C++ STL – Iterators and Algorithms

(Day 2, Part 1)

- Principle of Iterators
- Iterator Categories
- Iterator Traits

- Iterator-Interface to Algorithms
- Callbacks from Algorithms
- A Tour through Standard Algorithms

Principle of Iterators

Iterators provide a helping mechanism for container traversal, i.e.

- processing all elements in a container,
- either in a specific order,*
- or simply with the guarantee each element is visited exactly once.

> Note that STL-Iterators share the idea of the GoF Iterator Pattern but are far from its proposed implementation.

STL style Iterators

- access the current element via the overloaded `operator*`;
- advance to the next element via the overloaded `operator++`;
- end of traversal by comparison with a special end-point iterator

and do all of this expecting an interface in the vein of Duck-Typing.

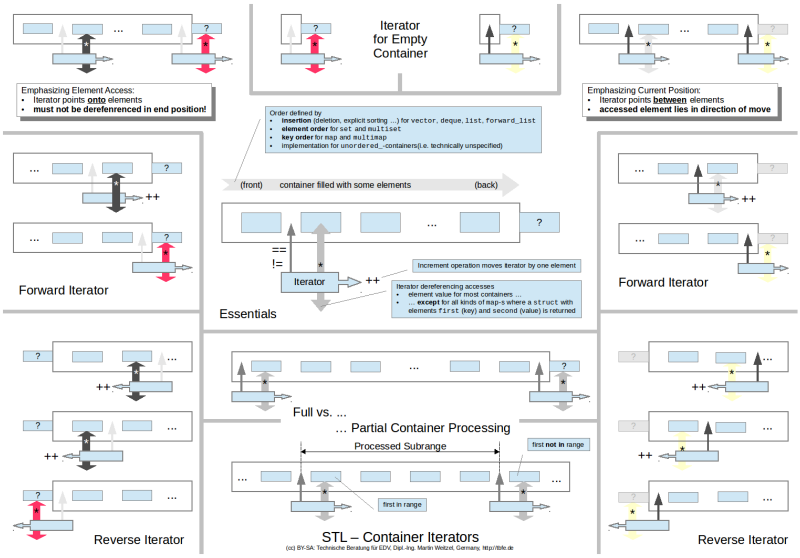
---

*: Then, of course, there needs to be an ordering criteria, which for sequential containers may be the sequential order created by insertion, and also trivially exists for the traditional *ordered associative* containers based on binary trees, **but not** for the hash-based containers (i.e. `std::unordered_set` and `std::unordered_map` and their `multi`-variants).

Visualising Iterators

Often iterators are visualised in graphics to support understanding their principle.

- One common abstraction is to show iterators **pointing to** an element – the element they would return when dereferenced.
- Another possible abstraction is to show iterators pointing **in between** elements – right before the element they would advance over next.



Visualising Iterators *Pointing To* Elements

Advantages

- It **is by far the most common** visualisation.
- It is immediately clear which element `operator*` will access.

Disadvantages

- Developers might first need to learn thinking in "asymmetric borders"*
- One more element must be assumed beyond the container-end to
  - visualise the iterator end position, and also
  - show how an iterator interacts with an empty container.
- It must be made **absolutely clear that this is a virtual element, introduced only for the purpose of visualisation**.

> **[!]** Any attempt to access that "end-point" Element is Undefined Behaviour according to the ISO Standard.

*: After being accustomed to that kind of thinking it often becomes second nature and thinking in symmetric borders may start feel unnatural.

Visualising Iterators *Pointing Between* Elements

Advantages

- No virtual elements outside the container need to be assumed.
- Ranges (of affected elements) are visualised in a most natural way.
- It usually goes without saying that the iterator must not be moved outside of the container.

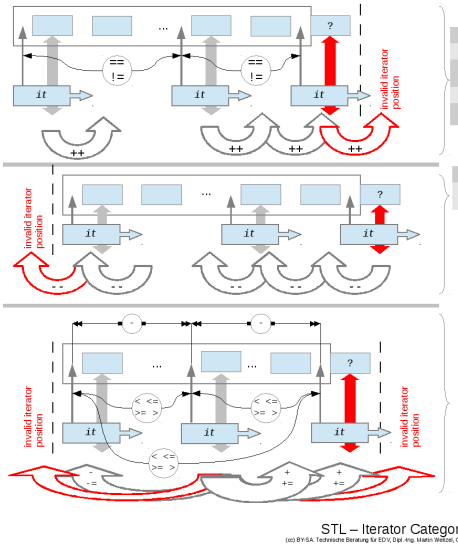
[!] **Nevertheless:**
Any attempt to move the iterator beyond the container borders Undefined Behaviour according to the ISO Standard.

