Good Design :
copy and move

- 값 타입(value types)은 일반적으로 복사 가능해야 함
  - 항상 복사 생성자와 복사 할당 연산자가 있어야 함
- 값 타입은 내용(contents)에 관한 것으로 값 타입이 복사되면 항상 두 개의 독립적인 값이 생기고 각자 수정될 수 있어야 함

- 예외
  - 클래스 계층구조의 인터페이스는 그렇지 않아야 함
  - 리소스 핸들(파일, 소켓 등)은 복사 가능할 수도 있고 그렇지 않을 수도 있음
  - 논리적, 성능상의 이유로 이동(move)하도록 타입이 정의될 수 있음 (unique_ptr 등)

복사 생성자(Copy 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& operator=(const Point& pt) {     // 복사 할당 연산자
       x = pt.x; y = pt.y;
      return *this;
   }
   int getX() const { return x ; }
   int getY() const { return y ; }
} ;
```

복사 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, pt2;
    pt1 = pt2 ;          // 1)
    pt1 = readPoint() // 2)
    Point pt3(pt1) ;   // 3)
    print(pt3) ;         // 4)
}
```

기본 복사 생성자

- 복사 생성자가 명시되지 않았다면 컴파일러가 복사 생성자를 자동으로 생성한다.
- 자동으로 생성된 복사 생성자는 데이터 멤버 별로 대입문을 실행한다.

```
// 자동으로 생성된 복사 생성자
Point(const Point& pt) { x = pt.x;  y = pt.y; };
```

Good Design : copyable (value type) class has a default constructor (C.43)

- 많은 표준 라이브러리 컨테이너나 언어 문법이 그들의 요소(elements)를 초기화 하기 위해 기본 생성자를 사용
  - 예를 들어, T arr[10], std::vector<T> vec(10)
- 모든 멤버가 기본 생성자를 갖는 클래스는 묵시적으로 기본 생성자를 가진다.

```
struct X {
        string s;
        vector<int> v;
    };
    X x; // means X{{}, {}}; that is the empty string and the empty vector
```

- built-in 타입은 적절하게 기본 생성되지 않을 수 있음

```
struct X {
        string s;
        int i;
    };
```

```
void f() {
    X x;    // x.s is initialized to the empty string; x.i is uninitialized
    cout << x.s << ' ' << x.i << '\n';
}
```

class MyStack

```
const int DEFAULT_MAX_SIZE = 10 ;
class MyStack {
    char* const items ;
    int top ;
    const int maxSize ;
public:
    MyStack(int _maxSize=DEFAULT_MAX_SIZE)
        : maxSize(_maxSize), items(new char[_maxSize]) {
        top = 0 ;
    }
    MyStack(const char* const str) : maxSize(strlen(str)+DEFAULT_MAX_SIZE),
        items(new char[strlen(str)+DEFAULT_MAX_SIZE]) {
        for ( int i = 0 ; i < strlen(str) ; i ++ ) items[i] = str[i] ;
        top = strlen(str) ;
    }
    void push(char c) { items[top++] = c; }
    char pop() { return items[--top]; }
    void print() const {
        for ( int i=0; i < top; i++ ) cout << items[i];
        cout << endl;
    }
    ~MyStack() noexcept { delete [] items; }
} ;
```

