5 Die Standard Template Library *(STL)*

5.1 Container
5.2 Iteratoren (*)
5.3 Funktionsobjekte
5.4 Algorithmen

(*) Auch in dieser Datei

Lehrbuch: 6.4
Kompendium, 3. Auflage: 8.12, 11.5
Kompendium, 4. Auflage: 11.5

Aufgabe

Verwaltung der Lieferanten einer Komponente in einem Container.

Dann legen Sie mal los und programmieren Sie mir das.

Aufgabenstellung ist vielleicht doch noch etwas dünn.
Sie muss konkretisiert werden.

Wie können wir das besser spezifizieren?

Wenn man nicht weiter weiß, überlegt man sich erst mal einen Test.

Den Test beschreiben erst mal mit Worten.

Verwaltung der Lieferanten einer Komponente in einem Container

Eine Komponente Hauptspeicher wird erzeugt.
Zwei verschiedene Lieferanten werden dem Container hinzugefügt und es wird überprüft, ob einer der Lieferanten ein Lieferant des Hauptspeichers ist.
Anschließend wird überprüft, ob ein dritter Lieferant kein Lieferant des Hauptspeichers ist.

Verwaltung der Lieferanten einer Komponente in einem Container (2)

In einem zweiten Test entfernen wir einen Lieferanten vom Hauptspeicher und überprüfen, ob der Lieferant nun auch wirklich kein Lieferant des Hauptspeichers mehr ist.

Verwaltung der Lieferanten einer Komponente in einem Container

```
bool containerTest() {
    //Eine Komponente Hauptspeicher wird erzeugt.
Komponente hauptspeicher;
   // 2 verschiedene Lieferanten werden dem Container hinzugefügt
Lieferant aa("AA");      hauptspeicher.addLieferant(aa);
Lieferant ab("AB");      hauptspeicher.addLieferant(ab);
   // und es wird überprüft, ob AA ein Lieferant des Hauptspeichers ist.
if (hauptspeicher.hatLieferant(aa) == false) { return false;}
   // Es wird überprüft, ob ein 3. kein Lieferant des Hauptspeichers ist.
Lieferant xx("XX");
if (hauptspeicher.hatLieferant(xx) == true) { return false;}

return true;
}
```

Verwaltung der Lieferanten einer Komponente in einem Container (2)

```
bool containerTest2 () {
Komponente hauptspeicher;
Lieferant aa("AA");      hauptspeicher.addLieferant(aa);
Lieferant ab("AB");      hauptspeicher.addLieferant(ab);

/* In einem zweiten Test entfernen einen Lieferanten vom Hauptspeicher
        und prüfen, dass der Lieferant nun auch wirklich kein Lieferant des
        Hauptspeichers mehr ist.*/
hauptspeicher.entferneLieferant(aa);
if (hauptspeicher.hatLieferant(aa) == true) { return false;}
if (hauptspeicher.hatLieferant(ab) == false) { return false;}

return true;
}
```

Klasse Lieferant

```
class Lieferant {
   public:
   Lieferant(string n): name(n) {};
   string getName() const {return name;}
private:
   Lieferant(const Lieferant&); /* verbotener Aufruf */
   Lieferant& operator=(const Lieferant&); /* verbotener Aufruf */
  string name;
};
```

```
bool operator==(const Lieferant& l1, const Lieferant& l2) {
   return l1.getName() == l2.getName();
}
```

Klasse Lieferant seit C++11

```
class Lieferant {
   public:
   Lieferant(string n): name(n) {};
   string getName() const {return name;}
   Lieferant(const Lieferant&) = delete;
   Lieferant& operator=(const Lieferant&) = default;
private:
    string name;
};
```

```
bool operator==(const Lieferant& l1, const Lieferant& l2) {
   return l1.getName() == l2.getName();
}
```

Klasse Lieferant: Initialisierung auf verschiedene Artem

```
class Lieferant {
   public:
   Lieferant(string n, string vn): name(n) {vorn=vn;};
string name;
string vorn;
};
```

```
class Lieferant {
   public:
   Lieferant(string n, string vn): name(n), vorn(vn) {};
string name;
string vorn;
};
```

```
class Lieferant {
   public:
   Lieferant(string n, string vn)
            {name=n; vorn=vn;};
string name;
string vorn;
};
```

