggf. notwendige Anpassungen des Elementtyps:

```
using namespace std;
vector<double> v;

// from standard input float-s appended to vector ...
copy(istream_iterator<float>(cin), istream_iterator<float>(),
     back_inserter(v));

// ... to classic array (widening to double) ...
double data[100]; const auto N = sizeof data / sizeof data[0];

// ... (protecting against overflow) ...
if (v.size() < N) v.resize(N);

// ... (remembering filling state) ...
const auto endp = copy(v.begin(), v.end(), data);

// ... to set (truncating to integer) ...
set<int> s; copy(data, endp, inserter(s, s.begin()));

// ... to stdout with semicolon and space after each value
copy(s.begin(), s.end(), ostream_iterator<int>(cout, "; "));
```

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Kopieren zwischen gleichartigen Container-Typen

Obwohl mit dem `std::copy`-Algorithmus *1:1-Kopien* gleichartiger Container problemlos möglich sind, wird hier eine Zuweisung eher sinnvoll sein, also:

```
std::vector<MyClass> v1, v2;
…
v2 = v1;
```

Und nicht (evtl. nach einem `v2.clear()`):

```
std::copy(v1.begin(), v1.end(), std::back_inserter(v2));
```

> Insbesondere kann der spezifisch für eine Container-Klasse definierte Zuweisungs-Operator spezielle Eigenschaften der jeweiligen Abspeicherungsart berücksichtigen.*

*: Im Gegensatz zum generischen Kopieren werden dabei oft PODs (plain old data types) als Elementtyp erkannt und dann eine unterschiedliche Spezialisierung verwendet, welche die Operation in ein `std::memmove` oder `std::memcpy` überführt.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Einige typische Algorithmen in Beispielen

Die folgenden Programmfragmente gehen jeweils aus von einem STL-Container

```
std::vector<int> v;
```

der mit einigen Datenwerten gefüllt wurde.

Zählen der Elemente mit dem Wert 542

```
auto n = std::count(v.begin(), v.end(), 542);
```

Suchen des ersten Elements mit dem Wert 542

```
const auto f = std::find(v.begin(), v.end(), 542);
if (f != v.end()) {
    // erstes (passendes) Element gefunden
    …
}
else {
    // kein passendes Element vorhanden
    …
}
```

Löschen aller Elemente mit dem Wert 542

```
const auto end = std::remove(v.begin(), v.end(), 542);
```

Die Variable `end` enthält nun einen Iterator, der das neue (logische) Ende des (von der Anzahl der Elemente gesehen größeren) Containers bezeichnet.

```
// nicht mehr zum Inhalt zu zaehlende  Elemente loeschen
v.erase(end, v.end());
```

(i) Code wie der gerade gezeigte ist vor allem in generischen Templates zweckmäßig, bei denen die Klasse von `v` völlig offen gehalten werden soll.*

*: Wird ein bestimmter, typischer und auch häufiger Anwendungsfall von solchen Templates deutlich sub-optimal behandelt, kann immer noch eine entsprechende Spezialisierung erfolgen.

Callbacks aus Algorithmen

Prinzipiell sind Algorithmen zwar Code aus einer Bibliothek, nicht selten enthält dieser aber "Rückrufe" an die Applikation.

Dafür existieren drei Möglichkeiten:

- Klassische (C-) Funktionen und Übergabe als Funktionszeiger:
  - hierbei wird nur Name der Funktion angegeben;
  - die anschließenden runden Klammern entfallen.
- Objekte mit überladener `operator()` Member-Funktion - sogenannte Funktoren:
  - oft wird eine Objekt-Instanz direkt bei der Argumentübergabe erzeugt;
  - in diesem Fall enthalten nachfolgenden runde Klammern
    - die Argumentliste Konstruktors oder
    - bleiben leer (= Default Konstruktur)
- C++11 Lambdas

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Callback-Beispiel mit Funktor

Ein typisches Beispiel für die Notwendigkeit eines Call-Backs besteht im Fall des Algorithmus `std::for_each`, wobei einer Schleife über alle Elemente eines Containers übergeben wird, was mit jedem Element zu tun ist, z.B. ausgeben.

