



# Programmieren 3

## C++

### Vorlesung 9: Algorithmen der Standard Template Library

Prof. Dr. Dirk Kutscher  
Dr. Olaf Bergmann

# Terminplan

23.11.2023: Vorlesung 9

**28.11.2023: Abgabe Übung 3**

30.11.2023: Vorlesung 10

**07.12.2023: Vorlesung entfällt**

14.12.2023: Das Semester im Schnelldurchlauf

**19.12.2023: Abgabe Übung 4**

21.12.2023: Probeklausur

04.01.2024: Fragen und Antworten

**12.01.2024: Klausur (T-Foyer)**

# Lehrevaluation

Vorlesung



<https://evasys.hs-emden-leer.de/evasys/online.php?pswd=AXVYM>

Praktikum



<https://evasys.hs-emden-leer.de/evasys/online.php?pswd=PNHLF>

# Wiederholung

# Templates

- Code/Algorithmen **wiederverwenden**
- Typsicher ohne Pointer-Casting
- Templates: Funktion oder Klasse...
  - einmal *definieren*
  - für verschiedene Typen *instanziiieren*
  - bei Bedarf *spezialisieren*

## Quicksort in C

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

- `base`: zu sortierendes Array mit `nmem` 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;  
}
```

# Funktions-Templates

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

## Herleiten von Template-Argumenten beim Instanziiieren

```
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

## Verwendung

Instanziierung

```
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;
}
```

```
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 Iteratoren 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

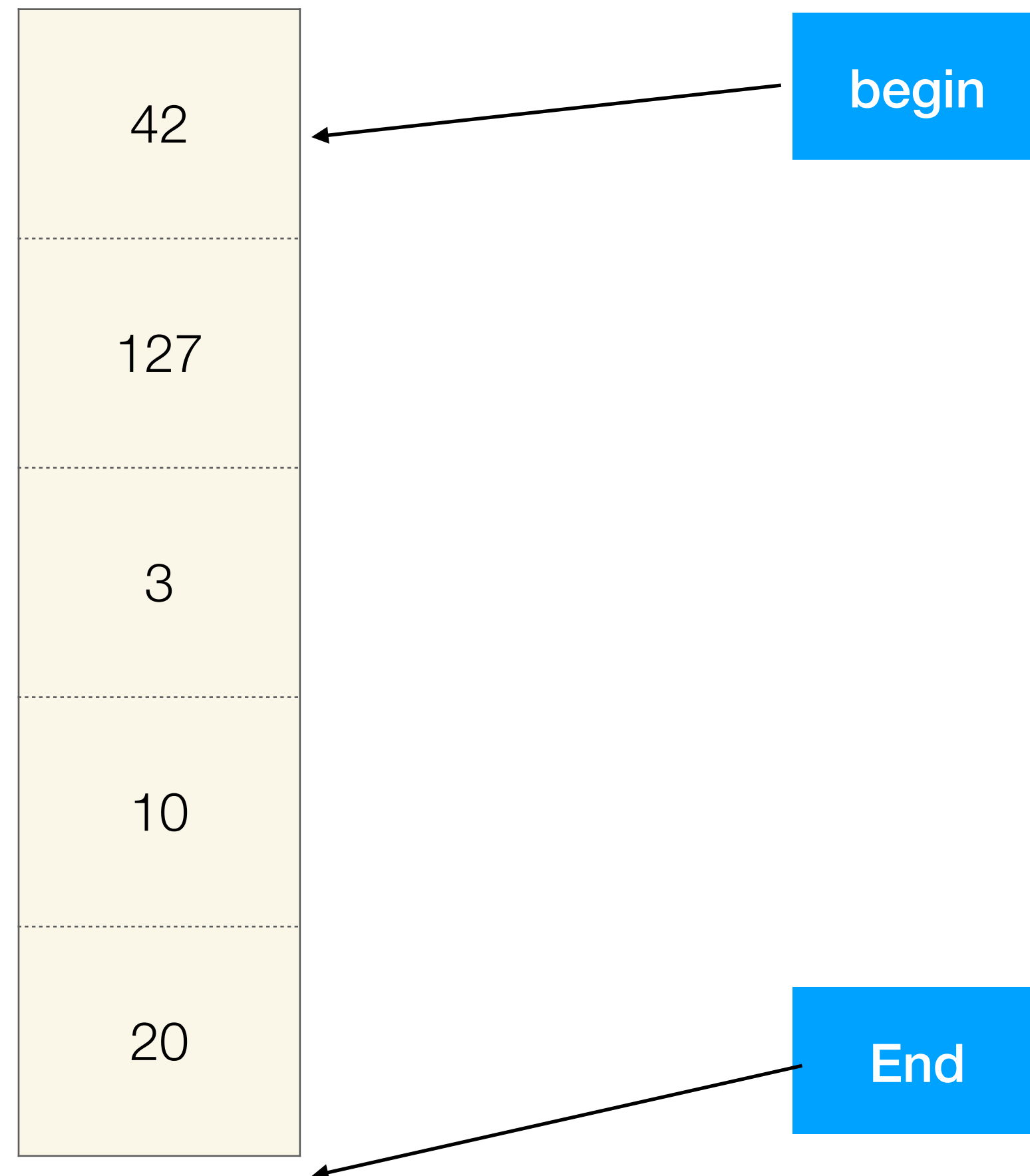
Sequence containers implement data structures which can be accessed sequentially.

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

# C++ und Container

**Container mit Werten**      **Iteratoren:**  
**Verweise auf “Positionen”**  
**im Container**

**Algorithmen:**  
**Generische Funktionen,**  
**die über Iteratoren**  
**auf Container zugreifen**

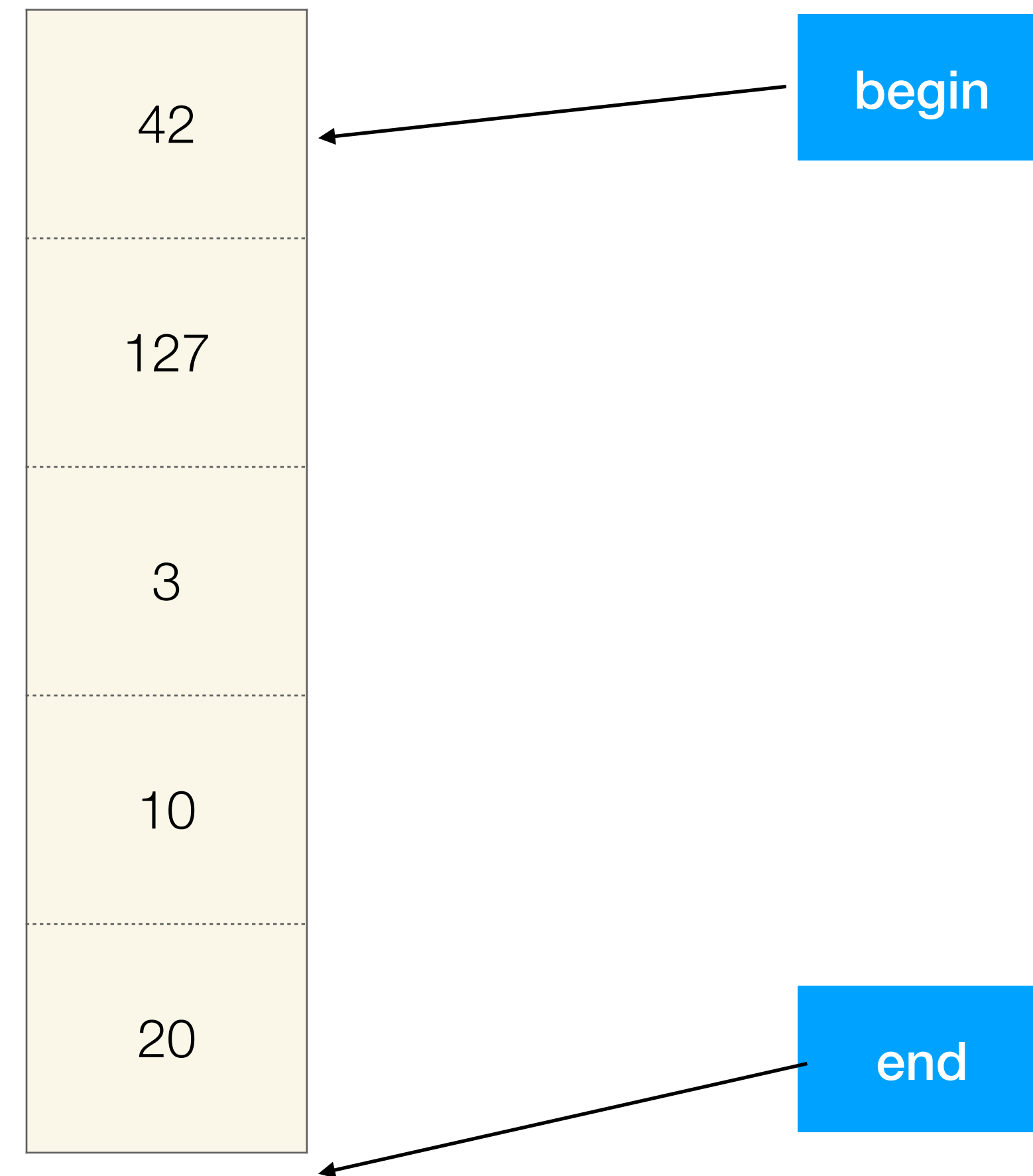


