

4. C++ vertieft (I)

Kurzwiederholung: Vektoren, Zeiger und Iteratoren
Bereichsbasiertes for, Schlüsselwort auto, eine Klasse für Vektoren,
Indexoperator, Move-Konstruktion, Iterator.

Lernziele

- Schlüsselwort **auto**
- Bereichsbasiertes **for**
- Kurzwiederholung der Dreierregel
- Indexoperator
- Move Semantik, X-Werte und Fünferregel
- Eigene Iteratoren

Wir erinnern uns...

```
#include <iostream>
#include <vector>
using iterator = std::vector<int>::iterator;

int main(){
    // Vector of length 10
    std::vector<int> v(10);
    // Input
    for (int i = 0; i < v.size(); ++i)
        std::cin >> v[i];
    // Output
    for (iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << " ";
}
```

Wir erinnern uns...

```
#include <iostream>
#include <vector>
using iterator = std::vector<int>::iterator;

int main(){
    // Vector of length 10
    std::vector<int> v(10); ← Das wollen wir genau verstehen!
    // Input
    for (int i = 0; i < v.size(); ++i)
        std::cin >> v[i];
    // Output
    for (iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << " ";
}
```

Wir erinnern uns...

```
#include <iostream>
#include <vector>
using iterator = std::vector<int>::iterator;
```

```
int main(){
```

```
    // Vector of length 10
```

```
    std::vector<int> v(10);
```

← Das wollen wir genau verstehen!

```
    // Input
```

```
    for (int i = 0; i < v.size(); ++i)
```

```
        std::cin >> v[i];
```

```
    // Output
```

```
    for (iterator it = v.begin(); it != v.end(); ++it)
```

```
        std::cout << *it << " ";
```



Das geht besser!

```
}
```

4.1 Nützliche Tools

Auf dem Weg zu elegantem, weniger komplizierten Code

auto

Das Schlüsselwort **auto** (ab C++11):

Der Typ einer Variablen wird inferiert vom Initialisierer.

auto

Das Schlüsselwort **auto** (ab C++11):

Der Typ einer Variablen wird inferiert vom Initialisierer.

```
int x = 10;
```


auto

Das Schlüsselwort **auto** (ab C++11):

Der Typ einer Variablen wird inferiert vom Initialisierer.

```
int x = 10;  
auto y = x; // int
```

auto

Das Schlüsselwort **auto** (ab C++11):

Der Typ einer Variablen wird inferiert vom Initialisierer.

```
int x = 10;  
auto y = x; // int  
auto z = 3; // int
```

auto

Das Schlüsselwort **auto** (ab C++11):

Der Typ einer Variablen wird inferiert vom Initialisierer.

```
int x = 10;  
auto y = x; // int  
auto z = 3; // int  
std::vector<double> v(5);
```

auto

Das Schlüsselwort **auto** (ab C++11):

Der Typ einer Variablen wird inferiert vom Initialisierer.

```
int x = 10;  
auto y = x; // int  
auto z = 3; // int  
std::vector<double> v(5);  
auto i = v[3]; // double
```

Schon etwas besser...

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

int main(){
    std::vector<int> v(10); // Vector of length 10

    for (int i = 0; i < v.size(); ++i)
        std::cin >> v[i];

    for (auto it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }
}
```

Bereichsbasiertes `for` (C++11)

```
for (range-declaration : range-expression)  
    statement;
```

- **range-declaration:** benannte Variable vom Elementtyp der durch range-expression spezifizierten Folge.
- **range-expression:** Ausdruck, der eine Folge von Elementen repräsentiert via Iterator-Paar `begin()`, `end()`, oder in Form einer Initialisierungsliste.

Bereichsbasiertes `for` (C++11)

```
for (range-declaration : range-expression)  
    statement;
```

- **range-declaration:** benannte Variable vom Elementtyp der durch range-expression spezifizierten Folge.
- **range-expression:** Ausdruck, der eine Folge von Elementen repräsentiert via Iterator-Paar `begin()`, `end()`, oder in Form einer Initialisierungsliste.

Bereichsbasiertes `for` (C++11)

```
for (range-declaration : range-expression)  
    statement;
```

- **range-declaration:** benannte Variable vom Elementtyp der durch range-expression spezifizierten Folge.
- **range-expression:** Ausdruck, der eine Folge von Elementen repräsentiert via Iterator-Paar `begin()`, `end()`, oder in Form einer Initialisierungsliste.

```
std::vector<double> v(5);
```


Bereichsbasiertes for (C++11)

```
for (range-declaration : range-expression)  
    statement;
```

- **range-declaration:** benannte Variable vom Elementtyp der durch range-expression spezifizierten Folge.
- **range-expression:** Ausdruck, der eine Folge von Elementen repräsentiert via Iterator-Paar `begin()`, `end()`, oder in Form einer Initialisierungsliste.

```
std::vector<double> v(5);  
for (double x: v) std::cout << x; // 00000
```

Bereichsbasiertes for (C++11)

```
for (range-declaration : range-expression)  
    statement;
```

- **range-declaration:** benannte Variable vom Elementtyp der durch range-expression spezifizierten Folge.
- **range-expression:** Ausdruck, der eine Folge von Elementen repräsentiert via Iterator-Paar `begin()`, `end()`, oder in Form einer Initialisierungsliste.

```
std::vector<double> v(5);  
for (double x: v) std::cout << x; // 00000  
for (int x: {1,2,5}) std::cout << x; // 125
```

Bereichsbasiertes for (C++11)

```
for (range-declaration : range-expression)
    statement;
```

- **range-declaration:** benannte Variable vom Elementtyp der durch range-expression spezifizierten Folge.
- **range-expression:** Ausdruck, der eine Folge von Elementen repräsentiert via Iterator-Paar `begin()`, `end()`, oder in Form einer Initialisierungsliste.

