Copy versus Move: swap

```
std::vector<int> a, b;
swap(a, b);

template <typename T>
void swap(T& a, T& b){
  T tmp(a);
  a = b;
  b = tmp;
}
template <typename T>
void swap(T& a, T& b){
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);
}
```

swap.cpp

`T tmp(a);`

- Allocates `tmp` and each element from `tmp`.
- Copy each element from `a` to `tmp`.
- Deallocates `tmp` and each element from `tmp`.

`T tmp(std::move(a));`

- Redirects the pointer from `tmp` to `a`.

Move Semantics: std::move

The function std::move moves its resource.

- std::move
  - Needs the header <utility>.
  - Converts its argument into a rvalue reference.
  - The compiler applies move semantics.
  - Is under the hood a static_cast to a rvalue reference

    ```
    static_cast<std::remove_reference<decltype(arg)>::type&&>(arg);
    ```

Copy semantics is a fallback for move semantics.

Move Semantics: STL

Each container of the STL and `std::string` has two new construtors:

- Move constructor
- Move assignment operator

- These new member functions take its arguments as **non-constant** rvalue references .

- Example

```
vector{
   vector(vector&& vec);                       // move consturctor
   vector& operator = (vector&& vec);          // move assignmen operator
   vector(const vector& vec);                  // copy constructor
   vector& operator = (const vector& vec);     // copy assignment operator
   . . .
```

The classical copy constructor and copy assignment operator take its argument as **constant** lvalue reference.