STL-Algorithmen

Die STL definiert auch viele nützliche Algorithmen, die in `<algorithm>` oder `<numeric>` definiert sind: Sortieren, Zählen, Summieren, Modifizieren, Kopieren, Schleifen …

Im Folgenden sei `c` irgendein Container-Typ, der Integer enthält, wie z.B. `std::array<int, dim>, std::vector<int>, std::list<int>, int[]` usw.

`std::for_each()`: Schleife als Funktionsaufruf (in `<algorithm>`)

Dieser Algorithmus wendet eine „Funktion“ `f` auf jedes einzelne Element eines Bereiches an. Der Bereich wird dabei durch 2 Iteratoren beschrieben: Der erste zeigt auf den Anfang und der zweite **hinter das Ende** des Bereiches. Solche Iteratoren erhält man bei Containern z.B. durch `c.begin(), std::begin(c), c.crend(), std::crbegin(c)` (`r` steht hier für reverse, `c` für `const`). Dieser Algorithmus ist eine Alternative zu `range-for`, weil er auch Teile des Containers durchlaufen kann und auch die Reihenfolge umdrehen kann.

Die Funktion `f()` (oder ein Lambda oder ein aufrufbares Objekt) muss ein Containerelement verarbeiten können (hier `int`). Bei Referenzübergabe des Elements (`int&`) kann der Container auch verändert werden!

```
#include <algorithm>
…
void f(int x)  // gib das Quadrat aus
{    std::cout << x << "hoch 2 = " << x*x << '\n'; }
…

for_each(begin(c), end(c), f);      // von vorn nach hinten
for_each(cbegin(c), cend(c), f);    // von vorn nach hinten, KEINE Änderung
for_each(rbegin(c), rend(c), f);    // von hinten nach vorn
```

Selbstverständlich darf man auch ein Lambda statt einer Funktion nehmen und dieses z.B. gleich im `for_each()` Konstrukt definieren:

```
std::for_each(cbegin(c), cend(c),
  [](auto x){ std::cout << x << "hoch -1 = " << 1./x << '\n'; }
 );
```

Quadriere alle Elemente des Containers (d.h. **ändere den Containerinhalt**):

```
std::for_each(begin(c), end(c), [](auto& x){ x *= x; });
```

`std::sort()` : Sortierung (in `<algorithm>`)

Zum Sortieren eignen sich nur Container mit Index-Zugriff, also z.B. `std::array`, `std::vector`, nicht aber `std::list, std::set, std::map` …

```
std::sort(std::begin(c), std::end(c));          // aufsteigend sortieren

std::sort(std::begin(c), std::end(c),
     [](auto x, auto y) { return x > y;});      // absteigend sortieren
```

Das optionale 3. Argument (fehlt es, wird aufsteigend sortiert) beschreibt, wie sortiert werden soll. Es werden 2 Containerelemente übergeben und das Ergebnis muss `true` sein, wenn sie in der richtigen Reihenfolge sind, sonst `false`. Solche Funktionen (oder Lambdas) müssen daher den Ergebnistyp `bool` haben und 2 Containerelemente vergleichen. Boolsche Funktionen nennt man auch **Prädikate.**

Der Includefile `<functional>` enthält schon einige nützliche Funktionen oder Prädikate, die man nicht unbedingt durch selbstgeschriebene Lambdas nachbilden muss:

```
std::sort(std::begin(c), std::end(c), std::greater<>{});
      // absteigend sortieren
```

`std::count_if()` : Zählen, wenn (in `<algorithm>`)

```
std::count_if(std::cbegin(c), std::cend(c),
     [](auto x){ return x % 3 == 0;});  // zähle Vielfache von 3
```

Das 3. Argument beschreibt, wann ein `x` gezählt werden soll. Es ist also ein Prädikat, das ein Containerelement als Argument hat. Es sollte den Container nie ändern und deshalb das Containerelement als Kopie oder konstante Referenz erhalten.