Zunächst ein Funktor, der dies leistet:

```
struct PrintWords {
    void operator()(const std::string &e) {
        cout << ": " << s << "\n";
    }
};
```

> Bei der Verwendung folgen dem Klassennamen des Funktors runde Klammern.

Dies gilt zumindest dann, wenn - wie oft üblich - ein (temporäres) Objekt der Funktor-Klasse als Argument an `std::for_each` übergeben wird:

```
std::for_each(v.begin(), v.end(), PrintWords());
```

Vergleich mit Funktion

Mit einer Funktion sieht das Beispiel so aus:

```
void print_words(const std::string &s) {
    cout << ": " << s << "\n";
};
```

Hier entfallen die Klammern, da der Name der Funktion weitergegeben wird.

```
std::for_each(v.begin(), v.end(), print_words);
```

Lokale Daten in Funktoren

Einer der Vorteile von Funktoren ist, dass sich im Funktor lokale Variable sauber kapseln lassen.

Hierzu ein leicht modifiziertes Beispiel, bei dem der Funktor die Argumente durchnummeriert:

```
struct PrintWordsEnumerated {
    void operator()(const std::string &s) {
        std::cout << ++n << ": " << s << "\n";
    }
    PrintWordsEnumerated() : n(0) {}
private:
    int n;
};
```

Parameterübergabe an Funktor

Über zusätzliche Member-Daten, die im Konstruktor des Funktors initialisiert werden, lassen sich auch Argumente aus der Aufruf-Umgebung weiterreichen:

```
struct PrintWordsEnumerated {
    void operator()(const std::string &e) {
        os << ++n << ": " << s << "\n";
    }
    PrintWordsEnumerated(std::ostream &os_) : n(0), os(os_) {}
private:
    int n;
    std::ostream &os;
};
```

Diese sind dann bei der Verwendung zu versorgen:

```
 std::for_each(v.begin(), v.end(), PrintWords(std::cout));
```

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Parameter aus Aufrufumgebung

Die mit der Übergabe von Parametern geschaffene Flexibilität ist spätestens dann wichtig, wenn es sich um Informationen handelt, die in der Aufruf-Umgebung in lokalen Variablen oder Argumenten vorliegen:

```
void foo(std::ostream &output) {
    …
    std::for_each(v.begin(), v.end(), PrintWords(output));
    …
}
```

Callback-Beispiel mit Lambda

Die in C++ neu eingeführte Lambdas haben gegenüber Funktoren den Vorteil, dass der ausgeführte Code direkt als Parameter des aufgerufenen Algorithmus zu sehen ist und nicht an einer mehr oder weniger weit davon entfernten Stelle steht.*

Grundlegendes Beispiel mit Lambda

Im einfachsten Fall greift das Lambda nur auf das vom Aufrufer übergebene Argument und evtl. globale Variable bzw. Objekte zu:

```
std::for_each(v.begin(), v.end(),
              [](const std::string &s) {
                      std::cout << s << '\n';
              }
);
```

*: Mit modernen IDEs, welche die Implementierung einer Funktion oder Klasse als Pop-Up zeigen, sobald man kurze Zeit den Mauszeiger darüber ruhen lässt, spielt dieser Nachteil aber nur eine geringe Rolle.

Callback-Beispiel mit Lambda

Lambda mit Capture List

Zum Zugriff in den Sichtbarkeitsbereich des umgebenden Blocks muss die *Capture-List* verwendet werden:

```
void foo(std::ostream &output) {
…
std::for_each(v.begin(), v.end(),
              [&output](const std::string &s) {
                  output << s << '\n';
              }
);
```

Darin werden die übergebenen Bezeichner aufgelistet, ggf. mit vorangestelltem &-Zeichen, wenn Referenz-Übergabe erfolgen soll.

