University of Applied Sciences
HOCHSCHULE EMDEN·LEER Networked Systems

Programmieren 3
C++

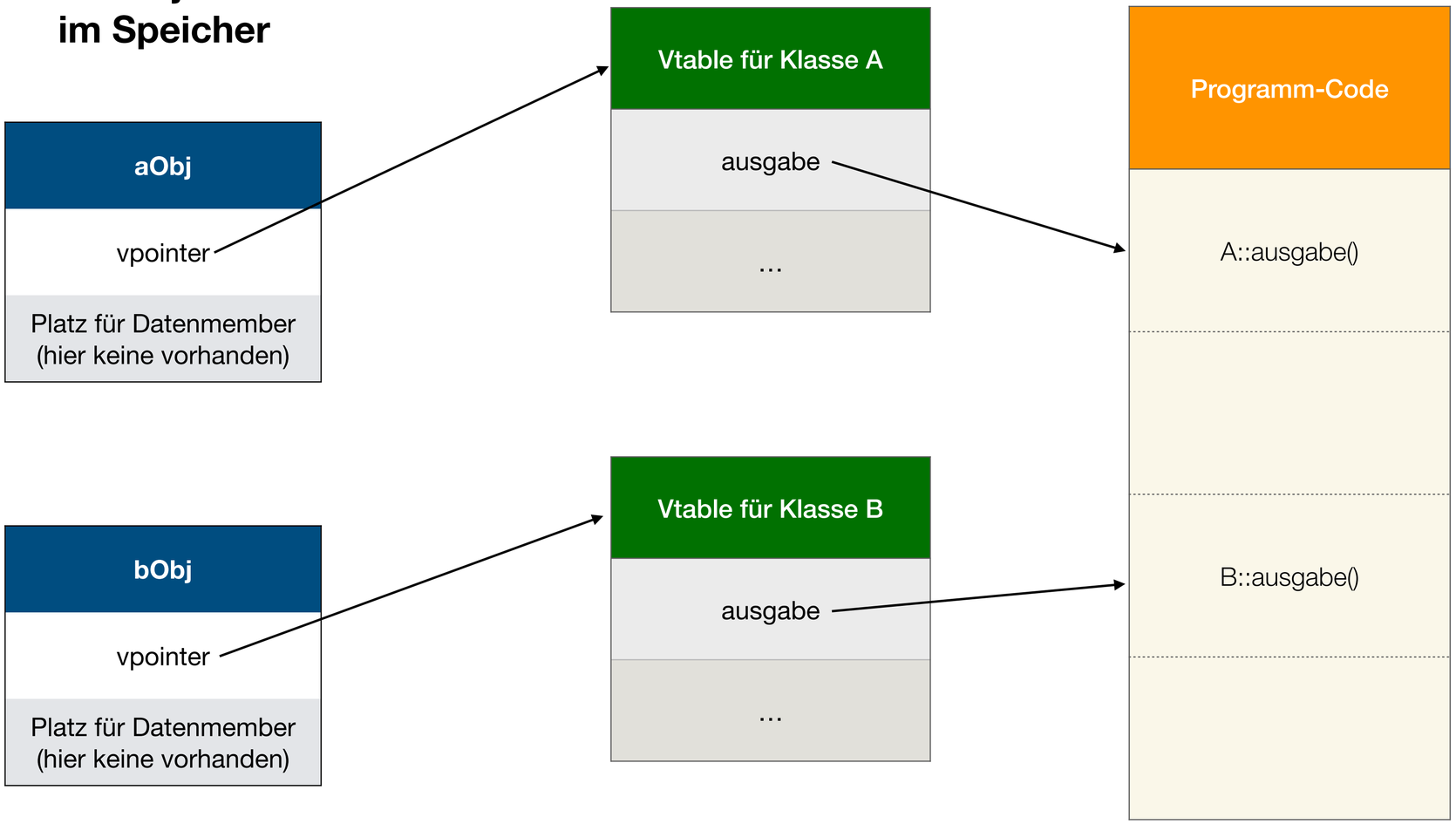
Vorlesung 08: Overloading, Templates, Standard-Template Library

Prof. Dr. Dirk Kutscher
Dr. Olaf Bergmann

Wiederholung

Polymorphie mit virtuellen Funktionen

Repräsentation von Objekten im Speicher



Beim Aufruf von ausgabe() auf einem Objekt vom Typ A oder B wird *zur Laufzeit* ermittelt, welche Funktion wirklich ausgeführt werden muss.

Zusammenfassung

- **Virtuelle Funktionen**
  - Funktionsnamen von Basisklassen bei abgeleiteten Klassen wiederverwenden
  - Das heißt, die Funktion in der abgeleiteten Klasse neu definieren
  - Zur Laufzeit (über Typinformationen im Objekt) die passende Funktion bestimmen und ausführen
- **Rein virtuelle Funktionen (*pure virtual functions*)**
  - Basisklasse gibt nur Interface vor, definiert aber gar keine Funktion
  - Von solchen Klassen kann man keine Objekte erzeugen (Abstrakte Basisklassen)
  - Abgeleitete Klassen *müssen* dann die entsprechenden Funktionen definieren
- **Zuweisungs- und Vergleichsoperationen**
  - Müssen in der Regel auf Eigenschaften das abgeleiteten Klasse zugreifen
  - Schwierig, dies ohne Verrenkungen zu vermeiden
  - Daher besser *nicht* als virtuelle Funktionen definieren

