

6. C++ vertieft (I)

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

Wir erinnern uns...

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

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

Das wollen wir doch genau verstehen!

Und zumindest das scheint uns zu umständlich!

165

166

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
```

167

Etwas besser...

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

int main(){
    std::vector<int> v(10,0); // Vector of length 10

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

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

168

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;
```

169

Ok, das ist cool!

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

int main(){
    std::vector<int> v(10,0); // Vector of length 10

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

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

170

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

171

Eine Klasse für Vektoren

```
class vector{
    int size;
    double* elem;
public:
    // constructors
    vector(): size{0}, elem{nullptr} {};

    vector(int s):size{s}, elem{new double[s]} {}
    // destructor
    ~vector(){
        delete[] elem;
    }
    // something is missing here
}
```

172

Elementzugriffe

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

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    double get(int i) const;
    void set(int i, double d);
    int length() const;
}
```

173

Was läuft schief?

```
int main(){
    vector v(32);
    for (int i = 0; i<v.length(); ++i)
        v.set(i,i);
    vector w = v;
    for (int i = 0; i<w.length(); ++i)
        w.set(i,i*i);
    return 0;
}
```

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    double get(int i);
    void set(int i, double d);
    int length() const;
}
```

```
*** Error in 'vector1': double free or corruption
(!prev): 0x000000000d23c20 ***
===== Backtrace: =====
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5) [0x7fe5a5ac97e5]
```

174

Rule of Three!

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

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    vector(const vector &v);
    double get(int i);
    void set(int i, double d);
    int length() const;
}
```

175

Rule of Three!

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

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    vector(const vector &v);
    vector& operator=(const vector&v);
    double get(int i);
    void set(int i, double d);
    int length() const;
}
```

Jetzt ist es zumindest korrekt. Aber umständlich.

176

Eleganter geht so:

```
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(size, v.size);
    std::swap(elem, v.elem);
}
}
```

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    vector(const vector &v);
    vector& operator=(const vector&v);
    double get(int i);
    void set(int i, double d);
    int length() const;
}
```

177

Arbeit an der Fassade.

Getter und Setter unschön. Wir wollen einen Indexoperator.
Überladen! So?

```
class vector{
...
double operator[] (int pos) const{
    return elem[pos];
}
void operator[] (int pos, double value){
    elem[pos] = value;
}
}
```

Nein!

178

Referenztypen!

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

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    vector(const vector &v);
    vector& operator=(const vector&v);
    double operator[] (int pos) const;
    double& operator[] (int pos);
    int length() const;
}
```

179

Soweit, so gut.

```
int main(){
    vector v(32); // Constructor
    for (int i = 0; i<v.length(); ++i)
        v[i] = i; // Index-Operator (Referenz!)

    vector w = v; // Copy Constructor
    for (int i = 0; i<w.length(); ++i)
        w[i] = i*i;

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

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    vector(const vector &v);
    vector& operator=(const vector&v);
    double operator[] (int pos) const;
    double& operator[] (int pos);
    int length() const;
}
```

180

Anzahl Kopien

Wie oft wird `v` kopiert?

```
vector operator+ (const vector& l, double r){
    vector result (l); // Kopie von l nach result
    for (int i = 0; i < l.length(); ++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 zwei Mal kopiert.

181

Move-Konstruktor und Move-Zuweisung

```
class vector{
...
    // move constructor
    vector (vector&& v): size(0), elem{nullptr} {
        swap(v);
    };
    // move assignment
    vector& operator=(vector&& v){
        swap(v);
        return *this;
    };
};
```

```
class vector{
public:
    vector ();
    vector(int s);
    vector ();
    vector(const vector &v);
    vector& operator=(const vector&v);
    vector (vector&& v);
    vector& operator=(vector&& v);
    double operator[] (int pos) const;
    double& operator[] (int pos);
    int length() const;
};
```

182

Erklärung

Wenn das Quellobjekt einer Zuweisung direkt nach der Zuweisung nicht weiter existiert, dann kann der Compiler den Move-Zuweisungsoperator anstelle des Zuweisungsoperators einsetzen.³ Damit wird eine potentiell teure Kopie vermieden.

Anzahl der Kopien im vorigen Beispiel reduziert sich zu 1.

³Analoges gilt für den Kopier-Konstruktor und den Move-Konstruktor.

183

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() {}
};
```

184

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;}
};
```

185

186

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

187

188

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 (genauer: zu einer R-Wert Referenz) machen. Dann kommt wieder Move-Semantik zum Einsatz. <http://en.cppreference.com/w/cpp/utility/move>

189

190

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+size;
    }
}
```

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    vector(const vector &v);
    vector& operator=(const vector&v);
    vector (vector&& v);
    vector& operator=(vector&& v);
    double operator[] (int pos) const;
    double& operator[] (int pos);
    int length() const;
    double* begin();
    double* end();
}
```

191

192

Const Iterator für den Vektor

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

```
class vector{
public:
    vector();
    vector(int s);
    ~vector();
    vector(const vector &v);
    vector& operator=(const vector&v);
    vector(vector&& v);
    vector& operator=(vector&& v);
    double operator[] (int pos) const;
    double& operator[] (int pos);
    int length() const;
    double* begin();
    double* end();
    const double* begin() const;
    const double* end() const;
}
```

193

Zwischenstand

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

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

194

Nützliche Tools (3): using (C++11)

using ersetzt in C++11 das alte typedef.

```
using identifier = type-id;
```

Beispiel

```
using element_t = double;
class vector{
    std::size_t size;
    element_t* elem;
...
}
```

195