

Back to Basic) Move semantic

Copy Semantics

```
std::vector<std::string> coll;
coll.reserve(3);

std::string s{getData()};

coll.push_back(s);

coll.push_back(getData());
```

Avoid objects with names

destruct temporary

Stack:

coll: 2 → [4 | 5 |]

s: 4 → [d | a | t | a | \0]

getData():

Heap:

[d | a | t | a | \0]

[d | a | t | a | 2 | \0]

```
std::vector<std::string> coll;
coll.reserve(3);

std::string s{getDate()};
coll.push_back(s);

coll.push_back(getData());
```

위의 방식대로 copy를 진행하게 된다면, 임시객체를 만든 뒤에, Copy를 진행하여 vector에 push_back을 진행한다.
위의 방식으로 진행하게 된다면, 임시 객체를 생성하고, copy를 진행하고, 임시 객체를 지우는데 추가적인 작업이 필요하게 된다.

move는 임시 객체의 주소를 복사를 하여, 임시 객체 자체가 움직이게 된다.
주의할 점은, std::move는 stack에 있는 객체 자체를 heap으로 움직이는 것이 아니다, 또한 stack에 남아있는 값은 재사용이 가능하다!!

Object with Names

std::move의 의미 : "I no longer need this value here"

example code

```
when you need an object/value multiple times:

std::string str(getData());

coll.push_back(str); // copy(still need the value of str)
coll2.push_back(std::Move(str)); // move(ok, no longer need the value)

When you deal with parameters:
void reinit(std::string& s)
{
    history.push_back(std::move(s)); // move(ok, no longer need the value)
    s = getDefaultVal();
}
```

std::move() 의 그림

```
std::vector<std::string> coll;
coll.reserve(3);
```

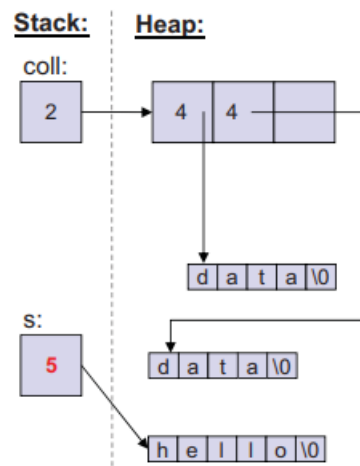
```
std::string s{getData()};
```

```
coll.push_back(s);
```

```
coll.push_back(std::move(s));
```

A moved-from library object is in a **valid but unspecified** state

```
std::cout << s;    // OK, some value
int i = s.size();  // OK, consistent value
s.append('.');     // OK, size() >= 1
char c1 = s[0];    // OK, some value
char c2 = s[5];    // Error (Undef.Behav.)
s = "hello";       // OK, specified state
...
```



- 그렇다면 `s`는 `std::move()`가 된 뒤에 어떻게 되는건가요??
 - moved-from library object는 valid 하지만, unspecified state 가 된다.
 - moved-from object를 다시 사용하는 것은 좋은 습관은 아니다.

Re-using Objects after std::move()

```
// read line-by-line from myStream and store it in a collection:
std::vector<std::string> allRows;
std::string row;
while (std::getline(myStream, row)) { // read next line into row
    allRows.push_back(std::move(row)); // and move it to somewhere
}
```

```
// swap two strings
void swap(std::string& a, std::string& b) {
    std::string tmp{std::move(a)};
    a = std::move(b); // assign new value to moved-from a
    b = std::move(tmp); // assign new value to moved-from b
}
```

Rvalue reference

```
template <typename T>
class vector {
public:
    ...
    // copy elem into the vector:
    void push_back(const T& elem);
    ...
    // move elem into the vector:
    void push_back(T&& elem);
    ...
};
```

now named
lvalue reference

declares
rvalue reference

```
#include <utility> // declares std::move()

std::vector<std::string> coll;
std::string s = getData();
...
coll.push_back(s); // copy s into coll

coll.push_back(getData()); // move temporary into coll

coll.push_back(s+s); // move temporary into coll

coll.push_back(std::move(s)); // move s into coll
// (no longer need s)

return coll;
```

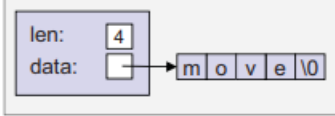
- Move semantic이 없는 이전 C++에서는 모든 객체는 container에 copy 가 되었다.
- 이는 불필요한 copy가 이루어졌다.
- C++11부터는 vector에 push_back을 진행하는 경우 && 마크를 사용해서, rvalue_push_back을 진행하게 되었다.
 - 임시 객체를 parameter로 주는 경우
 - std::move를 사용하는 경우