Disadvantages

- In common C++ literature this visualisation is hardly ever used.
- The *move direction* of the iterator must always be indicated to understand which element `operator*` will access.

Iterator Categories

- Forward Iterators
- Bidirectional Iterators
- Random Access Iterators



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` advanced to *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 to *n*-th previous element (next if *n* < 0)	
`it -= n`		
`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 if 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

Forward Iterators

Iterators modelling the ForwardIterator concept may (only) be advanced into one direction – i.e. forward from the begin to the end of a container.

Of the STL containers only one has iterators in that category, which is

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

including the respective `const_`-variant.

Bidirectional Iterators

Iterators modelling the BidirectionalIterator concept may be advanced in both directions – forward from the begin to the end of a container, or backward from the end (= reverse begin) to the begin (= reverse end).

> The BidirectionalIterator concept **includes** the capabilities of the ForwardIterator Concept.

Of the STL containers seven have iterators of that category, which are

- `std::list<T>::iterator`
- and iterators of all eight associative containers,

including the respective `const_`-, `reverse_`-, and `const_reverse_`-variants.

Random Access Iterators

Iterators modelling the RandomAccessIterator concept may efficiently access elements at any given (valid) position in a container.

> The RandomAccessIterator concept **includes** the capabilities of the BidirectionalIterator Concept.

Besides addition and subtraction of integral numbers (which move the iterator back or forth over that distance) by subtraction the size of the enclosed range* can be determined.

Of the STL containers three have iterators of that category, which are

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

including the respective `const_`-, `reverse_`-, and `const_reverse_`-variants.

---

*: As in stead "size of enclosed range" also the wording "number of enclosed elements" is a meaningful description, this again is a case where visualising iterators as pointing **in between** elements is much more natural.

Limit-Checking of Random-Access Iterators

With respect to the following it should be noted that the ISO standard specifies *Minimum-Requirements* only, which allow an highly efficient implementation, even if this is traded for safety.

> **[!]** When a Random Access Iterators participates in an operation that moves it outside the container borders the result is undefined.

Valid positions are also end-point positions, i.e. one beyond the container end for regular iterators and its begin for the `reverse_`-variants.

> **[!]** On subtraction the result is only meaningful if the involved iterators refer (to elements of*) the same container (including the end-point positions).

For any two iterators pointing to elements of different containers (of the same type – otherwise it were a compile time error) it is guaranteed that they never compare equal.

---

*: If the visualisation of iterators **pointing to** elements is preferred.

Iterator Traits

[Type Traits] in general are an advanced concept with many uses in compile time Meta-Programming, but not limited to that area.

With respect to iterators only some minimal basic knowledge needs to be acquired, which then can then be applied in "cookbook-style" where necessary.

The class `std::iterator_traits` provides type conversions from an iterator (type) to several related types, like the type resulting from dereferencing.

Its basic use is shown in an example on the next page.

For more information on iterator traits see:
http://en.cppreference.com/w/cpp/iterator/iterator_traits

Cookbook-Style Use of Iterator Traits

Typically iterator traits come in handy when algorithms are implemented.

If an algorithm

- shall not only work with iterators from the STL containers,
- but also for native pointers (into native array)

it may be difficult ar first glance to determine the type of a derefenrenced iterator in a generic way.

The following shows an elegant* solution applying `std::iterator_traits`:

```
template< … , typename Iterator, … >
void foo( … , Iterator it, … ) {
    typename std::iterator_traits<T>::value_type temp = *it++;
    …
}
```

*: Less elegant solutions might overload `foo` for pointer types.