Zugriffsrechte

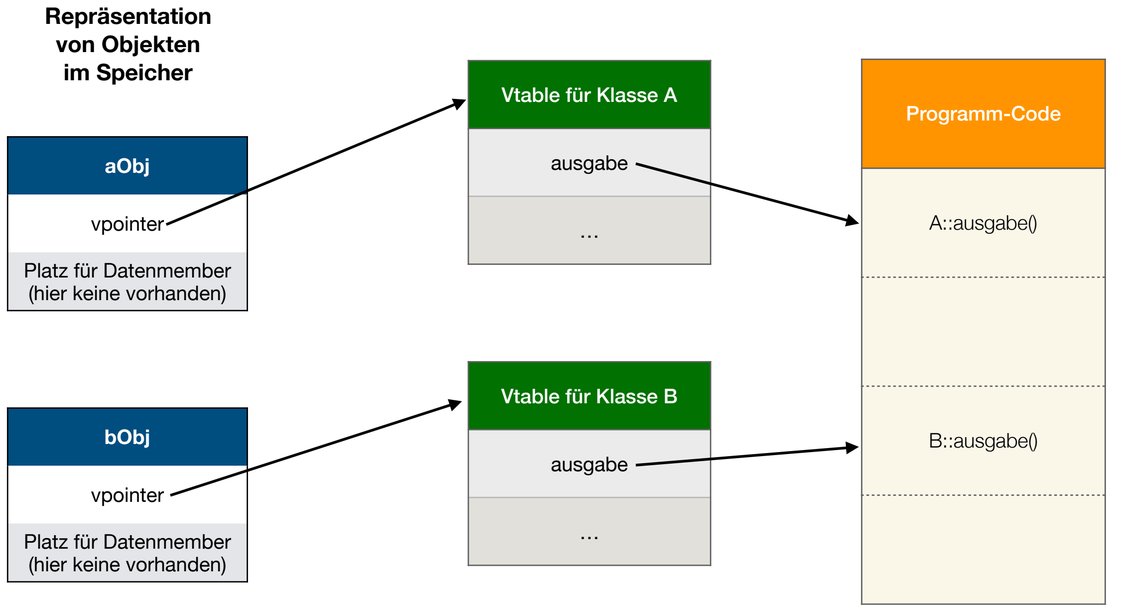
```
class Oberklasse {
private:                                          // Voreinstellung
  int oberklassePriv;
  void privateFunktionOberklasse();
protected:
  int oberklasseProt;
public:
  int oberklassePubl;
  void publicFunktionOberklasse();
};

// Oberklasse wird mit der Zugriffskennung public vererbt
class AbgeleiteteKlasse : public Oberklasse {
  int abgeleiteteKlassePriv;
public:
  int abgeleiteteKlassePubl;
  void publicFunktionAbgeleiteteKlasse() {
    oberklassePriv = 1;                           // Fehler: nicht zugreifbar
    // in einer abgeleiteten Klasse zugreifbar
    oberklasseProt = 2;
    // generell zugreifbar
    oberklassePubl = 3;
  }
};

int main() {
  AbgeleiteteKlasse objekt;
  int m = objekt.oberklassePubl;
  m = objekt.oberklasseProt;                      // Fehler: nicht zugreifbar
```

Zugriffsrecht in der Basisklasse	Zugriffsrecht in einer abgeleiteten Klasse
`private`	kein Zugriff
`protected`	`protected`
`public`	`public`

Man kann auch `private` oder `protected` erben
— meistens nicht sinnvoll

Typ-Informationen zur Laufzeit verwenden

- Virtuelle Funktionen machen das „automatisch“
- Man kann in C++ auch als Programmierer auf Typ-Informationen zugreifen
- Könnte man theoretisch in Methoden abgeleiteter Klassen verwenden, um herauszubekommen, von welchem Typ ein übergebenes Objekt wirklich ist
  - Achtung: sparsam verwenden
  - Meistens benötigt man das nicht



Typ-Informationen zur Laufzeit verwenden

```
#include <iostream>
#include <typeinfo>

class Base {
public:
  virtual const char *type(void) const {
    return typeid(*this).name();
  }
};

class Derived : public Base {};

int main() {
  Base b;
  Derived d;
  std::cout << b.type() << "  " << d.type() << std::endl;
}
```

(plattformabhängige) Repräsentation als Zeichenkette (nicht standardisiert)

Liefert Typ-Information zur Laufzeit.

- Achtung: sparsam verwenden
- Meistens benötigt man das nicht

Casting in C++

- Programmierer soll weiterhin Freiheit haben
- Aber: harmloses von gefährlichem Casting unterscheidbar machen
- Einige Cast-Operationen vom Compiler überprüfbar machen
- Daher unterschiedliche Cast-Operationen definiert:
  - `static_cast`: Typumwandlung zur Übersetzungszeit
  - `dynamic_cast`: Typumwandlung zur Laufzeit
  - `const_cast`: const-Eigenschaften entfernen
  - `reinterpret_cast`: Datentyp beliebig neu interpretieren

Hinweise zu `typeid()` und Cast-Operationen

- Sparsam verwenden!
- Andauernde Abfragen von Typen deuten auf Probleme in der Programmstruktur hin
- Harmoniert nicht mit Polymorphie. Besser: virtuelle Methoden
- Viele Laufzeit-Entscheidungen kann man dem Programm selbst überlassen: virtuelle Methoden

```cpp
#include <iostream>
#include <string>

using namespace std;

class Fahrrad {
  string marke;
public:
  virtual void printBike() {cout << "Marke...";}
  virtual Fahrrad* clone() const {return new Fahrrad(*this);}
};

class EBike: public Fahrrad {
  int kapazitaet;
public:
  virtual void printBike() {cout << "EBike-Marke...";}
  virtual Fahrrad* clone() const {return new EBike(*this);}
};

class MTB: public Fahrrad {
  int reifengroesse;
public:
  virtual void printBike() {cout << "MTB-Marke...";}
  virtual Fahrrad* clone() const {return new MTB(*this);}
};



Fahrrad* copyFahrad(const Fahrrad* fp) {
  return fp->clone();
}
```

Container in C++ (Stdlib)

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

`array` (C++11)	static contiguous array (class template)
`vector`	dynamic contiguous array (class template)
`deque`	double-ended queue (class template)
`forward_list` (C++11)	singly-linked list (class template)
`list`	doubly-linked list (class template)

