Vorlesung

Objektorientiertes Programmieren in C++

Teil 4 - WS 2026/27

Detlef Wilkening
www.wilkening-online.de
© 2026

10 Container & Iteratoren 2
10.1 Einführung 2
10.2 Vektoren 2
10.3 Container-Definitionen 6
10.4 Listen 7
10.5 Iteratoren 8
10.6 Arrays 12
10.7 Sets 13
10.8 Unsortiertes Set 19
10.9 Maps 20
10.10 Unsortierte Map 25
10.11 Algorithmen, Ranges und Lambdas 26
10.12 Weiteres 28

10 Container & Iteratoren

10.1 Einführung

In der Praxis muss man fast immer mit Mengen von Objekten arbeiten, sehr häufig auch mit dynamischen Mengen. Es gibt viele unterschiedliche Container-Typen, die allgemein und umfassend in Werken über „Algorithmen und Datenstrukturen“ besprochen werden. Hier ist nicht der Raum für tiefe Theorie, aber etwas muss sein. Denn die C++ Standard-Bibliothek bringt viele Container und Algorithmen mit, und man muss wissen wann man was einsetzt.

In diesem Kapitel werden wir nur die wichtigsten Container selber und Iteratoren kennenlernen. Weitere, aber nicht weniger wichtige Elemente wie z.B. Algorithmen, Ranges und Lambda-Ausdrücke, werden später erklärt.

10.2 Vektoren

Der Feld-, Wald- und Wiesen-Container der STL ist der Vektor. Er ist das Arbeitspferd unter den Containern – d.h. immer wenn Sie erstmal keinen Besseren wissen, dann ist der Vektor die erste Wahl.

Der Vektor implementiert ein dynamisches Array, d.h. einen Container bei dem die Elemente direkt hintereinander im Speicher stehen, und dessen Anzahl Elemente dynamisch wachsen oder schrumpfen kann.

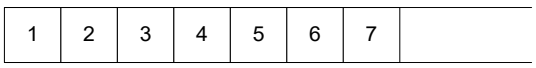

Abb. 10-1 : Speichermodell eines Vektors

Merkmale:

- Der Vektor ist ein sequentieller Container, d.h.:
  - die Elemente liegen hintereinander (in einer Sequenz), und
  - nur der Nutzer des Containers bestimmt die Reihenfolge der Elemente – der Container selber verändert die Reihenfolge nicht.
- Der Vektor unterstützt wahlfreien Zugriff, d.h. man kann direkt auf das n-te Element zugreifen (ohne Performance-Probleme).
- Der Vektor benötigt relativ wenig Speicher, da er kaum interne Verwaltungs-Informationen benötigt und die Elemente bündig im Speicher aneinander liegen.
- Anfügen neuer Elemente am Ende des Vektors und Löschen von Elementen am Ende des Vektors geht sehr schnell – siehe Abb. **10-2.**
- Muss dagegen vorne ein Element eingefügt oder gelöscht werden, so ist dies sehr aufwändig und damit langsam – siehe Abb. **10-2**. Einfüge- und Lösch-Operationen mitten im Vektor sind abhängig von der Position langsam oder schnell – im Mittel aber schlecht.
- Beim Suchen nach einem Element im Vektor, muss der Vektor von vorn nach hinten durchlaufen werden – d.h. die benötigte Suchzeit steigt linear zur Anzahl der Elemente im Vektor.

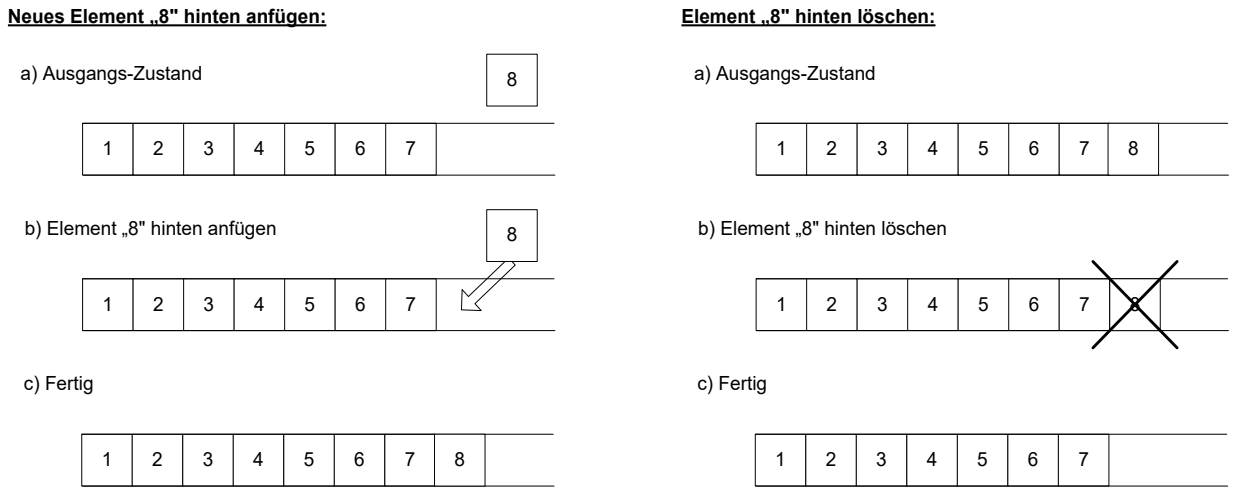

Abb. 10-2 : Hinten Einfügen und Löschen im Vektor

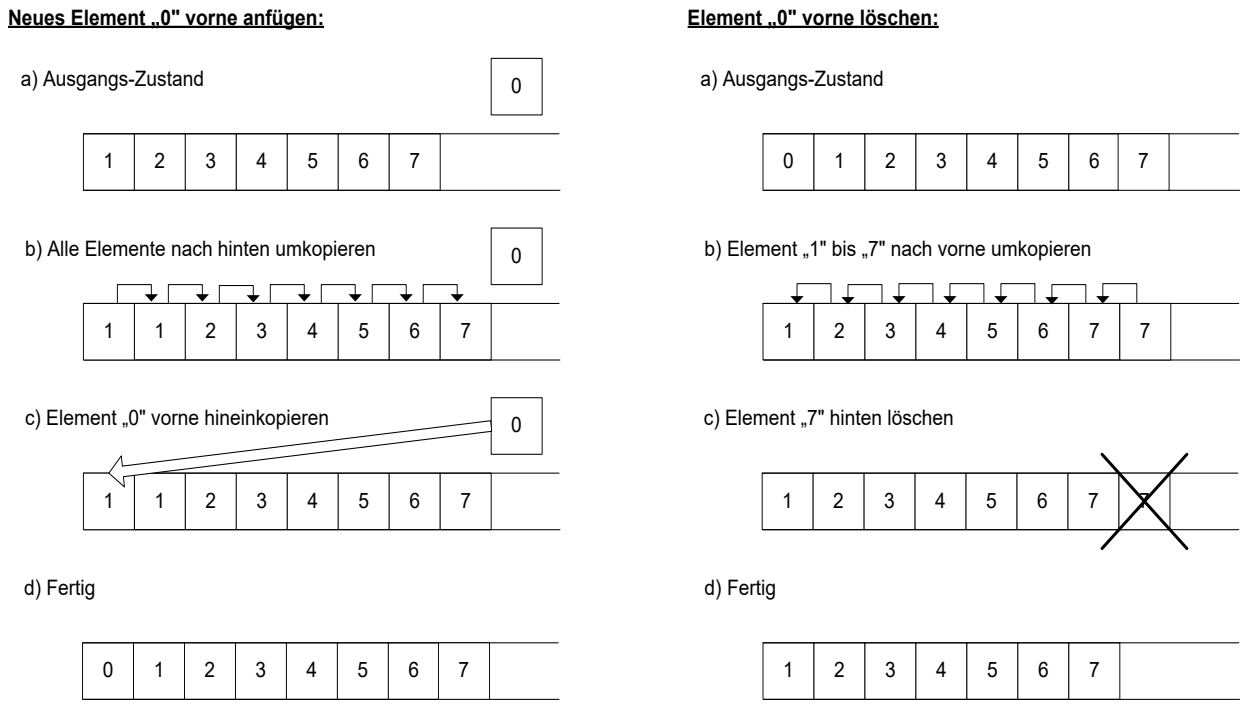

Abb. 10-3 : Vorne Einfügen und Löschen im Vektor

Schauen wir uns ein erstes Beispiel für Vektoren an:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> v;              // Ein Vector nur fuer int Objekte
   v.push_back(1);             // Fuegt das Element hinten an
   v.push_back(3);             // Dito...
   v.push_back(2);
   v.push_back(4);
   v.push_back(1);

   cout "Anzahl Elemente im Vektor: " << v.size() << '\n';

   for (vector<int>::size_type i=0; i<v.size(); ++i)
   {
      cout << v[i] << ' ';     // 1 3 2 4 1
   }
}
```

Ausgabe