```
std::vector<double> v(5);
for (double x: v) std::cout << x; // 00000
for (int x: {1,2,5}) std::cout << x; // 125
for (double& x: v) x=5;
```

Cool!

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

int main(){
    std::vector<int> v(10); // Vector of length 10

    for (auto& x: v)
        std::cin >> x;

    for (const auto x: v)
        std::cout << x << " ";
}
```

4.2 Speicherallokation

Bau einer Vektorklasse

Für unser genaues Verständnis

Wir bauen selbst eine Vektorklasse, die so etwas kann!

Für unser genaues Verständnis

Wir bauen selbst eine Vektorklasse, die so etwas kann!

Auf dem Weg lernen wir etwas über

- **RAII (Resource Acquisition is Initialization) und Move-Konstruktion**

Für unser genaues Verständnis

Wir bauen selbst eine Vektorklasse, die so etwas kann!

Auf dem Weg lernen wir etwas über

- **RAII (Resource Acquisition is Initialization) und Move-Konstruktion**
- **Index-Operatoren und andere Nützlichkeiten**

Für unser genaues Verständnis

Wir bauen selbst eine Vektorklasse, die so etwas kann!

Auf dem Weg lernen wir etwas über

- **RAII (Resource Acquisition is Initialization) und Move-Konstruktion**
- **Index-Operatoren und andere Nützlichkeiten**
- **Templates**

Für unser genaues Verständnis

Wir bauen selbst eine Vektorklasse, die so etwas kann!

Auf dem Weg lernen wir etwas über

- **RAII (Resource Acquisition is Initialization) und Move-Konstruktion**
- **Index-Operatoren und andere Nützlichkeiten**
- Templates
- Exception Handling

Für unser genaues Verständnis

Wir bauen selbst eine Vektorklasse, die so etwas kann!

Auf dem Weg lernen wir etwas über

- **RAII (Resource Acquisition is Initialization) und Move-Konstruktion**
- **Index-Operatoren und andere Nützlichkeiten**
- Templates
- Exception Handling
- Funktoren und Lambda-Ausdrücke

Für unser genaues Verständnis

Wir bauen selbst eine Vektorklasse, die so etwas kann!

Auf dem Weg lernen wir etwas über

- **RAII (Resource Acquisition is Initialization) und Move-Konstruktion**
- **Index-Operatoren und andere Nützlichkeiten**
- Templates
- Exception Handling
- Funktoren und Lambda-Ausdrücke

heute

Eine Klasse für (double) Vektoren

```
class Vector{
public:
    // constructors
    Vector(): sz{0}, elem{nullptr} {};
    Vector(std::size_t s): sz{s}, elem{new double[s]} {}
    // destructor
    ~Vector(){
        delete[] elem;
    }
    // (something is missing here)
private:
    std::size_t sz;
    double* elem;
}
```

Elementzugriffe

```
class Vector{
    ...
    // getter. pre: 0 <= i < sz;
    double get(std::size_t i) const{
        return elem[i];
    }
    // setter. pre: 0 <= i < sz;
    void set(std::size_t i, double d){
        elem[i] = d;
    }
    // size property
    std::size_t size() const {
        return sz;
    }
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    double get(std::size_t i) const;
    void set(std::size_t i, double d);
    std::size_t size() const;
}
```

(Vector Schnittstelle)

Was läuft schief?

```
int main(){
    Vector v(32);
    for (std::size_t i = 0; i!=v.size(); ++i)
        v.set(i, i);
    Vector w = v;
    for (std::size_t i = 0; i!=w.size(); ++i)
        w.set(i, i*i);
    return 0;
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    double get(std::size_t i) const;
    void set(std::size_t i, double d);
    std::size_t size() const;
}
```

(Vector Schnittstelle)

Was läuft schief?

```
int main(){
    Vector v(32);
    for (std::size_t i = 0; i!=v.size(); ++i)
        v.set(i, i);
    Vector w = v;
    for (std::size_t i = 0; i!=w.size(); ++i)
        w.set(i, i*i);
    return 0;
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    double get(std::size_t i) const;
    void set(std::size_t i, double d);
    std::size_t size() const;
}
```

(Vector Schnittstelle)

```
*** Error in 'vector1': double free or corruption
(!prev): 0x0000000000d23c20 ***
===== Backtrace: =====
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5) [0x7fe5a5ac97e5]
...
```


Rule of Three!

```
class Vector{  
...  
public:  
    // copy constructor  
    Vector(const Vector &v)  
        : sz{v.sz}, elem{new double[v.sz]} {  
        std::copy(v.elem, v.elem + v.sz, elem);  
    }  
}
```

```
class Vector{  
public:  
    Vector();  
    Vector(std::size_t s);  
    ~Vector();  
    Vector(const Vector &v);  
    double get(std::size_t i) const;  
    void set(std::size_t i, double d);  
    std::size_t size() const;  
}
```

(Vector Schnittstelle)

Rule of Three!

```
class Vector{
...
    // assignment operator
    Vector& operator=(const Vector& v){
        if (v.elem == elem) return *this;
        if (elem != nullptr) delete[] elem;
        sz = v.sz;
        elem = new double[sz];
        std::copy(v.elem, v.elem+v.sz, elem);
        return *this;
    }
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    Vector(const Vector &v);
    Vector operator=(const Vector&v);
    double get(std::size_t i) const;
    void set(std::size_t i, double d);
    std::size_t size() const;
}
```

(Vector Schnittstelle)

Rule of Three!

```
class Vector{
...
    // assignment operator
    Vector& operator=(const Vector& v){
        if (v.elem == elem) return *this;
        if (elem != nullptr) delete[] elem;
        sz = v.sz;
        elem = new double[sz];
        std::copy(v.elem, v.elem+v.sz, elem);
        return *this;
    }
}
```