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

`set`	collection of unique keys, sorted by keys (class template)
`map`	collection of key-value pairs, sorted by keys, keys are unique (class template)
`multiset`	collection of keys, sorted by keys (class template)
`multimap`	collection of key-value pairs, sorted by keys (class template)

https://en.cppreference.com/w/cpp/container

std::map

Mal angucken…

https://en.cppreference.com/w/cpp/container/map
https://en.cppreference.com/w/cpp/utility/pair

```
#include <map>
#include <string>
#include <iostream>

using namespace std;

int main() {
  map<string, int> preis;


  preis["Toastbrot"]=2;
  preis["Schokolade"]=3;
  preis["Kaffee"]=10;

  cout << preis["Kaffee"] << endl;

}
```

Übung

- Schreiben Sie ein Programm, in dem eine `map<string, string>` eine Reihe von Wertpaaren aufbewahrt (z. B. Zuordnung: Land → Hauptstadt)
- Geben sie alle Wertpaare nacheinander aus.
- Warum eignet sich der Container `map` nicht für folgende Zuordnungen:
  - Vorname → Name
  - Stadt → Postleitzahl

Übung

```
 1 #include <map>
 2 #include <string>
 3 #include <iostream>
 4
 5 using namespace std;
 6
 7 int main() {
 8   map<string, string> capitals = {
 9     { "Deutschland", "Berlin" },
10     { "Frankreich", "Paris" },
11     { "USA", "Washington, D.C." }
12   };
13
14   capitals["Samoa"] = "Apia";
15
16   for (auto const &c : capitals) {
17     cout << c.first << ": " << c.second << endl;
18   }
19 }
```

Overloading (Überladen)

```
#include <iostream>
#include <string>

using namespace std;

void print(int a) {
  cout << "int: " << a << endl;
}

void print(float a) {
  cout << "float: " << a << endl;
}

void print(const string& a) {
  cout << "string: " << a << endl;
}
```

```
int main() {
  print(42);
  print(127.2F);
  print("Hello");
}
```

Overloading (Überladen)

```
class Fahrrad {
  string marke;
  string modell;

public:
  Fahrrad(const string& ma, const string& mo)
    : marke(ma), modell(mo) {};
  void print() const {
    cout << "Marke: " << marke << endl
   << "Modell: " << modell
   << endl;}
};

void print(const Fahrrad& a) {
  cout << "Fahrrad:" << endl;
  a.print();
}
```

```
int main() {
  print(42);
  print(127.2F);
  print("Hello");
  print(Fahrrad("Giant", "Cold Rock"));
}
```

Operator-Overloading

```
#include <iostream>
#include <string>

using namespace std;

class Fahrrad {
  string marke;
  string modell;

public:
  Fahrrad(const string& ma, const string& mo)
    : marke(ma), modell(mo) {};
  void print(ostream& xout) const {
    xout << "Marke: " << marke << endl
    << "Modell: " << modell
    << endl;}
};

ostream& operator<<(ostream& xout, const Fahrrad& f) {
  f.print(xout);
  return xout;
}
```

```
int main() {
  cout << Fahrrad("Giant", "Cold Rock")
       << Fahrrad("Foo", "Bar");
}
```

Operator-Overloading

```
#include <iostream>
#include <string>

using namespace std;

class Fahrrad {
  string marke;
  string modell;

public:
  Fahrrad(const string& ma, const string& mo)
    : marke(ma), modell(mo) {}

  const Fahrrad& operator=(const Fahrrad& rhs) {
    marke=rhs.marke;
    modell=rhs.modell;
    return *this;
    }
};
```

```
int main() {
  Fahrrad f1("Giant", "Cold Rock");
  Fahrrad f2("Foo", "Bar");

  f2=f1;
}
```

Operator-Overloading

Zeile	Element-Funktion	Syntax	Ersetzung durch
1	nein	`x ⊗ y`	`operator⊗(x,y)`
2		`⊗ x`	`operator⊗(x)`
3		`x ⊗`	`operator⊗(x,0)`
4	ja	`x ⊗ y`	`x.operator⊗(y)`
5		`⊗ x`	`x.operator⊗()`
6		`x ⊗`	`x.operator⊗(0)`
7		`x = y`	`x.operator=(y)`
8		`x(y)`	`x.operator()(y)`
9		`x[y]`	`x.operator[](y)`
10		`x->`	`(x.operator->())->`
11		`(T)x`	`x.operator T()`
12		`static_cast<T>(x)`	`x.operator T()`
13		`new T`	`T::operator new(sizeof(T))`
14		`delete p`	`T::operator delete(p)`
15		`new T[n]`	`T::operator new[](n * sizeof(T))`
16		`delete[] p`	`T::operator delete[](p)`

`T` ist Platzhalter für einen Datentyp, `p` ist vom Typ `T*`.

Templates

- Generelle Idee: Code wiederverwenden
  - Typsicher, ohne auf Pointer-Casting zurückgreifen zu müssen (wie in C)
- Templates: Funktion/ Klasse einmal definieren
  - Und dann für verschiedene Typen instanziieren

Quicksort in C

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*comp)(const void *, const void *));
```

- `base`: zu sortierendes Array mit `nmemb` Elementen
- `size`: Größe eines Elements
- `comp`: Vergleichsfunktion
- Benutzung

```
int cmp(const void *p, const void *q) {
  return strcmp(*(char **)p, *(char **)q);
}

int main(int argc, char **argv) {
  qsort(++argv, --argc, sizeof(char *), cmp);
  while(argc-- > 0) {
     printf("%s\n", *argv++);
  }
  return 0;
}
```

Templates

Beispiel: quicksort als Template-Funktion in C++

Template-Parameter

```
template <typename T>
void qsort(const T* base, int nmemb, int (comp)(const T& e1, const T& e2)) {
  // qsort-Implementierung
  // würde irgendwann comp(element 1, element 2) aufrufen
}

int intCmp(const int& e1, const int& e2) {
  return e1<e2;
}

int main() {
  int intArray[10];
  qsort<int>(intArray, 10, intCmp);
}
```

Instanziierung der Template-Funktion

Funktions-Templates