```
Anzahl Elemente im Vektor: 5
1 3 2 4 1
```

- Mit „vector<int> v;“ wird ein Objekt „v“ angelegt, das vom Typ „std::vector<int>“ ist. Der exakte Typ ist sogar noch länger – Sie erinnern sich, dass man in C++ schnell lange Typ-Namen bekommt.
- In diesem Vektor kann man nur int-Objekte speichern – dies wird durch die Angabe des Element-Typs in den spitzen Klammern erreicht. Sie müssen immer einen Element-Typ angeben – Vektoren ohne Element-Typ (d.h. ohne spitze Klammern) sind ein Fehler.

- Mit der Element-Funktion „push_back“ kann man hinten an den Vektor Elemente anfügen. Der Vektor kümmert sich intern automatisch um Speicherplatz und all diese Dinge.
- Die aktuelle Anzahl an Elementen im Vektor kann mit der Element-Funktion „size“ erfragt werden. Der Rückgabe-Typ der Funktion ist „std::vector<int>::size_type“.
- Der Typ „std::vector<int>::size_type“ ist ein integraler unsigned Typ, der auf jeden Fall groß genug ist, die Anzahl an Elementen in einem „std::vector<int>“ abzubilden. Der wahre zugrunde liegende Typ ist vom Standard nicht definiert.
- Mit dem Index-Operator „[ ]“ kann mit einem 0-basierten Index vom Typ „std::vector<int>::size_type“ auf den n-te Element lesend oder schreibend zugegriffen werden (wahlfreier Zugriff).

Ein weiteres Beispiel:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
   vector<string> v;              // Ein Vector nur fuer String Objekte
   v.push_back("Axel");
   v.push_back("Max");
   v.push_back("Tim");
   v.push_back("Bernd");

   for (vector<string>::size_type i=0; i<v.size(); )
   {
      cout << v[i++];
      if (i != v.size())
      {
         cout << ", ";
      }
   }
}
```

Ausgabe
```
Axel, Max, Tim, Bernd
```

Alternativ zum aufwändigen Typ „vector<string>::size_type“ zur Definition der Zählvariable „i“ in der For-Schleife bietet sich die Auto-Variante an, die hier wieder viel kürzer und einfacher ist. Das folgende Beispiel entspricht abgesehen von der Verwendung von „auto“ exakt dem letzten Beispiel.

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
   vector<string> v;
   v.push_back("Axel");
   v.push_back("Max");
   v.push_back("Tim");
   v.push_back("Bernd");

   for (auto i=0; i<v.size(); )       // Falls Sie hier eine Warnung bekommen
   {                                  // -> Erklaerung im Text unten
      cout << v[i++];
      if (i != v.size())
      {
         cout << ", ";
      }
```

```
    }
    cout << '\n';
}
```

Ausgabe
```
Axel, Max, Tim, Bernd
```

10.3 Container-Definitionen

Man kann bei allen Containern der C++ Standard-Bibliothek den Container auch direkt mit Elementen vorbelegen – dies geschieht mit den geschweiften Klammern:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> v{ 1, 3, 2, 4 };         // Vorbelegung des Vektors
   cout "Anzahl: " << v.size() << '\n';

   v.push_back(1);
   v.push_back(8);
   cout "Anzahl: " << v.size() << '\n';

   for (vector<int>::size_type i=0; i<v.size(); ++i)
   {
      cout << v[i] << ' ';
   }
}
```

Ausgabe
```
Anzahl Elemente im Vektor: 4
Anzahl Elemente im Vektor: 6
1 3 2 4 1 8
```

Da der Compiler in diesem Fall den Typ der Elemente aus den Initial-Werten deduzieren kann, darf man hier die explizite Element-Typ Angabe in den spitzen Klammern auch weglassen. Achtung, bedenken Sie auch hier bitte, dass Zeichenketten-Konstanten keine Strings sind.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector v{ 11, 22, 44 };      // Definition ohne <int> moeglich
                                // - da der Compiler den Typ deduzieren kann

   for (vector<int>::size_type i=0; i<v.size(); ++i)
   {
      cout << v[i] << ' ';
   }
}
```

Ausgabe
```
11 22 44
```

Beides (Vorbelegung und Weglassen des Element-Typs bei Vorbelegung) funktioniert mit allen Containern, und nicht nur mit dem Vektor. In den Beispielen und Musterlösungen finden sie mal diese und mal jene Variation, damit Sie etwas Übung bekommen.

10.4 Listen

Ein weiterer dynamischer sequentieller Container ist die doppelt-verkettete Liste. Im Gegensatz zum Vektor liegen die Elemente nicht direkt hintereinander, sondern können beliebig im Speicher verteilt sein und sind untereinander verkettet.

Abb. 10-4 : Speichermodell eines doppelt-verketteten Liste - todo

Der Hauptvorteil der Liste gegenüber dem Vektor sind die Einfüge- und Lösch-Operationen, die unabhängig von der Position überall gleichschnell sind. Benötigen Sie also einen dynamischen sequentiellen Container mit schnellen wahlfreien Einfüge- und Lösch-Operationen, dann ist wahrscheinlich die Liste sinnvoller als der Vektor.

Abb. 10-5 : Einfügen und Löschen in der doppelt-verketteten Liste - todo

Aber die Liste hat auch Nachteile gegenüber dem Vektor. Sie ist nicht so speichersparend, denn jedes Element benötigt einen Verwaltungs-Overhead, und vor allem macht der wahlfreie Zugriff auf die Listenelemente keinen Sinn. Um das n-te Element zufinden, muss die Liste von vorne über die ersten n Elemente iterieren – dies ist im Schnitt sehr langsam und nicht empfehlenswert. In C++ unterstützt die Liste daher auch keinen wahlfreien Zugriff, damit Sie nicht versehentlich eine extrem langsame Operation verwenden.

Ein erstes Beispiel für Listen:

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
   list<int> l;              // Eine Liste nur fuer int Objekte
   l.push_back(1);
   l.push_back(2);
   l.push_back(3);

   list<int>::size_type count = l.size();
   cout "Anzahl Elemente in der Liste: " << count << '\n';
}
```

```
Ausgabe
Anzahl Elemente in der Liste: 3
```

Da man in die Liste auch vorne effizient einfügen kann, gibt es für die Liste u.a. auch die Element-Funktion „push_front“, die ein Element vorne einfügt.

Aber wie läuft man jetzt über die Liste? Der obige Ansatz vom Vektor mit einer Zählschleife und einem Lauf-Index kann ja nicht funktionieren. Eine Liste hat ja keinen wahlfreien Zugriff, da der niemals performant sein kann.