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    Vector(const Vector &v);
    Vector operator=(const Vector&v);
    double get(std::size_t i) const;
    void set(std::size_t i, double d);
    std::size_t size() const;
}
```

(Vector Schnittstelle)

Jetzt ist es zumindest korrekt. Aber umständlich.

Weiterleitung des Konstruktors

```
public:  
// copy constructor  
// (with constructor delegation)  
Vector(const Vector &v): Vector(v.sz)  
{  
    std::copy(v.elem, v.elem + v.sz, elem);  
}
```

Copy-&-Swap Idiom

```
class Vector{
...
    // Assignment operator
    Vector& operator= (const Vector&v){
        Vector cpy(v);
        swap(cpy);
        return *this;
    }
private:
    // helper function
    void swap(Vector& v){
        std::swap(sz, v.sz);
        std::swap(elem, v.elem);
    }
}
```

Copy-&Swap Idiom

```
class Vector{
...
    // Assignment operator
    Vector& operator= (const Vector&v){
        Vector cpy(v);
        swap(cpy);
        return *this;
    }
private:
    // helper function
    void swap(Vector& v){
        std::swap(sz, v.sz);
        std::swap(elem, v.elem);
    }
}
```

copy-and-swap idiom: alle Felder von `*this` tauschen mit den Daten von `cpy`. Beim Verlassen von `operator=` wird `cpy` aufgeräumt (dekonstruiert), während die Kopie der Daten von `v` in `*this` verbleiben.

Begriffsklärung: Idioms und Patterns

Idiom und **(Design) Pattern** sind Begriffe aus dem **Software-Engineering**

- Design Patterns (Entwurfsmuster) sind *allgemeine, in der Regel nicht sprachspezifische*, wiederverwendbare Lösungen für häufig auftretende Entwurfsprobleme. Sie erfassen bewährte Verfahren und beschreiben in der Regel Beziehungen und Interaktionen zwischen Klassen und Objekten. Beispiel: visitor-pattern.
- Idioms sind hingegen *sprachspezifische* Methoden zur Implementierung bestimmter häufig auftretender Aufgaben oder Schritte, z.B. das RAI-Idiom oder das Copy-&Swap-Idiom in C++. Idioms umfassen üblicherweise auch weniger Code als Patterns.

Arbeit an der Fassade.

Getter und Setter unschön. Wir wollen einen Indexoperator.

Arbeit an der Fassade.

Getter und Setter unschön. Wir wollen einen Indexoperator.

Überladen!

Arbeit an der Fassade.

Getter und Setter unschön. Wir wollen einen Indexoperator.

Überladen! So?

```
class Vector{
...
    double operator[] (std::size_t pos) const{
        return elem[pos];
    }

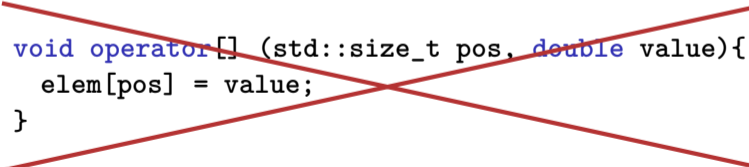
    void operator[] (std::size_t pos, double value){
        elem[pos] = value;
    }
}
```

Arbeit an der Fassade.

Getter und Setter unschön. Wir wollen einen Indexoperator.

Überladen! So?

```
class Vector{  
...  
    double operator[] (std::size_t pos) const{  
        return elem[pos];  
    }  
  
    void operator[] (std::size_t pos, double value){  
        elem[pos] = value;  
    }  
}
```



Nein!

Referenztypen!

```
class Vector{
...
    // for non-const objects
    double& operator[] (std::size_t pos){
        return elem[pos]; // return by reference!
    }
    // for const objects
    const double& operator[] (std::size_t pos) const{
        return elem[pos];
    }
}
```

Soweit, so gut.

```
int main(){
    Vector v(32); // constructor
    for (int i = 0; i<v.size(); ++i)
        v[i] = i; // subscript operator

    Vector w = v; // copy constructor
    for (int i = 0; i<w.size(); ++i)
        w[i] = i*i;

    const auto u = w;
    for (int i = 0; i<u.size(); ++i)
        std::cout << v[i] << ":" << u[i] << " "; // 0:0 1:1 2:4 ...
    return 0;
}
```

4.3 Iteratoren

Wie man bereichsbasiertes **for** unterstützt.

Bereichsbasiertes for

Wir wollten doch das:

```
Vector v = ...;  
for (auto x: v)  
    std::cout << x << " ";
```

Bereichsbasiertes for

Wir wollten doch das:

```
Vector v = ...;  
for (auto x: v)  
    std::cout << x << " ";
```

Dafür müssen wir einen Iterator über **begin** und **end** bereitstellen.

Iterator für den Vektor

```
class Vector{  
...  
    // Iterator  
    double* begin(){  
        return elem;  
    }  
    double* end(){  
        return elem+sz;  
    }  
}
```

(Zeiger unterstützen Iteration)

Const Iterator für den Vektor

```
class Vector{  
...  
    // Const-Iterator  
    const double* begin() const{  
        return elem;  
    }  
    const double* end() const{  
        return elem+sz;  
    }  
}
```

Zwischenstand

```
Vector Natural(int from, int to){  
    Vector v(to-from+1);  
    for (auto& x: v) x = from++;  
    return v;  
}
```

```
int main(){  
    auto v = Natural(5,12);  
    for (auto x: v)  
        std::cout << x << " "; // 5 6 7 8 9 10 11 12  
    std::cout << std::endl;  
        << "sum = "  
        << std::accumulate(v.begin(), v.end(),0); // sum = 68  
    return 0;  
}
```

Vector Schnittstelle

```
class Vector{
public:
    Vector(); // Default Constructor
    Vector(std::size_t s); // Constructor
    ~Vector(); // Destructor
    Vector(const Vector &v); // Copy Constructor
    Vector& operator=(const Vector&v); // Assignment Operator
    double& operator[] (std::size_t pos); // Subscript operator (read/write)
    const double& operator[] (std::size_t pos) const; // Subscript operator
    std::size_t size() const;
    double* begin(); // iterator begin
    double* end(); // iterator end
    const double* begin() const; // const iterator begin
    const double* end() const; // const iterator end
}
```

4.4 Effizientes Speicher-Management*

Wie man Kopien vermeidet

Anzahl Kopien

Wie oft wird `v` kopiert?

