X. Move-semantics and rvalue-references

Prerequisites

Let's look at an example of an old problem:

```
vector<string> v; // realized as we did it
v.push_back("abc");
```

This code (if we think, that vector is implemented the way we did in the first semester) creates **2 string** instead of one. To understand what is going on, look at this:

```
void push_back(const T& x) {
    ...
    new (ptr) T(x);
    ...
}
```

So, basically, we create a temporary string, that is then copied to the place it reside in, which is inefficient.

Consider resizing when capacity equals size. When we make a new array, we **have to** call c-tor to copy all elements to the new array. Let's say we hold string of length n, then we would have to copy **a lot** of memory.

Consider another example:

What we would like to achieve is to somehow be able to *cheaply move* object to it's final destination.

To be precise, in these examples:

```
vector<string> v;
string s = "abc";
v.push_back(s);
```

We want to move s to v's end, **and we don't care what happens to *s***, it can even be empty afterwards. Or here:

```
struct Data {
    string s;
    Data(const string& s) : s(s) {}
};
```

We would like to *put the argument into the member of our class.* (We don't want unnecessary copying)

Shared pointers could solve a problem like this, but it would add indirection to our program, and also will also slow it down.

Let's look at **the swap that we know**:

```
void swap(T& x, T& y) {
    T t = x;
    x = y;
    y = t;
}
```

It works in linear time relative to the size of x and y. If we want it to be faster, we could make it swap to work this way: swap the members of x and y recursively.

One solution for the first problem, was the following:

```
...
v.emplace_back("abc");
```

Where *emplace_back* takes the arguments of c-tor for the type, and constructs it in-place.

```
void emplace_back(const Args&... args)
// Again, the implemntation is from our point of view
// (when we don't know move semantics)
```

However there is a much better solution for the problem in general.

A new view on the problem (move-semantics)

On the level of interface, we added *std::move* function and operations, that use move-semantics. For example if type T supports move semantics (every basic type does), the following will work in *O(1)*:

```
void swap(T& x, T& y) {
    T t = std::move(x);
    x = std::move(y);
    y = std::move(t);
}
```

It may look like exatly like recursive swapping, **however** it generalizes on temporary objects, that compiler already moves by default. That's why swapping is not as general as moving.

Our first example also works faster, and we don't even have to do anything:

```
vector<string> v;
v.push_back("abc");
```

However, the class must implement move-operations to use all these novelties. It is already true for all standard types and std containers. **How to do it for custom types**? Let's consider the *vector* class:

```
template <typename T>
class vector:
    T* arr;
    size_t sz;
```

```
    size_t cap;
public:
    // ...
```

Right now we didn't add anything, and so std::move will work for us **exactly like copying**. Let's add a naive realization of move-ctor:

```
template <typename T>
class vector:
    T* arr;
    size_t sz;
    size_t cap;
public:
    // ...
    vector(vector&& other)
        : str(other.str)
        , sz(other.sz)
        , cap(other.cap)
    {
        other.str = nullptr;
        other.sz = other.cap = 0;
    }
```

So, we basically need to:

- Take the members of *other* and *put them* into our members
- Clear the members of *other*

We set size and capacity of *other* to 0 (because we guarantee, that the elements from 0 to sz - 1 are valid)

Note: we don't use std::move on basic types, because **std::move of basic types is equivalent to copying**. So it is meaningless to move basic types.

Naive implementation of move-operator=:

```
    ...
    vector& operator=(vector&& other) {
        auto copy = std::move(other);
        swap(copy);
        return *this;
    }
```

Note: we don't want to do this:

```
    ...
    vector& operator=(vector&& other) {
        swap(other);
        return *this;
    }
```

Because we usually expect, that after a move-operation, *other* becomes empty.

Note: the new c-tor of *Data* struct is as follows:

```
struct Data {
    string str;
    Data(string&& str)
        : str(std::move(str))
    {}
    Data(const std::string& str)
        : str(str)
    {}
};
```

In fact, the compiler chooses the right c-tor, based on overload rules (which means, that taking away const-ness is illegal, in which case the copy-ctor will be used).

Note: default move c-tor just std::move's everything in everything, **which would be incorrect in vector or string**. It is correct, if and only if the underlying members also support move-semantics.

Rule of Five

If one of these is non-trivial:

- Move-ctor
- Move-operator=
- Copy-ctor
- Copy-operator=
- Destructor

Then all of them must be defined

Note: if none of those are defined, than everything is defined **by default**.

Interesting fact

Examples of classes that allow move-semantics, but not copying:

- std::unique_ptr
- std::thread

std::move implementation

The following implementation is naive, but works in ~90% of cases:

```
template <typename T>
T&& move(T& x) {
    return static_cast<T&&>(x);
}
```

So, **in reality**, it is just a way to **make the compiler use the overload we want**.