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
```

```
   list<int> l;
   l.push_back(1);
   l.push_back(2);
   l.push_back(3);
   for (list<int>::size_type i=0; i<l.size(); ++i)
   {
      cout << l[i] << ' ';     // Compilerfehler, kein wahlfreier Zugriff vorhanden
   }
}
```

Die Lösung sind Iteratoren. Nehmen Sie Abstand von Laufvariablen und wahlfreiem Zugriff, programmieren Sie *echtes* C++ und nutzen Sie zumindest Iteratoren. Eine zweite Lösung ist die range-basierte For-Schleift, die intern Interatoren nutzt. Oder Sie benutzen die Algorithmen oder Ranges aus der Standard-Bibliothek. Dazu später mehr.

10.5 Iteratoren

Iterator – allgemeines Konzept um auf die *Elemente* eines Objekts sequentiell zugreifen zu können, ohne die zugrundlegende Repräsentation zu kennen. Hier ein erstes Beispiel mit einem Vektor.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> v { 1, 3, 2, 4 };

   // Nutzung eines Non-Const-Iterators - schreibende und lesende Zugriffe erlaubt
   for (vector<int>::iterator it=begin(v); it!=end(v); ++it)
   {
      cout << *it << ' ';       // 1 3 2 4
      *it += 4;                 // Schreibender Zugriff
   }
   cout << '\n';

   // Nutzung eines Const-Iterators - nur lesende Zugriffe erlaubt
   for (vector<int>::const_iterator it=begin(v); it!=end(v); ++it)
   {
      cout << *it << ' ';       // 5 7 6 8
   }
}
```

```
Ausgabe
1 3 2 4
5 7 6 8
```

- Die Funktion „begin“ des Containers gibt den Iterator auf das erste Element zurück. Dieser Iterator wird auch Start- oder Beginn-Iterator genannt.
- Der Typ des Iterators ist der Container-Typ mit dem zugeordneten Iterator-Typ – hier im Beispiel „std::vector<int>::iterator“ bzw. „std::vector<int>::const_iterator“.
- In der STL ist der Iterator-Typ daher fest mit dem Container-Typ (hier „std::vector<...>“) und dem Element-Typ (hier „int“) verbunden.
- Meistens existieren sowohl Non-Const- als auch Const-Iteratoren – wie im Beispiel beim Vektor. Über einen Non-Const-Iterator können die Elemente in einem Container gelesen und auch verändert werden, über einen Const-Iterator können sie nur gelesen werden.
- Um auf das vom Iterator referenzierte Element zuzugreifen, muss man den Dereferenzierungs-Operator „*“ verwenden, oder bei Element-Zugriff von Klassen den

Pfeil-Operator „->“ – siehe übernächstes Beispiel.

- Einen Iterator kann man mit dem Operator ++ (es gibt ihn in Prä- und Postfix-Notation) auf das nächste Element setzen. Achtung – der Präfix Operator ist bei Iteratoren performanter, sollte also vorgezogen werden.
- Es gibt einen ausgezeichneten Iterator für einen Container, den „Ende-Iterator“ – er wird von der Funktion „end“ zurückgegeben. Er zeigt auf das virtuelle erste Element, das nicht mehr im Container ist. D.h. wenn der Lauf-Iterator gleich dem End-Iterator ist, sind alle Elemente des Containers abgearbeitet worden.
- Auf das vom Ende-Iterator referenzierte Element darf nicht zugegriffen werden – es existiert ja gar nicht. Es ist ja nur ein *virtuelles Element*, um das Ende des Containers anzuzeigen.

Abb. 10-6 : Start- und Ende-Iterator für einen Vektor - todo

Einer der vielen Vorteile von Iteratoren ist, dass sie vom zugrunde liegenden Container abstrahieren, und daher unabhängig von ihm sind. Neben dem Vektor funktionieren sie also auch problemlos für Listen und andere STL Container, aber auch für viele andere Mengen bis hin zu berechneten Mengen. Aber wie auch immer – mit Iteratoren kann man jetzt z.B. über Listen laufen:

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
   list<int> l { 1, 3, 2, 4 };

   // Nutzung eines Non-Const-Iterators - schreibende und lesende Zugriffe erlaubt
   for (list<int>::iterator it=begin(l); it!=end(l); ++it)
   {
      cout << *it << ' ';       // 1 3 2 4
      *it += 4;                 // Schreibender Zugriff
   }
   cout << '\n';

   // Nutzung eines Const-Iterators - nur lesende Zugriffe erlaubt
   for (list<int>::const_iterator it=begin(l); it!=end(l); ++it)
   {
      cout << *it << ' ';       // 5 7 6 8
   }
}
```

```
Ausgabe
1 3 2 4
5 7 6 8
```

Neben dem Zugriff auf das Element mit dem auf den Iterator angewandten Dereferenzierungs-Operator „*“ gibt es noch eine zweite Möglichkeit für Element-Zugriffe mit dem Pfeil-Operator „->“. Schauen Sie sich das folgende Beispiel an:

```
#include <iostream>
#include <list>
#include <string>
using namespace std;

int main()
{
   list<string> l;
```

```
   l.push_back("Backe");
   l.push_back("backe");
   l.push_back("Kuchen");

   // 1. Schleife - Ausgabe der Strings
   for (list<string>::const_iterator it=begin(l); it!=end(l); ++it)
   {
      cout << *it << ' ';
   }
   cout << '\n';

   // 2. Schleife - Ausgabe der String-Laengen mit dem Punkt-Operator '.'
   for (list<string>::const_iterator it=begin(l); it!=end(l); ++it)
   {
      cout << (*it).length() << ' ';
   }
   cout << '\n';

   // 3. Schleife - Ausgabe der String-Laengen mit dem Pfeil-Operator '->'
   for (list<string>::const_iterator it=begin(l); it!=end(l); ++it)
   {
      cout << it->length() << ' ';
   }
   cout << '\n';
}
```

Ausgabe

```
Backe backe Kuchen
5 5 6
5 5 6
```

- Die erste Schleife gibt die String-Elemente in der Liste aus.
- Wollen Sie aber stattdessen die Längen der Strings ausgeben, müssen Sie die Element-Funktion „length()“ auf die String-Objekte ausführen. Aufgrund der Operator-Prioritäten der Operatoren „*“ und „.“ müssen Sie – wie in der zweiten Schleife umgesetzt – Klammern benutzen, ansonsten würde das „length()“ auf den Iterator angewandt werden (der Punkt Operator hat höhere Priorität als der Dereferenzierungs-Operator).
- In der dritten Schleife sehen Sie die Alternativ-Form mit dem Pfeil-Operator “->” für den Element-Zugriff über Iteratoren. Er ist die Kurzform von „(*).“ und macht semantisch genau dasselbt – ist nur viel kürzer und prägnanter in der Nutzung. Gewöhnen Sie sich direkt an ihn – die Langform „(*).“ ist den meisten Programmierern viel zu lang und wird daher nur wenig genutzt.

Achtung – nochmal der Hinweis: greifen Sie niemals über einen Ende-Iterator eines Containers auf das Element zu. Es existiert nicht, da der Ende-Iterator ja nur auf ein virtuelles Element zeigt. Das Verhalten Ihres Programms ist beim Dereferenzieren des Ende-Iterators undefiniert.

Hinweis – in C++ sind Iteratoren sehr wichtig (wichtiger als in vielen anderen Sprachen). Wir nutzen sie nicht nur, um über Container zu laufen, sondern über jede Art von Mengen – so z.B. um in der Filesystem-Library über die Dateien eines Verzeichnisses.

10.5.1 Iteratoren mit „auto“

Die Erfahrung zeigt, dass sich Einsteiger in die Sprache C++ mit den Typen der Iteratoren sehr schwer tun, da diese nicht intuitiv sind. Mit C++11 kann die Schleife einiges vereinfacht werden, indem auch hier die automatische Typ-Deduktion mit „auto“ eingesetzt wird. Hier

nochmal das obige Listen-Beispiel mit „auto“.

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
   list<int> l;
   l.push_back(1);
   l.push_back(3);
   l.push_back(2);
   l.push_back(4);

   // Nutzung von "auto" fuer die Iterator Variablen-Definition
   for (auto it=begin(l); it!=end(l); ++it)
   {
      cout << *it << ' ';                          // 1 3 2 4
   }
}
```

Ausgabe
```
1 3 2 4
```

In diesem Fall deduziert der Compiler bei „auto“ den Typ des Iterators zu einem non-const Iterator. Wenn Sie aber, was sicherer ist, einen Const-Iterator bekommen wollen, dann nutzen Sie die Funktionen „cbegin“ und „cend“ – sie liefern immer einen Const-Iterator zurück. Auch dies funktioniert natürlich bei allen Containern, und nicht nur der Liste.

