C++ FOR (Mittwochvormittag)

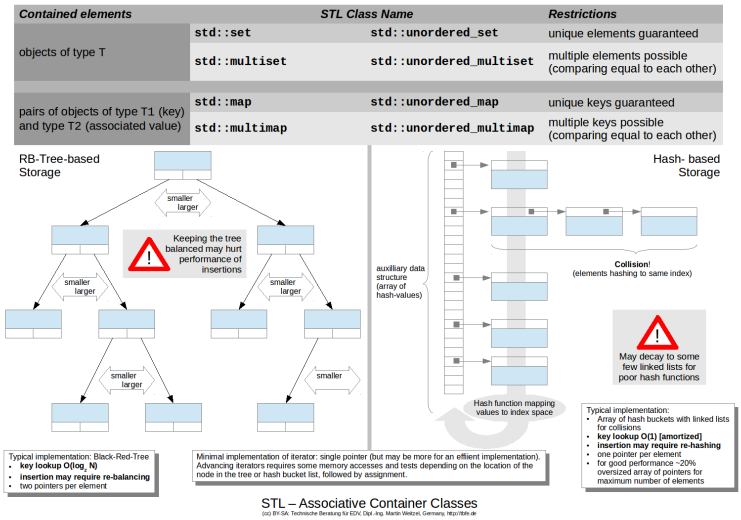
Kürzere Pausen werden jeweils nach Bedarf eingelegt.

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

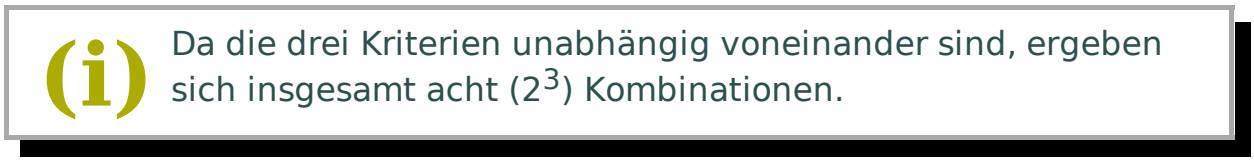


*: Eine stets balancierte Variante des Binärbaums.

Kombination der Varianten

Die verfügbaren assoziative 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



Speicherverfahren Red-Black-Tree

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

Die folgende Tabelle zeigt den Zusammenhang zwischen der Tiefe eines balancierten Binärbaums und der Anzahl der Einträge, die enthalten sind, wobei die Tiefe der Anzahl der Vergleiche, die (maximal) nötig sind, um in bei entsprechend vielen Einträgen einen bestimmten Schlüssel zu finden.

Tiefe	und minimale .. maximale Zahl der Einträge
1	1 .. 1
2	2 .. 3
5	16 .. 31
7	64 .. 127
10	512 .. 2023
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)

*: Eine Binärbaum ist balanciert, wenn allen Zweige eine möglichst gleichmäßige gleichmäßige Tiefe haben. Ein unbalancierter Binärbaum entspricht im Extremfall einer einfach verketteten Liste.

Speicherverfahren Hashing

Unter dem Begriff "Hashing" wird (auch) ein Abspeicherungsverfahren verstanden, welches - unter der Voraussetzung einer gut streuenden Hash-Funktion - Suchzeiten 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 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 Soirtieralgorithmus (z.B. Bubblesort)

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 als Index innerhalb eines Arrays festlegt, dessen Gesamtgröße wiederum dem Streubereich der Hash-Funktion entspricht.

Anhängig von der Anzahl der Datenwerte und der Größe des Streubereichs kann es weder theoretisch noch praktisch eine eindeutige Abbildung des Schlüssels auf den Streubereich geben.

*: Bei den verschiedenen Varianten des set gibt es keinen dem Schlüssel zuegeordneten Datenwert. Alternativ könnte man dort den Wert auch als Schlüssel ansehen.

Entstehung von Überläufern

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

Gut streuende Hashfunktionen liefern zwar bei einer bi zu 80..90%-igen Belegung des zur Verfügung stehenden Index-Raumes nur selten Überläufer, dennoch kann dieser Fall prinzipiell immer auftreten.