```
Vector operator+ (const Vector& l, double r){
    Vector result (l);
    for (std::size_t i = 0; i < l.size(); ++i)
        result[i] = l[i] + r;
    return result;
}
int main(){
    Vector v(16);
    v = v + 1;
    return 0;
}
```

Anzahl Kopien

Wie oft wird `v` kopiert?

```
Vector operator+ (const Vector& l, double r){
    Vector result (l); // Kopie von l nach result
    for (std::size_t i = 0; i < l.size(); ++i)
        result[i] = l[i] + r;
    return result; // Dekonstruktion von result nach Zuweisung
}

int main(){
    Vector v(16); // Allokation von elems[16]
    v = v + 1; // Kopie bei Zuweisung!
    return 0; // Dekonstruktion von v
}
```

Anzahl Kopien

Wie oft wird `v` kopiert?

```
Vector operator+ (const Vector& l, double r){  
    Vector result (l);  
    for (std::size_t i = 0; i < l.size(); ++i)  
        result[i] = l[i] + r;  
    return result;  
}  
  
int main(){  
    Vector v(16);  
    v = v + 1;  
    return 0;  
}
```

`v` wird (mindestens) zwei Mal kopiert.

Move-Konstruktor und Move-Zuweisung

```
class Vector{
...
    // move constructor
    Vector (Vector&& v): Vector() {
        swap(v);
    };
    // move assignment
    Vector& operator=(Vector&& v){
        swap(v);
        return *this;
    };
}
```

Vector Schnittstelle

```
class Vector{
public:
    Vector();
    Vector(std::size_t s);
    ~Vector();
    Vector(const Vector &v);
    Vector& operator=(const Vector&v);
    Vector (Vector&& v);
    Vector& operator=(Vector&& v);
    const double& operator[] (std::size_t pos) const;
    double& operator[] (std::size_t pos);
    std::size_t size() const;
}
```

Wenn das Quellobjekt einer Zuweisung direkt nach der Zuweisung nicht weiter existiert, dann kann der Compiler den Move-Zuweisungsoperator anstelle des Zuweisungsoperators einsetzen.⁵ Damit wird eine potentiell teure Kopie vermieden.

Anzahl der Kopien im vorigen Beispiel reduziert sich zu 1.

⁵Analoges gilt für den Kopier-Konstruktor und den Move-Konstruktor.

Illustration zur Move-Semantik

```
// nonsense implementation of a "vector" for demonstration purposes
class Vec{
public:
    Vec () {
        std::cout << "default constructor\n";}
    Vec (const Vec&) {
        std::cout << "copy constructor\n";}
    Vec& operator = (const Vec&) {
        std::cout << "copy assignment\n"; return *this;}
    ~Vec() {}
};
```

Wie viele Kopien?

```
Vec operator + (const Vec& a, const Vec& b){  
    Vec tmp = a;  
    // add b to tmp  
    return tmp;  
}
```

```
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```

Wie viele Kopien?

```
Vec operator + (const Vec& a, const Vec& b){  
    Vec tmp = a;  
    // add b to tmp  
    return tmp;  
}  
  
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```

Ausgabe
default constructor
copy constructor
copy constructor
copy constructor
copy assignment

4 Kopien des Vektors

Illustration der Move-Semantik

```
// nonsense implementation of a "vector" for demonstration purposes
class Vec{
public:
    Vec () { std::cout << "default constructor\n";}
    Vec (const Vec&) { std::cout << "copy constructor\n";}
    Vec& operator = (const Vec&) {
        std::cout << "copy assignment\n"; return *this;}
    ~Vec() {}
    // new: move constructor and assignment
    Vec (Vec&&) {
        std::cout << "move constructor\n";}
    Vec& operator = (Vec&&) {
        std::cout << "move assignment\n"; return *this;}
};
```

Wie viele Kopien?

```
Vec operator + (const Vec& a, const Vec& b){  
    Vec tmp = a;  
    // add b to tmp  
    return tmp;  
}
```

```
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```


Wie viele Kopien?

```
Vec operator + (const Vec& a, const Vec& b){  
    Vec tmp = a;  
    // add b to tmp  
    return tmp;  
}  
  
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```

Ausgabe
default constructor
copy constructor
copy constructor
copy constructor
move assignment

3 Kopien des Vektors

Wie viele Kopien?

```
Vec operator + (Vec a, const Vec& b){  
    // add b to a  
    return a;  
}
```

```
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```

Wie viele Kopien?

```
Vec operator + (Vec a, const Vec& b){  
    // add b to a  
    return a;  
}
```

```
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```

Ausgabe
default constructor
copy constructor
move constructor
move constructor
move constructor
move assignment

1 Kopie des Vektors

Wie viele Kopien?