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
   list l { 1, 3, 2, 4 };

   // Nutzung von "cbegin" und "cend"
   for (auto it=cbegin(l); it!=cend(l); ++it)
   {
      cout << *it << ' ';
   }
}
```

Ausgabe
```
1 3 2 4
```

10.5.2 Range-basierte For-Schleife

Schon als es allgemein um die Schleifen von C++ ging, haben wir die neue range-basierte For-Schleife für Container kennen gelernt.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> v;
   v.push_back(2);
   v.push_back(4);
   v.push_back(8);
   v.push_back(1);

   for (int value : v)
   {
      cout << value << ' ';
```

```
    }
    cout << '\n';
}
```

Ausgabe
```
2 4 8 1
```

Bei der neuen C++11 For-Schleife definiert man zuerst im Schleifen-Kopf eine Variable (das Schleifen-Objekt), die die einzelnen Objekte im Container während der Durchläufe aufnimmt – und danach gibt man nur noch nach einem Doppelpunkt den Container an:

10.5.3 Fazit

Iteratoren sind ein wichtiges Konzept in C++ – nicht nur in der STL. In C++ werden viele Dinge mit Iteratoren erschlagen, und man kann mit Iteratoren viel mehr machen, als Sie im Augenblick ahnen – das Tutorial kann dies nur ansatzweise zeigen. Hier sollten Sie aber schon mal lernen, dass Iteratoren wichtig sind und Sie sich direkt an sie gewöhnen sollten. Also, wenn Sie über Container laufen: „Immer mit Iteratoren, niemals mit Index“. In den meisten Fällen ist es übrigens noch besser, wenn Sie statt expliziter Schleifen einen Algorithmus nutzen, aber das bekommen wir erst später.

10.6 Arrays

Seit C++11 gibt es in der C++ Standard-Bibliothek ein Array fester Container – wobei die Größe zur Compile-Zeit bekannt sein muss, d.h. die Größe muss eine Compile-Zeit-Konstante sein. Da das Array im Gegensatz zum Vektor eine feste Größe hat, ist er bzgl. Speicherverbrauch und Performance noch etwas besser als der Vektor – solange man die dynamische Größe des Vektors eben nicht benötigt.

Ansonsten ist das Array – wie der Vektor – ein sequentieller Container und bietet wie dieser auch den wahlfreien Zugriff an. Aber wie alle STL-Container unterstützt das Array natürlich auch Iteratoren.

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
   array<int, 2> a1;                                  // Achtung!! (*)
   cout << "size:  " << a1.size() << '\n';
   cout << "a1[0]: " << a1[0] << '\n';
   cout << "a1[1]: " << a1[1] << '\n';

   array<int, 4> a2 = { 11, 22, 33 };                 // (**)
   cout << "size:  " << a2.size() << '\n';
   cout << "a2[0]: " << a2[0] << '\n';
   cout << "a2[1]: " << a2[1] << '\n';
   cout << "a2[2]: " << a2[2] << '\n';
   cout << "a2[3]: " << a2[3] << '\n';

   array<int, 4>::const_iterator it = begin(a2);      // Hier ist in C++11 auch wieder
   array<int, 4>::const_iterator eit = end(a2);       // "auto" eine Alternative
   for (; it!=eit; ++it)
   {
      cout << *it << ' ';
   }
```

```
}
```

```
Mögliche Ausgabe
size:  2
a1[0]: -858993460
a1[1]: -858993460
size:  4
a2[0]: 11
a2[1]: 22
a2[2]: 33
a2[3]: 0
11 22 33 0
```

Wird ein Array Objekt ohne Initialwerte konstruiert (siehe Zeile (*) im Beispiel), so werden die Anfangs-Werte nicht zwingend initialisiert – dies ist abhängig vom normalen Verhalten des Element-Typs als lokale Variable. Alle elementaren Datentypen (u.a. „int's“) haben einen rein zufälligen Startwert – daher die „mögliche Ausgabe“ mit den beiden *komischen* (zufälligen) Werten.

Alternativ kann ein Array Objekt mit einer in geschweiften Klammern gesetzte komma-separierten Liste von Elementen initialisiert werden – siehe Zeile (**) im folgenden Beispiel. Hierbei gilt es folgende Situationen zu unterscheiden:

- Zu wenig Elemente angegeben, Typen mit zufälligen Startwerten (z.B. „int's“)
  => Initialisierung der restlichen Werte mit Null-Bytes.
- Zu wenig Elemente angegeben, Typen mit festen Startwerten (z.B. „string's“)
  => *normale* Initialisierung der restlichen Werte
- Anzahl Elemente in der Liste passt zum Array
  => alles wie erwartet
- Zu viele Elemente angegeben
  => Compiler-Fehler

10.7 Sets

Sets sind in C++ geordnete Mengen.

- Eine Menge ist ein Container, in dem jedes Element nur einmal vorhanden ist. Wird versucht das identische Element ein weiteres Mal in ein Set eingefügt, so wird dies ignoriert – das Set ändert sich also nicht. Mehr Details siehe Kapitel 10.7.2.
- In C++ sind die Elemente im Set geordnet über den Kleiner-Operator „<“ des Element-Typs. Wird daher über ein Set iteriert, so werden die Elemente im Set geordnet ausgegeben (von kleineren zu größeren Elementen) – siehe Beispiel.
- Die Elemente werden in den Container mit „insert“ statt mit „push_back“ eingefügt. Der Name soll hierbei ausdrücken, dass die Elemente nicht hinten angehängt sondern irgendwo in den Container einsortiert werden. Ein Set ändert die Reihenfolge der Elemente im Container selbständig – im Gegensatz zu den sequentiellen Containern wie z.B. Array, Vektor oder Liste.
- Ihre Hauptvorteile sind die implizierte Sortierung und das relativ schnelle Suchen, das in Kapitel 10.7.5 besprochen wird.

```
#include <iostream>
```

```
#include <set>
using namespace std;

int main()
{
   set<int> s;      // Ein Set von Int-Elementen
   s.insert(4);     // Einfuegen der Int-Elemente erfolgt unsortiert
   s.insert(1);
   s.insert(6);
   s.insert(3);
   s.insert(2);
   s.insert(5);

   // Ausgabe der Int-Elemente erfolgt sortiert
   for (set<int>::const_iterator it=begin(s); it!=end(s); ++it)
   {
      cout << *it << ' ';
   }
   cout << '\n';
}
```

```
Ausgabe
1 2 3 4 5 6
```

Beachten Sie in diesem Beispiel bitte das unsortierte Einfügen der Elemente, die aber trotzdem vorhandene sortierte Ausgabe.

Die Ordnung im Set wird default-mässig über den Kleiner-Operator “<” hergestellt. Daher kann man nur Typen in ein Set einfügen, für die der Kleiner-Operator definiert ist. Möchte man Typen ohne Kleiner-Operator einfügen, so muss man entweder selber den Kleiner-Operator definieren oder das Set mit einer anderen Vergleichs-Funktion definieren (siehe Kapitel 10.7.3).

10.7.1 Zeichen- und String-Sortierung

Für Zeichen und Strings ist der Kleiner-Operator definiert, daher können beide problemlos als Element-Typ von Sets benutzt werden. In diesem Fall werden die Zeichen bzw. Texte aber *nur* nach ihrer Zeichen-Kodierung geordnet, es ist keine korrekte lexikalische Ordnung. Solange man ohne Berücksichtigung von Klein- oder Großschreibung und Umlauten auskommt ist das Ergebnis aber meist alphabetisch sortiert und reicht für unsere Einsteiger-Zwecke.

```
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
{
   set<string> s;
   s.insert("horst");
   s.insert("wilhelmine");
   s.insert("alexander");
   s.insert("detlef");
   for (set<string>::const_iterator it=begin(s); it!=end(s); ++it)
   {
      cout << *it << '\n';
   }
}
```

```
Mögliche Ausgabe – abhängig von der Zeichenkodierung Ihrer Plattform
alexander
detlef
```

```
horst
wilhelmine
```

Sollen aber Klein- als auch Großschreibung korrekt berücksichtigt werden, so versagt der Kleiner-Operator für Zeichen:

```
#include <iostream>
#include <set>
using namespace std;

int main()
{
   set<char> s;
   s.insert('a');
   s.insert('A');
   s.insert('b');
   s.insert('B');
   s.insert('c');
   s.insert('C');
   for (set<char>::const_iterator it=begin(s); it!=end(s); ++it)
   {
      cout << *it << ' ';
   }
   cout << '\n';
}
```

```
Mögliche Ausgabe – abhängig von der Zeichenkodierung Ihrer Plattform
A B C a b c
```

Das Ergebnis ist auf den meisten Plattformen das Obige (Groß- vor Klein-Buchstaben), da so die Zeichen-Kodierung im ASCII-Code ist.

10.7.2 Mehrfache Elemente und Element-Identität

Wird ein Element mehrfach in ein Set eingefügt, so wird nur das erste Mal berücksichtigt. Weiteres Einfügen des identischen Objekts wird ignoriert und verändert das Set nicht.

