Copy versus Move Semantic

In this lesson, we will compare the performance of the copy and move semantic for the containers in the Standard Template Library (STL).

WE'LL COVER THE FOLLOWING

A lot has been written on the advantages of the move semantic over the copy semantic. Rather than an expensive copy operation, we can use a cheap move operation. Let's break than down further.

There is one subtle difference between copy and move semantic: if we create a new object based on an existing one, the copy semantic will copy the elements of the resource, while the move semantic will move the elements of the resource. Of course, copying is expensive, and moving is cheap, but there are additional serious consequences to this technique:

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

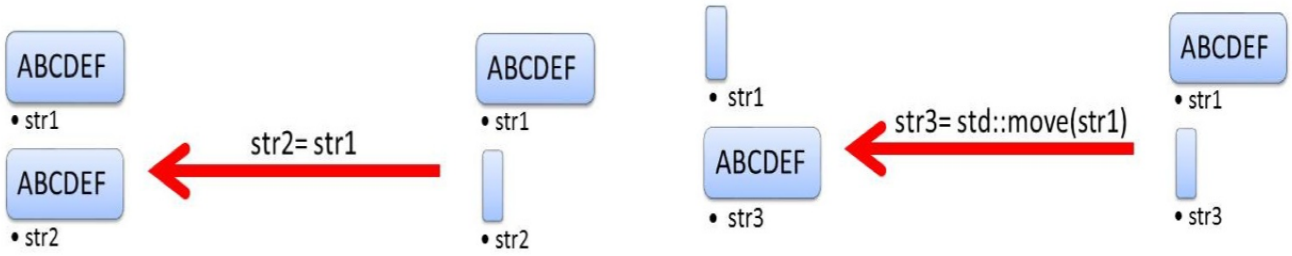
The second point is demonstrated clearly with `std::string`.

Copy vs. Move

Copy	Move

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

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



In copy semantic, both strings `str1` and `str2` have the same content “ABCDEF” after the copy operation. So, what is the difference between copy and move semantic?

The string `str1` is in opposition with the copy semantic afterward empty. 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 ensures that the source of the move semantic is no longer needed.

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

Some Important Points to Remember

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

A class supports **move semantic** if the class has both 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 for the move constructor and move assignment operator.

std::swap

Below is an example of the process of using the move semantic and the copy semantic to swap two variables. The copy version resembles the way that we performed this function with 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

This is what the `T tmp(a);` command essentially performs:

1. Allocates `tmp` in stack and elements of `tmp` in the heap.
2. Copies each element from `a` to `tmp`.

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

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

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