```
find()  
count()  
sort()  
copy()  
...
```

**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` : Dereferenzieren (Element zurückgeben)

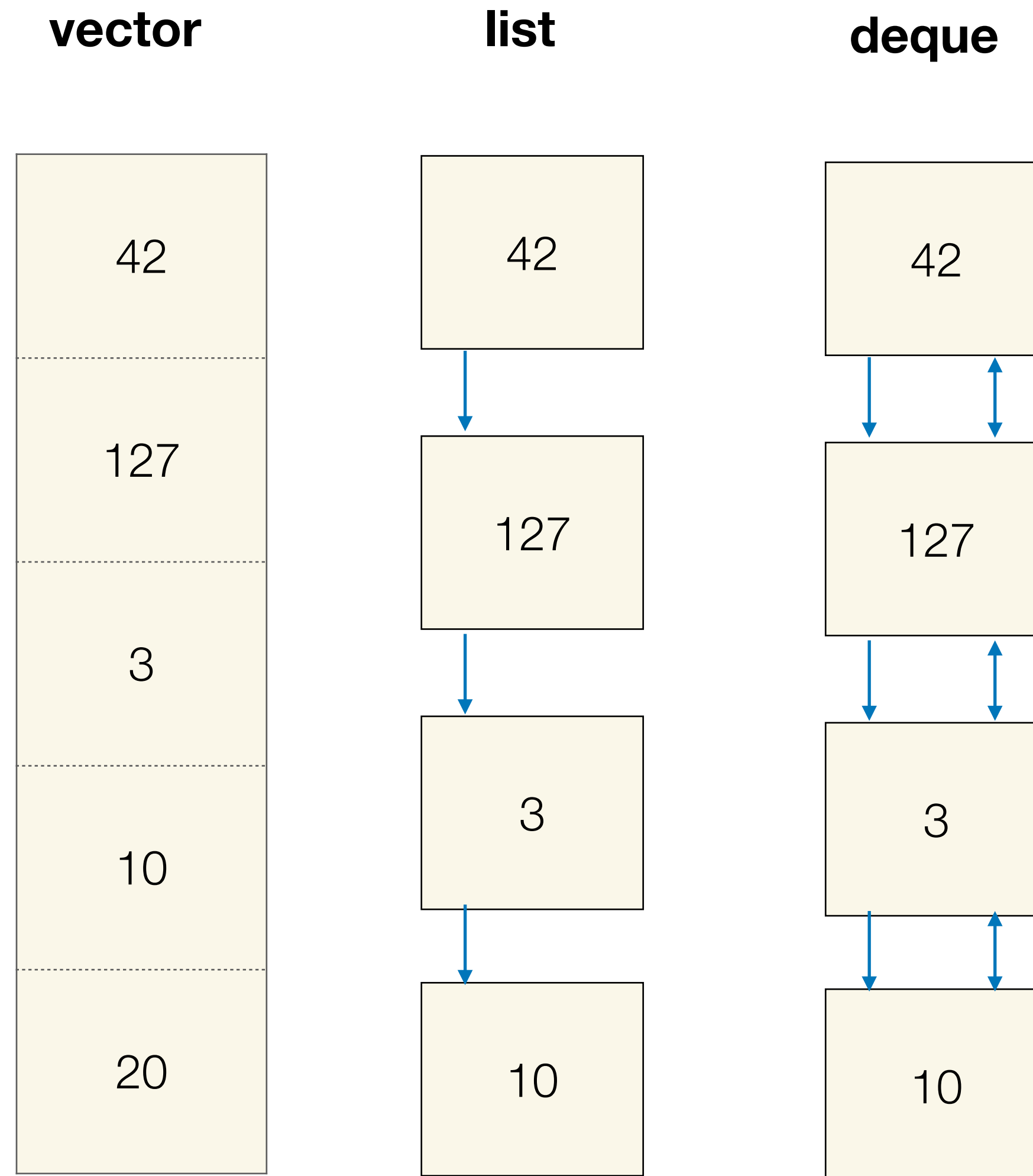


# Iteratoren

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

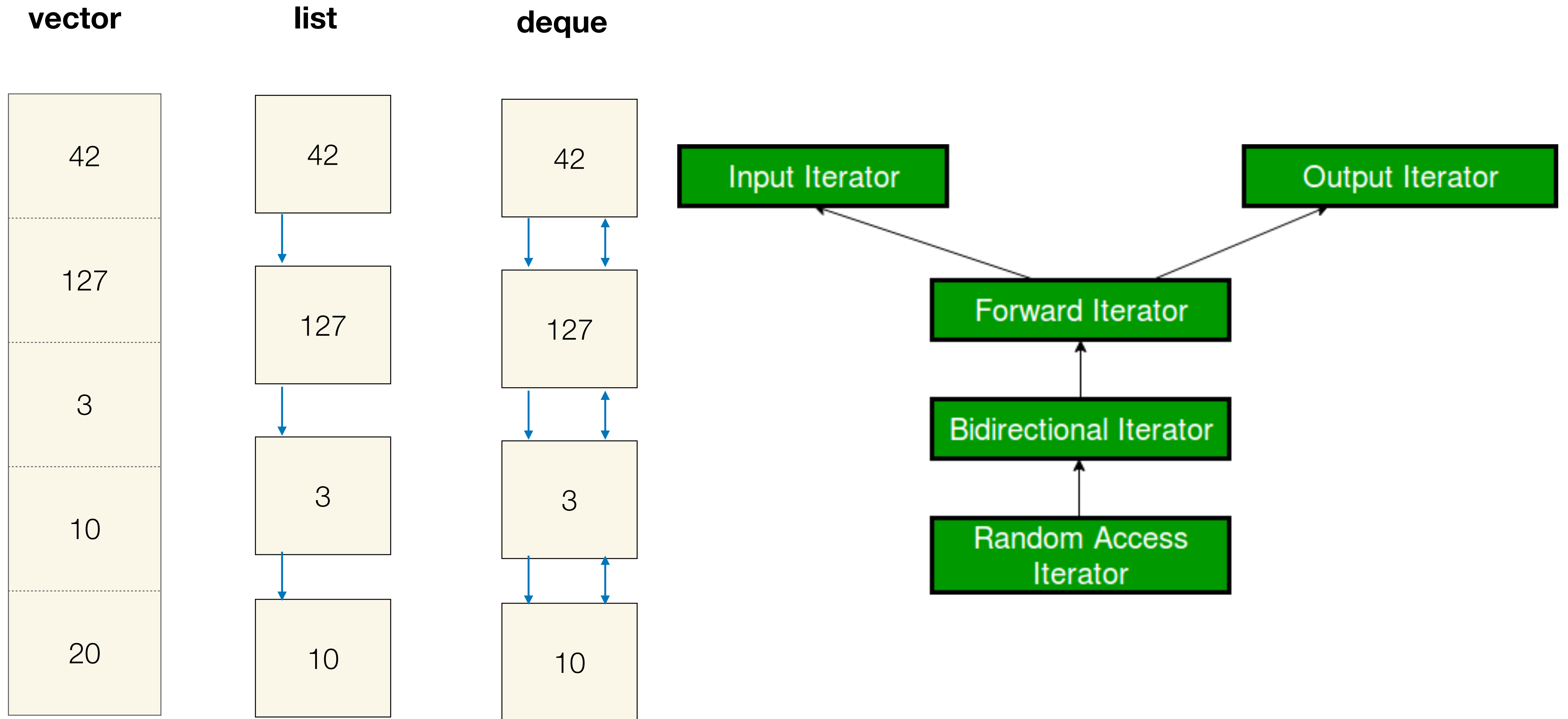