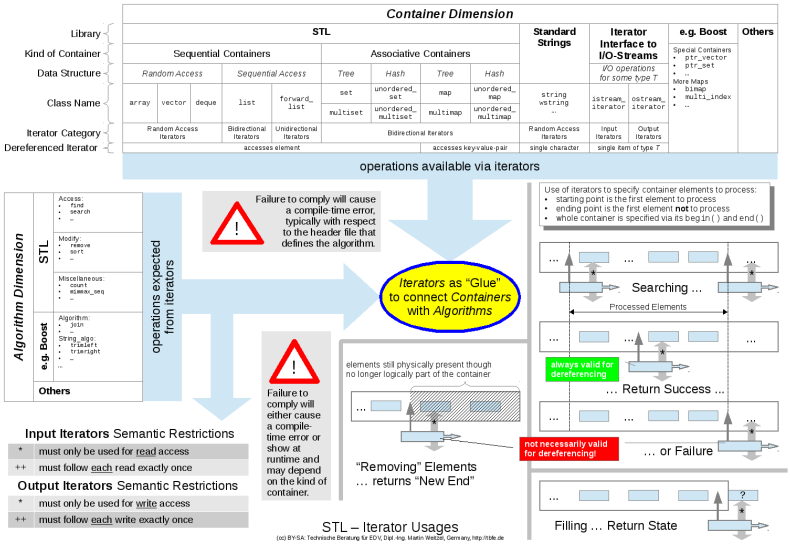
Iterator-Interface to Algorithms

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

- Input-Iteratoren
- Output-Iteratoren

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

- Füllstands- und ...
- ... Löschungs-Anzeige



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.*

*: Selbst unter Berücksichtigung der Tatsache, dass nicht alle Algorithmen für alle Container sinnvoll sind, würde die Zahl der erforderlichen Implementierungen immer noch bei mehreren hundert liegen.

Container-Achse

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

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

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

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

Input-Iterator

Input-Iteratoren sind *abwechselnd*

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

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

Output-Iterator

Output-Iteratoren sind *abwechselnd*

- für den *Schreib-Zugriff* zu dereferenzieren (*)
- und *weiter zu schalten* (++).

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

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.

[!] Wird nur ein Teilbereich eines Containers `c` übergeben, der sich nicht bis zu dessen Ende erstreckt, darf zur Prüfung auf Erfolg oder Misserfolg nicht mit `c.end()` verglichen werden.

Gescheiterte Suche

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

- immer auf die Endposition

des zur Bearbeitung übergebenen Bereichs zeigt.

[!] Beim Suchen in einem leeren Container oder Bereich wird stets die Endposition des Containers bzw. des Bereichs geliefert.

Zustandsanzeige nach Füllen

Beim Füllen eines Containers ist es üblich, dass Algorithmen

- den (neuen) Füllstand des Containers

durch einen entsprechenden Iterator als Rückgabewert anzeigen.

> Dies ist sinnvoll und praktisch, um den Container ab dieser Stelle später weiter füllen zu können.

Löschungs-Anzeige durch neues (logisches) Ende

Da die Algorithmen für eine große Zahl von Containern funktionieren sollen, finden gibt es einige aus pragmatischer Sicht sinnvolle aber Besonderheiten.

Eine davon betrifft die Tatsache, dass "löschende" Algorithmen nicht davon ausgehen, den bearbeiteten Container verkleinern zu können.

> Dadurch funtionieren diese Algorithmen auch für die eingebauten Arrays und die STL-Klasse `std::array`.

Beim Löschen von Elementen aus Containern wird nur "logisch gelöscht", d.h.

- die enthaltenen Elemente erden ggf. anders angeordnet, und
- als Rückgabewert ein Iterator geliefert, der das "neue Ende" bezeichnet.

Iteratoren und Algorithmen

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

Iteratoren sind in der Regel einfache (Helfer-) Klassen, welche abhängig vom Container, für den sie jeweils zuständig sind, eine vollkommen unterschiedliche 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.

Beispiel: Implementierung des copy-Algorithmus

Zum besseren Verständnis, wie mittels Iteratoren die Algorithmen eine besondere Flexibilität erzielen, 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 von einen normalen Iterator – wie von `v.begin()` geliefert – nur bereits vorhandene Elemente überschrieben aber keine neuen Elemente angefügt werden können, ist es wichtig, den Ziel-Container vor dem Kopieren auf die notwendige (Mindest-) Größe zu bringen.

