Move vs. Copy

In this lesson we'll see how the std::move utility proves more efficient than copying.

WE'LL COVER THE FOLLOWING

The function `std::move`, defined in the header `<utility>`, empowers the compiler to move its resource. In the so-called *move semantic*, the values from the source object are moved to the new object. Afterward, the source is in a *well-defined* but not specified state. Most of the times that is the default state of the source. By using `std::move`, the compiler converts the source `arg` to a rvalue reference: `static_cast<std::remove_reference<decltype(arg)>::type&&>(arg)`.

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` example below.

If the compiler can not apply the *move semantic*, it falls back to the *copy*

If the compiler can not apply the *move semantic*, it falls back to the *copy semantic*.

Vector elements

```
#include <utility>
//...
std::vector<int> myBigVec(10000000, 2011);
std::vector<int> myVec;

myVec = myBigVec;              // copy semantic
myVec = std::move(myBigVec);   // move semantic
```

Strings

```
#include <iostream>
#include <utility>

// Driver code
int main()
{
        std::string str1 = "abcd";
        std::string str2 = "efgh";
        std::cout << "str1: " << str1 << std::endl;
        std::cout << "str2: " << str2 << "\n\n";

        // Copying
        str2 = str1;              // copy semantic
        std::cout << "After copying" << std::endl;
        std::cout << "str1: " << str1 << std::endl;
        std::cout << "str2: " << str2 << "\n\n";

        str1 = "abcd";
        str2 = "efgh";

        // Moving
        str2 = std::move(str1);
        std::cout << "After moving" << std::endl;
        std::cout << "str1: " << str1 << std::endl;
        std::cout << "str2: " << str2 << "\n\n";
}
```

In **line 22**, `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.

Classes

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.

User-defined data types

User-defined data types can support move and copy semantics as well.

Example

```
class MyData{
    MyData(MyData&& m) = default; // move constructor
    MyData& operator = (MyData&& m) = default; // move assignment
    MyData(const MyData& m) = default; // copy constructor
    MyData& operator = (const myData& m) = default; // copy assignment
};
```

The move semantic has priority over the copy semantic.

> 🗝️ **To move is cheaper than to copy**
> The move semantic has two advantages. Firstly, it is often a good idea to use cheap moving instead of expensive copying. So there is no superfluous allocation and deallocation of memory necessary. Secondly, there are objects, which can not be copied, e.g., a thread or a lock.

Now, let's discuss another useful function – `std::forward`.