**Beispiel (in stdlib etwas anders definiert)**

# 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--` für `list` nicht definiert

# Varianten von Iteratoren



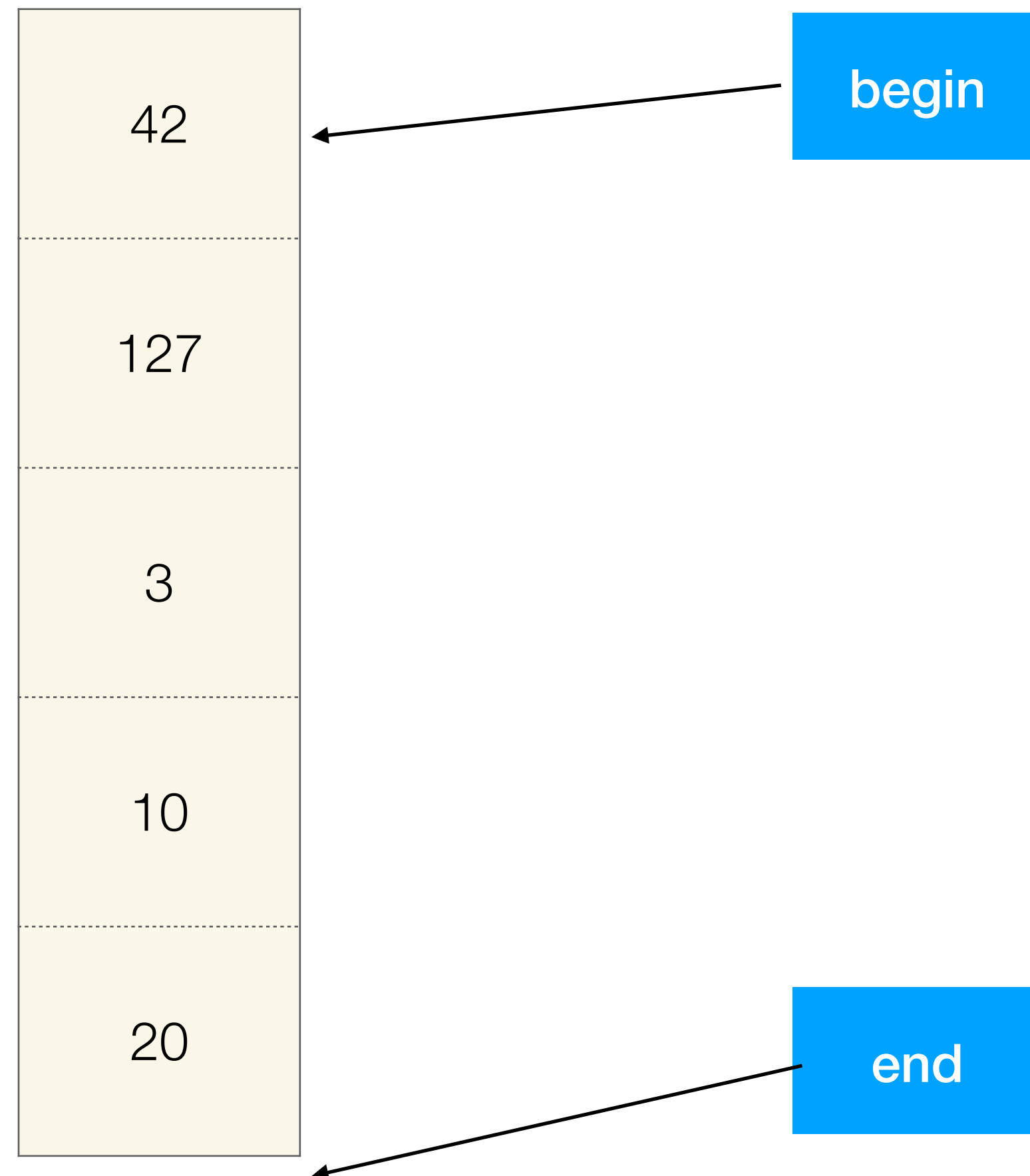


# Algorithmen

Container mit Werten

Iteratoren:  
Verweise auf “Positionen”  
im Container

Algorithmen:  
Generische Funktionen,  
die über Iteratoren  
auf Container zugreifen