Definition bestimmt die Reihenfolge, nicht Schreibweise hinter Doppelpunkt

```
class Lieferant {
   public:
   Lieferant(string n, string vn): name(n), vorn(vn) {};
string name;
string vorn;
};
```

```
class Lieferant {
   public:
   Lieferant(string n, string vn): vorn(vn), name(n) {};
string name;
string vorn;
};
```

Klasse Lieferant

```
class Lieferant {
public:
   Lieferant(string n): name(n) {};
   string getName() const  {return name;}
private:
   string name;
};
```

```
bool operator==(const Lieferant& l1,
                const Lieferant& l2) {
   return l1.getName() == l2.getName();
}
```

```
Lieferant l1("Hugo");
Lieferant l2("Hans");
Lieferant l3("Hugo");

if (l1 == l2) {cout << "gleich ";}
if (l1 == l3) {cout << "gleich2";}
```

Ausgabe ?

1. gleich
2. gleich2
3. gleich gleich2
4. Keine Ausgabe

Klasse Lieferant

```
class Lieferant {
public:
   Lieferant(string n): name(n) {};
   string getName() const  {return name;}
private:
   string name;
};
```

```
bool operator==(const Lieferant& l1,
                const Lieferant& l2) {
   return l1.getName() == l2.getName();
}
```

```
Lieferant l1("Hugo");
Lieferant l2("Hans");
Lieferant l3("Hugo");

if (l1 == l2) {cout << "gleich ";}
if (l1 == l3) {cout << "gleich2";}
```

Ausgabe ?

1. gleich
2. gleich2
3. gleich gleich2
4. Keine Ausgabe

1. __ gleich _-_ gleich2 __ gleich gleich2 __ Keine Ausgabe

Klasse Komponente

Code auf dieser Seite verstanden?
1. Verstehe nur Bahnhof
2. Verstehe das Problem, aber nicht jedes Schlüsselwort
3. Verstehe die Schnittstelle und den kompletten Code

```
class Komponente {
public:
   Komponente() {};
   void addLieferant(const Lieferant& );
   void entferneLieferant(const Lieferant& );
   bool hatLieferant(const Lieferant& li) const;
private:
    Komponente(const Komponente&); /* verbotener Aufruf */
    Komponente& operator=(const Komponente&); /* verbotener Aufruf */
   vector<Lieferant> container;
};
```

```
void Komponente::addLieferant(const Lieferant& li) { /*…  */ }
void Komponente:: entferneLieferant(const Lieferant& li ) { /*…  */ }
bool Komponente:: hatLieferant(const Lieferant& li) const { /*…  */ }
```

Klasse Komponente (2)

```
void Komponente::addLieferant(const Lieferant& li) {
   container.push_back(li);
}
```

```
void Komponente:: entferneLieferant(const Lieferant& li ) { /*…  */ }
bool Komponente:: hatLieferant(const Lieferant& li) const { /*…  */ }
```