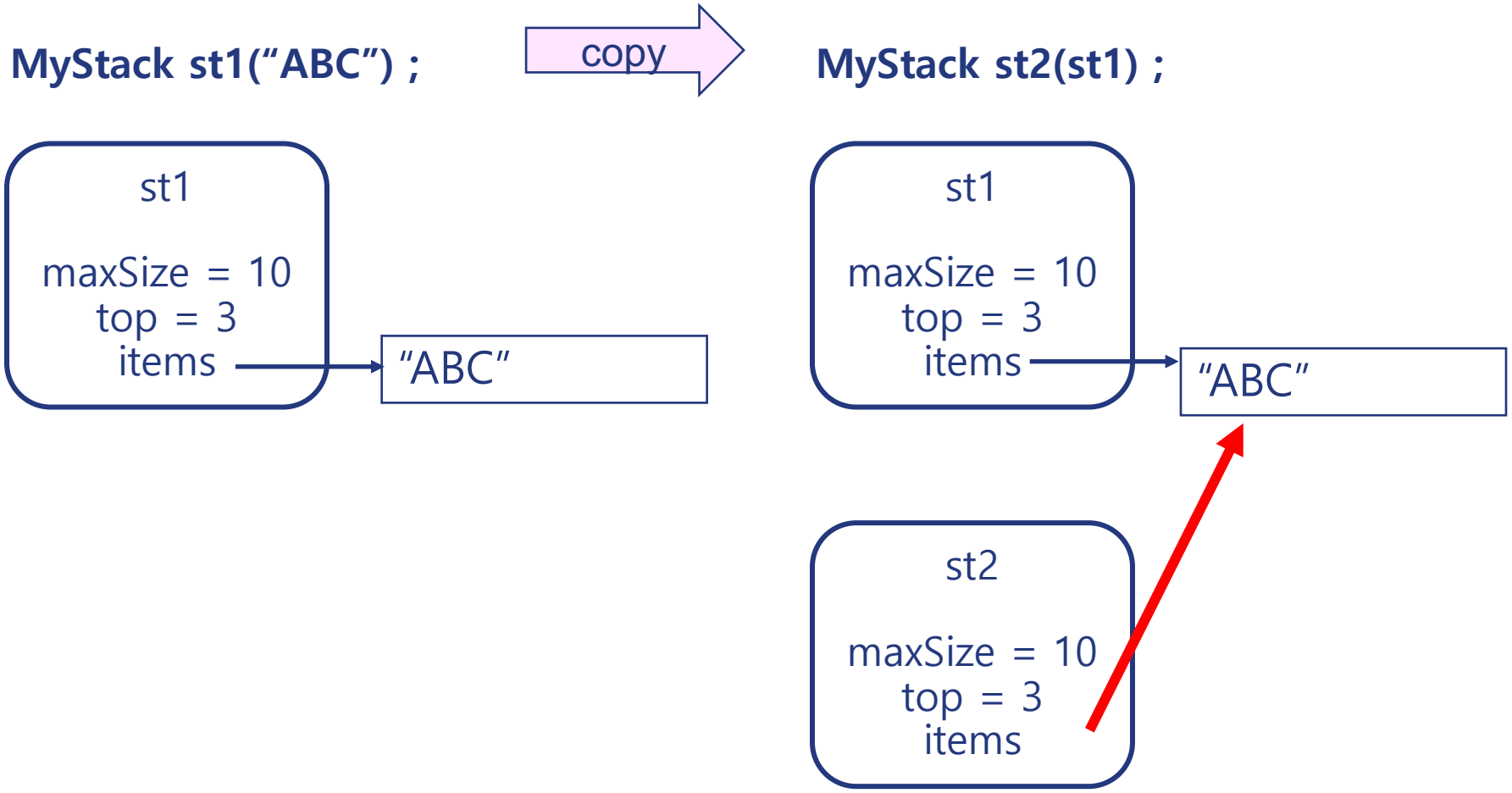
기본 복사 생성자의 문제점

```
int main() {
    MyStack st1("ABC") ;
    MyStack st2(st1) ; // default copy constructor invoked

    st2.pop();
    st2.push('D') ;
    st2.print();     // ABD

    st1.print();     // ABD, not ABC
}
```

Shallow Copy



Solution - Copy Constructor

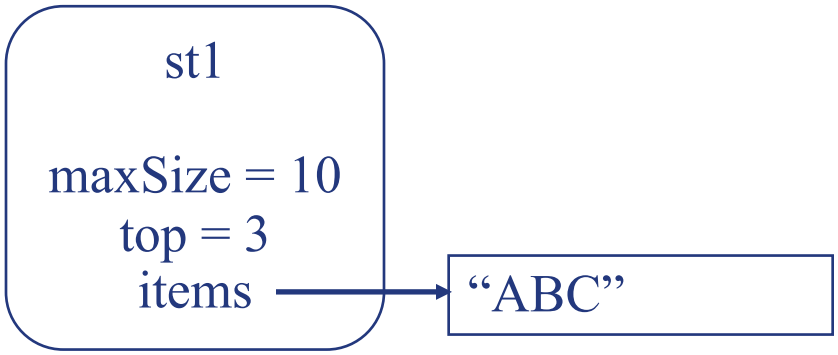
- Default action of assignment: memberwise copy
- The Problem: shallow copy
- The Solution: copy constructor for deep copying
- The Copy Constructor for class X
  - X(const X&)

