Move Semantics (Since C++11)

- 변수(좌측값)는 복사 하고, 임시값(우측값)은 데이터를 이동하자!
  - 좌측값: int a = 3; //&a로 변수 a의 주소를 알 수 있음
  - 우측값: int a = 3; //&3으로 리터럴의 주소를 알 수 있나?
- 우측값으로 객체를 생성하는 방법 제공하자!
- 복사하는 비용을 줄이자!
  - 컴파일러: 복사 생략(copy collision), 반환값 최적화(return value optimization)
  - 예) swap(Rectangle a, Retangle b) 를 하는 경우 (tmp=a; a=b; b=tmp)
- 외부 리소스(예, 동적 메모리, unique_ptr)의 소유권을 가져오자!

```
int main() {
    Point pt1 ;                          //&  연산자로 주소를 알 수 있나? l-value
    //1) readPoint() 함수의 리턴값  //&readPoint() 로 주소를 알 수 있는가?
    pt1 = readPoint() ;   // but, return value optimization
    //2) 우측값으로 객체 생성 가능한가?
    Point pt2(readPoint());
    //3) 인자값 Point (10, 20) 은 임시 객체
    print( Point {10, 20} );  // but, copy collision
}
```

rvalue (추가)

- rvalue 는 lvalue 가 아닌 나머지들(?)
- rvalue 예시
  - 임시 객체, 리터럴 상수, 함수 반환 값 (lvalue 참조 제외), built-in operators의 연산 결과
- rvalue 는 보통 짧은 lifetime 을 가지며, const나 함수의 반환 값으로 사용되지 않음
- Address of an rvalue cannot be taken by built-in address-of operator:
  - &int(), &i++[3], &42, and &std::move(x) are invalid.
- An rvalue can't be used as the left-hand operand of the built-in assignment or compound assignment operators.

```
int main() {
   String s1;
   // const char* -> String, but 임시 객체
   s1 = "hello"
   String s2;
   s2 + s2 = s2;  //s2 + s2 의 연산 결과는 rvalue
}
```

```
   void func (int&& x);
   int i = 3;
   func (i);          // error
   func (42);         // OK!, literal
   func (v.size());  // OK!, 임시 객체
}
```

이동 생성자(move Constructor)

```
class Point {
    int x, y ;
public:
    Point(int x=0, int y=0) { this->x = x ; this->y = y ; }          // 일반 생성자
    Point(const Point& pt) { x = pt.x ; y = pt.y ; }                  // 복사 생성자
    Point(Point&& pt) noexcept                                        // 이동 생성자
                : x{std::move(pt.x)}, y{std::move(pt.y)}              // move
                { pt.x = 0; pt.y=0 ; }                                // clear
    Point& operator=(Point&& pt) noexcept {                           // 이동 할당 연산자
        x = std::move(pt.x); y = std::move(pt.y);                     // move
        pt.x = 0; pt.y=0 ;                                            // clear
        return *this;
    }
    int getX() const { return x ; } int getY() const { return y ; }
} ;
```

이동 생성자의 호출 상황

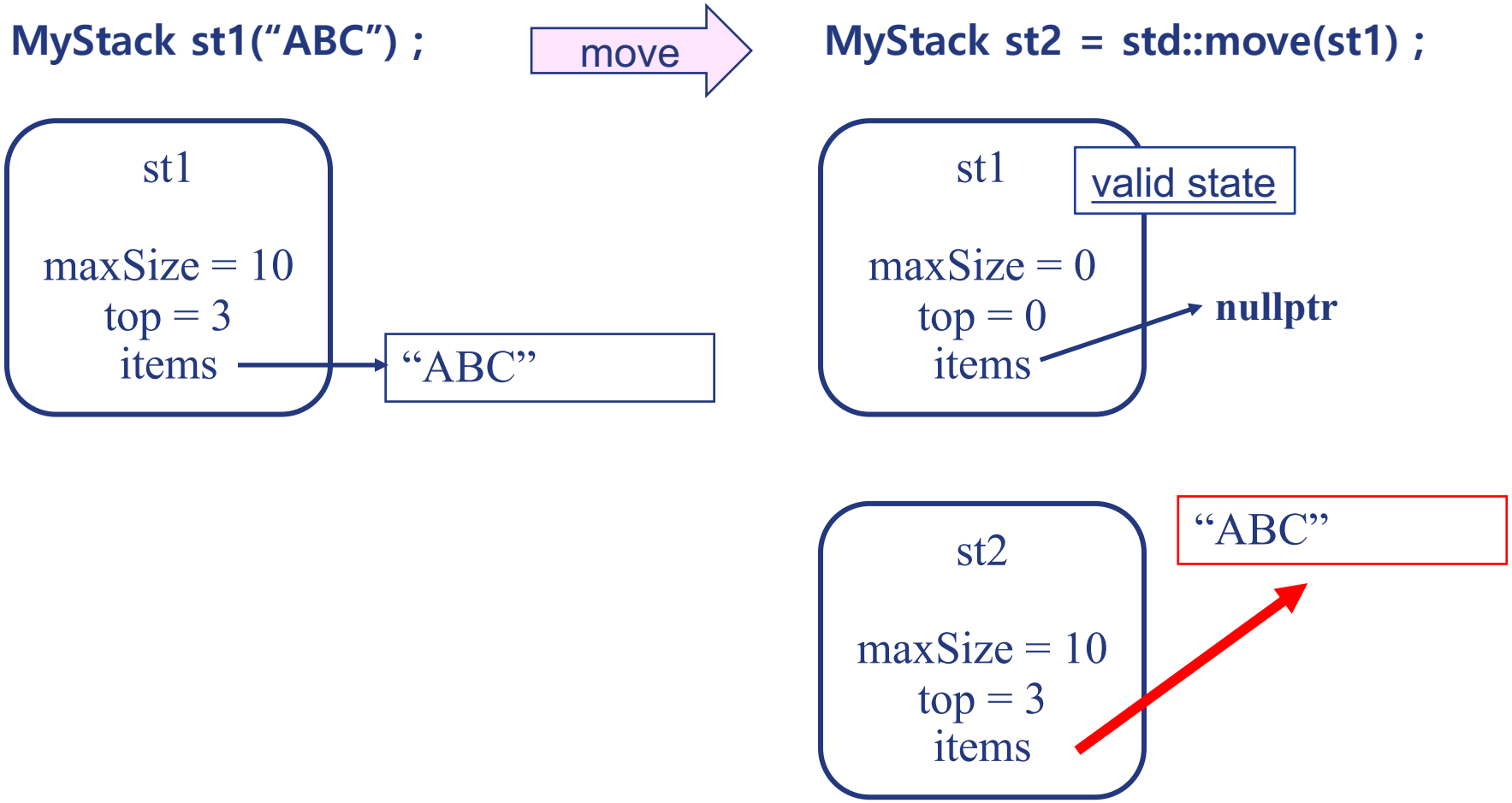
```
Point readPoint() {
    int x, y ;
    cin >> x >> y ;
    return Point(x, y) ;
}
void print(const Point pt) {
    cout << pt.getX() << ", " << pt.getY() << endl ;
}

int main() {
    Point pt1 ;
    // copy elision,  return value optimization by compiler automatically
    pt1 = readPoint() ; //so, try std::move(readPoint())
    print( Point {10, 20} ); //so, try std::move(Point {10, 20})
}
```