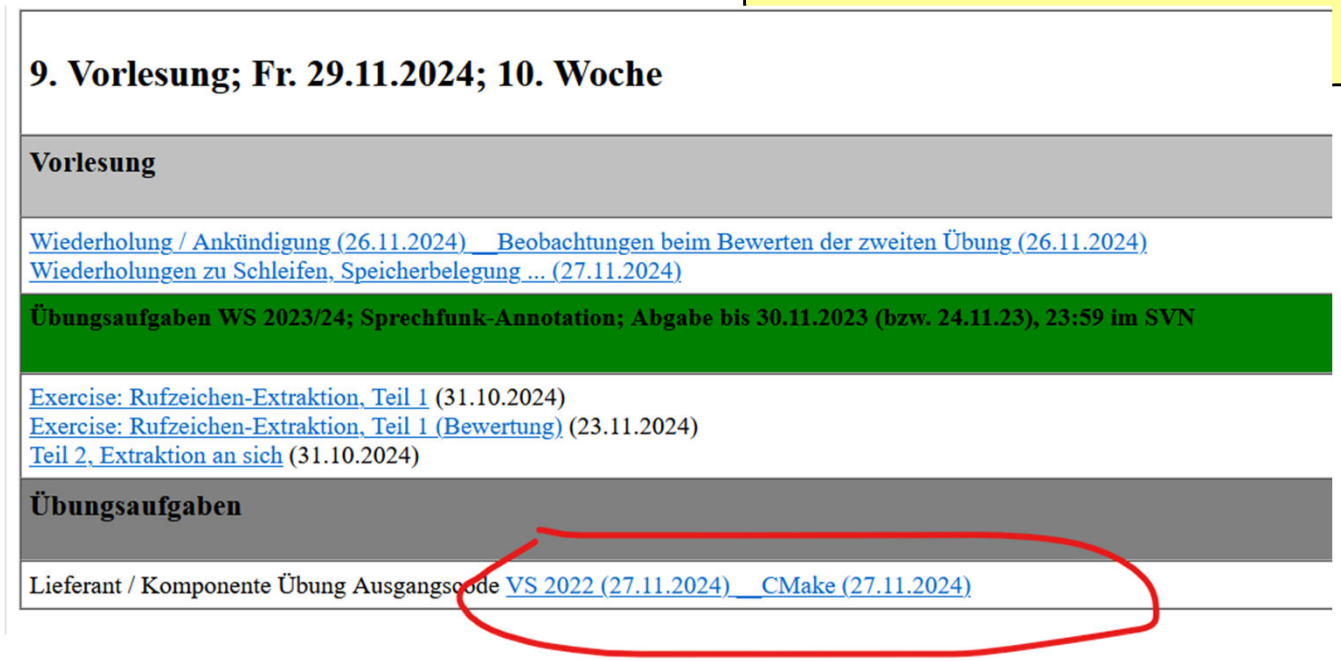
Implementieren Sie die Methode hatLieferent und sorgen dafür,
dass die beiden Tests laufen.
Code finden Sie in Lieferant-01.7z
hatLieferant reicht erst mal. Den Rest bauen wir später zusammen.

Packen Sie den Code aus dem Zip-Archiv
LieferantenKomponenten.sln durch Doppelklick starten
Linux bzw. CLion User müssen sich selber ein Projekt bauen bzw.
ich habe ein Code mit Cmake-Datei hochgeladen

Sie sind dran

Wenn Sie fertig sind, Schauen Sie sich die weiteren Folien hier schon mal an und verwenden statt einer einfachen for-Schleife Iteratoren.

Code auf dieser Seite verstanden?
1. Tests mit hatLieferanten laufen bei mir
2. Sie können weitermachen, ich werde es hier und heute nicht implementieren.



9. Vorlesung; Fr. 29.11.2024; 10. Woche

Vorlesung

Wiederholung / Ankündigung (26.11.2024) __Beobachtungen beim Bewerten der zweiten Übung (26.11.2024)
Wiederholungen zu Schleifen, Speicherbelegung ... (27.11.2024)

Übungsaufgaben WS 2023/24; Sprechfunk-Annotation; Abgabe bis 30.11.2023 (bzw. 24.11.23), 23:59 im SVN

Exercise: Rufzeichen-Extraktion, Teil 1 (31.10.2024)
Exercise: Rufzeichen-Extraktion, Teil 1 (Bewertung) (23.11.2024)
Teil 2, Extraktion an sich (31.10.2024)

Übungsaufgaben

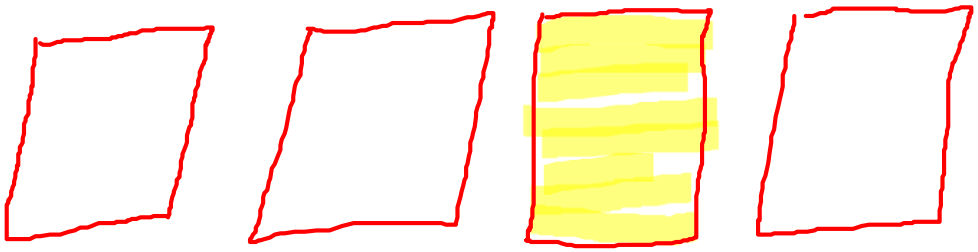
Lieferant / Komponente Übung Ausgangscode VS 2022 (27.11.2024) __CMake (27.11.2024)

Klasse Komponente (3)

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   for (int i=0; i<container.size(); ++i)  {
       if (container[i] == li) {
          return true;
          }
       }
    return false;
}
```

Klasse Komponente (3)

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   for (int i=0; i<container.size(); ++i)  {
       if (container[i] == li) {
          return true;
          }
       }
    return false;
}
```



Klasse Komponente (3)

Container werden normalerweise mit Iteratoren durchlaufen.