```
find()  
count()  
sort()  
copy()  
...
```

**Als Templates  
definiert!**

# Algorithmen: find (3)

```
template<class Iterator, class ElementTyp>
Iterator myFind(Iterator first,
               Iterator last, ElementTyp suchwert) {

    while(first != last) {
        if(*first == suchwert)
            break;
        first++;
    }

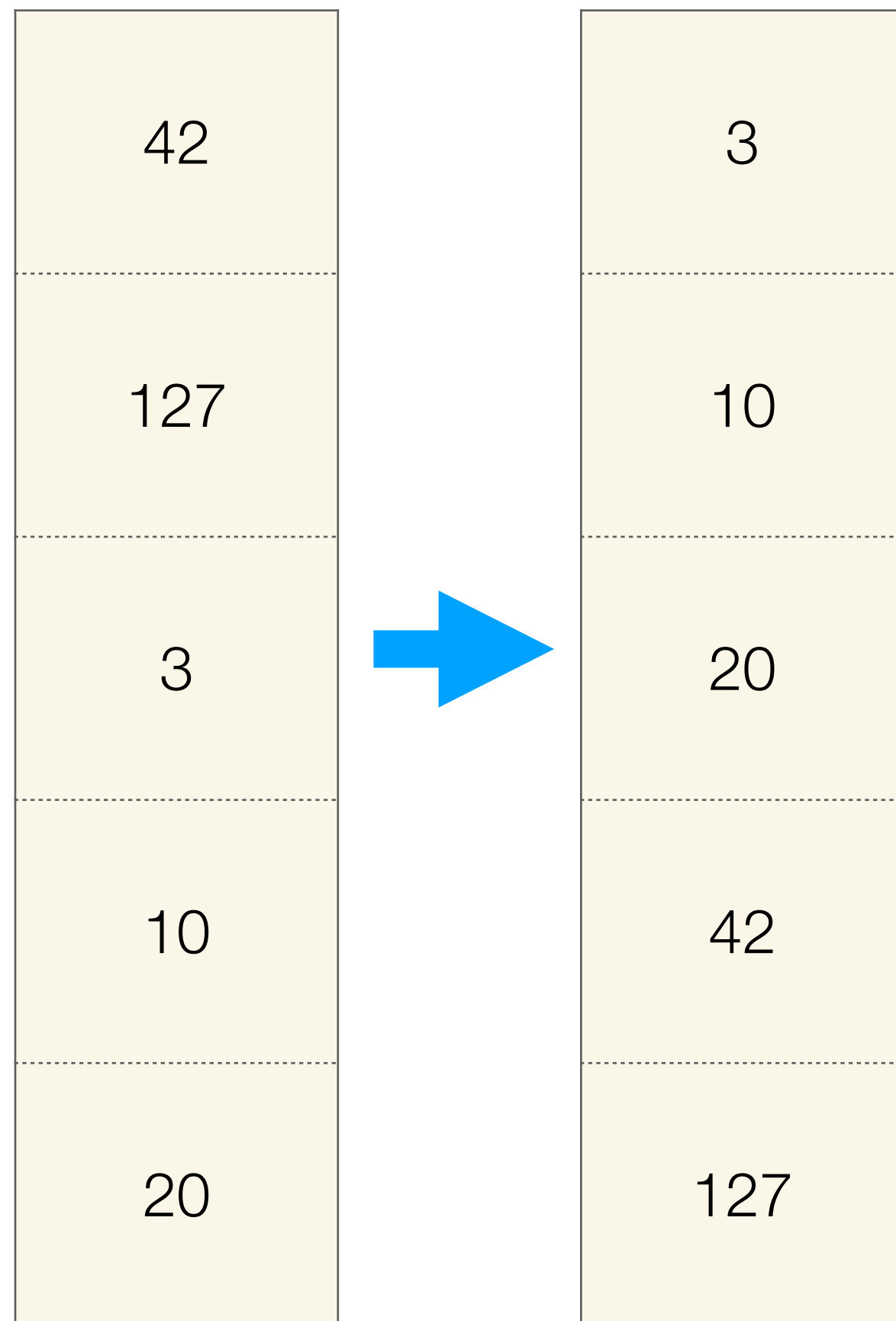
    return first;
}
```

```
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;
}
```

# `std::sort`



- Werte innerhalb eines Bereichs in aufsteigender Reihenfolge sortieren
- Natürlich auch wieder für beliebige Container-Klassen und Elementtypen
- Sortier-Algorithmus?

# sort Implementieren

```
template<class T>
void print(const T& val) {
    cout << val << " ";
}

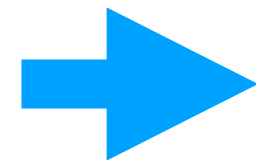
int main() {
    vector<int> zahlen{42,43,44,42,1,2,5,42};
    vector<char> zeichen{'a', 'b', 'a', 'c', 'd'};

    mySort(zahlen.begin(), zahlen.end());
    for_each(zahlen.begin(), zahlen.end(), print<int>);
    cout << endl;