```
bool kleiner(int a, int b) {
  return a<b;
}

bool kleiner(float a, float b) {
  return a<b;
}
```

Funktions-Templates

```
bool kleiner(int a, int b) {
  return a<b;
}

bool kleiner(float a, float b) {
  return a<b;
}
```

bequemer:

```
template<class T>
bool kleiner(T a, T b) {
  return a<b;
}
```

Funktions-Templates

```
template<class T>
bool kleiner(T a, T b) {
  return a<b;
}

template bool kleiner<int>(int, int);
// instanziiert kleiner<int>

template bool kleiner<>(char, char);
// instanziiert kleiner<char>,
// Template-Argument wird
// automatisch abgeleitet

template bool kleiner(float, float);
// instanziiert kleiner<float>,
// Template-Argument wird
// automatisch abgeleitet
```

Explizites Instanziieren

Funktions-Templates

```
template<class T>
bool kleiner(T a, T b) {
  return a<b;
}
```

Implizites Instanziieren:

```
int main() {
  bool b=kleiner<int>(5,10); // instanziiert und
                             // ruft kleiner<int>(int, int) auf

  bool c=kleiner<>('a','b'); // instanziiert und
                             // ruft kleiner<char>(char, char) auf

  bool d=kleiner(float(1), float(2));
                             // instanziiert und
                             // ruft kleiner<float>(float, float) auf
}
```

Klassen-Templates

```
template <typename T> class SimpleStack {
public:
  static const unsigned int MAX_SIZE{20};
  bool empty() const { return anzahl == 0; }

  bool full() const { return anzahl == MAX_SIZE; }

  auto size() const { return anzahl; }

  void clear() {
    anzahl = 0;
  }

  const T& top() const;
  void push(const T& x);
private:
  unsigned int anzahl{0};
  T array[MAX_SIZE];
};

template <typename T>
const T& SimpleStack<T>::top() const {
  assert(!empty());
  return array[anzahl - 1];
}


template <typename T>
void SimpleStack<T>::push(const T& x) {
  assert(!full());
  array[anzahl++] = x;
}
```

Template-Parameter

Definition der Member-Funktionen

Klassen-Templates

```
template <typename T> class SimpleStack {
public:
  static const unsigned int MAX_SIZE{20};
  bool empty() const { return anzahl == 0; }

  bool full() const { return anzahl == MAX_SIZE; }

  auto size() const { return anzahl; }

  void clear() {
    anzahl = 0;
  }

  const T& top() const;
  void push(const T& x);
private:
  unsigned int anzahl{0};
  T array[MAX_SIZE];
};

template <typename T>
const T& SimpleStack<T>::top() const {
  assert(!empty());
  return array[anzahl - 1];
}


template <typename T>
void SimpleStack<T>::push(const T& x) {
  assert(!full());
  array[anzahl++] = x;
}
```

Verwendung

Instanziierung

```
int main() {
  SimpleStack<int> einIntStack;

  int i{100};
  while (!einIntStack.full()) {
    einIntStack.push(i++);
  }
  cout << "Anzahl : " << einIntStack.size()
       << '\n';

  cout << "oberstes Element: “
       << einIntStack.top() << '\n';


  SimpleStack<double> einDoubleStack;