Anhängen an Vektoren

Hilfsklasse: `back_insert_iterator`

Ohne den Ziel-Container zuvor auf eine passende Größe zu bringen, lässt sich eine sichere Variante des Kopierens auch erreichen, indem neue Elemente mit `push_back` angefügt werden:*

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

> Da die in diesem und vielen folgenden Beispielen an Algorithmen übergebenen Argumentlisten recht lang sind, erfolgt ein Umbruch innerhalb der Argument-Liste, um die Lesbarkeit zu verbessern.

Dabei werden auch unterschiedliche Formatierungs-Stile demonstriert.*

---

*: Die oben verwendete asymmetrische Klammerung der Argumentliste ist sicher Geschmackssache, dupliziert aber den oft auch bei geschweiften Klammern üblichen Stil.

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 Instanziierungs-Typ schließen kann.

Eine Hilfsfunktion* vermeidet diese Redundanz:

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

---

*: Diese Funktion ist zwar Bestandteil der Standard-Bibliothek, 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.

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(s, s.begin()));
```

Etwas ungewöhnlich ist hier das zusätzliche Argument. Dessen Zweck ist es, einen Optimierungshinweis* zu geben, den der Insert-Iterator ggf. verwendet, um die Suche nach der Einfügeposition abzukürzen.

> Üblicherweise wird – wenn kein wirklicher 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 dereferenzierten 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-Konstruktor erzeugten Gegenstück zunächst `false`.
Die Operationen `*` und `++` werden auf eine Eingabe mit `operator>>` abgebildet, wobei die eingelesene Variable genau den 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 verschiedenen Container-Typen*

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

*: Im Rahmen der Zuweisungskompatibilität erfolgt beim Kopieren zwischen unterschiedlichen Containern auch die Typ-Konvertierung des Elementtyps.

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.

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());
```

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 from Algorithms

- Generelle Möglichkeiten für Callbacks
- Callback mit Funktor
- Callback mit Lambda
- C++14 Erweiterungen für Lambdas
- Prädikate für Algorithmen (generell)
- Prädikate für Algorithmen (Beispielserie))
- Algorithmen vs. spezifische Member-Funktionen

Generelle Möglichkeiten für Callbacks

Prinzipiell sind Algorithmen zwar Code aus einer Bibliothek, häufig enthalten diese aber "*Callbacks*" 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 Konstruktor)
- C++11 Lambdas

Callback 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. es ausgeben.

Zunächst ein Funktor, der dies leistet:

```
struct PrintWords {
    void operator()(const std::string &e) {
        cout << ": " << e << "\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 &e) {
    cout << ": " << e << "\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 << ": " << e << "\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(),
              PrintWordsEnumerated(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 in lokalen Variablen oder Argumenten vorliegen:

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

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

Datentypen in Funktoren

Diese richten sich nach der gewünschten Zugriffsart:

Zugriffsart	Daten-Member	Konstruktor-Argument
über Kopie	(konstanter) Wert	Wert oder (konstante) Referenz
direkt, nur lesend	konstante Referenz	(konstante) Referenz
auch schreibend	Referenz	Referenz

Referenzen für Daten-Member dürfen keinesfalls mit Wertübergabe im Konstruktor kombiniert werden:

```
class SomeFunctor {
    …
    T data1;
    const T &data2;
public:
    …
    SomeFunctor(const T& d1, T d2)
        : data1(d1)  // OK
        , data2(d2)  // SERIOUS PROBLEM HERE !!
    {}
};
```

Callback 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.

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.

Lambda mit privaten Daten

Ein Funktor kann problemlos private Daten besitzen, welche ausschließlich dem überladenen Funktionsaufruf zur Verfügung stehen.

In C++11 sind die Möglichkeiten zur Kapselung in dieser Hinsicht für ein Lambda etwas eingeschränkt: lokale Daten können nicht wirklich privat sein sondern sind auch in der Aufrufumgebung sichtbar:

```
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';
                  }
    );
    …
}
```

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

C++14 Extensions to Lambdas

With C++14 lambdas have been further extended:*

- Lambdas may use `auto` as argument type, thus allowing generic lambdas, similar to functors with a templated `operator()`.
- The capture list may define so-called init-captures, giving the same options as local (independent) data members of a functor, with (possibly context dependant) initialisation.

*: For the curious: C++14 also fixed a minor oversight in the lambda syntax specification of C++11: While the idea was to allow the lambda argument list to be completely dropped if empty, including its parentheses if empty, that was forgotten in the syntax for `mutable` lambdas.

C++14 `auto`-typed Lambda Arguments

Prior to C++14 argument types of lambdas had to be spelled out explicitly, which could sometimes be wordy and inconvenient:*