```
#include <iostream>
#include <set>
using namespace std;

int main()
{
   set<int> s;
   s.insert(1);                    // '1' zum ersten Mal einfuegen
   s.insert(1);                    // '1' wieder einfuegen -> wird ignoriert
   s.insert(2);
   s.insert(1);                    // und nochmal die '1' -> wird wieder ignoriert

   cout << "Anzahl Element im Set: " << s.size() << '\n';
   for (set<int>::const_iterator it=begin(s); it!=end(s); ++it)
   {
      cout << *it << ' ';
   }
   cout << '\n';
}
```

```
Ausgabe
Anzahl Element im Set: 2
1 2
```

Trotz dreimaligen Einfügens ist im Beispiel nur einmal die „1“ im Set. Das Set ist halt eine Menge – und nach dem ersten Einfügen ist die „1“ schon in der Menge und kann kein weiteres Mal hinzugefügt werden.

10.7.3 Eigene Sortierung

Dieses Kapitel können Sie noch nicht verstehen, da es u.a. auf Klassen, Operator-Überladung und Funktions-Objekten beruht. Trotzdem möchte ich hier kurz vorstellen, wie Sie ein Set mit einer eigenen Sortierung definieren können.

Sie definieren einfach eine Funktions-Objekt-Klasse (*) mit dem Funktions-Objekt-Operator für den Vergleich (**) und übergeben ihn als zweiten Template-Parameter an das Set (***). Schon wird im folgenden Beispiel über die Länge der Strings sortiert.

```
#include <iostream>
#include <set>
#include <string>
using namespace std;

class StringLengthCmp                                          // (*)
{
public:
   bool operator()(const string& s1, const string& s2) const // (**)
   {
      return s1.length() < s2.length();
   }
};

int main()
{
   set<string, StringLengthCmp> s;                             // (***)
   s.insert("aa");
   s.insert("bbbb");
   s.insert("c");
   s.insert("ddd");
   for (auto it=begin(s); it!=end(s); ++it)
   {
      cout << *it << '\n';
   }
}
```

```
Ausgabe
c
aa
ddd
bbbb
```

Wie Sie an der Ausgabe sehen, werden in diesem Beispiel die Strings nicht nach der Zeichenkodierung sondern nach ihrer Länge sortiert.

Hinweis – der Iterator-Typ im Beispiel ist „set<string, StringLengthCmp>::const_iterator“).

10.7.4 Implementierung

Im Prinzip könnte uns die Implementierung des Sets egal sein – aber zum Verständnis der Vor- und Nachteile eines Sets hilft ein grundsätzliches Verständnis einer Set-Implementierung. Aber Achtung – der ISO C++ Standard legt die Implementierung nicht fest – er beschreibt nur das äußere Verhalten des Containers (hier des Sets) und seiner Funktionen inkl. des Speicherverbrauchs und der Zeitkomplexität. Die zurzeit typische Implementierung eines Sets ist ein *„ausbalancierter binärer Rot-Schwarz Baum“* – da dieser die Anforderungen des Standards erfüllt und gleichzeitig eine sehr gute Performance

aufweist.

Abb. 10-7 : Speichermodell eines Sets todo

Die Elemente im Set werden in Form eines Baums (mit der Wurzel oben) geordnet. Der Baum ist ein binärer Baum, da jeder Knoten bis zu zwei (binär) Kind-Knoten hat. Das Besondere an diesem Baum ist, dass für jeden Knoten gilt: Elemente links von ihm sind alle kleiner, Elemente rechts sind alle größer. Achtung – dies gilt für jeden Knoten auf jeder Ebene.

Einfügen im Set ist im Prinzip relativ problemlos und performant, da nur die richtige Stelle gefunden werden und dort nur ein neuer Knoten hinzugefügt werden muss. Analog zum Einfügen ist auch der Aufwand beim Löschen.

Abb. 10-8 : Einfügen im Set todo

In der Praxis sind Einfügen und Löschen dann aber doch nicht so einfach. Es könnte nämlich passieren, dass der Baum *„entartet“*, d.h. die einzelnen Zweige unterschiedlich tief sind – im Extremfall der Baum zu einer Art Liste mutiert. Würden wir ein Set auf diese einfache Weise implementieren, so würde das Hinzufügen von „1“, „2“, „3“, „4“ und „5“ (in genau dieser Reihenfolge) den Baum wie in der folgenden Abbildung entarten lassen.

Abb. 10-9 : Entstehung eines entarteten Baums todo

Das Set wäre zwar immer noch sortiert und immer noch ein korrektes Set, aber es würde seinen zweiten Hauptvorteil – das schnelle Suchen (siehe nächstes Kapitel 10.7.5) verlieren. Darum darf der Baum nicht entarten, sondern muss stattdessen immer *„ausbalanciert“* bleiben (daher alle Zweige sollten immer die ungefähr gleiche Tiefe haben). Das Verfahren, mit dem diese Ausbalancierung erreicht wird, arbeitet intern häufig mit sogenannten „rot/schwarz“ Markierungen.

Nochmal zurück: die zurzeit typische Implementierung eines Sets ist ein *„ausbalancierter binärer Rot-Schwarz Baum“* – und jetzt verstehen Sie hoffentlich im Prinzip diese Aussage.

10.7.5 Suchen im Set

Neben dem Speicherbedarf, der Performance von typischen Operationen wie Einfügen und Löschen, und speziellen Funktionen wie z.B. wahlfreier Zugriff gibt es weitere Kriterien, nach denen man Container beurteilen muss. Das wohl wichtigste Kriterium nach diesen Basis-Sachen ist die Such-Performance, da Suchen zu den typischsten Aufgaben auf Containern gehört. Bevor wir uns die Such-Performance von Sets anschauen, lassen Sie uns nochmal kurz die Such-Performance der sequentiellen Container (Array, Vektor und Liste) zum Vergleich erarbeiten.

© Detlef Wilkening 2026

Wenn Sie ein Element in einem sequentiellen Container suchen, dann bleibt Ihnen nichts anderes übrig, als von vorne bis hinten über den Container zu laufen, und Element für Element mit Ihrem Such-Element zu vergleichen. Es ist wie im echten Leben – wenn Sie die Herz-Dame in einem Kartenspiel suchen, dann müssen Sie den gesamten Karten-Stapel durchgehen, bis Sie sie haben - oder am Ende wissen, das sie fehlt. Man nennt dies eine lineare Suche, da man den Container linear durchlaufen muss. Das Problem ist, dass das Suchen so ziemlich lange dauert und vor allem mit der Anzahl an Elementen im Container wächst: doppel soviele Elemente im Container, doppelt solange Suchzeit. Zehn mal soviele Elemente im Container, zehn mal solange Suchzeit. Bei z.B. 1000 Elementen im Container sind im Schnitt 500 Zugriffe notwendig.

Abb. 10-10 : Suchen im sequentiellen Container todo

Ist das bei Sets anders? Muss man da nicht auch linear den Container durchlaufen? Nein, schauen Sie sich nochmal einen typischen Set-Baum an:

Abb. 10-11 : Suchen im Set todo

An jedem Knoten müssen Sie nur die folgende Entscheidung treffen:

- Ist es das gesuchte Element? Wenn ja => fertig
- Ist das gesuchte Element kleiner? Wenn ja => in den Kindern links weitersuchen
- Ansonsten muss es größer sein => in den Kindern rechts weitersuchen

Mit jedem Zugriff finden Sie entweder das Element oder halbieren die Suchmenge. Für die Anzahl an Zugriffen um ein Element zu finden, ist also nicht mehr direkt die Anzahl an Elementen im Container verantwortlich, sondern die Tiefe des Baums – und die ergibt sich aus dem 2er-Logarithmus der Element-Anzahl. Bei 10 Elementen also nur 3-4 Zugriffe, bei 100 Elementen ca. 6-7 Zugriffe und bei 1000 Elementen ca. 10 Zugriffe.

In der folgenden Tabelle finden Sie den Vergleich für die Anzahl notwendiger Zugriffe beim Suchen in einem sequentiellen Container und einem binären Baum:

Elemente vs. Zugriffe	Sequentieller Container	Binärer-Baum
10 Elemente	5	4
100 Elemente	50	7
1.000 Elemente	500	10
1.000.000 Elemente	500.000	20
1.000.000.000 Elemente	500.000.000	30

Die Zahlen sollten für sich sprechen – das Suchen im Set ist um Klassen schneller als das Suchen in einem sequentiellen Container.

Und da die Suche im Set so schnell sein kann, gibt es natürlich auch eine so implementierte Find-Funktion auf Sets. Aber Achtung – die Find-Funktion gibt nicht das gefundene Element zurück, sondern einen Iterator darauf. Und falls es das Element im Set nicht gibt – dann den

© Detlef Wilkening 2026

Ende-Iterator des Containers.