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

Re-Hashing

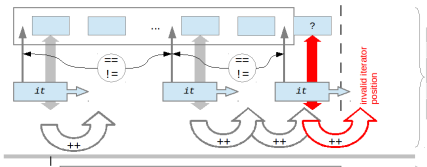
Ist der Umfang der Datenmenge unbekannt, muss der Indexraum möglichwerweise zu irgend einem Zeitpunkt vergrößert werden, wenn die Performance nicht leiden soll.

- Eine universell verwendbare Klasse wie `unordered_set` oder `unordered_map` (die jeweiligen `multi`-Varianten an dieser stillschweigend eingeschlossen) kann dies nahezu vollautomatisch tun.
- Dazu muss die Hash-Funktion angepasst werden, sodass 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 nun gültigen Hash-Funktion noch einmal neu berechnet werden.***

*: Mit Hilfe spezieller Techniken lässt sich der Aufwand hierfür halbieren, indem diese Neuberechnung nur für die Hälfte der Einträge reduziert wird.

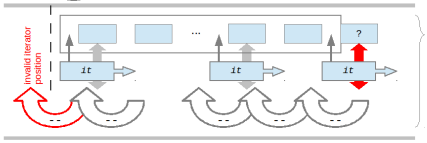
Iteratoren-Kategorien

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



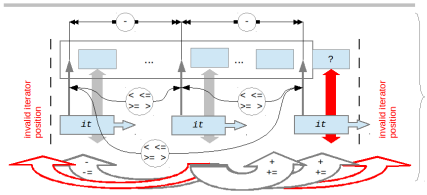
Operations of **Unidirectional Iterators**

	Effect	*Remarks*
`*it`	access referenced element	undefined at container end
`++it` `it++`	advance to next element (usual semantic for pre-/postfix version)	
`it == it`	compare for identical position	operands must denote existing element or end of same container
`it != it`	compare for different position	



Additional Operations of **Bidirectional Iterators**

	Effect	*Remarks*
`--it` `it--`	advance to previous element (usual semantic for pre-/postfix version)	undefined at container begin



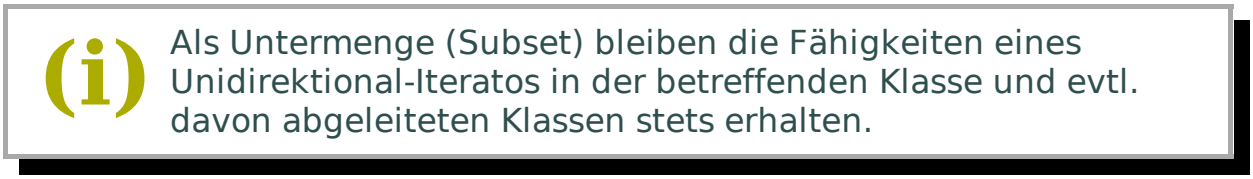
Additional Operations of **Random Access Iterators**

	Effect	*Remarks*
`it + n` `it += n`	`it` advanced by *n*-th next element (previous if *n* < 0)	resulting iterator position must be inside container (denoze existing element or end)
`it - n` `it -= n`	`it` advanced by *n*-th previous element (next if *n* < 0)	
`it - it`	number of increments to reach rhs `it` from lhs `it`	operands must denote existing element or end of same container
`it < it`	true lhs `it` before rhs `it`	
`it <= it`	true if lhs `it` not after rhs `it`	
`it >= it`	true if lhs `it` not before rhs `it`	
`it > it`	true if lhs `it` after rhs `it`	

STL – Iterator Categories

Unidirektional-Iteratoren

Iteratoren, welche der Kategorie der Unidirektionale Iteratoren hinzuzurechnen sind, 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::iterator::forward_list`.

Bidirektional-Iteratoren

Bidirektionale Iteratoren

Iteratoren, die der Kategorie der Bidirektional-Iteratoren hinzu zu rechnen sind, können sich - ihrem Namen entsprechend - zwar in zwei Richtung bewegen, nämlich vom Anfang eines Containers zu dessen Ende hin und umgekehrt, können dabei aber nicht "springen" (kein `operator+(std::size_t)`).