```
std::map<std::string, unsigned long> m;
…
std::map<std::string, unsigned long> other;
std::copy_if(m.cbegin(), m.cend(), std::inserter(other),
             [](const std::pair<std::string, unsigned long> &e) {
                 return e.second != 0;
             });
```

Clearly, in the above scenario `auto`-typed arguments for lambdas offer a big simplification:

```
std::copy_if(m.cbegin(), m.cend(), std::inserter(other),
             [](const auto &e) { return e.second != 0; });
```

---

*: The problem could be slightly alleviated if there were a type definition for the map, say `C`, allowing to refer to the element type inside as `C::value_type` (or if `C` were a dependant type as `typename C::value_type`).

C++14 Lambda Init-Captures

This C++14 extension allows a third form of capture:

```
// assuming local scope here
SomeType local;
…
… [local]( … ) { // capturing by value-copy
   … // access local copy read-only
  } …
… [local]( … ) mutable { // capturing by value-copy
    … // access local copy modifiable
  } …
… [&local]( … ) { // capturing by reference
    … // access and possibly modify local in outer scope
  } …
… [mine = local]( … ) { // init-capture (C++14 only)
    … // access mine modifiable, initial value from local
  } …
```

The initialisation of init-captures may also use arbitrary expressions.

```
… [mine = 2*local + 1] ( … ) { … } …
… [mine = std::make_shared<SomeType>(local)] ( … ) { … } …
```

Extended Type Deduction for C++14 Lambdas

With the extensions to lambdas in C++14 also two new contexts for type deduction were introduced:

- In case of `auto`-typed lambda arguments the type deduction rules are the same as for template functions (and hence **not** exactly the same as for `auto`-typed variables!)
- In case of init-captures the type of the newly introduced identifier is determined is like for `auto`-typed variables.

Init-Captures and Move-Construction

Init-captures solve a problem that lay outside the capabilities of lambdas as defined by C++11:

- When capturing by value-copy, only the copy-constructor applies.
  - In an init-capture, if the initialising expression is a temporary, the move construction may take place – given a move constructor exists and the compiler does not choose to prefer (N)RVO.

Therefore only init-captures allow the following (assuming `local` is a move-only type* or it is its last use and copying is expensive):

```
… [mine = std::move(local)] ( … ) { … } …
```

*: Like e.g. `std::unique_ptr`, `std::thread`.

C++14 Init-Capture Pitfalls

One possible pitfall with init-captures compared to value-copy is this:

[!] Classic (C-style) arrays are treated as value objects captured by value-copy but decay to pointers (to the first element) in init-captures.

Secure when capturing value-copies:

```
auto foo() {
    int primes[] = {2, 3, 5, 7};
    return [primes]( … ) {
                … // lambda-body
            };
};
```

Refers to invalidated stack memory:

```
auto foo() {
    int primes[] = {2, 3, 5, 7};
    return [p = primes]( … ) {
                … // lambda-body
            };
};
```

This can be used as an argument to prefer `std::array` over classic arrays, as then in both cases copying the content would be copied.

Prädikate für Algorithmen (generell)

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 Implementierung 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) {
        const auto tmp = *from++;
        if (pred(tmp))
            *dest++ = tmp;
    }
    return dest;
}
```

*: In die STL wurde der Algorithmus `std::copy_if` erst mit C++11 aufgenommen. Allerdings erfüllt das seit C++98 vorhandene `std::remove_copy_if` denselben Zweck, wenn man das Prädikat invertiert angibt.

Möglichkeiten zur Übergabe von Prädikaten

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
- C++11-Lambdas

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

Im folgenden werden in einem Container mit Ganzzahlen alle Elemente mit einem Wert kleiner als 42 gezählt.

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

Prädikat mit Funktionszeiger übergeben

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.

Prädikat mit Funktor übergeben

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 schneller (und oft sogar kleiner) sein als bei Übergabe eines Prädikats mittels Funktionszeiger.

