OPERATOR OVERLOADING RULE OF THREE

Problem Solving with Computers-II

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

GitHub

Will this code compile?

```
int main(){
  Complex p;
  p.conjugate();
  p.print();
}
```

Calls default constructor

call to destructor

```
class Complex
{
private:
    double real;
    double imag;
public:
    double getMagnitude() const;
    double getReal() const;
    double getImaginary() const;
    void print() const;
    void conjugate();
    void setReal(double r);
    void setImag(double r);
};
```

[✓ A. Yes]
B. No
C. I am not sure . . .

Will this code compile?

```
int main(){
  Complex p;
  Complex w(1, 2);
  p = w;
  p.conjugate();
  p.print();
}
```

calls parametrized constructor

```
class Complex
{
private:
    double real;
    double imag;
public:
    Complex(double re, double im):
real(re), imag(im){}
    double getMagnitude() const;
    double getReal() const;
    double getImaginary() const;
    void print() const;
    void conjugate();
    void setReal(double r);
    void setImag(double r);
};
```

1 2

A. Yes

[✓ B. No]

C. I am not sure . . .

because default constructor is no longer automatically generated by the compiler

Will this code compile?

```
int main(){
  Complex p;
  Complex w(1, 2);
  p = w;
  p.conjugate();
  p.print();
}
```

The constructor on the right works for both calls

```
class Complex
{
private:
    double real;
    double imag;
public:
    Complex(double re = 0, double im = 0):
real(re), imag(im){}
    double getMagnitude() const;
    double getReal() const;
    double getImaginary() const;
    void print() const;
    void conjugate();
    void setReal(double r);
    void setImag(double r);
};
```

default values for parameters

[✓ A. Yes]
B. No
C. I am not sure . . .

Operator Overloading

We would like to be able to perform operations on two objects of the class using the following operators:

<<

==

!=

+

-

and possibly others

Handout activity 1A

lhs operand   operator   rhs operand

```
cout << w;
```

ostream   complex

← This is actually a call to a function

Select the equivalent function call:

rhs   lhs

```
w.operator<<(cout);
```

~~A~~

member function of class ostream

lhs   rhs

```
cout.operator<<(w);
```

B

lhs   rhs

```
operator<<(cout, w);
```

C

free function

Overloading the << operator

```
int main(){
  Complex w(10, -5);
  w.conjugate();
  w.print();
}
```

```
int main(){
  Complex w(10, -5);
  w.conjugate();
  cout << w;
}
```

```
Before overloading the << operator
```

```
After overloading the << operator
```

```
operator<<(cout, w);
```

Select the function declaration that matches the above call

[✓ A]

```
void operator<<(ostream &out,
                const Complex &c);
```

rhs

B

```
void Complex::operator<<(ostream &out);
```

Overloading the + operator

```
p = q + w;
```

Goal: We want to apply the + operator to Complex type objects

New method: add()

```
int main(){
  Complex p;
  Complex q(2, 3);
  Complex w(10, -5);
  p = _______________;
  p.print();
}
```

```
int main(){
  Complex p;
  Complex q(2, 3);
  Complex w(10, -5);
  p = _______________;
  p.print()
}
```

Approach 1

Approach 2

New method: add()