```
#include <iostream>
#include <set>
using namespace std;

int main()
{
   set primes { 1, 2, 3, 5, 7, 11 } ;

   set<int>::const_iterator fit = primes.find(3); // '3' ist vorhanden
   if (fit!=end(primes))                          // -> gibt Iterator auf die 3 zurueck
   {
      cout << *fit << " ist im Set drin\n";
   }
   else
   {
      cout << "3 ist NICHT im Set drin\n";
   }

   fit = primes.find(4);                          // '4' ist nicht vorhanden
   if (fit!=end(primes))                          // -> gibt Ende-Iterator zurueck
   {
      cout << *fit << " ist im Set drin\n";
   }
   else
   {
      cout << "4 ist NICHT im Set drin\n";
   }
}
```

Ausgabe

```
3 ist im Set drin
4 ist NICHT im Set drin
```

10.8 Unsortiertes Set

Falls Sie nun glauben, die Suchzeiten von Sets sind nicht mehr zu toppen, so liegen Sie falsch. Ein anderes extrem schnelles Such-Verfahren ist das sogenannte Hashing, dass in C++ u.a. in den Containern „unordered_set und „unordered_map“ zum Zuge kommt. Man kann zeigen, dass ein Suchvorgang hier im Schnitt nur eine feste Anzahl von Zugriffen benötigt (typischerweise ~2), unabhängig von der Anzahl an Elementen im Container. Dafür verliert man den Vorteil der impliziten Sortierung. Ganz im Gegenteil ist die Reihenfolge der Elemente im Container nicht festgelegt, sondern quasi zufällig. Sie kann sich über die Lebensdauer eines Containers sogar ändern. Wir werden aus Zeitmangel den genauen Aufbau eines Hash-Containers hier nicht besprechen.

Das unsortierte Set (Header „unordered_set“) hat genau die gleiche Schnittstelle wie das sortierte Set, abgesehen von den Baum- bzw. Hash-spezifischen Funktionen, die wir hier aber nicht besprechen werden. Wir können also die Beispiele aus dem letzten Kapitel 1:1 auch für unsortierte Container verwenden.

```
#include <iostream>
#include <string>
#include <unordered_set>     // << Anderer Header
using namespace std;

int main()
{
   unordered_set<string> s { "Tom", "Eva" };   // << Anderer Container
   s.insert("Lea");
   s.insert("Max");
```

```
    s.insert("Ada");

    // Ausgabe der Elemente erfolgt zufaellig
    for (unordered set<string>::const iterator it=begin(s); it!=end(s); ++it)
    {
        cout << *it << ' ';
    }
    cout << '\n';
}
```

```
Mögliche Ausgabe (die Reihenfolge in einem Hash-Container ist nicht festgelegt)
Ada Tom Max Lea Eva
```

Beachten Sie in diesem Beispiel bitte die Reihenfolge der Ausgabe. Sie hat weder was mit der Reihenfolge des Einfügens zu tun, noch ist sie alphabetisch. Sie ist nur abhängig vom Hashing, und damit für uns quasi rein zufällig. Sie könnte bei Ihnen, mit einem anderen Compiler oder anderen Einstellungen auch noch wieder anders aussehen.

Das Besondere am Hashing ist das extrem schnelle Suchen – echt beeindruckend, oder?

Elemente vs. Zugriffe	Sequentieller Container	Binärer-Baum	Hashing
10 Elemente	5	4	~2
100 Elemente	50	7	~2
1.000 Elemente	500	10	~2
1.000.000 Elemente	500.000	20	~2
1.000.000.000 Elemente	500.000.000	30	~2

Wenn Ihr Problem auf ein Set angewiesen ist, dann sollten Sie das unsortierte Set bevorzugen, außer Sie benötigen die implizite Sortierung.

10.9 Maps

Wenn man die Sets verstanden hat, dann sind die Maps eigentlich keine Herausforderung mehr – Maps sind sehr ähnlich den Sets, stellen aber einen *„assoziativen“* Container da. Ein assoziativer Container ist ein Container, der einen Schlüssel (Key) mit einem Wert (Value) verbindet (assoziiert).

Schlüssel/Wert-Paare (oder auch Key/Value-Pairs) kennen wir aus dem realen Leben zuhauf. Unser Name ist der Schlüssel im Telefonbuch für unsere Telefonnummer (Wert), ein KFZ-Kennzeichen ist der Schlüssel für eine Auto/Halter-Kombination bei Studenten ist die Matrikel-Nr. der Schlüssel zu allen seinen Daten, Arbeitgeber vergeben Personalnummern als Schlüssel für die Personalakten, oder bei einer Konfigurations-Datei ist der Text-Key der Schlüssel zum eigentlichen Konfigurations-Wert. Assoziative Daten kommen in der Welt viel vor – und darum ist es hilfreich assoziative Container zur Verfügung zu haben.

Die Map speichert ihre Daten (die Schlüssel/Wert-Paare) wie das Set in einem Baum. Sortiert wird dieser aber nicht über den Kleiner-Operator „<“ des Paars, sondern nur über den Kleiner-Operator „<“ des Schlüssels. Damit kann z.B. nach einem Schlüssel gesucht

werden, und der zugehörige Wert ist schnell gefunden.

Abb. 10-12 : Speichermodell einer Map todo

Hinweise:

- Da eine Map „Schlüssel/Wert-Paare“ enthält, benötigt eine Map zwei Typen in spitzen Klammern: der erste Typ für den Schlüssel, der zweite für den Wert.
- Die letzlich in der Map gespeicherten Objekte haben dabei den Typ („Map-Value-Type“) „std::map<typ1, typ2>::value_type“. Achtung – zwei unterschiedliche Maps haben natürlich auch unterschiedliche Wert-Typen.
- Das Einfügen von Schlüssel/Wert Paaren geschieht mit der Element-Funktion „insert“. Aber Achtung – hierbei müssen die beiden Werte in geschweiften Klammern angegeben werden.
- Laufen über eine Map geschieht natürlich auch mit Iteratoren – aber achten Sie darauf, dass auch der Iterator die zwei Typ-Informationen für Schlüssel und Wert bekommt. Mit „auto“ wird das viel einfacher – siehe Beispiel danach.
- Der dereferenzierte Iterator einer Map ist nicht der Schlüssel oder der Wert, sondern natürlich das Map-Value-Paar, d.h. beide Elemente. Der Zugriff auf den Schlüssel bzw. den Wert geschieht dann über die Attribute „first“ (für den Schlüssel) bzw. „second“ (für den Wert) des Iterators.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
   map<int, string> m;
   m.insert({ 2, "Heinrich" });      // Achtung - die Werte stehen
   m.insert({ 3, "Ede" });           //   in geschweiften Klammern...
   m.insert({ 4, "Ansgar" });
   m.insert({ 1, "Willi" });
   m.insert({ 5, "Tom" });

   // Direkte Nutzung von "it->first" und "it->second"
   for (map<int, string>::const_iterator it=begin(m); it!=end(m); ++it)
   {
      cout << it->first << ' ' << it->second << '\n';
   }

   cout << '\n';

   // Umweg ueber Schluessel- und Wert-Variable "key" und "value"
   for (map<int, string>::const_iterator it=begin(m); it!=end(m); ++it)
   {
      int key = it->first;
      string value(it->second);
      cout << key << " -> " << value << '\n';
   }
}
```

Ausgabe

```
1 Willi
2 Heinrich
3 Ede
4 Ansgar
5 Tom

1 -> Willi
```

```
2 -> Heinrich
3 -> Ede
4 -> Ansgar
5 -> Tom
```

Beachten Sie bitte, dass die Einträge in der Map über den Schlüssel sortiert sind – wie man anhand der Ausgabe verifizieren kann – obwohl die Einträge unsortiert in die Map eingefügt worden sind.