Callback-Beispiel mit Lambda

Lambda mit "privaten Daten"

Im Gegensatz zu einem Funktor, der problemlos private Daten haben kann, welche nur dem überladenen Funktionsaufruf zur Verfügung stehen, sind die Möglichkeiten zur Kapselung in einem Lambda eingeschränkter:

```
void foo(std::ostream &output) {
    …
    int line_nr = 0;
    std::for_each(v.begin(), v.end(),
                  [&line_nr, &output](const std::string &s) {
                      output << ++line_nr << s << '\n';
                  }
    );
    …
}
```

Dies zeichnet zugleich den allgemeinen Weg vor, wie ein Lambda auf seine Aufrufumgebung nicht nur lesend sondern auch modifizierend einwirken kann.

Flexible Bedingungen (Prädikate) für Algorithmen

Viele STL-Algorithmen haben eine Variante, in der man ein Auswahlkriterium (Prädikat) flexibel angeben kann. Es handelt sich dabei typischerweise um die Algorithmus-Variante, die mit `_if` endet.

Als Beispiel zur Verdeutlichung des Prinzips kann wieder die Implemenierung eines Algorithmus zum Kopieren von einem Container in einen anderen dienen, diesmal beschränkt auf ausgewählte Elemente:*

```
template<typename T1, typename T2, typename T3>
T2 my_copy_if(T1 from, T1 upto, T2 dest, T3 pred) {
    while (from != upto) {
        auto tmp = *from++;
        if (pred(tmp))
            *dest++ = tmp;
    }
    return dest;
}
```

*: In die STL wurde ein solcher Algorithmus als `std::copy_if` erst mit C++11 aufgenommen. Allerdings erfüllt `std::remove_copy_if`, das seit C++98 vorhanden ist, mit einem invertierten Prädikat denselben Zweck.

Prädikate übergeben

Da es sich um eine Template handelt, gilt für das Prädikat `pred` lediglich, dass als aktuelles Argument etwas "Aufrufbares" (callable) anzugeben ist.

Damit kommen

- Funktionszeiger,
- Funktoren und
- Lambdas

in Frage, sofern diese als Rückgabewert ein `bool` liefern.*

Die folgenden Beispiele beziehen sich jeweils auf einen Container, der Ganzzahlen enthält. Darin werden die Werte gezählt, eelche kleiner als 42 sind.

*: Oder präziser: einen Typ, der ggf. in `bool` umgewandelt wird.

Mit Funktionszeiger

Zunächst muss die Funktion definiert werden:

```
bool lt42(int n) { return n < 42; }
```

Dann kann sie als Prädikat dienen:

```
… std::count_if( … , … , lt42);
```

[!] Viele Compiler generieren hier grundsätzlich einen Funktionsaufruf, womit diese Variante zur Laufzeit recht ineffizient sein kann.*

*: Die GNU-Compiler erzeugen seit einigen Jahren zumindest auf den höheren Optimierungsstufen aber deutlich besseren Code, indem sie bei sichtbarer Funktionsdefinition in Fällen wie dem Obigen eine Inline-Umsetzung vornehmen, selbst wenn die Funktion `lt42` nicht mit `inline` markiert wurde.

Mit Funktor

Zunächst muss die Funktor-Klasse definiert werden:

```
struct Lt42 {
    bool operator()(int n) const { return n < 42; }
};
```

Dann wird sie als Prädikat zu einem temporären Objekt instanziiert:

```
… std::count_if( … , … , Lt42());
```

Ist der Funktionsaufruf-Operator explizit oder - wie oben - implizit `inline`,* wird der erzeugte Code sehr schnell (und oft sogar kleiner) sein als bei Übergabe eines Prädikats mittels Funktionszeiger.

*: Hisichtlich des GCC (g++) sei aber daran erinnert, dass bei ausgeschalteter Optimierung (`-O0`) grundsätzlich *keine* Funktion als Inline-Funktion umgesetzt wird.