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   for (vector<Lieferant>::iterator  iter = container.begin();
         iter != container.end();     ++iter)  {
      if (*iter  == li) {
         return true;
         }
       }
     }
    return false;  }
```

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   for (int i=0; i<container.size(); ++i)  {
      if (container[i] == li) {
         return true;
         }
       }
    return false;
}
```

Iteratoren (2)

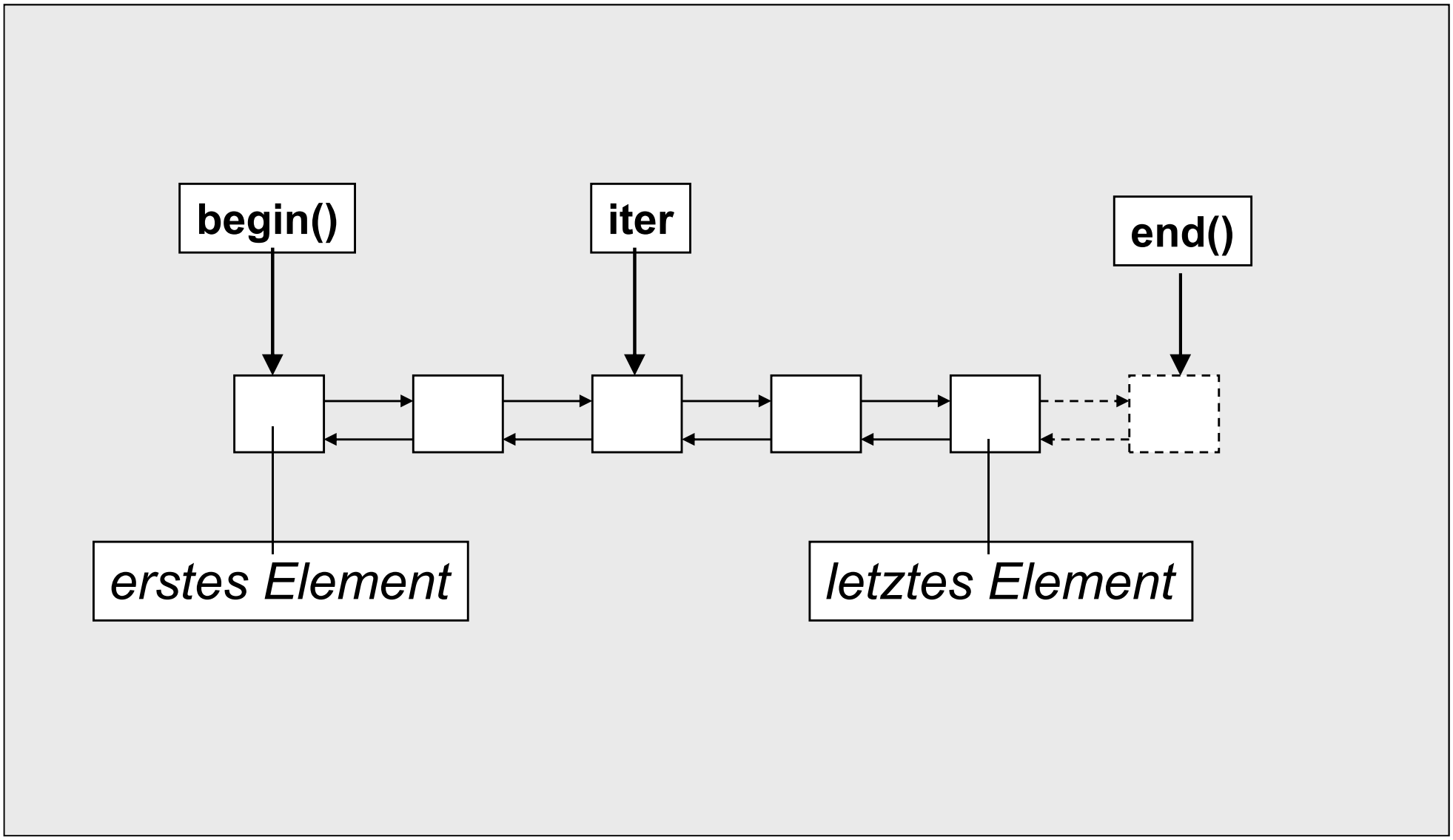
Die Schnittstelle der Iteratoren entspricht genau der Art und Weise wie Zeiger in C/C++ über Arrays wandern.