  double d{1.00234};
  while (!einDoubleStack.full()) {
    d = 1.1 * d;
    einDoubleStack.push(d);
    cout << einDoubleStack.top() << '\t';
  }
}
```

C++ Standard Template Library (STL)

- Datenstrukturen (Container) und Algorithmen
- Alle auf der Basis von Templates definiert
- Entkopplung von Algorithmen und Datenstrukturen durch Integratoren als Schnittstelle
- Algorithmen greifen über Iteratoren auf Container zu

`std::vector` und andere Container-Klassen

- Sind alle als Template-Klassen definiert
  - Daher schreiben wir immer `vector<int>`

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

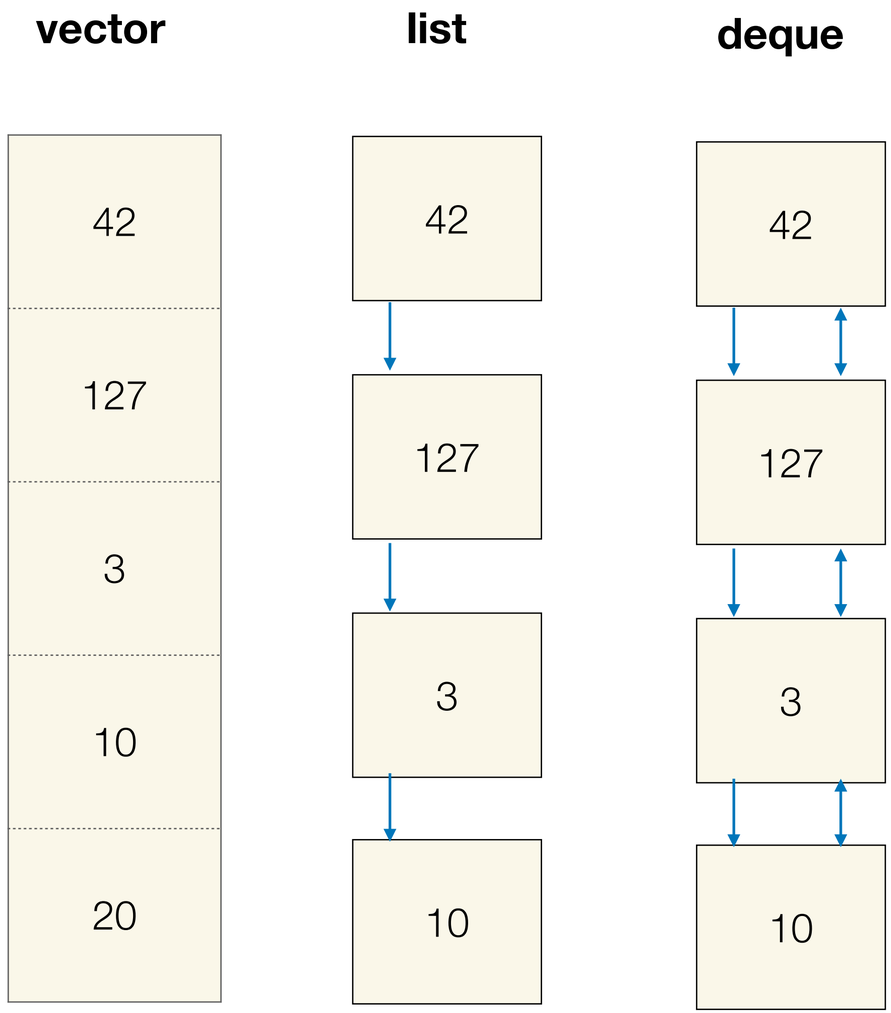
array (C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)
deque	double-ended queue (class template)
forward_list (C++11)	singly-linked list (class template)
list	doubly-linked list (class template)

https://en.cppreference.com/w/cpp/container

Sequenz-Container

- Unterschiedliche Container-Typen mit unterschiedlichen Eigenschaften
  - `std::list` — sequentielle Liste, gut wenn oft neue Elemente an beliebigen Stellen eingefügt werden sollen
  - `std::vector` — Block im Speicher, gut für effizienten Zugriff
  - `std::deque` — Double-Ended Queue, gut für Einfügen, Löschen am Anfang/Ende

Sequenz-Container



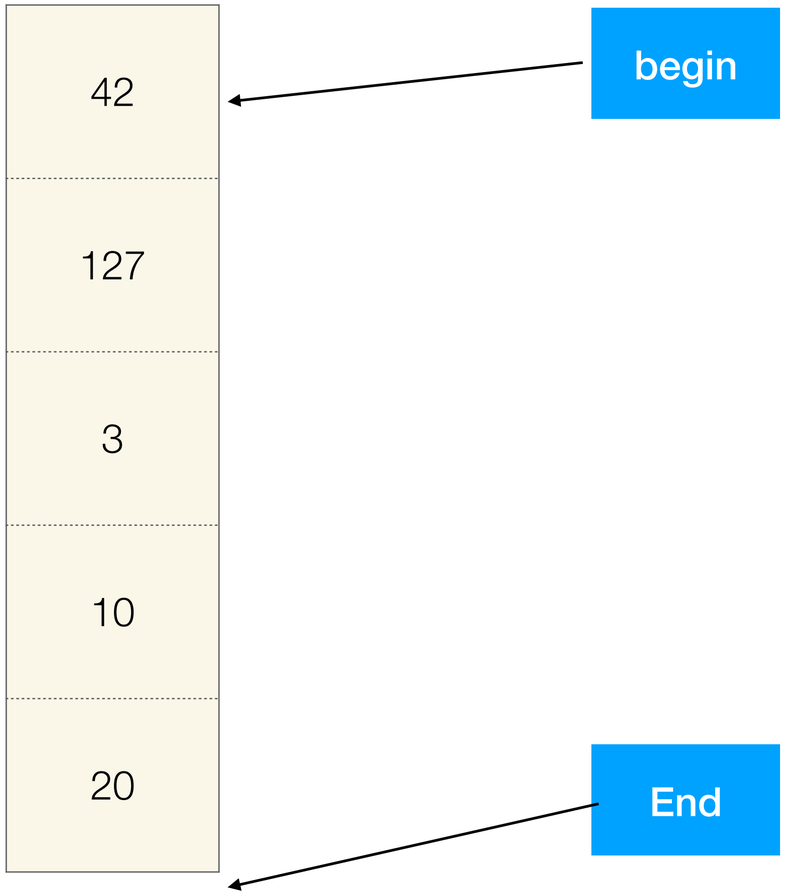
Wie würde man eine Funktion implementieren, die einen Wert in einem `vector`, einer `list` oder `deque` findet?

C++ und Container

Container mit Werten

Iteratoren:
Verweise auf “Positionen”
im Container

Algorithmen:
Generische Funktionen,
die über Iteratoren
auf Container zugreifen

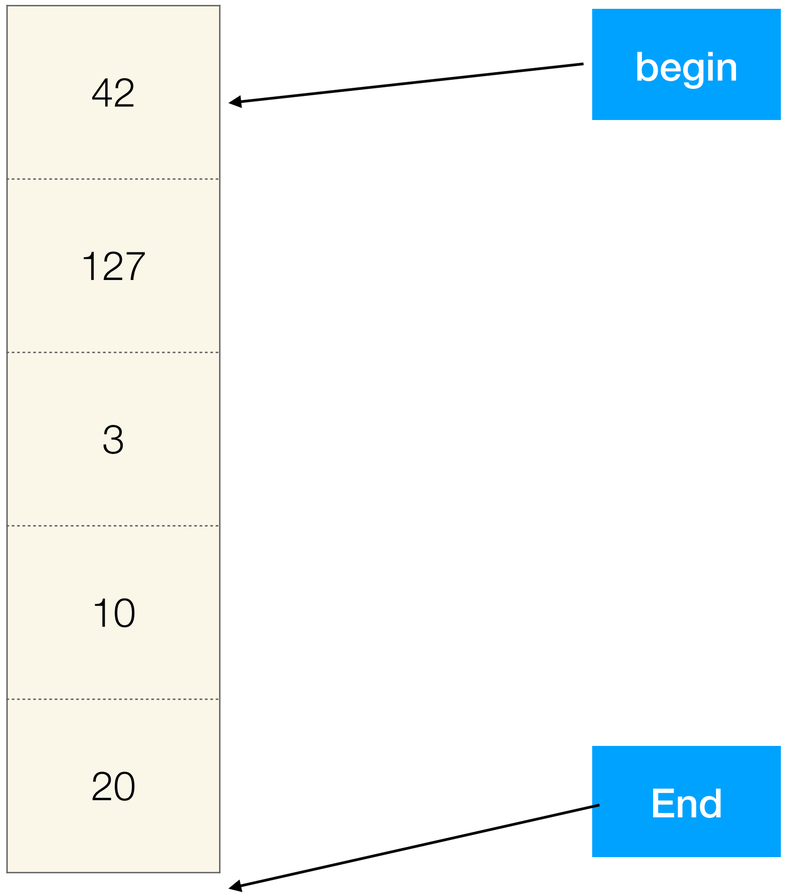