Hinweis – wer sich über die Attribut-Namen „first“ und „second“ wundert, und eigentlich die Namen „key“ und „value“ oder ähnlich erwartet hat – der hat gut aufgepasst, aber hier Pech. Man muss beim Programmieren häufiger 2 oder mehr Werte zu einem Paar oder einem Tuple zusammenfassen – und daher hat C++ hierfür fertige Klassen in der Bibliothek. Dies sind „std::pair“ und „std::tuple“. C++ hat bei der Map natürlich bestehende Element wiederverwendet – hier das „std::pair“ – und das bringt halt die allgemeinen Attribut-Namen „first“ und „second“ mit.

Hinweise und Vereinfachungen:

- Wie alle anderen Container, kann man die Map direkt mit den geschweiften Klammern initialisieren – die Schlüssel/Wert-Paare müssen hierbei wiederum auch in geschweifte Klammern gesetzt werden.
- Wie eben schon erwähnt, ist der Typ der Wert/Schlüssel-Paare in der Map „std::map<typ1, typ2>::value_type“. Genau so einen Typ erwartet die Element-Funktion „insert“ auch. Mit den geschweiften Klammern sagen wir dem Compiler, dass er aus den beiden Werten bitte so einen Typ erzeugen soll. Man kann dies aber auch explizit machen – siehe Beispiel. Aber das ist so viel mehr Schreibarbeit, dass das hoffentlich keiner macht.
- Für den Typ des Iterators bietet sich „auto“ an, oder man benutzt direkt die range-basierte For-Schleife, mit oder ohne „auto“ – siehe Zeile (*)
- Man kann hier auch die „structured-binding“ Initialisierung von C++ benutzen und sich den Value-Typ direkt in Schlüssel und Wert aufsplitten lassen – siehe Zeile (**).

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
   map<int, string> m {{ 2, "Heinrich" }};
   m.insert(map<int, string>::value_type(3, "Ede"));    // explizit
   m.insert({ 1, "Willi" });                            // implizit

   for (auto it = cbegin(m); it != cend(m); ++it)
   {
      cout << it->first << " -> " << it->second << '\n';
   }

   cout << '\n';

   for (auto x  : m)                                              // (*)
   {
      cout << x.first << " -> " << x.second << '\n';
   }

   cout << '\n';
```

```
   for (auto [key, value] : m)                                    // (**)
   {
      cout << key << " -> " << value << '\n';
   }
}
```

Ausgabe
```
1 -> Willi
2 -> Heinrich
3 -> Ede

1 -> Willi
2 -> Heinrich
3 -> Ede

1 -> Willi
2 -> Heinrich
3 -> Ede
```

Bemerkung – die Zeilen (*) und (**) im Beispiel sind nicht optimal – besser wären hier eine Const-Referenzen. Aber das werden wir erst später kennenlernen.

10.9.1 Zugriff in die Map mit dem Index-Operator

Eine wichtige Funktion bei Maps ist der Index-Operator „[key]“ (die eckigen Klammern). Im Prinzip implementiert er einen wahlfreien Zugriff – aber nicht auf das n-te Element (über den Index), sondern auf den Wert mit dem Schlüssel „key“ – und zwar sowohl lesend als auch schreibend.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
   map<string, string> m;
   m.insert({"Detlef", "Wilkening"});
   m.insert({"Bjarne", "Stroustrup"});
   m.insert({"Dennis", "Ritchie"});

   // Lesende Zugriffe
   cout << "Wert von Detlef: " << m["Detlef"] << '\n';
   cout << "Wert von Bjarne: " << m["Bjarne"] << '\n';
   cout << "Wert von Dennis: " << m["Dennis"] << '\n';

   // Schreibender Zugriff
   m["Detlef"] = "Wilky";

   // Lesender Zugriff
   cout << "Neuer Wert von Detlef: " << m["Detlef"] << '\n';
}
```

Ausgabe
```
Wert von Detlef: Wilkening
Wert von Bjarne: Stroustrup
Wert von Dennis: Ritchie
Neuer Wert von Detlef: Wilky
```

Aber was passiert beim Zugriff auf einen Schlüssel mit dem Index-Operator, wenn der Schlüssel (und damit auch der Wert) nicht in der Map vorhanden ist? Dann wird der Eintrag automatisch erzeugt – dieser Schlüssel mit einem neu erzeugten Wert-Objekt. Das Wert-

Objekt wird hierbei mit einem expliziten Standard-Konstruktor Aufruf erzeugt. Nun kennen Sie noch keine Konstruktoren, und daher auch noch keinen Standard-Konstruktor, aber im Augenblick reicht die Info, dass dann ein *„quasi-leeres“* Wert-Objekt erzeugt wird. Mehr lernen wir dazu später.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
   map<int, int> m1;
   m1[0];                // Erzeugt einen Map-Eintrag "0 ->  0"                    (*)
   m1[1] =  0;           // Erzeugt einen Map-Eintrag "1 ->  0"
   m1[2] = 22;           // Erzeugt einen Map-Eintrag "2 -> 22"

   for (map<int, int>::const_iterator it = begin(m1); it!=end(m1); ++it)
   {
      cout << it->first << " -> " << it->second << '\n';
   }
   cout << "Neu: " << m1[3] << "\n\n";                                          // (**)

   map<string, string> m2;
   m2["leer"];                                                                 // (***)
   m2["empty"] = "";                                                           // (****)
   m2["C++"] = "Programmiersprache";
   for (map<string, string>::const_iterator it = begin(m2); it!=end(m2); ++it)
   {
      cout << it->first << " -> \"" << it->second << "\"\n";
   }
}
```

```
Ausgabe
0 -> 0
1 -> 0
2 -> 22
Neu: 0

C++ -> "Programmiersprache"
empty -> ""
leer -> ""
```

Hinweise:

- Mit explizitem Standard-Konstruktor-Aufruf erzeugte elementare Datentypen sind ‚0’ initialisiert, d.h. z.B. Bool-Werte sind „false“ und Int-Werte sind ‚0’ – siehe Beispiel. Dies ist unabhängig davon, dass lokale Variablen dieser Typen ansonsten zufällig initialisiert sind.
- String-Objekte werden mit dem Standard-Konstruktor leer erzeugt.
- Die Zeilen (*), (**) und (***) im Beispiel benutzen die implizite Paar-Konstruktion ohne explizite Zuweisung, d.h. es wird der Eintrag nur mit Schlüssel und *leerem* Wert erzeugt.
- Die anderen impliziten Paar-Konstruktionen (z.B. Zeile (****)) überschreiben direkt den leeren Default-Wert. Bitte machen Sie sich klar, dass dieses Einfügen gegenüber der Insert-Funktion oben vielleicht syntaktisch schöner und kürzer ist, aber die Performance schlechter ist. Hier wird zuerst ein Leer-Wert erzeugt, und dieser wied dann wieder überschrieben.
- Zu guter Letzt noch ein Hinweis, den Sie noch nicht in voller Tiefe verstehen können: diese Methode der impliziten Map-Konstruktion funktioniert natürlich nur, wenn die Wert-Objekte default-konstruierbar sind, d.h. einen Standard-Konstruktor haben. Damit ist dann vor allem auch die gesamte Nutzung des Index-Operators „[ ]“ auf Maps nicht möglich,

© Detlef Wilkening 2026

wenn der Wert-Typ keinen Standard-Konstruktor hat.

- Der Index-Operators „[ ]“ auf Maps kann auch nur genutzt werden, wenn die Map nicht konstant ist („non-const“). Da es beim Operator „[ ]“ prinzipiell immer passieren kann das der Key nicht vorhanden ist und der Eintrag angelegt werden muss, ist der Operator „[ ]“ für konstante Maps nicht erlaubt.

10.9.2 Suchen in der Map

Einer der Haupt-Vorteile von Sets ist das schnelle Suchverhalten, da jeder Vergleich den Suchraum halbiert (siehe Kapitel 10.7.5). Da die Map intern wie ein Set aufgebaut ist – mit dem Schlüssel als Sortierkriterium, kann in einer Map genauso schnell nach dem Schlüssel gesucht werden – dazu exisitert auch hier die Element-Funktion „find“. Wird der Schlüssel gefunden, so gibt die Funktion einen Iterator auf den Eintrag (Schlüssel/Wert-Paar) zurück. Wurde kein passender Schlüssel gefunden, wird ein Ende-Iterator zurückgegeben.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
   map<int, string> m;
   m.insert({2, "Heinrich"});
   m.insert({3, "Ede"});
   m.insert({1, "Ansgar"});
   m.insert({4, "Tom"});

   map<int, string>::iterator fi = m.find(3);
   if (fi!=end(m))
   {
      cout << "3 -> \"" << fi->second << '\n';
   }

   fi = m.find(7);
   if (fi==end(m))
   {
      cout << "7 nicht gefunden\n";
   }
}
```

Ausgabe

```
3 -> "Ede
7 nicht gefunden
```

Die typische Anwendung eines assoziativen Containers ist die Suche nach einem Schlüssel, d.h. z.B. der Zugriff in ein Telefonbuch über den Namen. Wird die Suche nach einem Wert benötigt, so muss auch hier eine lineare Suche durchgeführt werden (wie bei sequentiellen Containern) – die natürlich langsam ist.

