6. C++ advanced (I)

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

What do we learn today?

- Keyword `auto`
- Ranged `for`
- Short recap of the Rule of Three
- Subscript operator
- Move Semantics, X-Values and the Rule of Five
- Custom Iterators

We look back...

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

We want to understand this in depth!

This looks too pedestrian

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
```

Slightly better...

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


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); // Vector of length 10

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

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


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
- Subscript operators and other utilities
- Templates
- Exception Handling
- Functors and lambda expressions

A class for (double) vectors

```
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;
}
```


Element access

```
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 Interface)

What's the problem here?

```
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 Interface)

```
*** 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 Interface)

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 Interface)

Now it is correct, but cumbersome.

More elegant this way (part 1):

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


More elegant this way (part 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);
  }
}
```

copy-and-swap idiom: all members of `*this` are exchanged with members of `cpy`. When leaving `operator=`, `cpy` is cleaned up (deconstructed), while the copy of the data of `v` stay in `*this`.

Syntactic sugar.

Getters and setters are poor. We want a subscript (index) operator.

Overloading! So?

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

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

No!

Reference types!

```
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];
  }
}
```


So far so good.

```
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;
}
```


Number copies

How often is v being copied?

```
Vector operator+ (const Vector& l, double r){
    Vector result (l); // copy of l to result
    for (std::size_t i = 0; i < l.size(); ++i)
        result[i] = l[i] + r;
    return result; // deconstruction of result after assignment
}
int main(){
    Vector v(16); // allocation of elems[16]
    v = v + 1;   // copy when assigned!
    return 0;    // deconstruction of v
}
```

v is copied (at least) twice

Move construction and move assignment

```
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;
}
```

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.[5] Expensive copy operations are then avoided.

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

[5] 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;
}
```

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;
}
```

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;
}
```

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);
}
```

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);
}
```

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 forced into an x-value. Then move-semantics are applied. http://en.cppreference.com/w/cpp/utility/move

std::swap & std::move

`std::swap` is implemented as above (using templates)

`std::move` can be used to move the elements of a container into another

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


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+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();
}
```

(Pointers support iteration)

Const Iterator for the vector

```
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;
}
```

Intermediate result

```
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;
}
```


Today's Conclusion

- Use `auto` to infer a type from the initializer.
- X-values are values where the compiler can determine that they go out of scope.
- Use move constructors in order to move X-values instead of copying.
- When you know what you are doing then you can enforce the use of X-Values.
- Subscript operators can be overloaded. In order to write, references are used.
- Behind a ranged `for` there is an iterator working.
- Iteration is supported by implementing an iterator following the syntactic convention of the standard library.