```
find()
count()
sort()
copy()
…
```

Alle als
Templates definiert!

Iteratoren

- Werden ähnlich wie Zeiger verwendet
  - `it++` : Iterator auf nächstes Element zeigen lassen
  - `it--` : Iterator auf vorheriges Element zeigen lassen
  - `*it` : Derefenzieren (Element zurückgeben)

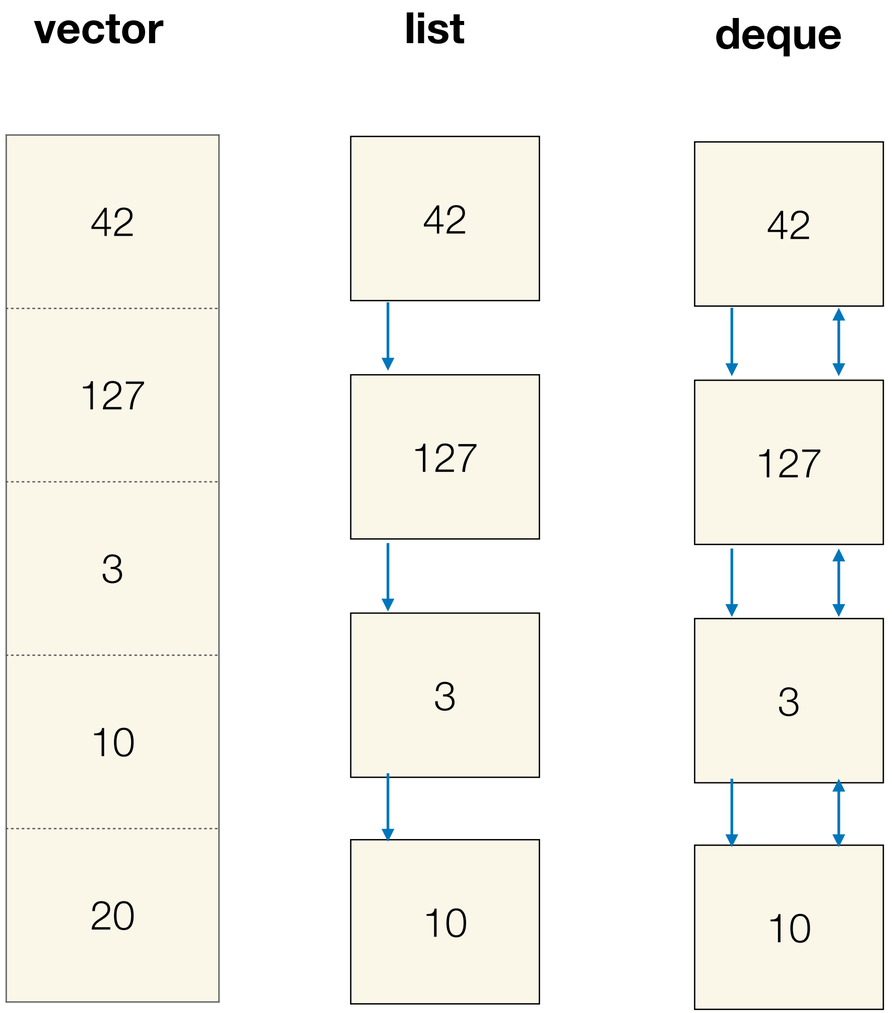


Iteratoren