class MyStack

```
const int DEFAULT_MAX_SIZE = 10 ;
class MyStack {
    char* const items = nullptr;
    int top =0;
    const int maxSize ;
public:
    …
   MyStack(MyStack&& stack) noexcept
              : items {std::move(stack.items)}, top {std::move(stack.top)},   //move
                maxSize {std::move(stack.maxSize)} {
        if (this != &stack) {
            stack.items = nullptr; stack.top = 0; stack.maxSize = 0;          //clear rhs
        }
   }
   MyStack& operator=(MyStack&& stack) noexcept {
        if (this == &stack) return *this;
        delete [] items ; items = nullptr;                                    //clear myself
        items = std::move(stack.items);                                       //move
        top = std::move(stack.top); maxSize = std::move(stack.maxSize);
        stack.items = nullptr;  stack.top = 0; stack.maxSize = 0;             //clear rhs
        return *this;
   }
   // MyStack& operator=(MyStack&& stack) noexcept {          //move-and-swap idioms?
   //    if(this == &stack) return *this;
   //     MyStack tmp(std::move(stack)); swap(tmp);          //write your own swap using std::swap
   //     return *this;
   //}
} ;
```

Move


MyStack st1("ABC") ;
move
MyStack st2 = std::move(st1) ;
st1
maxSize = 10
top = 3
items
"ABC"
st1
valid state
maxSize = 0
top = 0
items
nullptr
st2
maxSize = 10
top = 3
items
"ABC"

Good Design : 이동 연산은 이동을 수행해야 하며, 원본은 유효한 상태로 남겨야 한다. (C.64)

- That is the generally assumed semantics.
- After y = std::move(x) the value of y should be the value x had and x should be in a valid state.
- Ideally, that moved-from should be the default value of the type.
- 표준(standard)은 이동된 객체(moved-from objects)는 소멸될 수 있어야 된다는 것만 요구함
  - The standard library assumes that it is possible to assign to a moved-from object (x = std::move(y); y = z;)

```
class X {   // OK: value semantics
    public:
        X();
        X(X&& a) noexcept;  // X를 이동한다
        void modify();      // X의 값을 변경한다
        ~X() { delete[] p; }
    private:
        int* p;
        int sz;
};
```

```
X::X(X&& a) :p{a.p}, sz{a.sz}  { // 값을 가져간다
        a.p = nullptr;     // empty 상태가 된다
        a.sz = 0;
}
void use() {
        X x1{};
        // ...
        X y1 = std::move(x1);
        x1 = X{};   // OK
} // OK: x 는 소멸 가능하다
```

Good Design :
이동 연산은 자기 대입에 안전하게 하라 (C.65)

- If x = x changes the value of x, people will be surprised and bad errors can occur.
- people don't usually directly write a self-assignment that turn into a move
- In the case of self-assignment, a move assignment operator should not leave the object holding pointer members that have been deleted or set to nullptr.

```
class Foo {
    string s;
    int i;
public:
    Foo& operator=(Foo&& a);
    // ...
};
```

```
// OK, but there is a cost
Foo& Foo::operator=(Foo&& a) noexcept {
    if (this == &a) return *this;  // this line is redundant
    s = std::move(a.s);
    i = a.i;
    return *this;
}
```

Good Design :
이동 연산은 noexcept로 만들라 (C.66)

- 예외를 던지는 이동 연산은 대부분의 사람들의 합리적 가정에 위배됩니다.
- 예외를 던지지 않는 이동은 표준 라이브러리와 언어 특징들에서 더 효율적으로 사용될 것이다.

```
class Vector {
public:
    Vector(Vector&& a) noexcept :elem{a.elem}, sz{a.sz} { a.sz = 0; a.elem = nullptr; }
    Vector& operator=(Vector&& a) noexcept {
        elem = a.elem; sz = a.sz; a.sz = 0; a.elem = nullptr;
    }
    // ...
private:
    T* elem;
    int sz;
};
```