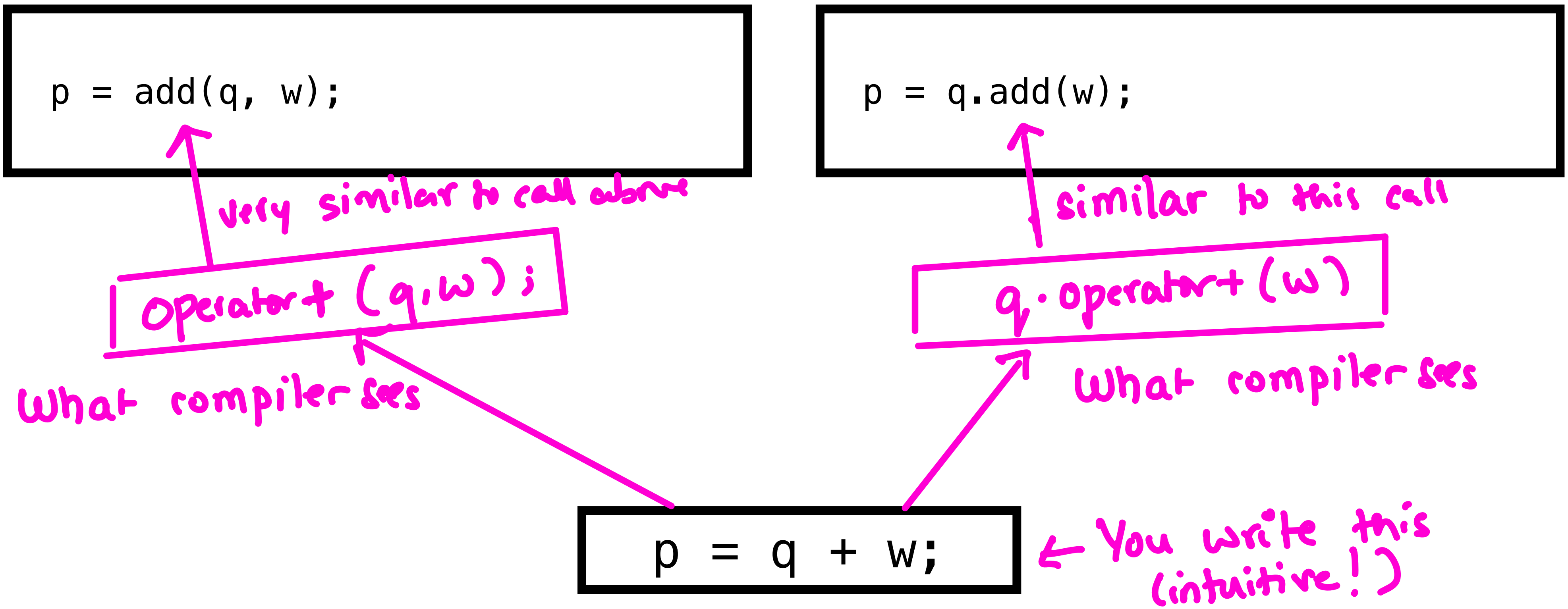
```
int main(){
  Complex p;
  Complex q(2, 3);
  Complex w(10, -5);
  p = add(q, w);
  p.print();
}
```

```
int main(){
  Complex p;
  Complex q(2, 3);
  Complex w(10, -5);
  p = q.add(w);
  p.print()
}
```

Approach 1

Approach 2

Overloading the + operator for Complex objects



```
p = add(q, w);
```

very similar to call above

operator+(q,w);

What compiler sees

```
p = q.add(w);
```

similar to this call

q.operator+(w)

What compiler sees

```
p = q + w;
```

← You write this (intuitive!)

Goal: We want to apply the + operator to Complex type objects

Handout Activity 1B:

Implement operator+ for Complex objects as a non-member function

```
Complex operator+(const Complex& lhs, const Complex& rhs) {
    // YOUR CODE:




}
```

Overloading Operators for IntList

In lab01 you will overload operators for the IntList ADT

==

!=

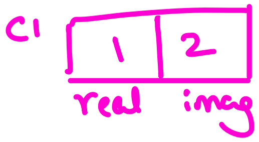
+ (list concatenation)

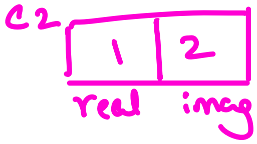
<< (overloaded stream operation to print the sequence)

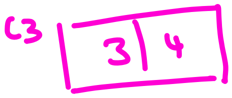
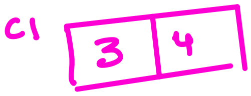
RULE OF THREE

If a class defines one (or more) of the following it should probably explicitly define all three:

1. Destructor → free any dynamic (heap) memory
2. Copy constructor → initialize a new object using an existing one
3. Copy assignment → copy the data of one object into another (both objects already exist in memory)