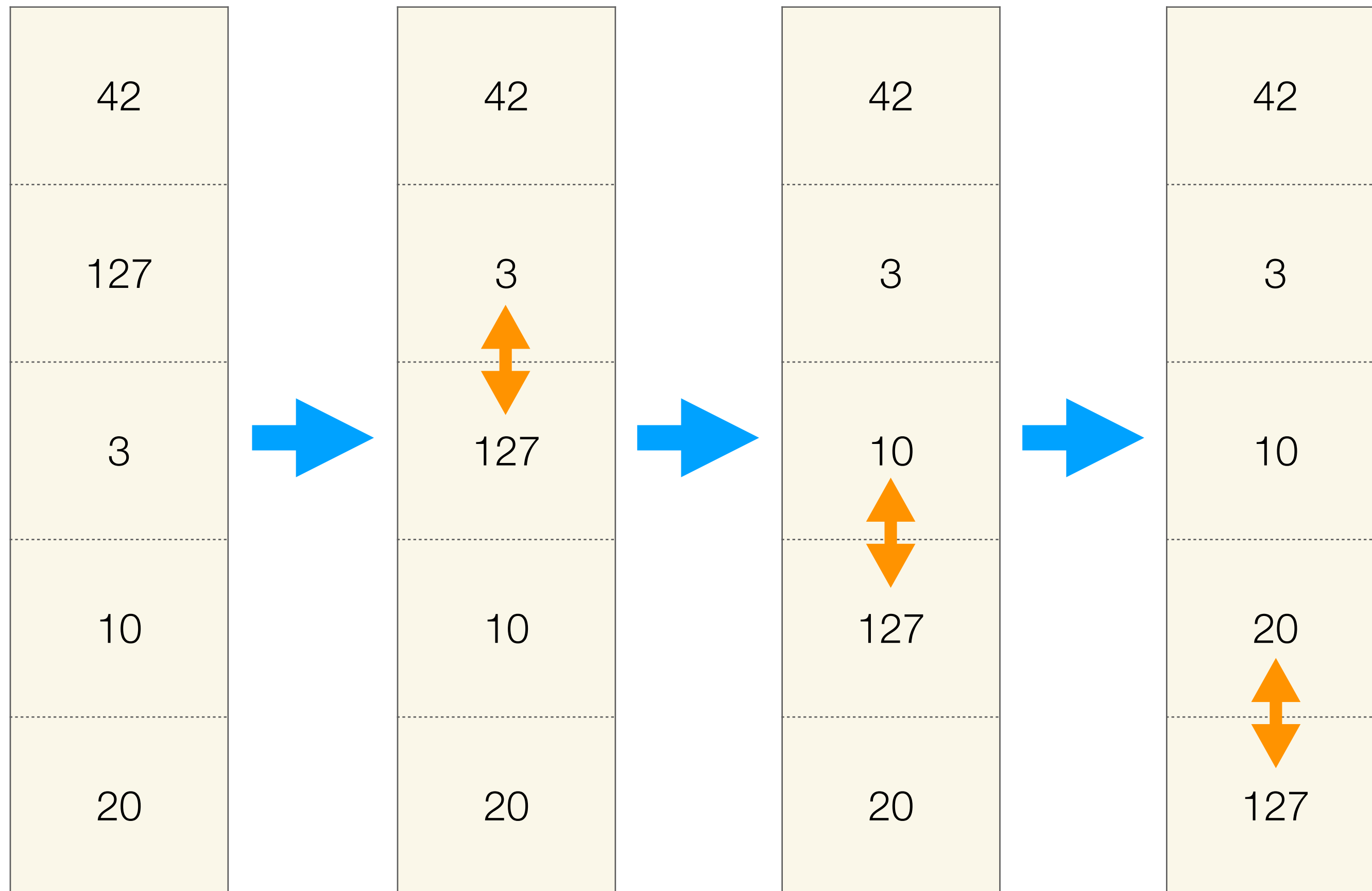
    mySort(zeichen.begin(), zeichen.end());
    for_each(zeichen.begin(), zeichen.end(), print<char>);
    cout << endl;
}
```

# Algorithmus (aufsteigende Reihenfolge) 1/3

42
127
3
10
20

