std::swap

```
template<typename T>
void swap(T& t1, T& t2) {
    T temp(t1);
    t1 = t2;
    t2 = temp;
}
```

- Standard implementering

- Prova med FltPtr:
- Onödig allokering/deallokering
- Skriv egen Swap

Swap idiom

- Basic idea:
  - Sometimes it is much faster to swap than doing three assignments.
- An example is our String implementation:
  - Assignment: Always copy of the characters, maybe one deallocation and one allocation on the heap.
  - Swap: Just swap the three members (char* data, int size and int capacity)

Swap idiom, how to make it work

- Assume that we write our own implementation of Vector to make it behave in some special way (e.g. being very conservative with extra memory allocation when the capacity is exceeded):

```
namespace MyLib {
    template <class T> MyVector {
        template<T>
        friend void swap(MyVector<T>& lhs, MyVector<T>& rhs) {
            using std::swap;
            swap(lhs.data, rhs.data);
            swap(lhs.capacity, rhs.capacity);
            swap(lhs.size, rhs.dize);
        }
        . . .
    }
}
```

- Problem: Our swap is now inside namespace MyLib. The standard swap is inside namespace std.
  But when a template use swap it has to find the right one!
- Solution, always write:

```
using std::swap;
swap(x, y);
```

- If the type of x is MyVector “our” swap function will be found by Argument Dependent Lookup, i.e. the C++ compiler will find the type of x and look up swap in that scope.
- If no swap is found there the compiler will find the generic std:swap instead.

Move constructors

- Sometimes an objects last use is to be copied from, e.g.

```
C Foo() {
    C temp;
    // here we construct the return value of the function
    return temp;
}
```

- If C has resources allocated we will copy them just to destroy the original. Better to give away the resources.

```
class FltPtr {
    float * ptr;
public:
    … a lot of code here
    FltPtr(FltPtr&& other) { //declaration of a move-constructor
        ptr = other.ptr;     //We steal it
        other.ptr = nullptr; //but leave other in a consistent state
    }
}
```

Swap med move constructors

```
template<typename T>
void swap(T& t1, T& t2) {
     T temp(std::move(t1));
     t1 = std::move(t2);
     t2 = std::move(temp);
}
```

Move assignment operator

```
class FltPtr {
    float * ptr;
public:
    … a lot of code here
    FltPtr(FltPtr&& other) { //declaration of a move-constructor
        ptr = other.ptr;     //We steal it
        other.ptr = nullptr; //but leave other in a consistent state
    }

    FltPtr& operator=(FltPtr&& other) {
    //declaration of a move-assignment
        using std::swap;
        swap(*this, other);
        return *this;
    }
};
```

std::move and compiler treatments of move constructors

```
template< class T >
typename std::remove_reference<T>::type&&
move(T&& t) noexcept;
```

- A simplified version!

```
template<class T>
T&& move(T& t) noexcept {
    return t;
}
```