```
Vec operator + (Vec a, const Vec& b){  
    // add b to a  
    return a;  
}
```

```
int main (){  
    Vec f;  
    f = f + f + f + f;  
}
```

Ausgabe
default constructor
copy constructor
move constructor
move constructor
move constructor
move assignment

1 Kopie des Vektors

Erklärung: Move-Semantik kommt zum Einsatz, wenn ein x-wert (expired) zugewiesen wird. R-Wert-Rückgaben von Funktionen sind x-Werte.

http://en.cppreference.com/w/cpp/language/value_category

Wie viele Kopien

```
void swap(Vec& a, Vec& b){  
    Vec tmp = a;  
    a=b;  
    b=tmp;  
}
```

```
int main (){  
    Vec f;  
    Vec g;  
    swap(f,g);  
}
```

Wie viele Kopien

```
void swap(Vec& a, Vec& b){  
    Vec tmp = a;  
    a=b;  
    b=tmp;  
}
```

```
int main (){  
    Vec f;  
    Vec g;  
    swap(f,g);  
}
```

Ausgabe
default constructor
default constructor
copy constructor
copy assignment
copy assignment

3 Kopien des Vektors

X-Werte erzwingen

```
void swap(Vec& a, Vec& b){
    Vec tmp = std::move(a);
    a=std::move(b);
    b=std::move(tmp);
}
int main (){
    Vec f;
    Vec g;
    swap(f,g);
}
```

X-Werte erzwingen

```
void swap(Vec& a, Vec& b){  
    Vec tmp = std::move(a);  
    a=std::move(b);  
    b=std::move(tmp);  
}  
  
int main (){  
    Vec f;  
    Vec g;  
    swap(f,g);  
}
```

Ausgabe
default constructor
default constructor
move constructor
move assignment
move assignment

0 Kopien des Vektors

X-Werte erzwingen

```
void swap(Vec& a, Vec& b){  
    Vec tmp = std::move(a);  
    a=std::move(b);  
    b=std::move(tmp);  
}  
  
int main (){  
    Vec f;  
    Vec g;  
    swap(f,g);  
}
```

Ausgabe
default constructor
default constructor
move constructor
move assignment
move assignment

0 Kopien des Vektors

Erklärung: Mit `std::move` kann man einen L-Wert Ausdruck zu einem X-Wert machen. Dann kommt wieder Move-Semantik zum Einsatz.

<http://en.cppreference.com/w/cpp/utility/move>

`std::swap` & `std::move`

`std::swap` ist (mit Templates) genau wie oben gesehen implementiert

`std::move` kann verwendet werden, um die Elemente eines Containers in einen anderen zu verschieben

```
std::move(va.begin(), va.end(), vb.begin())
```

Zusammenfassung

- Benutze **auto** um Typen vom Initialisierer zu inferieren.
- X-Werte sind solche, bei denen der Compiler weiss, dass Sie ihre Gültigkeit verlieren.
- Benutze Move-Konstruktion, um X-Werte zu verschieben statt zu kopieren.
- Wenn man genau weiss, was man tut, kann man X-Werte auch erzwingen.
- Indexoperatoren können überladen werden. Zum Schreiben benutzt man Referenzen.
- Hinter bereichsbasiertem **for** wirkt ein Iterator.
- Iteration wird unterstützt, indem man einen Iterator nach Konvention der Standardbibliothek implementiert.

5. C++ vertieft (II): Templates

Mit einigen (konvertierten) Folien von Malte Schwerhoff

Generische Programmierung

Ziel: Mache Code (Funktionen, Klassen) so allgemein wie möglich verwendbar

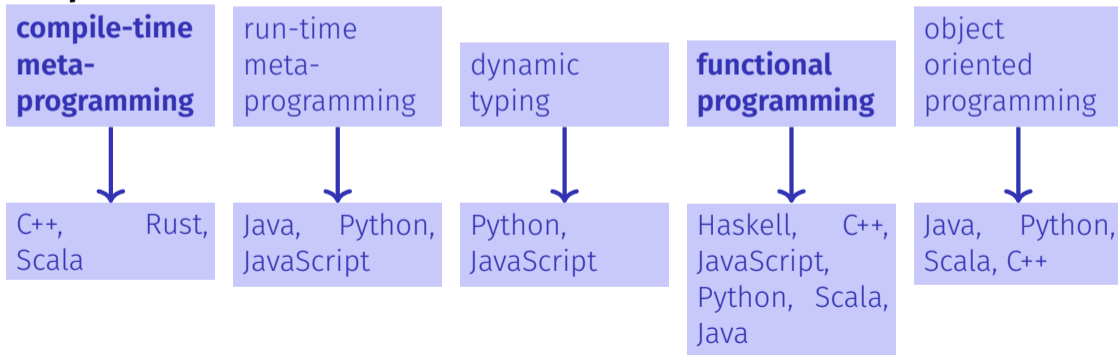
Bekannte Anwendungsfälle:

- **vector**: speichere Objekte jeden Typs, z.B. ints, strings ..
- **std::max(e1, e2)**: Gib das grössere zweier Objekte zurück
- **Exp* e = new ...; e.eval()**: evaluiere einen beliebigen Ausdruck (Variable, Konstante, Summe, Produkt, ...)

Generische Programmierung

Wie: Verschiedene Programmiersprachen verfolgen verschiedene Ansätze (und normalerweise mehr als einen), aber die Ideen sind grundsätzlich ähnlich.

Beispiele:



Motivation

Ziel: generische Vektor-Klasse und Funktionalität.

```
Vector<double> vd(10);  
Vector<int> vi(10);  
Vector<char> vi(20);  
  
auto nd = vd * vd; // norm (vector of double)  
auto ni = vi * vi; // norm (vector of int)
```

Parametrischer Polymorphismus

Typen als Template-Parameter

1. Ersetze in der konkreten Implementation einer Klasse den Typ, der generisch werden soll (beim Vektor: **double**) durch einen Stellvertreter, z.B. **T**.
2. Stelle der Klasse das Konstrukt `template<typename T>` voran (ersetze **T** ggfs. durch den Stellvertreter)..

Das Konstrukt `template<typename T>` kann gelesen werden als **“für alle Typen T”**.

Typen als Template Parameter

```
template <typename ElementType>
class Vector{
    std::size_t size;
    ElementType* elem;
public:
    ...
    Vector(std::size_t s):
        size{s},
        elem{new ElementType[s]}{}
    ...
    ElementType& operator[] (std::size_t pos){
        return elem[pos];
    }
    ...
}
```

Template Instanziierung

`Vector<typeName>` erzeugt Typinstanz von `Vector` mit `ElementType=typeName`.