```
template<typename T>
class Iteratortyp {
public:
   // Konstruktoren, Destruktor weggelassen
   bool operator==(const Iteratortyp<T>&) const;
   bool operator!=(const Iteratortyp<T>&) const;
   Iteratortyp<T>& operator++();                    // präfix
   Iteratortyp<T>  operator++(int);                 // postfix
   T& operator*() const;
   T* operator->() const;
private:
   // Verbindung zum Container ...
};
```

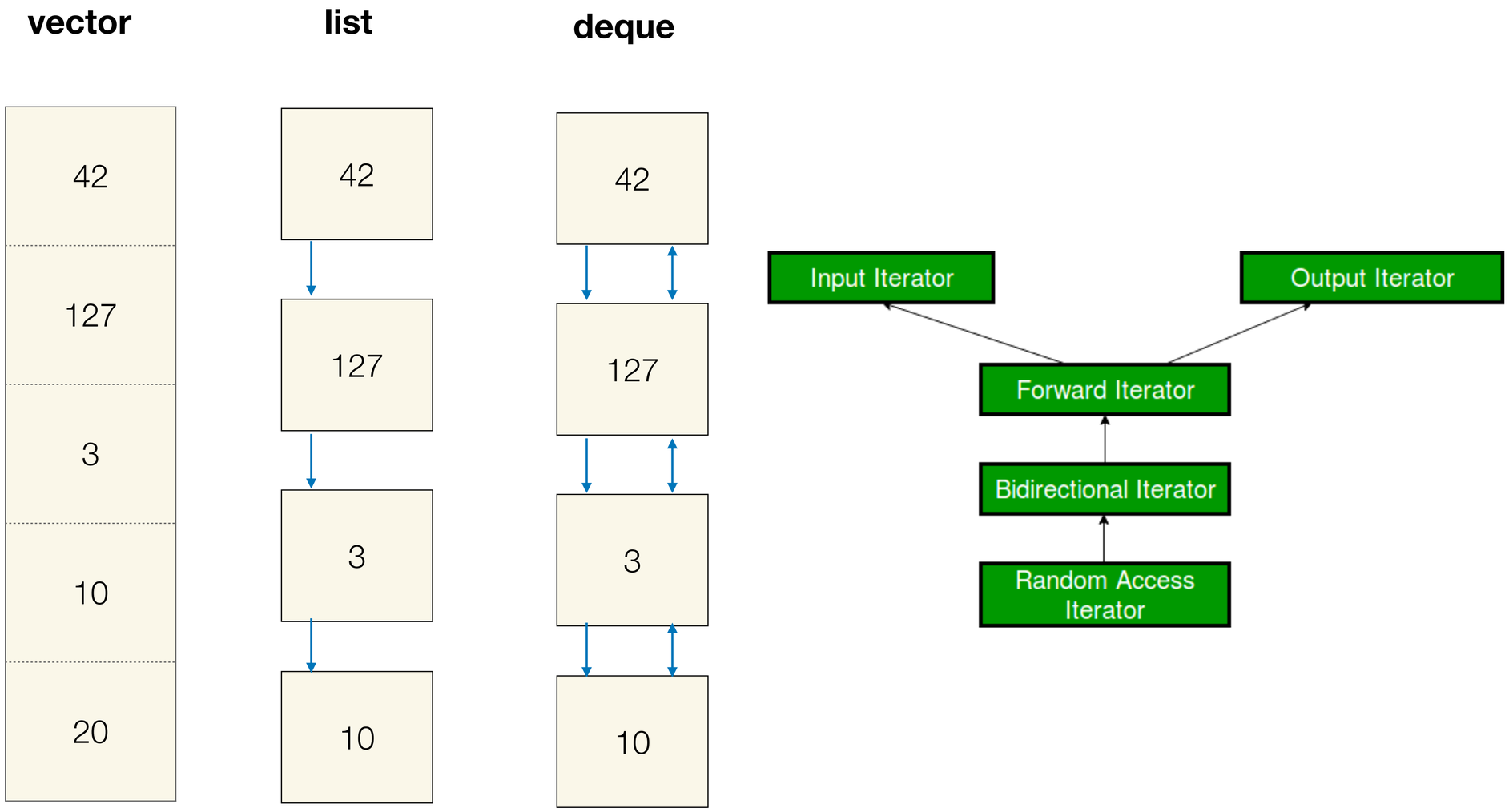
Beispiel (in stdlib etwas anders definiert)

Iteratoren und Container



- Unterschiedliche Algorithmen für Iterator-Methoden
- Abhängig von Container-Typ, für den ein Iterator definiert/ erzeugt wurde
- Einige Methoden sind nicht auf alle Container anwendbar
- z.B. `operator--` bei `list`

Varianten von Iteratoren



Iteratoren: Beispiel 1

```
#include <vector>
#include <iostream>

using namespace std;

int main() {
  vector<int> zahlen;

  zahlen.push_back(42);
  vector<int>::iterator it=zahlen.begin();
  cout << *it << endl;
}
```

Iteratoren: Beispiel 2

```
#include <vector>
#include <iostream>

using namespace std;

int main() {
  vector<int> zahlen{42,43,44};

  for(vector<int>::iterator it=zahlen.begin();
      it!=zahlen.end(); it++) {
    cout << *it << " ";
  }
  cout << endl;
}
```

Iteratoren: Beispiel 3

```
#include <vector>
#include <iostream>

using namespace std;

int main() {
  vector<int> zahlen{42,43,44};

  //   for(vector<int>::iterator it=zahlen.begin();
  for(auto it=zahlen.begin(); // Typ automatisch bestimmen
      it!=zahlen.end(); it++) {
    cout << *it << " ";
  }
  cout << endl;
}
```

Iteratoren: Beispiel 4

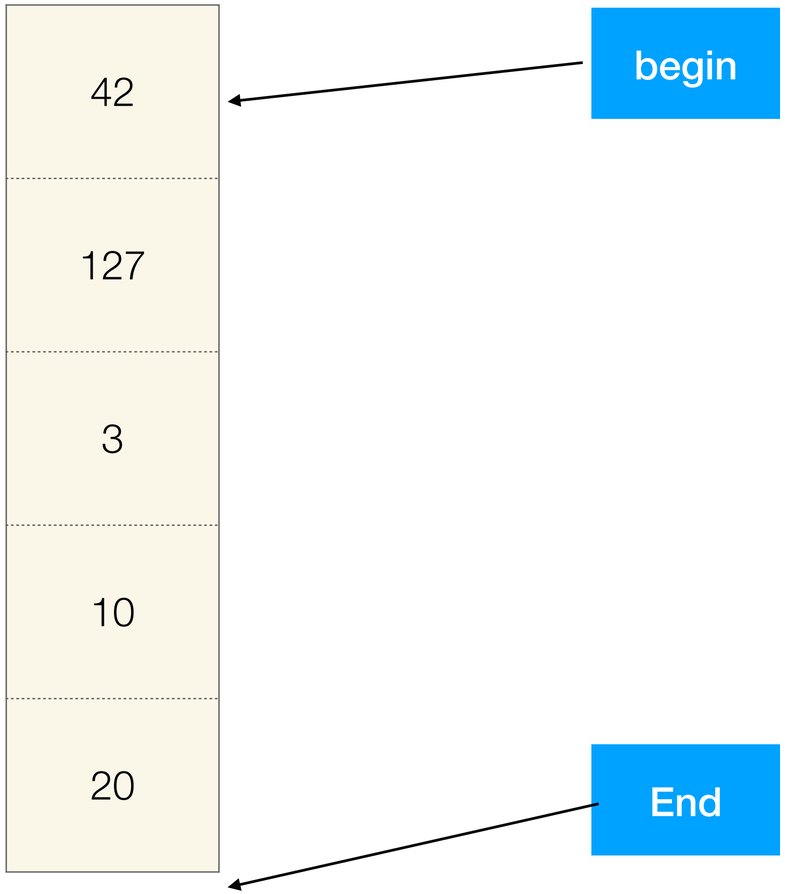
```
#include <vector>
#include <iostream>

using namespace std;

int main() {
  vector<int> zahlen{42,43,44};