Code	Function called	Memory Diagram
Complex c1(1,2);	Parameterized constructor	

Complex c2 = c1;
oR
Complex c2(c1);
	Copy constructor	

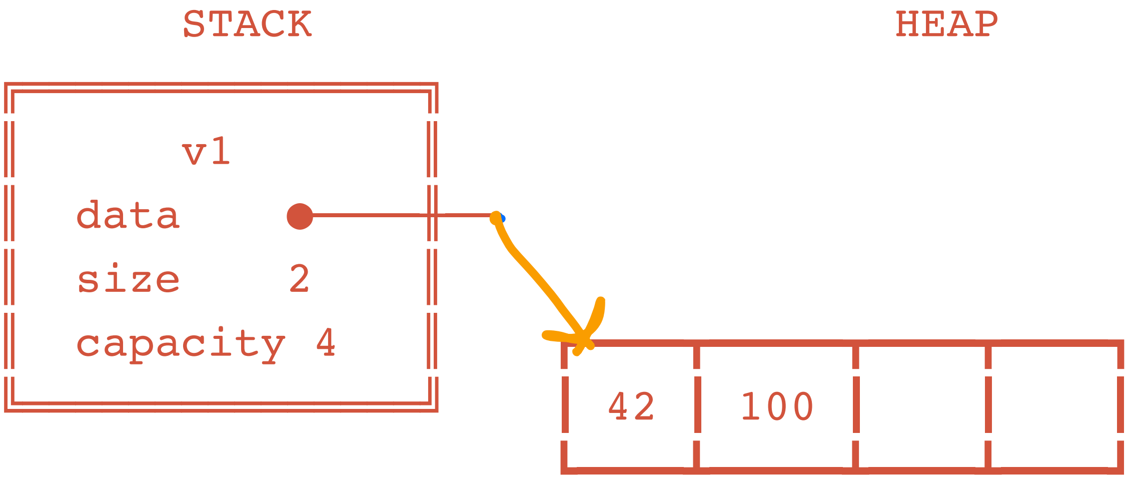
Complex c3 (3,4);	Parameterized constructor	
c1 = c3;	Copy assignment	

When an ADT does not use any dynamic memory default copy constructor & copy assignment operator works just fine!

THE CODE:

```
CustomVector v1;
v1.push_back(42);
v1.push_back(100);
```

MEMORY AFTER v1.push_back(100):



THE PROBLEM: SHALLOW COPY WITH DYNAMIC MEMORY

THE CODE:

```
CustomVector v1;
v1.push_back(42);
v1.push_back(100);
CustomVector v2 = v1;
```

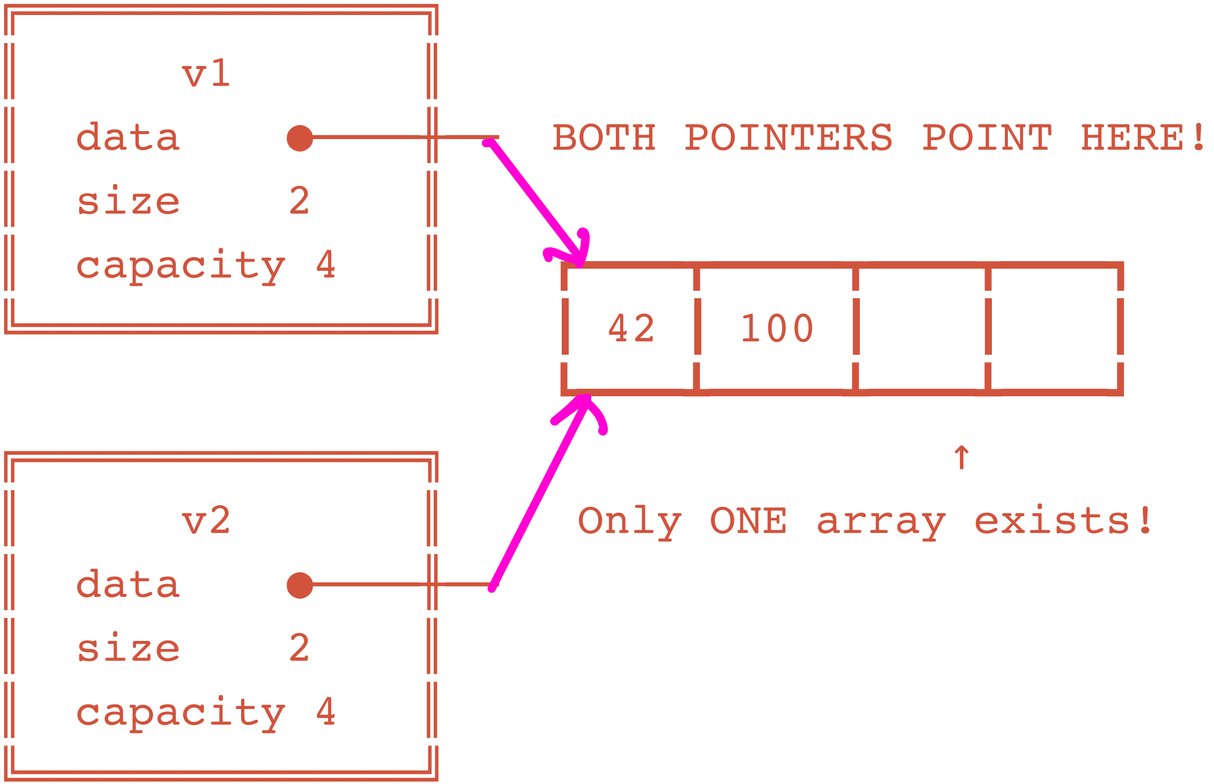
Default copy = SHALLOW
Copies pointers,
NOT the data!