```
Vector<double> x;           // vector of double
Vector<int> y;              // vector of int
Vector<Vector<double>> x;   // vector of vector of double
```

Bezeichnung: **Typinstanziierung**.

Type-checking

Templates sind weitgehend Ersetzungsregeln zur Instanzierungszeit und während der Kompilation. Es wird immer so wenig geprüft wie nötig und so viel wie möglich.

Beispiel

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){
        return left < right ? left : right;
    }
};
```

Beispiel

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){
        return left < right ? left : right;
    }
};

Pair<int> a(10,20); // ok
auto m = a.min(); // ok
Pair<Pair<int>> b(a,Pair<int>(20,30)); // ok
auto n = b.min(); // no match for operator< !
```

Parametrischer Polymorphismus II

Funktionentemplates

1. Ersetze in der konkreten Implementation einer Funktion den Typ, der generisch werden soll durch einen Namen, z.B. **T**,
2. Stelle der Funktion das Konstrukt `template<typename T>` voran (ersetze **T** ggfs. durch den gewählten Namen).

Funktionentemplates

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Typen der Aufrufparameter determinieren die Version der Funktion, welche (kompiliert und) verwendet wird:

```
int x=5;
int y=6;
swap(x,y); // calls swap with T=int
```

Sicherheiten

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Eine unverträgliche Version der Funktion wird nicht erzeugt:

```
int x=5;
double y=6;
swap(x,y); // error: no matching function for ...
```


.. auch mit Operatoren

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){ return left < right? left: right; }
    std::ostream& print (std::ostream& os) const{
        return os << "("<< left << "," << right<< ")";
    }
};

template <typename T>
std::ostream& operator<< (std::ostream& os, const Pair<T>& pair){
    return pair.print(os);
}
```

.. auch mit Operatoren

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){ return left < right? left: right; }
    std::ostream& print (std::ostream& os) const{
        return os << "("<< left << "," << right<< ")";
    }
};

template <typename T>
std::ostream& operator<< (std::ostream& os, const Pair<T>& pair){
    return pair.print(os);
}
```

```
Pair<int> a(10,20); // ok
std::cout << a; // ok
```

Praktisch!

```
// Output of an arbitrary container
template <typename T>
void output(const T& t){
    for (const auto& x: t)
        std::cout << x << " ";
    std::cout << "\n";
}

int main(){
    std::vector<int> v={1,2,3};
    output(v); // 1 2 3
}
```

Praktisch: eine generische Funktion, die man auf alle Container anwenden kann.

Welche Einschränkungen hat diese Funktion?

- Anforderungen an T?
- Anforderungen an Elemente von T?

Explizite Typangabe

```
// input of an arbitrary pair
template <typename T>
Pair<T> read(){
    T left;
    T right;
    std::cin << left << right;
    return Pair<T>(left,right);
}
...

auto p = read<double>();
```

Explizite Typangabe

```
// input of an arbitrary pair
template <typename T>
Pair<T> read(){
    T left;
    T right;
    std::cin << left << right;
    return Pair<T>(left,right);
}
...
```

```
auto p = read<double>();
```

Wenn der Typ bei der Instanzierung nicht inferiert werden kann, muss er explizit angegeben werden.

Typinferenz

- bei der Instanziierung müssen Template-Parameter explizit angegeben werden - es sei denn, der Compiler kann sie aus den angegebenen Argumenten ableiten
- Typinferenz wurde mit C++17 verbessert

```
auto p1 = Pair<int>(1,2); // OK
```

```
Pair<int> p2 = Pair(1,2); // C++14: error; C++17: OK
```

```
auto p3 = Pair(1,2); // C++14: error; C++17: OK
```

Mächtig!

```
template <typename T> // square number
T sq(T x){
    return x*x;
}
template <typename Container, typename F>
void apply(Container& c, F f){ // x <- f(x) forall x in c
    for(auto& x: c)
        x = f(x);
}
int main(){
    std::vector<int> v={1,2,3};
    apply(v,sq<int>);
    output(v); // 1 4 9
}
```

Zusammenfassung

- Templates verbessern die Wiederverwendbarkeit von Code, indem sie Klassen und Funktionen parametrisch machen in Bezug auf Typen
- Implementiert durch statische Code-Generierung (Meta-Programmierung zur Kompilierzeit)
 - Vorteile
 - ▶ Kein Laufzeit-Overhead (im Vergleich zu dynamischen Lösungen, z. B. Vererbung oder dynamische Typisierung)
 - ▶ Compiler kann generierten Code wie gewohnt optimieren
 - ▶ Typabhängige statische Spezialisierung möglich (siehe weiter unten)
 - Nachteile
 - ▶ Getrennte Kompilierung (.h vs. .cpp) nicht mehr möglich
 - ▶ Der resultierende Binärcode (Maschinencode) ist größer
 - ▶ Verzögerte Typprüfung, komplexere Fehlermeldungen (abschwächbar durch Konzepte, siehe weiter unten)

Concepts (C++20)

```
struct Student { ... }; // Lacks == and < operators
```

```
template <typename K>
struct BSTNode {
    K key;
    ...
    bool contains(K search_key) {
        if (search_key < key) ...
        else if (search_key == key) ...
        else ...
    }
};
```

```
auto n1 = BSTNode(8);
auto n2 = BSTNode("Howdy!");
auto n3 = BSTNode(new int(8));
auto n4 = BSTNode(Student("Omar"));
```

Code kompiliert ohne Probleme

- Ist das sinnvoll?
- Wenn nicht, warum kompiliert der Code?

Concepts (C++20)

```
auto n4 = BSTNode(Student("Omar"));  
n4.contains(Student("Ida")); // Error!
```

Code kompiliert nicht mehr

```
test.cpp:29:  
  In instantiation of  
  'bool BSTNode<K>::contains(K)  
  [with K = Student]':  
test.cpp:21:  
  no match for 'operator<' (operand types  
  are 'Student' and 'Student')  
    21 |     if (search_key < key)  
        |
```

Probleme:

- Die Instanziierung von **BSTNode<Student>** ist bereits unsinnig, wird aber vom Compiler akzeptiert
- Anforderung an konkrete **Ks** im Code verstreut, nicht Teil der Deklaration

Concepts (C++20)

Besser: explizite Anforderung, verhindert sinnlose Instanziierungen.