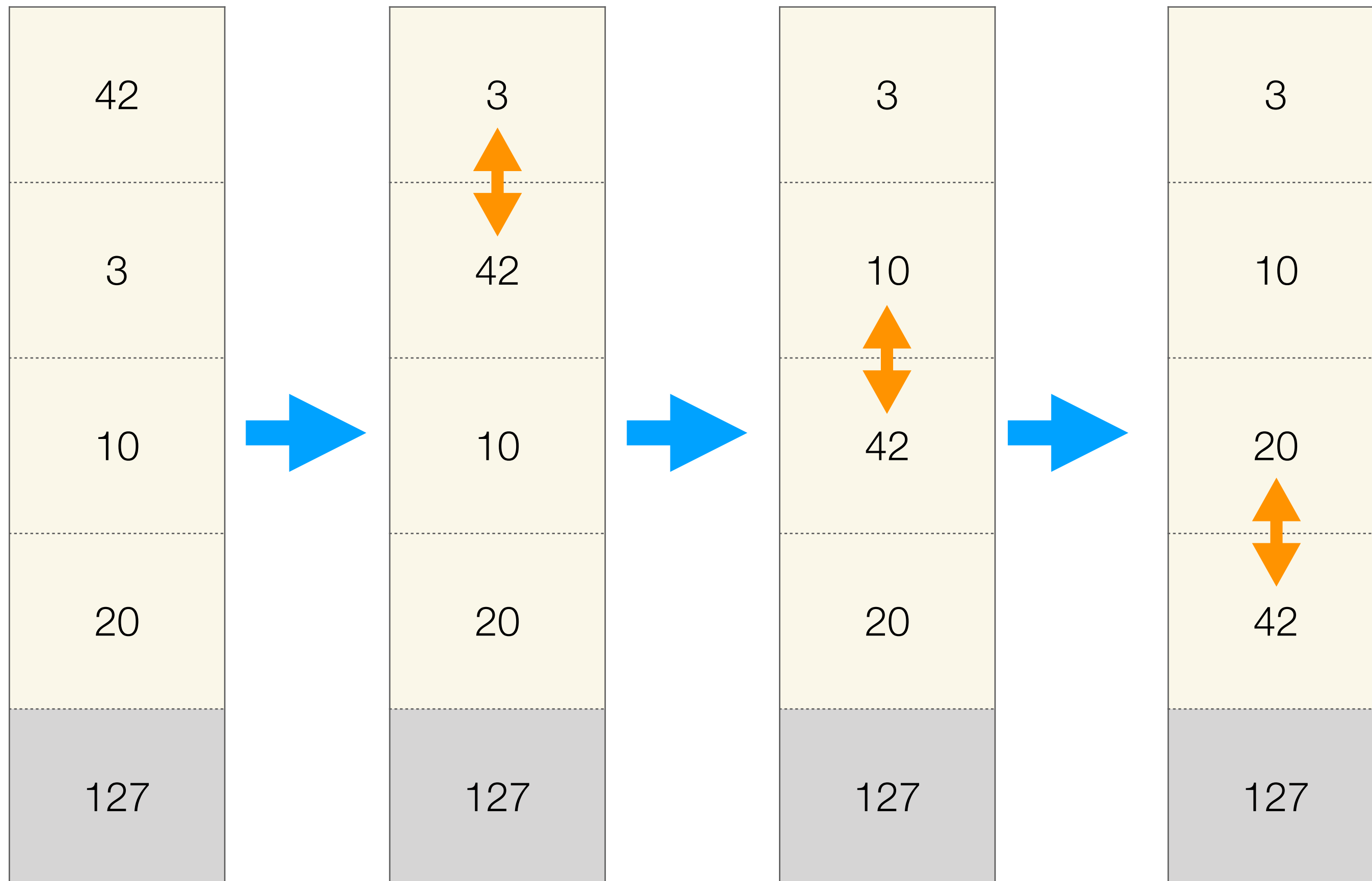


# Algorithmus (aufsteigende Reihenfolge) 1/3

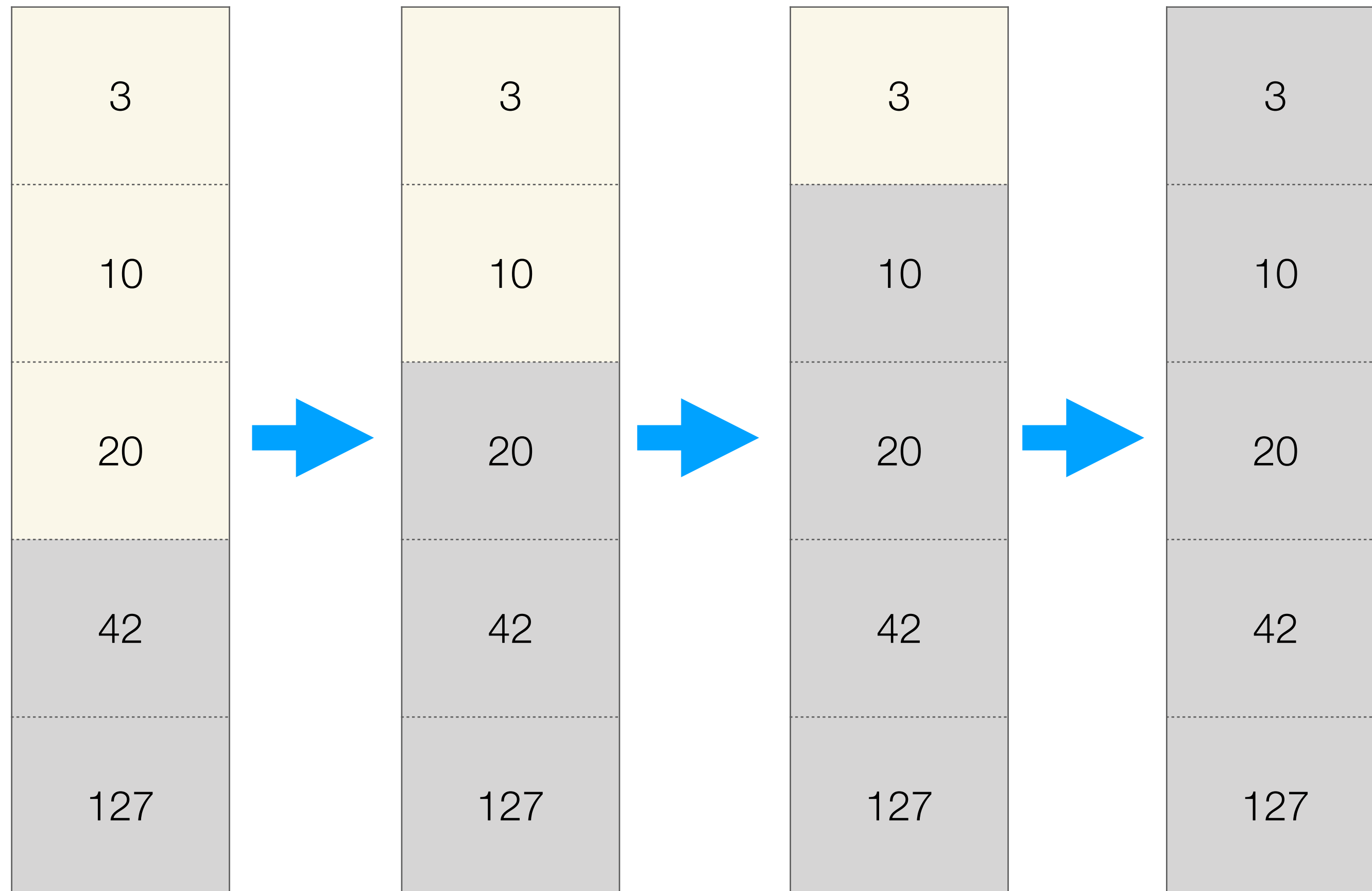




# Algorithmus (aufsteigende Reihenfolge) 2/3



# Algorithmus (aufsteigende Reihenfolge) 3/2



# sort Implementieren

