Copy vs. Move Semantics

In this lesson, we will compare the performance of the copy and move operations for the containers in the STL.

WE'LL COVER THE FOLLOWING

A lot has been written about the advantages of the **move** semantic over the **copy** semantic. Instead of an expensive copy operation, we can use a cheap move operation. But, what does that mean?

The subtle difference is that if we create a new object based on an existing one, the copy semantic will copy the elements of the existing resource, whereas the move semantic will move the elements of the resource. So, of course, copying is expensive and moving is cheap. But there are additional serious consequences.

1. With the copy semantic, it is possible that a std::bad_alloc will be thrown because our program is out of memory.
2. The resource of the move operation is in a *"valid but unspecified state"* afterward.

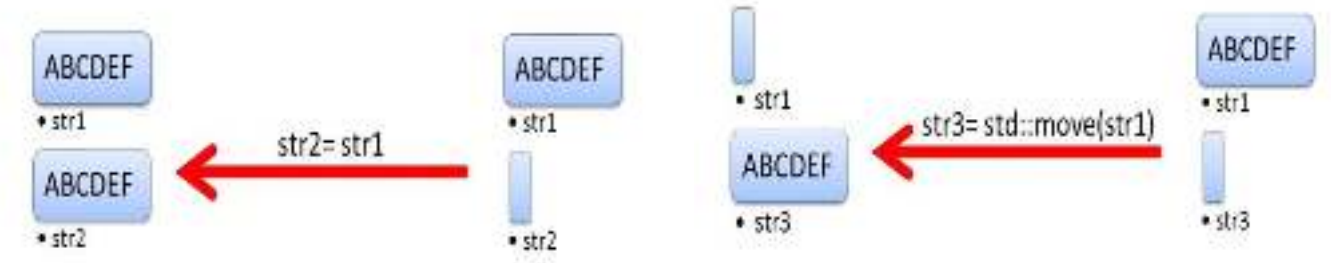
The second point can be explained well by `std::string`.

Copy

```
string str1("ABCDEF");
string str2;
str2 = str1;
```

Move

```
string str1("ABCDEF");
string str3;
str3 = std::move(str1);
```



In the copy semantic, both strings (`str1` and `str2`) have the same content, “ABCDEF”, after the copy operation. So, what’s the difference in the move semantic?

The string `str1` is empty after the move operation. This is not guaranteed, but is often the case. We explicitly requested the move semantic with the function `std::move`. The compiler will automatically perform the move semantic if it is sure that the source of the move semantic is not needed anymore.

> 💡 We will explicitly request the move semantic in our program by using std::move. Although it is called `std::move`, we should have a different mental picture in mind. When we move an object, we transfer ownership. **By moving, we give the object to someone else.**

Some important points

A class supports **copy semantics** if the class has a copy constructor and a copy assignment operator.

A class supports **move semantics** if the class has a move constructor and a move assignment operator.

If a class has a copy constructor, it should also have a copy assignment operator. The same holds true for the move constructor and move assignment operator.

`std::swap`

Below is an example of how move and copy semantics can be used to swap

two variables. The copy version represents the way we did it before C++11. It shows how the move semantic is more efficient and saves memory.

```
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);
}
```

Explanation

The `T tmp(a);` command essentially:

- allocates `tmp` and each element from `tmp`.
- copies each element from `a` to `tmp`.
- deallocates `tmp` and each element from `tmp`.

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

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

Further information

- std::bad_alloc
- std::move

Let's learn about move semantics in more detail in the next lesson.