6. C++ advanced (I)

Repetition: vectors, pointers and iterators, range for, keyword auto, a class for vectors, subscript-operator, move-construction, iterators

We look back...

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

We look back...

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

We want to understand this in depth!

We look back...

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

We want to understand this in depth!

At least this is too pedestrian

Useful tools (1): `auto` (C++11)

The keyword `auto`:

The type of a variable is inferred from the initializer.

Useful tools (1): `auto` (C++11)

The keyword `auto`:

The type of a variable is inferred from the initializer.

Examples

```
int x = 10;
```

Useful tools (1): `auto` (C++11)

The keyword `auto`:

The type of a variable is inferred from the initializer.

Examples

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

Useful tools (1): `auto` (C++11)

The keyword `auto`:

The type of a variable is inferred from the initializer.

Examples

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

Useful tools (1): `auto` (C++11)

The keyword `auto`:

The type of a variable is inferred from the initializer.

Examples

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

Useful tools (1): `auto` (C++11)

The keyword `auto`:

The type of a variable is inferred from the initializer.

Examples

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

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

Useful tools (2): range `for` (C++11)

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

range-declaration: named variable of element type specified via the sequence in range-expression

range-expression: Expression that represents a sequence of elements via iterator pair `begin()`, `end()` or in the form of an intializer list.

Useful tools (2): range `for` (C++11)

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

range-declaration: named variable of element type specified via the sequence in range-expression

range-expression: Expression that represents a sequence of elements via iterator pair `begin()`, `end()` or in the form of an intializer list.

Examples

Useful tools (2): range `for` (C++11)

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

range-declaration: named variable of element type specified via the sequence in range-expression

range-expression: Expression that represents a sequence of elements via iterator pair `begin()`, `end()` or in the form of an intializer list.

Examples

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

Useful tools (2): range `for` (C++11)

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

range-declaration: named variable of element type specified via the sequence in range-expression

range-expression: Expression that represents a sequence of elements via iterator pair `begin()`, `end()` or in the form of an intializer list.

Examples

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

Useful tools (2): range `for` (C++11)

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

range-declaration: named variable of element type specified via the sequence in range-expression

range-expression: Expression that represents a sequence of elements via iterator pair `begin()`, `end()` or in the form of an intializer list.

Examples

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

Useful tools (2): range `for` (C++11)

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

range-declaration: named variable of element type specified via the sequence in range-expression

range-expression: Expression that represents a sequence of elements via iterator pair `begin()`, `end()` or in the form of an intializer list.

Examples

```
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;
```

That is indeed 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: x)
    std::cout << i << " ";
}
```

For our detailed understanding

We build a vector class with the same capabilities ourselves!

For our detailed understanding

We build a vector class with the same capabilities ourselves!

On the way we learn about

For our detailed understanding

We build a vector class with the same capabilities ourselves!

On the way we learn about

- RAII (Resource Acquisition is Initialization) and move construction

For our detailed understanding

We build a vector class with the same capabilities ourselves!

On the way we learn about

- RAII (Resource Acquisition is Initialization) and move construction
- Index operators and other utilities

For our detailed understanding

We build a vector class with the same capabilities ourselves!

On the way we learn about

- RAII (Resource Acquisition is Initialization) and move construction
- Index operators and other utilities
- Templates

For our detailed understanding

We build a vector class with the same capabilities ourselves!

On the way we learn about

- RAII (Resource Acquisition is Initialization) and move construction
- Index operators and other utilities
- Templates
- Exception Handling

For our detailed understanding

We build a vector class with the same capabilities ourselves!

On the way we learn about

- RAII (Resource Acquisition is Initialization) and move construction
- Index operators and other utilities
- Templates
- Exception Handling
- Functors and lambda expressions

A class for vectors

```
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
}
```

Element access

```
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;
}
```

What's the problem here?

```
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;
}
```

What's the problem here?

```
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): 0x0000000000d23c20 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7fe5a5ac97e5]
```

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;
}
```

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;
}
```

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;
}
```

Now it is correct, but cumbersome.

More elegant this way:

```
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;
}
```

Syntactic sugar.

Getters and setters are poor. We want an index operator.

Syntactic sugar.

Getters and setters are poor. We want an index operator.

Overloading!

Syntactic sugar.

Getters and setters are poor. We want an index operator.

Overloading! So?

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

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

Syntactic sugar.

Getters and setters are poor. We want an index operator.

Overloading! So?

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

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

Nein!

Reference types!

```
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;
}
```

So far so good.

```
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;
}
```

Number copies

How often is `v` being copied?

```
vector operator+ (const vector& l, double r){
    vector result (l);
    for (int i = 0; i < l.length(); ++i) result[i] = l[i] + r;
    return result;
}

int main(){
    vector v(16);
    v = v + 1;
    return 0;
}
```

Number copies

How often is v being copied?

```
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); // allocation of elems[16]
    v = v + 1;   // copy when assigned!
    return 0;    // deconstruction of v
}
```

Number copies

How often is `v` being copied?

```
vector operator+ (const vector& l, double r){
    vector result (l);
    for (int i = 0; i < l.length(); ++i) result[i] = l[i] + r;
    return result;
}

int main(){
    vector v(16);
    v = v + 1;
    return 0;
}
```

`v` is copied twice

Move construction and move assignment

```
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;
}
```

Explanation

When the source object of an assignment will not continue existing after an assignment the compiler can use the move assignment instead of the assignment operator.[3] A potentially expensive copy operations is avoided this way.

Number of copies in the previous example goes down to 1.

[3] Analogously so for the copy-constructor and the move constructor

Illustration of the Move-Semantics

```
// 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() {}
};
```

How many Copy Operations?

```
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;
}
```

How many Copy Operations?

```
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;
}
```

Output

default constructor
copy constructor
copy constructor
copy constructor
copy assignment

4 copies of the vector

Illustration of the Move-Semantics

```
// 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;}
};
```

How many Copy Operations?

```
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;
}
```

How many Copy Operations?

```
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;
}
```

Output

default constructor
copy constructor
copy constructor
copy constructor
move assignment

3 copies of the vector

How many Copy Operations?

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

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

How many Copy Operations?

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

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

Output

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

1 copy of the vector

How many Copy Operations?

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

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

Output
default constructor
copy constructor
move constructor
move constructor
move constructor
move assignment

1 copy of the vector

Explanation: move semantics are applied when an x-value (expired value) is assigned. R-value return values of a function are examples of x-values.

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

How many Copy Operations?

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

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

How many Copy Operations?

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

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

Output
default constructor
default constructor
copy constructor
copy assignment
copy assignment

3 copies of the vector

Forcing x-values

```
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);
}
```

Forcing x-values

```
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);
}
```

Output

default constructor
default constructor
move constructor
move assignment
move assignment

0 copies of the vector

Forcing x-values

```
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);
}
```

Output
default constructor
default constructor
move constructor
move assignment
move assignment

0 copies of the vector

Explanation: With std::move an l-value expression can be transformed into an x-value. Then move-semantics are applied. http://en.cppreference.com/w/cpp/utility/move

Range `for`

We wanted this:

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

Range `for`

We wanted this:

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

In order to support this, an iterator must be provided via `begin` and `end` .

Iterator for the vector

```
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();
}
```

Const Iterator for the vector

```
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;
}
```

Intermediate result

```
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;
}
```

Useful tools (3): `using` (C++11)

`using` replaces in C++11 the old `typedef`.

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

Useful tools (3): `using` (C++11)

`using` replaces in C++11 the old `typedef`.

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

Beispiel

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