```
template<class Iterator>
void mySort(Iterator first, Iterator last) {

    if(first+1==last) return; // nur ein Element

    Iterator it=first;

    while(it != last-1) {
        Iterator next=it+1;
        if(*it > *next) swap(*it, *next);
        it++;
    }
    mySort(first, last-1);
}
```

```
template<class T>
void print(const T& val) {
    cout << val << " ";
}

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

    mySort(zahlen.begin(), zahlen.end());
    for_each(zahlen.begin(), zahlen.end(), print<int>);
    cout << endl;

    mySort(zeichen.begin(), zeichen.end());
    for_each(zeichen.begin(), zeichen.end(), print<char>);
    cout << endl;
}
```

# sort Implementieren

```
template<class Iterator>
void mySort(Iterator first, Iterator last) {

    if(first+1==last) return; // nur ein Element

    Iterator it=first;

    while(it != last-1) {
        Iterator next=it+1;
        if(*it > *next) swap(*it, *next);
        it++;
    }
    mySort(first, last-1);
}
```

Probleme?

```
template<class T>
void print(const T& val) {
    cout << val << " ";
}

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

    mySort(zahlen.begin(), zahlen.end());
    for_each(zahlen.begin(), zahlen.end(), print<int>);
    cout << endl;

    mySort(zeichen.begin(), zeichen.end());
    for_each(zeichen.begin(), zeichen.end(), print<char>);
    cout << endl;
}
```

# Verbessertes sort

```
template<class Iterator, class Compare>
void mySort(Iterator first, Iterator last, Compare comp) {

    Iterator prev = first, next = first;

    // prev != last sicherstellen, bevor next inkrementiert wird
    if (prev == last || ++next == last)
        return;

    /* Ein Containerdurchlauf. Am Ende ist next == last und
    * prev == last-1 */
    while(next != last) {
        if (comp(*next, *prev))
            swap(*prev, *next);

        prev = next++;
    }
    mySort(first, prev, comp); // Rekursion mit [first, last-1)
}
```

Erlaubt nun auch ForwardIterator  
und leere Container

# Vordefinierte Sortierreihenfolge

```
#include <functional>

template<class Iterator>
void mySort(Iterator first, Iterator last) {
    /* Aufruf mit Vergleichsfunktion < für den Elementtyp des Containers
     * auf den Iterator verweist */
    mySort(first, last, std::less<typename Iterator::value_type>());
}
```

```
int main() {
    vector<int> zahlen{42, 127, 3, 10, 20};
    forward_list<char> zeichen{'a', 'b', 'a', 'd', 'c'};

    // aufsteigende Sortierung
    mySort(zahlen.begin(), zahlen.end());
    for_each(zahlen.begin(), zahlen.end(), print<int>);
    cout << endl;

    // absteigende Sortierung einer einfach verketteten Liste
    mySort(zeichen.begin(), zeichen.end(), std::greater());
    for_each(zeichen.begin(), zeichen.end(), print<char>);
    cout << endl;
}
```

Instanzieren von less mit dem Element-Datentyp des Containers (typename, weil Iterator::value\_type ein Template-Datentyp ist: mySort ist abhängig von Iterator.)

Alternative Sortierreihenfolge (absteigend)



# sort mit selbstdefinierten Datentypen

```
/operations.h:384:21: error: invalid operands to binary expression ('const Quadrat'
and 'const Quadrat')
    {return __x < __y;}
               ^
quadrat.cpp:39:9: note: in instantiation of member function 'std::less<Quadrat>::op
erator()' requested here
    if (comp(*next, *prev))
        ^
```

```
struct Quadrat {
    int px; int py; int len;
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};
};

int main() {
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};

    mySort(q.begin(), q.end());
}
```

# sort mit selbstdefinierten Datentypen


```
bool operator<(const Quadrat& q1,  
               const Quadrat& q2) {  
    return q1.len < q2.len;  
}
```

```
struct Quadrat {  
    int px; int py; int len;  
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};  
};  
  
int main() {  
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};  
  
    mySort(q.begin(), q.end());  
}
```

# sort mit selbstdefinierten Datentypen

```
#include <functional>

template<class Iterator>
void mySort(Iterator first, Iterator last) {
    /* Aufruf mit Vergleichsfunktion < für den Elementtyp des Containers
     * auf den Iterator verweist */
    mySort(first, last, std::less<typename Iterator::value_type>());
}
```



```
bool operator<(const Quadrat& q1,
               const Quadrat& q2) {
    return q1.len < q2.len;
}
```

```
struct Quadrat {
    int px; int py; int len;
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};
};

int main() {
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};

    mySort(q.begin(), q.end());
}
```

# std::sort

```
template <class RandomAccessIterator, class Compare>  
void sort (RandomAccessIterator first,  
           RandomAccessIterator last,  
           Compare comp);
```

- Funktioniert so ähnlich wie unsere eigene Version

# std::sort