10.10 Unsortierte Map

Genauso wie bei Sets gibt es auch bei Maps eine unsortierte Map („unordered_map“ aus dem Header „unordered_map“), die intern auf dem Hashing-Verfahren beruht. Auch hier gilt wieder, dass die Schnittstelle identisch zur sortierten Map ist, und das Suchen extrem schnell ist.

```cpp
#include <iostream>
#include <string>
#include <unordered_map>     // << Anderer Header
using namespace std;

int main()
{
   unordered_map<string, string> m { { "BN", "Bonn" } };     // << Anderer Container
   m.insert({ "AC", "Aachen" });
   m.insert({ "HB", "Bremen" });
   m.insert({ "HH", "Hamburg" });

   for (auto [sign, city] : m)
   {
      cout << sign << " => " << city << '\n';
   }
}
```

```
Mögliche Ausgabe (die Reihenfolge in einem Hash-Container ist nicht festgelegt)
BN => Bonn
HH => Hamburg
AC => Aachen
HB => Bremen
```

Wie schon beim Set gilt auch hier die Empfehlung: Wenn Sie die implizite Sortierung nach dem Schlüssel nicht benötigen, bevorzugen Sie die unsortierte Map.

Anmerkung: zusammen mit dem Vektor ist in der Praxis die unsortierte Map der am häufigsten verwendetet Container.

10.11 Algorithmen, Ranges und Lambdas

Bislang haben wir mit den Inhalten der Container noch nicht viel gemacht. Wenn, dann sind wir über den gesamten Container gelaufen und haben alle Elemente ausgegeben. Hierfür haben wir entweder eine normale For-Schleife mit Iteratoren oder die range-basierte For-Schleife genutzt. Für ganz einfache Aufgabenn, die in einer Zeile erledigt werden können, ist das auch okay.

Bei komplizierteren Operationen auf dem Container bzw. den Elementen werden die Schleifen aber schnell groß und komplex. Dann sollten Sie von diesen Schleifen Abstand nehmen, und stattdessen Algorithmen, Ranges und Lambdas einsetzen. Alle diese Dinge werden aus Zeitmangel in der Vorlesung nicht besprochen – ich möchte sie hier daher ganz kurz vorstellen, auch wenn Sie viele Dinge im Detail nicht verstehen werden.

Die C++ Standard-Bibliothek enthält in C++23 116 fertige Algorithmen für alle möglichen Aufgaben. Die meisten Algorithmen gibt es in mehrern Formen (z.B. find in 2 Varianten, oder sort in 4 Varianten). Zählt man alle diese Varianten zusammen, so enthält die C++ Standard-Library weit über 250 fertige Algorithmen. Das folgende Beispiel zeigt mögliche Anwendungen der Algorithmen „accumulate“ und „sort“.

- Alle Algorithmen bekommen als die ersten beiden Argumente Start- und Ende-Iterator übergeben, die den zu bearbeitenden Bereich angeben. Man kann daher auch auf Teil-Bereichen eines Containers arbeiten.

- „accumulate“ aus dem Header „numeric“ summiert (accumuliert) alle Elemente aus dem Bereich auf. Den Start-Wert der Summation gibt man als dritten Parameter an. Man kann „accumulate“ auch eine andere Operation als die Summation mitgeben, zum Beispiel in Form eines Lambdas (siehe zweites „accumulate“ Beispiel).
- „sort“ aus dem Header „algorithm“ sortiert den Bereich mit dem Kleiner Operator. Man kann „sort“ auch eine andere Vergleichs-Operation mitgeben, zum Beispiel in Form eines Lambdas (siehe zweites „accumulate“ Beispiel).

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
using namespace std;

int main()
{
   vector v { 2, 4, 1, 5, 3 };

   // Summieren (accumulieren) der Elemente + 100
   auto sum = accumulate(cbegin(v), cend(v), 100);
   cout << "Summe+100: " << sum << '\n';

   // Produkt der Elemente
   auto pro = accumulate(cbegin(v), cend(v), 1, [](int a, int b){ return a*b; });
   cout << "Produkt: " << pro << '\n';

   for (int x : v) cout << x << ' ';
   cout << '\n';

   // Sortieren des Vektors
   sort(begin(v), end(v));
   for (int x : v) cout << x << ' ';
   cout << '\n';

   // Rueckwaerts-Sortieren des Vektors ohne das erste Element
   sort(++begin(v), end(v), [](int a, int b){ return a>b; });
   for (int x : v) cout << x << ' ';
}
```

Ausgabe

```
Summe+100: 115
Produkt: 120
2 4 1 5 3
1 2 3 4 5
1 5 4 3 2
```

So schön Algorithmen auch sind, sie haben ihre Grenzen. So ist es oft relativ aufwändig, sie zu kombinieren. Auch arbeiten sie nicht lazy, daher die Auswertung der Elemente beschränkt sich nicht auf die notwendigen Elemente. Und häufig ist die Syntax komplizierter als notwendig. C++20 hat Ranges mitgebracht, die quasi „Algorithmen 2.0“ sind. Hier noch ein kurzes Beispiel ohne nähere Erklärung mit den C++20 Ranges:

```
#include <iostream>
#include <ranges>
#include <vector>
using namespace std;

int main()
{
   vector v { 1, 2, 3, 4, 5, 6 };

   auto even = [](int i) { return 0 == i%2; };
   auto view = v | views::filter(even);
   for (int i : view)
```

```
    {
        cout << i << ' ';
    }
}
```

```
Ausgabe
2 4 6
```

10.12 Weiteres

Die C++ Standard-Bibliothek enthält folgende weiteren Container, die wir aus Zeitmangel nicht weiter besprechen können:

- Deque „std::deque“
- Einfach verkette Liste “std::forward_list” – seit C++11
- Multi-Set „std::multiset“
- Hash-Multi-Set „std::unordered_multiset“ – seit C++11
- Multi-Map „std::multimap“
- Hash-Multi-Map „std::unordered_multimap“ – seit C++11
- Stack „std::stack“
- Queue „std::queue“
- Priority-Queue „std::priority_queue“
- Bit-Set „std::bitset“
- Val-Array „std::valarray“
- Flatten-Sets und Maps seit C++23

Auch die Strings der C++ Standard-Bibliothek erfüllen alle Anforderungen an einen STL Container, d.h. sie unterstützen u.a. die entsprechenden inneren Typen und Iteratoren. D.h werden in manchen Büchern Strings unter den STL Containern eingeordnet. Und auch C-Arrays, die in der Vorlesung nicht besprochen werden, sind im Prinzip STL Container. Mit entsprechenden Wrappern lassen sich auch Streams zu Container ähnlichen Objekten machen, die entweder nur gelesen oder nur geschrieben werden können.

Alle STL Container halten ihre eigenen Kopien der zu speichernden Objekte, d.h. die übergebenen Objekte können danach von Ihnen verändert oder zerstört werden – auf die Objekte in den Containern hat das keinen Einfluss. U.a. deshalb müssen die Typen, die in STL-Containern gespeichert werden sollen, auch const-zuweisbar und const-kopierbar sein, was aber für die meisten C++ Typen zutrifft. Daher können aber z.B. Referenzen nicht in einem STL Container gespeichert werden.

© Detlef Wilkening 2026