`std::accumulate()` : Summieren etc. (leider in `<numeric>`)

```
std::accumulate(std::cbegin(c), std::cend(c), 0); // Summieren
```

Das 3. Argument ist hier der Startwert der Summation. Man kann ein optionales 4. Argument angeben, das die Art der Akkumulierung angibt (fehlt es, wird immer addiert). Es ist eine Funktion (oder Lambda etc.) mit 2 Argumenten: Das erste ist das Zwischenergebnis der Akkumulierung, das 2. ist das Container-Element, das akkumuliert werden soll. Der Rückgabewert muss das Ergebnis dieser Akkumulierung sein.

```
accumulate(cbegin(c), cend(c), 1.,
     [](auto product, auto x) { return product*x; }
     ); // Multipliziere alle Elemente mit Resultat

accumulate(cbegin(c), cend(c), 1., multiplies<>{}); // dasselbe
```

Ist z.B. `Rechteck` ein Typ, der ein Rechteck beschreibt (no na) und berechnet die Funktion `double flaeche(Rechteck r)` die Fläche von `r` und ist `vr` ein Container mit solchen Rechtecken, so ist die Gesamtfläche von `vr`

```
accumulate(cbegin(vr), cend(vr), 0.,
     [](double sum, Rechteck x) { return sum + flaeche(x); });
```

Ich habe die Typen hier explizit hingeschrieben anstatt `auto` zu verwenden!

STL-Algorithmen und die Zukunft: Ranges

Man könnte sich fragen, warum man Iteratoren anstatt der ganzen Container verwenden muss. `sort(c)` ist kürzer und einfacher zu verstehen als `sort(begin(c), end(c))`. Eine erste Antwort darauf ist, dass man mit den Iteratoren auch Containerteile beschreiben kann und auch eine andere Laufrichtung erreichen kann (von hinten nach vorn), während die erste Form nur Container als Ganzes von vorn nach hinten abarbeiten kann.

Aber ist das wirklich so?

C++20 gibt in Form der `ranges`-Bibliothek die Antwort darauf: Nein

Die Ranges-Bibliothek (ab C++20) definiert im Scope `std::ranges` alle Algorithmen der STL neu (und erweitert viele der Algorithmen um coole neue Möglichkeiten). Man kann diese neuen Dinge mit einem modernen Compiler ausprobieren, indem man sie aus dem Scope `std::ranges` verwendet:

```
std::sort(c.begin(), c.end()) <=>  std::ranges::sort(c)
```

Leider sind noch nicht alle Algorithmen in C++20 für Ranges umgeschrieben worden. So wird erst C++23 `std::ranges::accumulate` und weitere fehlende Algorithmen enthalten.

```
std::accumulate(c.begin(), c.end(), 0) <=>
     std::ranges::accumulate(c, 0)
```

Eine coole Erweiterung ist die optionale Verwendung von Projektionen in den Algorithmen. Beispiele für Projektionen sind die einzelnen Komponenten einer Struktur oder Klasse. Man kann z.B. einen sortierbaren `Rechteck`-Container `vr` (`Rechteck` mit Attributen `x, y`) nach den `y`-Werten sortieren. Derzeit geht das nur mithilfe eines Lambdas, das die Vergleichsoperation modelliert :

```
std::sort(vr.begin(), vr.end(),
     [](auto& a, auto& b){ return a.y < b.y; });
```

in C++20:

```
std::ranges::sort(vr, std::less<>{}, &Rechteck::y);
std::ranges::sort(vr, {}, &Rechteck::y);   // noch kürzer
```

Hat `Rechteck` zusätzlich eine Methode `flaeche()`, welche den Flächeninhalt berechnet, so sortiert man `vr` absteigend nach Flächeninhalt:

Vor C++20:

```
std::sort(r.begin(), r.end(), [](auto& a, auto& b)
     { return a.flaeche() > b.flaeche(); });
```

C++20:

```
std::ranges::sort(r, std::greater<>{}, &Rechteck::flaeche);
```

Die neuen Konstruktionen sind nicht nur kürzer, sondern auch einfacher zu verstehen und kosten auch keine zusätzliche Rechenzeit.

Containerteile und umgedrehte Laufrichtungen kann man als Views interpretieren, d.h. man sieht den Container unter einer Brille, die Teile wegfiltert oder die die Reihenfolge verändert. Und selbstverständlich wird man die Algorithmen dann auch auf solche Views anwenden können. In der Ranges-Bibliothek vordefinierte Views sind z.B. die Umkehrung der Reihenfolge, Verwendung einer Teilmenge etc. Und selbstverständlich werden solche Views auch kombiniert werden können.