class MyStack

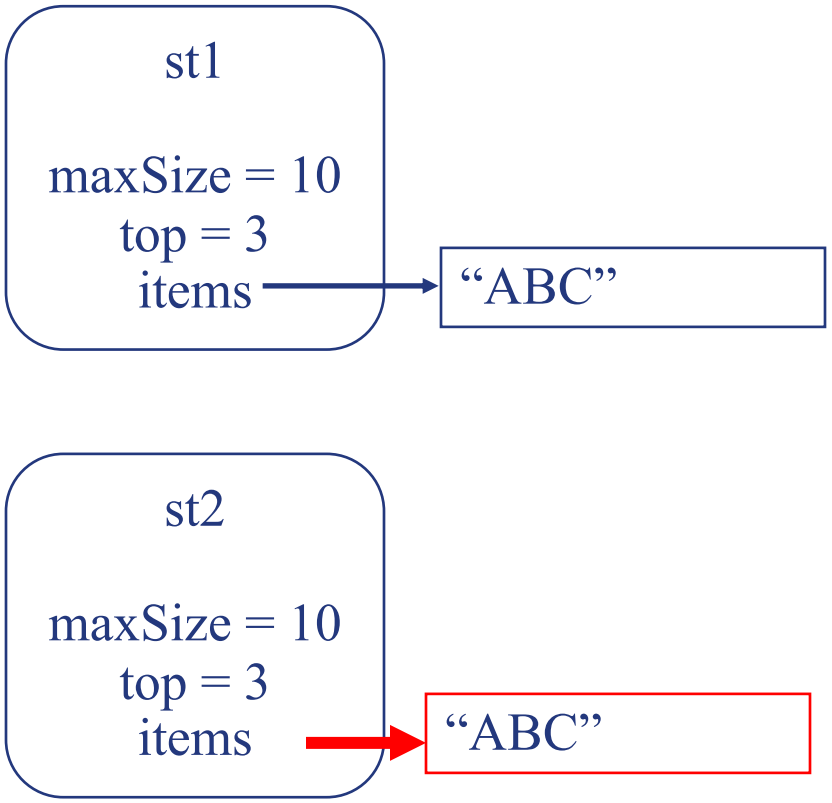
```
const int DEFAULT_MAX_SIZE = 10 ;
class MyStack {
    char* const items ;
    int top ;
    const int maxSize ;
public:
    MyStack(int _maxSize=DEFAULT_MAX_SIZE)
        : maxSize(_maxSize), items(new char[_maxSize]) {
        top = 0 ;
    }
    MyStack(const char* const str) : maxSize(strlen(str)+DEFAULT_MAX_SIZE),
        items(new char[strlen(str)+DEFAULT_MAX_SIZE]) {
        for ( int i = 0 ; i < strlen(str) ; i ++ ) items[i] = str[i] ;
        top = strlen(str) ;
    }
    MyStack(const MyStack& another) : items(new char[another.maxSize]),
                                top(another.top), maxSize(another.maxSize) {
        for ( int i=0; i < top; i++ ) items[i] = another.items[i];
    }
    //MyStack& operator=(const MyStack& stack) { //copy-and-swap idioms
    //        MyStack tmp(stack); swap(tmp);            //write your own swap using std::swap
    //        return *this;
    //}
} ;
```

Deep Copy

MyStack st1("ABC") ;



MyStack st2(st1) ;



Good Design :
복사 연산은 복사를 수행해야 한다 (C.61)

- 복사 연산은 복사를 수행해야 한다
  - After x = y, we should have x == y
  - After a copy x and y can be independent objects (value semantics, the way non-pointer built-in types and the standard-library types work)
  - or refer to a shared object (pointer semantics, the way pointers work)

```
class X {   // OK: value semantics
public:
    X();
    X(const X&);     // copy X
    void modify();   // change the value of X
    // ...
    ~X() { delete[] p; }
private:
    T* p;
    int sz;
};
```

```
bool operator==(const X& a, const X& b) {
    return a.sz == b.sz && equal(a.p, a.p +
a.sz, b.p, b.p + b.sz);
}
X::X(const X& a) :p{new T[a.sz]}, sz{a.sz} {
    copy(a.p, a.p + sz, p);
}
X x; X y = x;
if (x != y) throw Bad{};
x.modify();
// assume value semantics
if (x == y) throw Bad{};
```

Good Design :
복사 연산은 자기 대입에 안전하게 하라 (C.62)

- If x = x changes the value of x, people will be surprised and bad errors will occur (often including leaks)
- The standard-library containers handle self-assignment elegantly and efficiently

```
std::vector<int> v = {3, 1, 4, 1, 5, 9};
v = v;
// the value of v is still {3, 1, 4, 1, 5, 9}
```

- The default assignment generated from members that handle self-assignment correctly handles self-assignment

```
struct Bar {
    std::vector<std::pair<int, int>> v;
    std::map<std::string, int> m;
    std::string s;
};
Bar b;
b = b;   // correct and efficient
```