```
struct Student { ... }; // Lacks < operators
```

```
template <typename T>  
concept Comparable = requires(T t1, T t2) {  
    { t1 < t2 };  
};
```

```
template <Comparable K>  
struct BSTNode {  
    K key;  
    ...  
    bool contains(K search_key) { ... }  
};
```

```
auto n4 = BSTNode(Student("Omar")); // Error
```

```
test.cpp:43:  
    In substitution of 'template<class K>  
    BSTNode(K)-> BSTNode<K> [with K = Student]:  
test.cpp:23:  
    template constraint failure for  
    'template<class K> requires Comparable<K>  
    struct BSTNode  
test.cpp:4:  
    required for the satisfaction of Comparable<K>  
    [with K = Student]
```

Concepts (C++20)

```
#include <algorithm>
```

```
#include <vector>
```

```
#include <list>
```

```
template <typename Iter>
```

```
void clever_sort(Iter begin, Iter end) {
```

```
    if (!is_sorted(begin, end))
```

```
        std::sort(begin, end);
```

```
}
```

```
int main() {
```

```
    std::vector<int> data1 = {3,-1,10};
```

```
    clever_sort(data1.begin(), data1.end());
```

```
    std::list<int> data2 = {3,-1,10};
```

```
    clever_sort(data2.begin(), data2.end());
```

```
}
```

```
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:62,
   from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h: In instantiation of 'void std::_sort(
  _RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = std::_List_iterator<int>;
  _Compare = __gnu_cxx::__ops::_Iter_less_iter]':
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h:4842:18: required from 'void std::sort(_RAIter, _RAIter)
  [with _RAIter = std::_List_iterator<int>]'
<source>:10:14: required from 'void clever_sort(Iter, Iter) [with Iter = std::_List_iterator<int>]'
<source>:18:14: required from here
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h:1955:50: error: no match for 'operator-' (operand types
  are 'std::_List_iterator<int>' and 'std::_List_iterator<int>')
1955 |         std::_lg(__last - __first) * 2,
      |         ~~~~~^~~~~~
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algos.h:67,
   from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:61,
   from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_iterator.h:560:5: note: candidate: 'template<class _IteratorL,
  class _IteratorR> constexpr decltype ((_y.base() - __x.base())) std::operator-(const std::reverse_iterator<_Iterator
  >&, const std::reverse_iterator<_IteratorR>&)'
560 |     operator-(const reverse_iterator<_IteratorL>& __x,
      |     ~~~~~~
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_iterator.h:560:5: note: template argument deduction/
  substitution failed:
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:62,
   from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h:1955:50: note: 'std::_List_iterator<int>' is not derived
  from 'const std::reverse_iterator<_Iterator>'
1955 |         std::_lg(__last - __first) * 2,
      |         ~~~~~^~~~~~
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algos.h:67,
   from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:61,
   from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_iterator.h:1639:5: note: candidate: 'template<class _IteratorL,
  class _IteratorR> constexpr decltype ((_x.base() - __y.base())) std::operator-(const std::move_iterator<_IteratorL>&,
  const std::move_iterator<_IteratorR>&)'
1639 |     operator-(const move_iterator<_IteratorL>& __x,
      |     ~~~~~~
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_iterator.h:1639:5: note: template argument deduction/
  substitution failed:
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:62,
   from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h:1955:50: note: 'std::_List_iterator<int>' is not derived
  from 'const std::move_iterator<_IteratorL>'
1955 |         std::_lg(__last - __first) * 2,
      |         ~~~~~^~~~~~
```

Concepts (C++20)

```
#include <algorithm>
#include <vector>
#include <list>
#include <iterator>

template<std::random_access_iterator Iter>
void clever_sort(Iter begin, Iter end) {
    if (!is_sorted(begin, end))
        std::sort(begin, end);
}

int main() {
    std::vector<int> data1 = {3,-1,10};
    clever_sort(data1.begin(), data1.end());

    std::list<int> data2 = {3,-1,10};
    clever_sort(data2.begin(), data2.end());
}
```

```
In function 'int main()':
<source>:18:14: error: no matching function for call to 'clever_sort(std::__cxx11::list<int>::
    iterator, std::__cxx11::list<int>::iterator)'
   18 |   clever_sort(data2.begin(), data2.end());
      |   ~~~~~
<source>:8:6: note: candidate: 'template<class Iter> requires random_access_iterator<Iter>
    void clever_sort(Iter, Iter)'
     8 |   void clever_sort(Iter begin, Iter end) {
       |   ~~~~~
<source>:8:6: note: template argument deduction/substitution failed:
<source>:8:6: note: constraints not satisfied
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/compare:39,
    from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_pair.h:65,
    from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/utility:70,
    from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:60,
    from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/concepts: In substitution of 'template<
    class Iter> requires random_access_iterator<Iter> void clever_sort(Iter, Iter) [with Iter
    = std::_List_iterator<int>]':
<source>:18:14: required from here
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/concepts:67:13: required for the
    satisfaction of 'derived_from<typename std::__detail::__iter_concept_impl<_Iter>::type,
    std::random_access_iterator_tag>' [with _Iter = std::_List_iterator<int>]
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/iterator_concepts.h:660:13:
    required for the satisfaction of 'random_access_iterator<Iter>' [with Iter = std::
    _List_iterator<int>]
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/concepts:67:28: note: 'std::
    random_access_iterator_tag' is not a base of 'std::bidirectional_iterator_tag'
   67 |   concept derived_from = __is_base_of(_Base, _Derived)
      |   ~~~~~
      |   ~~~~~
```

Spezialisierung

- Grundidee: allgemeine Implementierung für beliebige Typen, aber spezialisierte (in der Regel: effizientere) Implementierung für bestimmte Typen
- Anwendbar auf Datenstrukturen (z. B. platzsparendere interne Darstellung) und Algorithmen (z. B. effizientere Sortierung)
- Beispiele
 - Allgemeiner `vector<T>`, aber platzkomprimierter `vector<bool>`
 - Quicksort für einen Container mit wahlfreiem Zugriff, ansonsten Mergesort

Spezialisierung