Copy constructor를 사용하는 경우

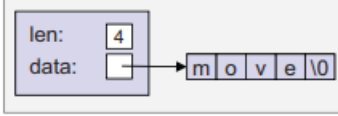
Strings With Move Semantics

coll.push_back(x)

```
string x = "move";
x: s:
```



```
string e1 = x;
e1: s:
```



```
class string {
private:
    int len; // current number of characters
    char* data; // array of characters

public:
    // create a full copy of s:
    string(const string& s)
    : len{s.len} { // copy length
        if (len > 0) { // if not empty
            data = new char[len+1]; // - new memory
            memcpy(data, s.data, // - copy chars
                len+1);
        }
    }
};
```

Copy constructor uses const lvalue reference

```
public:
    // create a copy of s with its content moved:
    string(string&& s)
    : len{s.len}, // copy length and
      data{s.data} { // copy pointer to memory
        s.data = nullptr; // erase memory at source
                          // to really move ownership
        s.len = 0;
    }
    ...
};
```

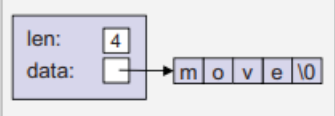
Move constructor uses rvalue reference

move constructor를 사용하는 경우

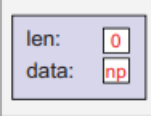
Strings With Move Semantics

coll.push_back(x)

```
string x = "move";
x:
```

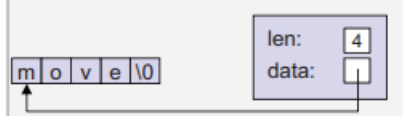


```
string e1 = x;
e1: s:
```



coll.push_back(std::move(e1))

```
string e2 = std::move(e1);
e2:
```



std::move(e1) is equivalent to static_cast<string&&>(e1)

```
class string {
private:
    int len; // current number of characters
    char* data; // array of characters

public:
    // create a full copy of s:
    string(const string& s)
    : len{s.len} { // copy length
        if (len > 0) { // if not empty
            data = new char[len+1]; // - new memory
            memcpy(data, s.data, // - copy chars
                len+1);
        }
    }
};
```

Copy constructor uses const lvalue reference

```
public:
    // create a copy of s with its content moved:
    string(string&& s)
    : len{s.len}, // copy length and
      data{s.data} { // copy pointer to memory
        s.data = nullptr; // erase memory at source
                          // to really move ownership
        s.len = 0;
    }
    ...
};
```

Move constructor uses rvalue reference

- 왼쪽에 있는 class를 보면, lvalue reference를 사용한다(Copy constructor). 왼쪽의 코드를 말로 표시한다면 "너는 다음에 다시 사용할 객체를 나에게 주는구나, 그 객체를 바탕으로 새로운 객체를 생성할게"로 된다.
- 오른쪽 코드는 rvalue reference를 사용한다(Move constructor). &&를 사용하며, 오른쪽의 코드를 말로 표현한다면 "너는 다음에 사용하지 않을 객체를 나에게 주는구나, 그 객체를 바탕으로 새로운 객체를 생성할게"가 된다.

정리

• void foo(const Type&)

- pass value without creating a copy
- can bind to everything

read-only access

- in parameter to read

• void foo(Type&)

- pass named entity to return a value
- only non-const named object (lvalues)

write access

- (in)out parameter

• void foo(Type&&)

- pass value that is no longer needed
- only objects without name or with move() (rvalues)

move access

- in parameter to adopt value

• void foo(const Type&&)

- possible, but semantic contradiction
- usually covered by const Type&

Don't use const when returning by value

```
const std::string getValue(); // forward decl.
...
std::vector<std::string> coll;
...
coll.push_back(getValue()); // copies
```

의미론적인 contradiction이 이루어진다. 나는 객체의 소유권을 steal할 것인데, const가 있다는 것은 말이 안된다. 그러나 C++에서는 이것을 허용은 한다. std::move를 사용한 뒤에서 container내에서 수정을 하려고 하면 const는 없어진다. 그러므로, std::move를 사용할 객체라고 한다면, const를 사용하지 말아라!

Move Semantics in Class

1. moved-from 객체의 상태는 `valid` 하지만 `unspecified state` 이다.
2. 만약 `move semantic` 이 없는 경우 `copy semantic` 만 사용할 수 있다.
 - `move-only type` 은 `std::stream`, `stream` 같은 타입이 있다.
3. default move operation은 생성된다.