Im Unterschied zu Zeigern können Iteratoren aber mit beliebigen Containern und nicht nur mit Arrays umgehen. Sie können aber auch mit normalen Arrays umgehen.

Iteratoren von verschiedenen Containern besitzen unterschiedliche Typen, aber die gleiche Schnittstelle.

Um mit Containern arbeiten zu können, stellen Container entsprechende Elementfunktionen bereit, die beiden wichtigsten sind ***begin()*** und ***end()***.

Die Iteratoren *begin* und *end*



Klasse Komponente (3)

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   for (vector<Lieferant>::iterator  iter = container.begin();
         iter != container.end();     ++iter)  {
      if (*iter  == li) {
         return true;
         }
       }
     }
    return false;  }
```

Code auf dieser Seite verstanden

1. Verstehe nur Bahnhof
2. Prinzip der Iteratoren als eine Art Zeiger verstanden
3. Verstanden, könnte Code dieser Methode nun selbst schreiben
4. Kannte Iteratoren schon vorher

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   for (int i=0; i<container.size(); ++i)  {
      if (container[i] == li) {
         return true;
         }
      }
    return false;
}
```

Iteratoren (3)

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

list<float>  cont;
for (int i=1; i<4; ++i) {
   cont. push_front(i*2.2);
   cont. push_back(i*1.1);
}

// Alle Elemente über Iteratoren
// ausgeben:
list<float>::iterator iter = cont.begin();

for (; iter != cont.end(); ++iter) {
   cout << *iter << ' ';
}
```

container::begin() liefert einen Iterator, der die Position des ersten Elements im Container repräsentiert.

container::end() liefert einen Iterator, der die Position **hinter** dem letzten Element im Container repräsentiert.

Falls *begin()* gleich *end(),* ist der Container leer (*container::empty()* liefert *true*).

Iteratoren (4)

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

list<int>  cont;
for (int i=1; i<3; ++i) {
   cont. push_back(i);
}
```

```
// Alle Elemente über Iteratoren
// ausgeben:
list<int>::iterator iter = cont.begin();

for (; iter != cont.end(); ++iter) {
   cout << *iter << ' ';
}
```

Ausgabe ?

1. 0 1
2. 1 2 1 2
3. 1 2
4. 1 1

Iteratoren (4)

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

list<int>  cont;
for (int i=1; i<3; ++i) {
   cont. push_back(i);
}
```

Ausgabe ?

1. 0 1
2. 1 2 1 2
3. 1 2
4. 1 1

```
// Alle Elemente über Iteratoren
// ausgeben:
list<int>::iterator iter = cont.begin();

for (; iter != cont.end(); ++iter) {
   cout << *iter << ' ';
}
```

Ergebnis

__ 0 1 __ 1 2 1 2

_-_ 1 2 __ 1 1

Iteratoren (4)

Der Typ des Iterators ist jeweils im zugehörigen Container per typedef festgelegt, z.B.:

```
class list {
  public:
    typedef ...   iterator;
   ...
};
```

Der genaue Typ ist implementierungsabhängig.

Daneben gibt es auch den Typ **const_iterator**.
Bei ihm ist über den *-Operator (bzw. **->**-Operator) nur ein konstanter (nicht modifizierender) Zugriff auf das Element, auf das der Iterator verweist, möglich.

Hiervon ist jedoch der Typ „***const iterator***“ zu unterscheiden.

Iteratoren (6)

```
for (cont.begin(); iter != cont.end(); ++iter) {
   cout << *iter << ' ';
}
```

Sofern die Elemente des Containers Objekte mit Komponenten sind, sind (normalerweise) auch der Operator „**->**“ und „**.**“ (Punkt) definiert:

```
iter-> komp
(*iter). komp
```

In einigen Implementierungen der STL kann der „->“-Operator aber evtl. nicht definiert sein. *(*iter).komp* funktioniert allerdings immer.

Klasse Komponente (2)