CustomVector v2=v1

MEMORY AFTER a v2 = v1:

STACK

HEAP



v1
data ●
size 2
capacity 4

BOTH POINTERS POINT HERE!

42	100		

↑
Only ONE array exists!

v2
data ●
size 2
capacity 4

THE PROBLEM: SHALLOW COPY WITH DYNAMIC MEMORY

WHEN BOTH GO OUT OF SCOPE:

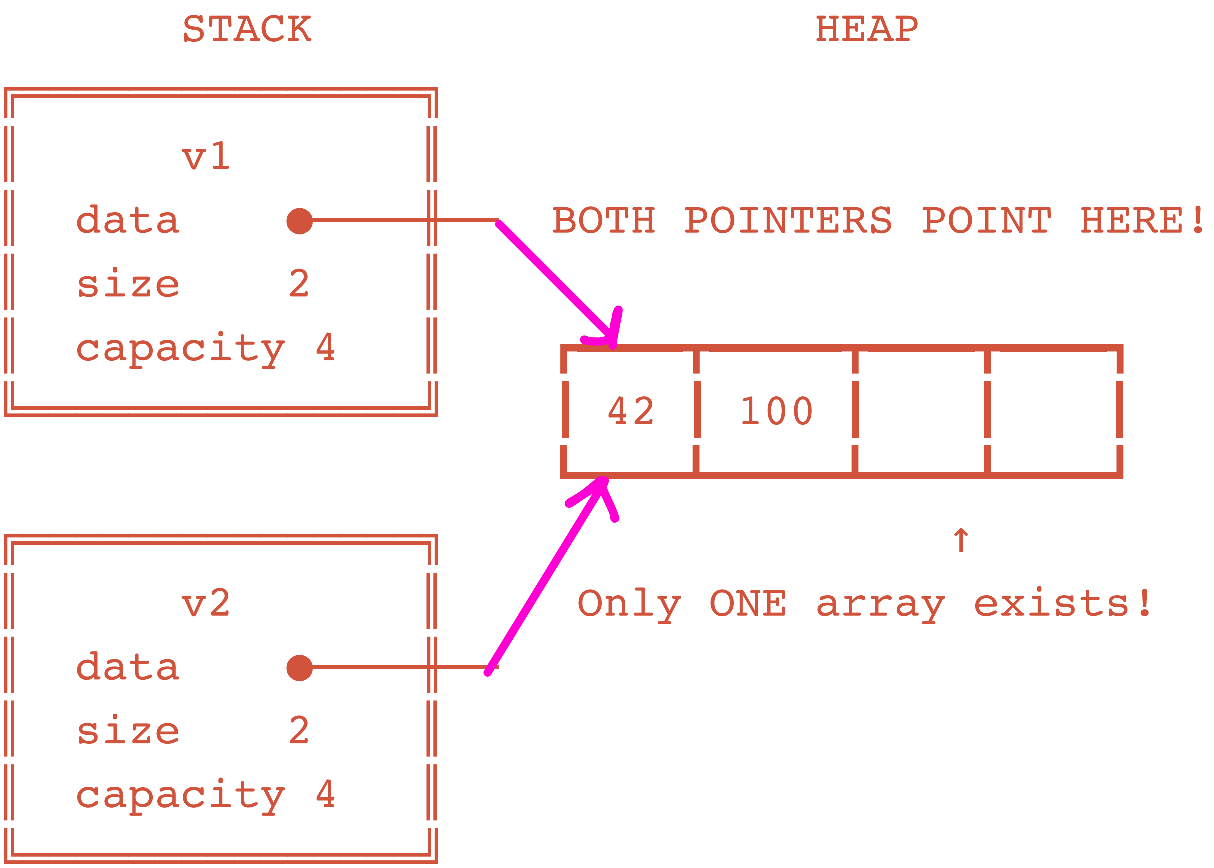
Step 1: v2's destructor runs → delete[] data; ✓ Frees the array

Step 2: v1's destructor runs → delete[] data; ✗ CRASH!