Um Bidirektional-Iteratoren handelt es sich bei

- `std::list<T>::iterator` und `std::list::const_iterator¸ sowie
- allen Iteratoren für alle assoziativen Container*

*: Die Iteratoren für den "const"- und "reverse"-Fall sind hier stillschweigend mit eingeschlossen.

Iteratoren mit wahlfreiem Zugriff

Iteratoren, die der Kategorie der Random-Access-Iteratoren hinzu zu rechnen sind, können - ihrem Namen entsprechend - innerhalb des zugehörigen Containers effizient wahlfrei zugreifen.

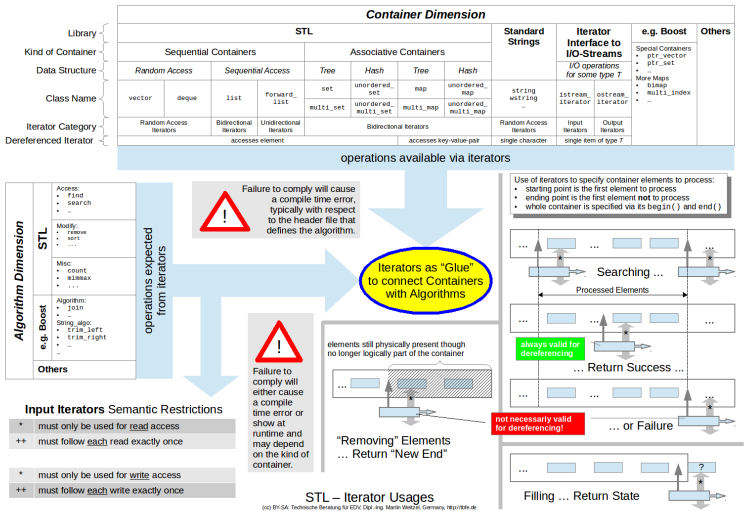
Iterator-Konventionen

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

- Input-Iteratoren
- Output-Iteratoren

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

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



Iteratoren als verbindendes Element

Mit Iteratoren als verbindendes Element zwischen Containern und Algorithmen sind bei ca. 40 Algorithmen (-Familien) und 12 Containern im ISO-Standard nur 40 und keine 480 (= 12 × 40) Implementierungen notwendig.*

*: Die Tatsache, dass nicht alle Algorithmen sinnvoll mit allen Containern zusammenarbeiten sei an dieser Stelle einmal außen vor gelassen. Würde man sie berücksichtigen, käme man wahrscheinlich immer noch auf einige hundert erforderliche Implementierungen im Gegensatz zu einer einzigen (Template-basierten), die ein bestimmten Algorithmus nun lediglich erfordert.

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.

Algorithmen-Achse

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

Input-Iterator-Kategorie

Input-Iteratoren sind ggf. abwechelnd *zu dereferenzieren für den* nur lesenden, exklusiven *Zugriff* und weiterzuschalten (`operator++( … )`).

Output-Iterator-Kategorie

Output-Iteratoren sind ggf. abwechelnd

- zu derefenenzieren für den schreibenden, exklusiven Zugriff
- und weiterzuschalten (`operator++( … )`).

Erfolgreiche Suche

Beim Suchen (und verwandten Container-Operationen) wird der Erfolg damit angezeigt, dass der als Rückgabewert gelieferte Iterator insofern **immer gültig** ist, als er auf eine Position innerhalb des zur Bearbeitung übergebenen Daten-Bereichs zeigt.

Fehlgeschlagene Suche

Beim Suchen wird der Misserfolg damit angezeigt, dass der als Rückgabewert gelieferte Iterator insofern **immer gültig** ist, als er auf eine Position innerhalb des zur Bearbeitung übergebenen Bereichs zeigt.

Zustandsanzeige nach Füllen

Beim Füllen eines Containers ist es üblich, den (neuen) Füllzustand des Containers durch den Wert eines Iterators anzuzeigen.

Löschanzeige durch neues (logisches) Ende

Da die Algorithmen für eine große Zahl von Containern funktionieren sollen, finden an einigen Stellen einige aus pragmatischen Sicht "merkürdige" (= des Merkens würdige) Entscheidungen statt.

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

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 unterschiedlichen Zielen abgelegt werden.