```
struct Quadrat {
    int px; int py; int len;
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};
};

bool kleiner(const Quadrat& q1, const Quadrat& q2) {
    return q1.len < q2.len;
}

template<class T>
void print(const T& val) {
    cout << val << " ";
}

template<>
void print<Quadrat>(const Quadrat& q) {
    cout << "Quadrat mit Kantenlaenge " << q.len << ", ";
}

int main() {
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};

    std::sort(q.begin(), q.end(), kleiner);
    for_each(q.begin(), q.end(), print<Quadrat>);
    cout << endl;
}
```

# print für spezielle Datentypen

```
template <class T>
void print(const T &val) {
    cout << val << endl;
}
```

**generische Templatefunktion  
für alle Typen T**

```
template <>
void print<Quadrat>(const Quadrat &q) {
    cout << "Quadrat mit Kantenlaenge " << q.len << endl;
}
```

**Spezialisierung für Quadrat**

```
struct Quadrat {
    int px; int py; int len;
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};
};

int main() {
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};

    mySort(q.begin(), q.end());
    print("Quadrat"); print(-123);
    for_each(q.begin(), q.end(), print<Quadrat>);
    print(Quadrat{9,12,3});
}
```



# `std::for_each`

- Häufige Aufgabe bei Container-Objekten: Alle oder ausgewählte enthaltenen Objekte angucken/bearbeiten/selektieren
- Unterschiedliche Muster anwendbar

# C-Style for-Schleife

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

using namespace std;

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    for(int i=0; i<zahlen.size(); i++) {
        cout << zahlen[i] << " ";
    }
}
```

# for-Schleife mit C++-Iteratoren

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

using namespace std;

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    for(auto i=zahlen.begin(); i!=zahlen.end(); i++) {
        cout << *i << " ";
    }
}
```

# std::for\_each

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

using namespace std;

void print(int val) {
    cout << val << " ";
}

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    for_each(zahlen.begin(), zahlen.end(), print);
}
```

# std::for\_each

```
template< class InputIt, class UnaryFunction > (1)  
constexpr UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f ); (since C++20)
```

- Wendet die übergebene Funktion auf alle Elemente in dem angegebenen Bereich an
- Iteratoren werden dereferenziert und die entsprechenden Objekte dann der Funktion übergeben
- Funktion kann daher nur ein Argument bekommen
- Funktion gibt den Funktor (f) zurück → z. B. für Zustand verwenden
- **Vorteile**
  - **Übersichtlich (eine Zeile für Schleife)**
  - **Ggf. Einfache Wiederverwendung von Funktionen**

# std::copy

```
template<...>
... my_copy(...) {
}
```

**stellt sicher, dass Speicher in result  
angelegt wird (result.reserve(...))**



```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};
    vector<int> result;

    my_copy(zahlen.begin(), zahlen.end(), back_inserter(result));
    for_each(result.begin(), result.end(), print);
}
```

# std::copy

```
template<class Input, class Output>
Output my_copy(Input first1, Input last1, Output output) {
    while(first1 != last1) {
        *output++=*first1++;
    }
    return output;
}
```

**stellt sicher, dass Speicher in result  
angelegt wird (result.reserve(...))**



```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};
    vector<int> result;

    my_copy(zahlen.begin(), zahlen.end(), back_inserter(result));
    for_each(result.begin(), result.end(), print);
}
```



# std::copy\_if

Kopiert nur, wenn das übergebene Prädikat zutrifft

```
bool istGerade(int i) {  
    return (i % 2) == 0;  
}  
  
int main() {  
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};  
    vector<int> result;  
  
    copy_if(zahlen.begin(), zahlen.end(), back_inserter(result), istGerade);  
    for_each(result.begin(), result.end(), print);  
}
```

# std::transform

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

using namespace std;

void print(int val) {
    cout << val << " ";
}

int mal2(int val) {return val*2;}

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    transform(zahlen.begin(), zahlen.end(), zahlen.begin(), mal2);
    for_each(zahlen.begin(), zahlen.end(), print);
}
```

# std::transform

```
template<...>  
... my_transform(...) {  
}
```

```
int main() {  
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};  
  
    my_transform(zahlen.begin(), zahlen.end(), zahlen.begin(), mal2);  
    for_each(zahlen.begin(), zahlen.end(), print);  
}
```

# std::transform

```
template<class Input, class Output, class Func>
Output my_transform(Input first1, Input last1, Output output, Func f) {
    while(first1 != last1) {
        *output++ = f(*first1++);
    }
    return output;
}
```

```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    my_transform(zahlen.begin(), zahlen.end(), zahlen.begin(), mal2);
    for_each(zahlen.begin(), zahlen.end(), print);
}
```

# std::bind: Argumente binden

- Erzeugt Funktor
  - Argumente können festgelegt werden
  - Platzhalter `_1`, `_2`, `_3` ... für nicht gebundene Argumente

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

```
using namespace std;
using namespace std::placeholders;
```

```
int mult(int a, int b) {
    return a * b;
}
```

```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};
```

```
    transform(zahlen.begin(), zahlen.end(), zahlen.begin(), bind(mult, 5, _1);
    for_each(zahlen.begin(), zahlen.end(), print);
}
```

**Binäre Funktion,**  
`bind` erzeugt unären Funktor,  
Erstes Argument festgelegt (= 5)

**Platzhalter für Werte von `zahlen`**

# std::accumulate

accumulate()  
reduce()

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

using namespace std;

int mult(int a, int b) {
    return a * b;
}

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1};

    int result = accumulate(zahlen.begin(), zahlen.end(), 1, mult);
    cout << result << endl;
}
```

**Binäre Funktion**

**Initialwert für Parameter init**

# std::accumulate

```
template<...>
... my_accumulate(...) {
}
```

```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    int result = my_accumulate(zahlen.begin(), zahlen.end(), 1, mult);
    cout << result << endl;
}
```



# std::accumulate

```
template<class Input, class Type, class Func>
Type my_accumulate(Input first, Input last, Type init, Func f) {
    while(first != last) {
        init = f(init, *first);
        first++;
    }
    return init;
}
```

```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    int result = my_accumulate(zahlen.begin(), zahlen.end(), 1, mult);
    cout << result << endl;
}
```