```
bool Komponente:: hatLieferant(const Lieferant& li) const { /*…  */ }
```

Implementieren Sie die obige Methode nochmals und zwar nun mit Iteratoren
Verwenden Sie Ihren vorherigen Code oder Sie finden den Ausgangscode in in Lieferant-02.7z

Packen Sie den Code aus dem Zip-Archiv
DynVektorAlsKlasse-Loes.sln durch Doppelklick starten (doofer Name)
Linux oder CLion User selber Projekt bauen.

Sie sind dran

Wenn Sie fertig sind, Schauen Sie sich die weiteren Folien hier schon mal an und setzen bei der nächsten Aufgabe fort.

Code auf dieser Seite verstanden?

1. Tests mit hatLieferanten mit Iteratoren laufen bei mir
2. Sie können weitermachen, ich werde es hier und heute nicht implementieren.

Überladung des Increment-Operators

```
class iterator  /*...*/
{
   /*...*/

   iterator& operator++() {
      „nächste Position markieren“
      return *this;
   }

   iterator operator++(int) {
      iterator temp = *this;
      „nächste Position markieren“
      return temp;
   }

   /*...*/
};
```

```
list<int>:::iterator iter1, iter2;
. . .
// iter1 zeige auf Liste mit
// den Elementen 1,2,3,4,
// und zwar auf 1

iter2 = ++iter1;
// iter2 und iter1 zeigen nun auf 2

// iter1 zeige vorher auf 2
iter2 = iter1++;
// iter1 zeigt nun auf 3,
// iter2 zeigt auf 2
```

Klasse Komponente (4): Besser mit const_iterator

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   for (auto iter = container.begin(); iter != container.end();     ++iter)  { …
```

```
bool Komponente:: hatLieferant(const Lieferant& li) const {

   for (auto iter : container)  { … // iter wäre hier vom Typ Lieferant (Kopie)
```

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   for (vector<Lieferant>::const_iterator  iter = container.begin();
          iter != container.end();     ++iter)  {
      if (*iter  == li) {
          return true;
          }
       }
     }
    return false;
}
```

Dieser Suchvorgang kommt ganz häufig vor:
Von Anfang bis Ende prüfen, ob ein
Element des Containers gleich einem anderen ist.

Klasse Komponente (4): Besser mit const_iterator

```
bool Komponente:: hatLieferant(const Lieferant& li) const {

  for (const auto& iter : container)  { … // iter wäre hier vom Typ
```

```
if (iter  == li) {
        return true;
        }
      }
    }
   return false;
}
```

Dieser Suchvorgang kommt ganz häufig vor:
Von Anfang bis Ende prüfen, ob ein
Element des Containers gleich einem anderen ist.

Klasse Komponente (5): Besser Algorithmus „find“

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
 vector<Lieferant>::const_iterator iter=
        find(container.begin(), container.end(),  li);
    if (iter == container.end()) { return false;}
    else {return true;}
}
```

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   for (vector<Lieferant>::const_iterator  iter = container.begin();
         iter != container.end();     ++iter)  {
      if (*iter  == li) {
         return true;
         }
       }
     }
    return false;  }
```

Klasse Komponente (5): Besser mit const_iterator

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
  if (find(container.begin(), container.end(),  li) == container.end()) {
     return false;}
  else {return true; }  }
```

```
bool Komponente:: hatLieferant(const Lieferant& li) const {

  return (find(container.begin(), container.end(),  li) != container.end());
  }
```

```
auto hlp = find(container.begin(), container.end(),  li);
```

Was war neu?

1. Stoff von heute aus diesem Foliensatz kannte ich schon
2. Stoff von heute aus diesem Foliensatz kannte ich schon und das eine und/oder andere wurde nochmals verfestigt/geklärt.
3. Ca. 90% kannte ich schon
4. Ca. 50% kannte ich
5. Weniger als 50% kannte ich schon

Wir bauen uns unser eigenes find.

```
int array[100];
/* ..*/
  find(&array[10],  &array[49], 99)

template <typename T, typename W>
T  find (T  anf, T ende, W value) {
    while (anf != ende  && !(*anf == value) ) { // Reihenfolge wichtig
        ++anf;
     }
     return anf;

}
```

```
bool Komponente:: hatLieferant(const Lieferant& li) const {
   return (find(container.begin(), container.end(),  li) != container.end());
   }
```

Algorithmus *find*

```
template <typename InputIter, typename Tp>
inline InputIter find(InputIter first, InputIter last, const Tp& val) {
   while (first != last && *first != val) {
      ++first;
   }
   return first;
}
```