```
template <>
class Pair<bool>{
    short both;
public:
    Pair(bool l, bool r):both{(l?1:0) + (r?2:0)} {};
    std::ostream& print (std::ostream& os) const{
        return os << "(" << both % 2 << "," << both / 2 << ")";
    }
};
```

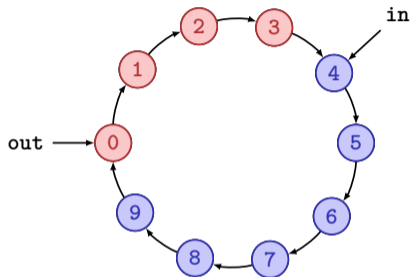
Spezialisierung

```
template <>
class Pair<bool>{
    short both;
public:
    Pair(bool l, bool r):both{(l?1:0) + (r?2:0)} {};
    std::ostream& print (std::ostream& os) const{
        return os << "(" << both % 2 << "," << both / 2 << ")";
    }
};

Pair<int> i(10,20); // ok -- generic template
std::cout << i << std::endl; // (10,20);
Pair<bool> b(true, false); // ok -- special bool version
std::cout << b << std::endl; // (1,0)
```

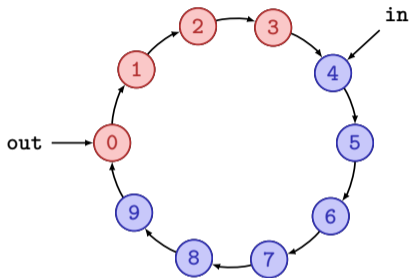

Templateparametrisierung mit Werten

```
template <typename T, int size>  
class CircularBuffer{  
    T buf[size] ;  
    int in; int out;
```



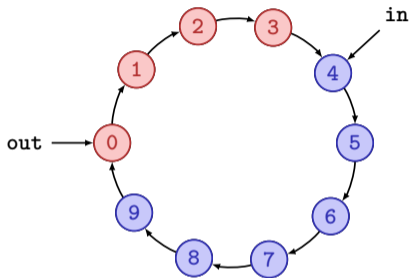
Templateparametrisierung mit Werten

```
template <typename T, int size>
class CircularBuffer{
    T buf[size] ;
    int in; int out;
public:
    CircularBuffer():in{0},out{0}{};
    bool empty(){
        return in == out;
    }
    bool full(){
        return (in + 1) % size == out;
    }
}
```



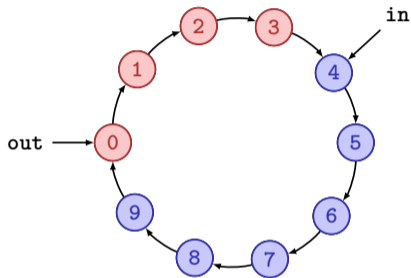
Templateparametrisierung mit Werten

```
template <typename T, int size>
class CircularBuffer{
    T buf[size] ;
    int in; int out;
public:
    CircularBuffer():in{0},out{0}{};
    bool empty(){
        return in == out;
    }
    bool full(){
        return (in + 1) % size == out;
    }
    void put(T x); // declaration
    T get();      // declaration
};
```



Templateparametrisierung mit Werten

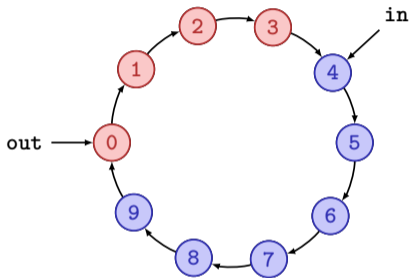
```
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}
```



Templateparametrisierung mit Werten

```
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}
```

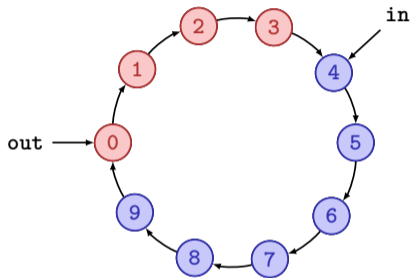
```
template <typename T, int size>
T CircularBuffer<T,size>::get(){
    assert(!empty());
    T x = buf[out];
    out = (out + 1) % size;
    return x;
}
```



Templateparametrisierung mit Werten

```
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}
```

```
template <typename T, int size>
T CircularBuffer<T,size>::get(){
    assert(!empty());
    T x = buf[out];
    out = (out + 1) % size;
    return x;
}
```



← Optimierungspotential, wenn $size = 2^k$.

Spezialisierung

Spezialisierung hat viele verschiedene Anwendungsfälle und kann in unterschiedlichen Formen auftreten:

- Optimierung durch Spezialisierung der Typen: z.B. `vector<T>` generisch, `vector<bool>` spezifisch
- Optimierung durch Spezialisierung für Werte: z.B. `std::array<T, n>`,
- Typspezifische Implementierung durch Spezialisierung: z.B. `std::hash<T>`, verwendet z.B. von `std::unordered_set`.
Auf dieser Idee basieren `traits`.