Already freed!

⚠ DOUBLE DELETION = UNDEFINED BEHAVIOR (crash or corruption)

MEMORY AFTER CustomVector v2 = v1:



STACK

HEAP

v1

data ●

size 2

capacity 4

BOTH POINTERS POINT HERE!

42	100		

↑

Only ONE array exists!

v2

data ●

size 2

capacity 4

THE PROBLEM: SHALLOW COPY WITH DYNAMIC MEMORY

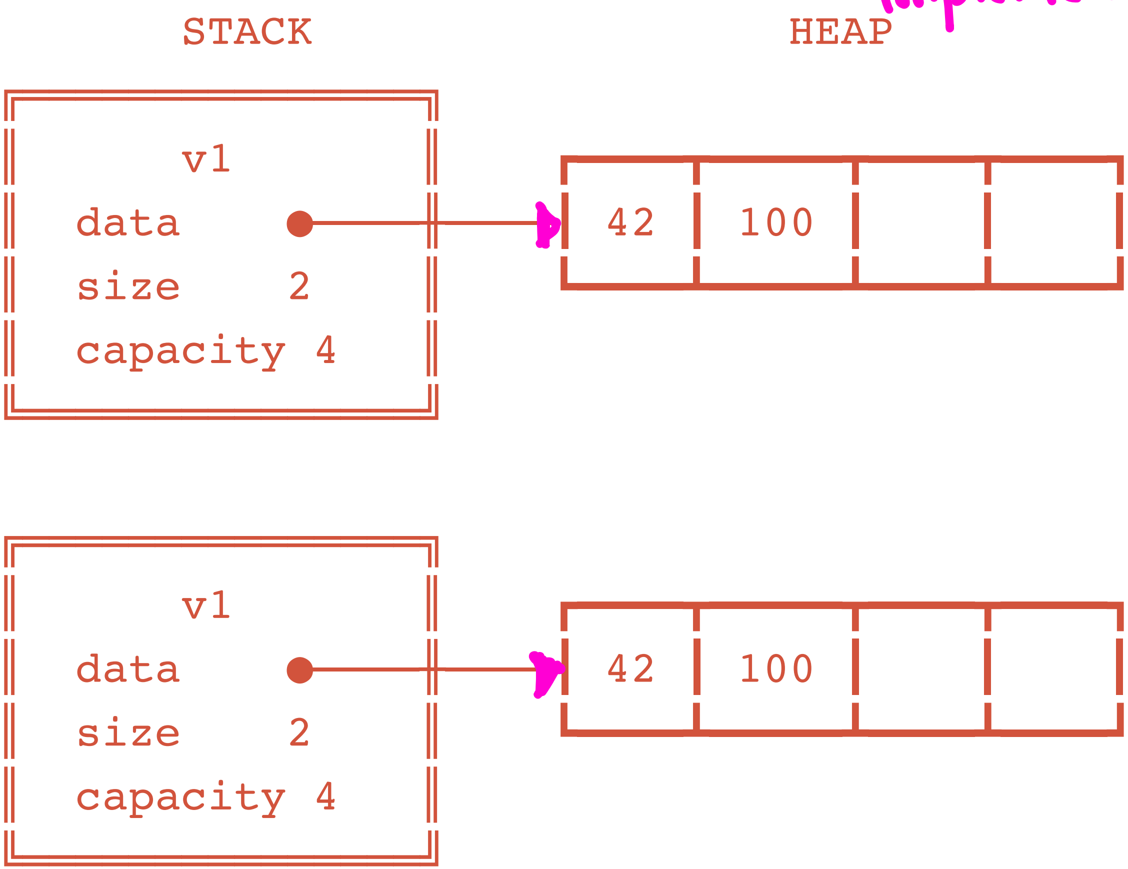
THE SOLUTION:

THE BIG THREE:

1. Destructor
2. Copy Constructor
3. Copy Assignment

(Deep copy needed!)

MEMORY AFTER CustomVector v2 = v1: (If copy constructor is correctly implemented)



Handout Activity Part 3:

Now apply the Rule of Three to CustomList!

This is in preparation for the upcoming lab