Beispiel: Implementierung des copy-Algorithmus

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

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

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

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

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

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

[!] Insbesondere wenn `v2.size()` kleiner ist als `v1.size()` werden mit einiger Wahrscheinlichkeit fremde Speicherbereiche überschreiben.

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 in v erreichen, indem man die neuen Elemente stets mit push_back hinten anfügt:*

```
copy(v1.begin(), v1.end(),
        std::back_insert_iterator<std::vector<int>>(v2)
);
```

*: Das Umbrechen längerer Zeilen bei der Argumentbergabe erscheint (nicht nur) hier sinnvoll, um die Argumente nach ihrer Bedeutung zu gruppieren. Die dabei zum Einsatz kommende Assymetrie ist dagegen Geschmackssache, und entspricht einem vebreiteten, häufig bei geschweiften (Block-) Klammern angewandten Stil.

Anhängen an Vektoren (2)

Hilfsfunktion: `back_inserter`

Die - redundant erscheinende - Angabe von `int` als Datentyp des Containers v2 ist technisch der Tatsache geschuldet, dass ein (temporäres) Objekt der Template-Klasse `std::back_insert_iterator` erzeugt werden muss und deren Konstrukor nicht aus dem Argumenttyp nicht auf den Instanziierungstyp schließen kann.

Eine Hilfsfunktion* vermeidet diese Redundanz:

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

---

* Diese Funktion ist zwar Bestandteil der Standardbibliothek, muss also nicht selbst geschrieben werden, aber ihre Implementierung ist 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` auch für die Klassen

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

und es existiert auch ein `std::front_insert_iterator`, der mit den Klassen

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

verwendet werden kann.*

*: Allgemeiner ausgedrückt sind die Anforderungen beim `std::back_insert_iterators` dass der zur Instanziierung verwendete Container eine `push_back` Member-Funktion hat und entsprechend `push_front` beim `std::front_insert_iterator`.

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

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

In assoziative Container einfügen (2)

Hilfsfunktion: `insert_iterator`

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

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

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

---

* 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). In diesen Fällen 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 ausführen, 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;
…
copy(fli.begin(), fli.end(),
     std::ostream_iterator<long>(std::cout, ";")
);
```

Stream-Iteratoren (2)

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 erzeugtes Objekt liefert im Vergleich mit einem per Default-Konstrukor erstellten Objekt 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 dabei den "Gut"-Zustand verlässt, liefert der damit verbundene Istream-Iterator bei künftigen 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 damit auch klassische Arrays bearbeitet werden.*

Kopieren AUS einem klassischen 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:

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

Oder:

```
copy(data, data + ndata, … );
```

Zeiger als Iteratoren (2)

Kopieren IN ein 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 stillschweigend überschrieben werden.

Kopieren zwischen unterschiedlichen Container-Typen

Beim Kopieren ist auch Kombination unterschiedlicher Container möglich und es erfolgen - falls notwendig - automatische Anpassungen des Elementtyps.

```
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, "; "));
```

Kopieren zwischen gleichartigen Container-Typen

Obwohl mit dem copy-Algorithmus auch *1:1-Kopien* gleichartiger Containern möglich sind, wird eine Zuweisung eher sinnvoll sein, also:

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

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

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

Insbesondere kann der spezifisch in einer Container-Klasse definierte Zuweisungs-Operator besondere Eigenschaften der jeweiligen Abspeicherungsart berücksichtigen.*

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

Einige typische Algorithmen in Beispielen

Die folgenden Programmfragmente gehen jeweils aus von deinem 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);
```

Einige typische Algorithmen in Beispielen (2)

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
 …
}
```

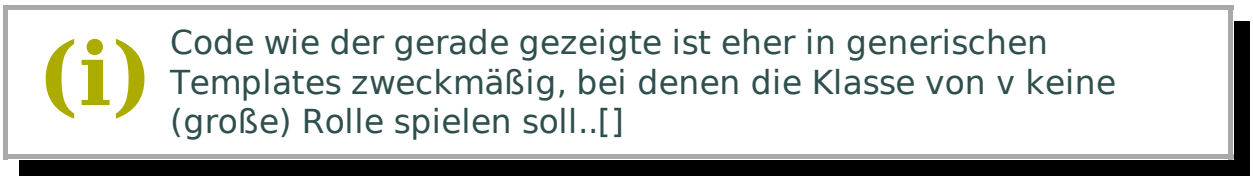
Einige typische Algorithmen in Beispielen (3)

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 ggf. loeschen
v.resize(end - v.begin());
```



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

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 damit 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 meist üblich ein (temporäres) Objekt der Funktor-Klasse direkt als drittes 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 beim Aufruf, da der Name der Funktion weitergegeben wird.