★중요한 C++ 생성자 법칙

1. 만약 `move semantic`과 `copy constructor` 모두 지정하지 않았다면, `move semantic`과 `copy constructor` 모두 기본 구현을 자동으로 생성한다.
2. 만약 `copy constructor`를 default로 지정하거나 구현을 한 경우 `move semantic`은 이루어지지 않는다.

모두 지정하지 않은 경우

```
class Cust {
private:
    std::string first;
    std::string last;
    int val;
public:
    Cust(const std::string& f, const std::string& l, int v)
        : first{f}, last{l}, val{v} {
    }

    // no copy constructor

    // no move constructor

    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
        return strm << "[" << c.val << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;
Cust c1{"Joe", "Fox", 77};
v.push_back(std::move(c1)); // moves c1
std::cout << "c1: " << c1 << '\n'; // c1: [77: ??? ???]
```

- **Move semantics is enabled** because no other special member function is user-declared
- Unless a move is not implementable

- `copy constructor`와 `move constructor` 모두 없는 경우 `compiler`가 자동적으로 생성한다.
- 결과적으로 `c1` 객체에는 `77`은 존재하고(`copy constructor`)이기 때문, 나머지 `string`은 존재하지 않는다(`undefined state`)

copy constructor만 구현을 한 경우

```
class Cust {
private:
    std::string first;
    std::string last;
    int val;
public:
    Cust(const std::string& f, const std::string& l, int v)
        : first{f}, last{l}, val{v} {
    }

    Cust(const Cust&) = default; // copy constructor

    // no move constructor

    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
        return strm << "[" << c.val << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;
Cust c1{"Joe", "Fox", 77};
v.push_back(std::move(c1)); // copies c1
std::cout << "c1: " << c1 << '\n'; // c1: [77: Joe Fox]
```

- **Move semantics is disabled** because of user-declared other special member function
- **Copying used as fallback**

- `copy constructor`만 생성한 경우 `compiler`는 자동적으로 `move constructor`를 생성하지 않는다.

Move Semantics and Special Member Functions

```
class Cust {
private:
    std::string first;
    std::string last;
    int val;
public:
    Cust(const std::string& f, const std::string& l, int v)
        : first{f}, last{l}, val{v} {
    }
    Cust(const Cust& c) // copy constructor
        : first{c.first}, last{c.last}, val{c.val} {
    }
    Cust(Cust&& c) noexcept // move constructor
        : first{std::move(c.first)}, last{std::move(c.last)}, val{c.val} {
        c.val *= -1;
    }
    friend std::ostream& operator << (std::ostream& strm, const Cust& c) {
        return strm << "[" << c.val << ": " << c.first << " " << c.last << "]";
    }
};

std::vector<Cust> v;
Cust c1{"Joe", "Fox", 77};
v.push_back(std::move(c1)); // moves c1
std::cout << "c1: " << c1 << '\n'; // c1: [-77: ??? ???]
```

Mark move constructor with **noexcept**, if implemented and it never throws

Parameter **c** has no move semantics unless marked with **move()** again, (the caller no longer needs the value, but we might need it multiple times)

- move_constructor를 생성하는 경우, &&를 사용했더라도, 초기화 부분에서도 std::move를 사용해야 한다. ★
- move_constructor를 구현하는 경우 noexcept를 사용하는 습관을 들이자.(이동 연산자, 이동 대입 연산자)
- move_constructor를 구현하는 경우, parameter에 const를 사용하지 말자!!
 - 만약 const를 사용하게 되면, 복사를 진행하기 때문이다.

상속을 하는 경우 주의할 점

- **Declared virtual destructors disable move semantics**
 - Moving special member functions are **not generated**
 - **If and only if** a polymorphic base class has **members** expensive to copy, it might make sense to declare/define move operations
- **Don't declare destructors in derived classes** (unless you have to)

```
class Person { // polymorphic base class with virtual functions
protected:
    std::string id; // to support move semantics for id, declare move functions
public:
    ...
    virtual void print() const = 0;
    virtual ~Person() = default; // disables move semantics for members
};

class Customer : public Person { // derived polymorphic class
protected:
    std::vector<int> data; // move semantics for data enabled without special function
public:
    ...
    virtual void print() const override;
    virtual ~Customer() = default; // disables move semantics for members
};
```

생성자를 다시 선언하지 말아라!! move_semantic을 할 수 없기 때문이다.

Perfect Forwarding

- c++에는 아직 issue가 있고 improvement할 여지가 있다.