Mit C++11-Lambda

Hier ist das Prädikat unmittelbar bei der Übergabe zu sehen:

```
… std::count_if( … , … , [](int n) { return n < 42; });
```

Mit Standard-Bibliotheksfunktion

Diese mit C++98 eingeführte Möglichkeit wirkt sehr unleserlich und wird evtl. auch deshalb nur selten benutzt:

```
… std::count_if( … , … , std::bind2nd(std::less<int>(), 42));
```

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Callback mit Zeiger auf spezifische Funktion

Dies ist die aus der C-Programmierung bekannte Technik.

Beispiel: Ausgeben aller Elemente mit einem Wert ungleich 524:

```
bool notEq524(int n) { return n != 524; }

void foo(const std::vector<int> &v) {
    …
    std::copy_if(v.begin(), v.end(),
                 std::ostream_iterator<int>(cout, " "),
                 notEq524
    );
    …
}
```

Callback mit spezifischem Funktor

Dies ist eine für C++-typische Technik.

Beispiel: Kopieren aller Elemente, die nicht den Wert 524 haben:

```
struct NotEq524 {
    bool operator()(int n) const { return n != 524; }
};
void foo(const std::vector<int> &v) {
    …
    copy_if(v.begin(), v.end(),
            std::ostream_iterator<int>(std::cout, " "),
            NotEq524()
    );
    …
}
```

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Callback mit allgemeiner verwendbarem Funktor

Funktoren mit Member-Daten

Der im folgenden verwendete Funktor kann in allen Vergleichen benutzt werden, welche auf die Prüfung hinauslaufen, ob eine Ganzzahl *ungleich* einem bestimmten Wert ist:

```
class NotEq {
    const int cmp;
public:
    NotEq(int c) : cmp(c) {}
    bool operator()(int n) const { return n != cmp; }
};
void foo(const std::vector &v) {
    …
    std::copy_if(v.begin(), v.end(),
                 std::ostream_iterator<int>(std::cout, " "),
                 NotEq(524)
    );
    …
}
```

Übergabe lokal sichtbarer Variablen*

Dieser Weg steht (nur) einem Funktor mit Member-Daten offen:

```
void foo(const std::vector &v, int hide) {
    …
    std::copy_if(v.begin(), v.end(),
                 std::ostream_iterator<int>(cout, " "),
                 NotEq(hide)
    );
    …
}
```

*: Dies schließt die Parameter einer Funktion ein, da diese im Wesentlichen den lokal sichtbaren Variablen entsprechen, lediglich mit einer (garantierten) Initialisierung beim Funktionsaufruf mit vom Aufrufer versorgten Werten.

Callback mit C++11-Lambda

Ohne Nutzung der Capture-List

```
void foo(const std::vector &v) {
    …
    std::copy_if(v.begin(), v.end(),
                 std::ostream_iterator<int>(std::cout, " "),
                 [](int n) { n != 524; }
    );
    …
}
```

Mit Nutzung der Capture-List

```
void foo(const std::vector &v, int hide) {
    …
    std::copy_if(v.begin(), v.end(),
              std::ostream_iterator<int>(std::cout, " "),
                 [hide](int n) { n != hide; }
    );
    …
}
```

Generische Algorithmen vs. spezifische Member-Funktionen

Bei löschenden Algorithmen wird manchmal vergessen, dass der Container damit nicht tatsächlich verkleinert wird. Vielmehr werden nur Elemente umkopiert oder getauscht und als Ergebnis wird ein Iterator zurückgegeben, der das neue logische Ende markiert.

> Ist die Klasse des Containers genau festgelegt, steht alternativ zu einem generischen Algorithmus mitunter eine spezifische Member-Funktion zur Verfügung. Deren Verwendung ist dann in aller Regel effizienter.

Mit `std::list<std::string> li` beispielsweise statt

```
auto end = std::remove(li.begin(), li.end(), 542);
li.erase(end, li.end());
```

besser folgendes:

```
li.remove(542);
```

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH

Übung

Ziel der Aufgabe:

STL-Container und Algorithmen verwenden.

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

(CC) BY-SA: Dipl.-Ing. Martin Weitzel im Auftrag von MicroConsult Training & Consulting GmbH