1. C++ vertieft (I)

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

Was lernen wir heute?

- 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 << " ";
}
```

Das wollen wir genau verstehen!

Das sieht umständlich aus!

Nützliche Tools (1): `auto` (C++11)

Das Schlüsselwort `auto`:

Der Typ einer Variablen wird inferiert vom Initialisierer.

Beispiele

```
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 << " ";
  }
}
```


Nützliche Tools (2): 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.

Beispiele

```
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;
```


Ok, das ist 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 << " ";
}
```


Für unser genaues Verständis

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
- Funktoren und Lambda-Ausdrücke

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;
}
```


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;
}
```

```
*** 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;
}
```

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;
}
```

Jetzt ist es zumindest korrekt. Aber umständlich.

Eleganter geht so (Teil 1):

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


Eleganter geht so (Teil 2):

```
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);
  }
}
```


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] = double;
  }
}
```

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;
}
```

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


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
}
```

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;
    };
}
```

```
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;
}
```

Erklärung

Wenn das Quellobjekt einer Zuweisung direkt nach der Zuweisung nicht weiter existiert, dann kann der Compiler den Move-Zuweisungsoperator anstelle des Zuweisungsoperators einsetzen.[1] Damit wird eine potentiell teure Kopie vermieden.

Anzahl der Kopien im vorigen Beispiel reduziert sich zu 1.

[1] Analoges 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;
}
```

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;
}
```

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;
}
```

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);
}
```

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);
}
```

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())
```


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;
    }
}
```

```
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;
  double* begin();
  double* end();
}
```


Const Iterator für den Vektor

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

}
```

```
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;
  double* begin();
  double* end();
  const double* begin() const;
  const double* end() const;
}
```

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;
}
```


Heutige 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.

Übung heute

Implementieren Sie einen *dynamischen* Vektor von `int`s, der *Indexoperatoren* anbietet und über *effizientes Speichermanagement* verfügt.