  //   for(vector<int>::iterator it=zahlen.begin();…
  for(auto z : zahlen) {  // Iterieren und Derefenzieren
    cout << z << " ";
  }
  cout << endl;
}
```

Algorithmen

Container mit Werten

Iteratoren:
Verweise auf “Positionen”
im Container

Algorithmen:
Generische Funktionen,
die über Iteratoren
auf Container zugreifen




```
find()
count()
sort()
copy()
…
```

Alle als
Templates definiert!

Algorithmen: find (1)

```
int main() {
  vector<int> zahlen{42,43,44};
  int suchwert=43;

  vector<int>::iterator it;

  for(it=zahlen.begin();
      it!=zahlen.end(); it++) {
    if(*it==suchwert) break;
  }

  if(it!=zahlen.end())
    cout << "gefunden: " << *it << endl;
  else
    cout << "nicht gefunden." << endl;

}
```

Algorithmen: find (2)

```
vector<int>::iterator find(vector<int>::iterator start,
           vector<int>::iterator ende, int suchwert) {
  vector<int>::iterator it;

  for(it=start;
      it!=ende; it++) {
    if(*it==suchwert) break;
  }
  return it;
}

int main() {
  vector<int> zahlen{42,43,44};

  vector<int>::iterator it=find(zahlen.begin(), zahlen.end(), 43);

  if(it!=zahlen.end())
    cout << "gefunden: " << *it << endl;
  else
    cout << "nicht gefunden." << endl;
}
```

Algorithmen: find (3)

```
template<class Iterator, class ElementTyp>
Iterator myFind(Iterator start,
          Iterator ende, ElementTyp suchwert) {
  Iterator it;

  for(it=start;
      it!=ende; it++) {
    if(*it==suchwert) break;
  }

  return it;
}
```

Algorithmen: find (3)

```
template<class Iterator, class ElementTyp>
Iterator myFind(Iterator start,
          Iterator ende, ElementTyp suchwert) {
  Iterator it;

  for(it=start;
      it!=ende; it++) {
    if(*it==suchwert) break;
  }

  return it;
}
```

```
int main() {
  vector<int> zahlen{42,43,44};
  vector<string> woerter{"eins", "zwei", "drei"};

  auto it=myFind(zahlen.begin(), zahlen.end(), 43);
  auto it2=myFind(woerter.begin(), woerter.end(), "zwei");

  if(it2!=woerter.end())
    cout << "gefunden: " << *it2 << endl;
  else
    cout << "nicht gefunden." << endl;
}
```

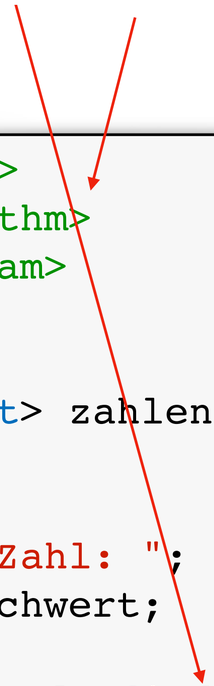
Alternative `find`-Implementierung

Könnte so z.B. in stdlib implementiert sein

```
1 template<class InputIterator, class T>
2   InputIterator find (InputIterator first, InputIterator last, const T& val)
3 {
4   while (first!=last) {
5     if (*first==val) return first;
6     ++first;
7   }
8   return last;
9 }
```

Beispiel

find
(Funktion aus <algorithm>



```cpp
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
  std::vector<int> zahlen{42, 3, 10};
  int suchwert;

  std::cout << "Zahl: ";
  std::cin >> suchwert;

  auto result = std::find(std::begin(zahlen), std::end(zahlen), suchwert);

  if(result !=  std::end(zahlen))
    std::cout << "element " << *result << " gefunden." << std::endl;
  else
    std::cout << "element nicht gefunden." << std::endl;
}
```

Beispiel

`find`
(Funktion aus `<algorithm>`

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
  std::vector<int> zahlen{42, 3, 10};
  int suchwert;

  std::cout << "Zahl: ";
  std::cin >> suchwert;

  auto result = std::find(std::begin(zahlen), std::end(zahlen), suchwert);

  if(result !=  std::end(zahlen))
    std::cout << "element " << *result << " gefunden." << std::endl;
  else
    std::cout << "element nicht gefunden." << std::endl;
}
```

`find` liefert Iterator

Vergleich mit Iterator für “Ende”

Algorithmen: count (1)

```
int main() {
  vector<int> zahlen{42,43,44,42,1,2,5,42};
  vector<char> zeichen{'a', 'b', 'a', 'c', 'd'};

  int c=myCount(zahlen.begin(), zahlen.end(), 42);
  cout << "42 " << c << " mal gefunden." << endl;

  c=myCount(zeichen.begin(), zeichen.end(), 'a');
  cout << "a " << c << " mal gefunden." << endl;
}
```

Algorithmen: count (1)

```
template<class Iterator, class ElementTyp>
int myCount(Iterator first,
            Iterator last, ElementTyp suchwert) {
  int c=0;

  while(first!=last) {
    if(*first==suchwert) c++;
    first++;
  }
  return
}
```

```
int main() {
  vector<int> zahlen{42,43,44,42,1,2,5,42};
  vector<char> zeichen{'a', 'b', 'a', 'c', 'd'};

  int c=myCount(zahlen.begin(), zahlen.end(), 42);
  cout << "42 " << c << " mal gefunden." << endl;

  c=myCount(zeichen.begin(), zeichen.end(), 'a');
  cout << "a " << c << " mal gefunden." << endl;
}
```