Der Aufruf dieser Funktion erfolgt aus der Implementierung von `std::for_each` heraus.

```
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 damit sauber kapseln lassen.

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

```
struct PrintWordsEnumerated {
 void operator(const std::string &e) {
  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) {
  cout << ++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));
```

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 ebenfalls in Variablen oder - noch typischer - if Aufrufargumenten vorliegen:

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

Dies ist nur noch mit Funktoren sauber abzubilden, mit einer Funktion scheidet diese Technik aus.

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 zeigt, sobald man kurze Zeit den Mauszeiger darüber ruhen lässt, spielt dieser Nachteil aber eine immer geringere 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) {
         std::cout << s << '\n';
        }
 );
```

Hier werden die übergebenen Bezeichner aufgelistet, ggf. mit vorangestelltem &-Zeichen, wenn Referenz-Übergabe gewünscht ist.

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 etwas eingeschränkter:

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

Dies zeichnet auch den allgemeinen ein Weg vor, auf dem 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) sehr flexibel angeben kann. Es handelt sich dabei typischerweise um die Algorithmus-Variante, die mit `_if` endet.

Als Beispiel kann wieder der die Implemenierung eines Algorithmus zum Kopieren von einem Container in einen anderen dienen:*

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

* Der STL wurde ein solcher Algorithmus erst mit C++11 hinzugefügt. Allerdings erfüllt das schon mit C++98 eingeführte `std::remove_copy_if` denselben Zweck, wenn man das Prädikat invertiert.

Prädikate übergeben

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

Damit kommen

- Funktionszeiger,
- Funktoren und
- Lambdas

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

Alle der folgenden Beispiele beziehen sich auf einen Container, der Ganzzahlen enthält und zählen darin die Werte, die kleiner als 42 sind.

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

Prädikate übergeben (2)

Mit Funktionszeiger

Zunächst muss die Funktion definiert werden:

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

Dann kann sie als Prädikat dienen:

```
… count_if( … , … , lt42);
```

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

* Die GNU-Compiler erzeugen seit einigen Jahren zumindest auf den höheren Optimierungsstufen aber 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.

Prädikate übergeben (3)

Mit Funktor

Zunächst muss die Funktor-Klasse definiert werden:

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

Dann wird sie als Prädikat zu einem temporären Objekt instanziiert:

```
… count_if( … , … , Lt42());
```

Ist der Funktionsaufruf-Operator explizit oder - wie oben - implizit `inline`,* wird der erzeugte Code sehr schnell (und meist auch kleiner) sein als bei der 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.

Prädikate übergeben (4)

Mit C++11-Lambda

Hier ist das Prädikat unmittelbar bei der Übergabe zu sehen:

```
… count_if( … , … , [](int v) { return v < 42; });
```

Mit Standard-Bibliotheksfunktion

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

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

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 v) { return v != 524; }

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

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* zu einem bestimmten Wert ist:

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

Callback mit allgemeiner verwendbarem Funktor (2)

Übergabe lokal sichtbarer Variablen*

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

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

Callback mit C++11-Lambda (2)

Mit Nutzung der Capture-List

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

Generische Algorithmen vs. spezifische Member-Funktionen

Insbesondere bei löschenden Algorithmen besteht oft die Gefahr, die Tatsache zu übersehen, dass der Container damit nicht tatsächlich verkleinert sondern nur Elemente umkopiert und ein neues Ende zurückgegeben wird.

> Ist die Klasse des Containers genau festgelegt, sollte - wenn diese Alernative existiert - aufdie Verwendung eines (generischen) Algorithmus verzichtet und eine ggf. vorhandene spezifische Member-Funktion vorgezogen werden.

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);
```

Übung

Ziel der Aufgabe:

STL-Container und Algorithmen verwenden.

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