*: In Bezu auf den GCC (g++) sei aber daran erinnert, dass bei ausgeschalteter Optimierung (`-O0`) grundsätzlich *keine* Funktion als `inline`-Funktion umgesetzt wird.

Prädikat mit C++11-Lambda übergeben

Hier ist das Prädikat unmittelbar bei der Übergabe zu sehen:

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

Alternative mit Standard-Bibliotheksfunktion

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

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

Prädikate für Algorithmen (Beispielserie)

Abschließend eine weitere Beispielserie zum Vergleich der verschiedenen Möglichkeiten, Algorithmen mit Prädikaten als Bedingungstests zu versorgen:

In der ersten Gruppe geschieht dies ohne jegliche Flexibilität – alles ist für den jeweiligen Einzelfall "hart kodiert".

- Funktionszeiger
- Funktor
- Lambda

In der zweiten Gruppe (für welche Funktionszeiger bereits nicht mehr mächtig genug sind), besteht mehr Flexibilität – wobei teilweise ähnliche Techniken aus einem unterschiedlichen Grund zur Anwendung kommen.

- Parametrisierter Funktor
- Zugriff auf lokale Daten mit Funktor
- Zugriff auf lokale Daten mit Lambda

Prädikat mit spezifischem Funktionszeiger

Dies ist die aus der C-Programmierung bekannte Technik.

Das Beispiel gibt alle Elemente mit einem Wert ungleich 524 aus:

```
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
    );
    …
}
```

Prädikat mit spezifischem Funktor

Dies ist eine für C++-typische Technik.

Das Beispiel gibt alle Elemente mit einem Wert ungleich 524 aus:

```
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()
    );
    …
}
```

Prädikat mit spezifischem Lambda

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

Prädikat mit allgemeiner verwendbarem Funktor

Der im folgenden verwendete Funktor kann in allen Vergleichen benutzt werden, welche auf die Prüfung hinauslaufen, ob eine Ganzzahl ungleich zu einem gegebenen 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)
    );
    …
}
```

Prädikat mit Funktor und Zugriff auf lokalen Kontext

Um auf lokale Variablen oder Parameter* einer aufrufenden Funktion zugreifen zu können, muss ein Funktor entsprechende Daten-Member besitzen und sein Konstruktor muss diese initialisieren:

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

*: Die Parameter einer Funktion entsprechen nahezu ihren lokalen Variablen. Es kommt lediglich die (garantierte) Initialisierung mit vom Aufrufer anzugebenden Werten hinzu.

Prädikat mit Lambda und Zugriff auf lokalen Kontext

Um auf lokale Variablen oder Parameter* einer aufrufenden Funktion zugreifen zu können, muss ein Lambda die Capture-List verwenden:

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

Lambdas in Member-Funktionen einer Klasse können hier auch `this` angeben. Damit besteht innerhalb des Lambdas Zugriff auf alle Member (Daten und Funktionen) desjenigen Objekts, für welches die das Lambda enthaltende Member-Funktion ausgeführt wird.

*: Die Parameter einer Funktion entsprechen nahezu ihren lokalen Variablen. Es kommt lediglich die (garantierte) Initialisierung mit vom Aufrufer anzugebenden Werten hinzu.

STL Algorithmen vs. spezifische Member-Funktionen

Ist die Klasse des Containers genau festgelegt, steht alternativ zu einem generischen Algorithmus eventuell eine spezifische Member-Funktion zur Verfügung, deren Verwendung in der Regel einfacher und effizienter ist.

Beispielsweise wird bei löschenden Algorithmen gerne vergessen, dass der Container dadurch nicht tatsächlich kleiner wird.* Zwar könnte man das Problem so lösen

```
std::list<std::string> li;
…
auto end = std::remove(li.begin(), li.end(), 542);
li.erase(end, li.end());
```

aber kompakter und effizienter geht es so:

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

*: Vielmehr werden – wie bereits dargestellt – nur Elemente umkopiert oder getauscht und als Ergebnis wird ein Iterator zurückgegeben, der das neue logische Ende anzeigt.

A Tour through Standard Algorithms

Please gather here for coach to show you around :-).

For a more practical approach to discover selected algorithms on your own, visit the Mini Projects focussing on algorithms.

For more information on the STL Algorithm Library see:
http://en.cppreference.com/w/cpp/algorithm