

OPERATOR OVERLOADING (CONTD.)

DYNAMIC RESOURCE MANAGEMENT

Problem Solving with Computers-II

C++

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

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



Today's goals

- Operator overloading (contd)
- Dynamic memory and common errors
- We want to understand the what, why, and how of the C++ Big Three:
 - Destructor
 - Copy constructor
 - Copy assignment operator

Overloading the + operator for Complex objects

```
z = x + y + w;
```

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

Overloading the << operator

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

Before overloading the << operator

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

After overloading the << operator

```
cout << w;
```

Select any equivalent C++ statement:

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

A

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

B

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

C

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

Select the function declaration that does NOT match the above call

A

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

B

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

C

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

Operator Overloading

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

<<

==

!=

+

-

and possibly others

Dynamic Memory: common errors

- Memory Leak: Program does not free memory allocated on the heap.
- Segmentation Fault: Code tries to access an invalid memory location

C++Big Four: Special functions of any C++ class

- Constructor
- Destructor
- Copy constructor
- Copy assignment operator

The compiler automatically generates default versions for all of these, but you can provide user-defined implementations.

RULE OF THREE

If a class uses dynamic memory, you usually need to provide your implementation of the destructor. If a class implements one (or more) of the following it should probably implement all three:

1. Destructor
 2. Copy constructor
 3. Copy assignment
- What is the behavior of these defaults?
 - What is the desired behavior ?
 - How should we over-ride these methods?

```
void test_0(){  
    IntList x;  
    x.push_front(10);  
    x.print();  
}
```

Assume:

- * **Default destructor**
- * **Default copy constructor**
- * **Default copy assignment**

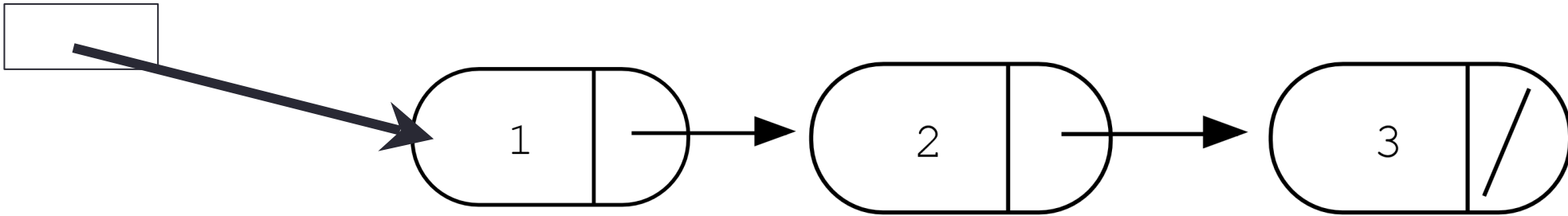
What is the result of running the above code?

- A. Compiler error
- B. Memory leak
- C. Segmentation fault
- D. None of the above

Concept Question

```
IntList::~~IntList(){  
    delete head;  
}
```

head



```
class Node {  
    public:  
        int data;  
        Node *next;  
};
```

Which of the following objects are deleted when the destructor of IntList is called?

(A): head pointer

(B): only the first node

(C): A and B

(D): All the nodes of the linked list

(E): A and D

Copy constructor

- Parameterized constructor whose first argument is a class object
- **initializes a (new) object using an existing object**

In which of the following cases is the copy constructor called?

A. `IntList x;`
`IntList y;`

B. `Complex(1, 2);`
`Complex p2(p1);`

C. `Complex* p1 = new Complex(1, 2);`

D. B & C

E. A, B & C

Behavior of default copy constructor

```
void test_copy_constructor() {  
    IntList x;  
    x.push_front(10);  
    x.push_front(20);  
    IntList y(x);  
    // calls the copy c'tor  
    x.clear();  
    y.print();  
}
```

Assume:

destructor: user-defined

copy constructor: default

copy assignment: default

What is the output?

A. No output

B. 20, 10

C. Segmentation fault

Copy assignment (operator=)

- For existing objects x, y, this statement calls the operator= function:

`x = y;`

- Default behavior: Copies the member variables of rhs object (y) to lhs object (x)

```
Complex x(1, 2);
```

```
Complex y;
```

```
y = x;
```

```
cout << y;
```


Behavior of default copy assignment

`x : 1 -> 2 -> 5 -> null`

```
void default_assignment_1(IntList& x){  
    IntList y;  
    y = x;  
}
```

* What is the behavior of the default assignment operator?

Assume:

- * **User-defined** destructor
- * **Default copy constructor**
- * **Default copy assignment**

Behavior of default copy assignment

```
void test_default_assignment_2() {  
    IntList x, y;  
    x.push_front(10);  
    x.push_front(20)  
    y = x;  
    y.print()  
}
```

What is the result of running the above code?

- A. Prints 20, 10
- B. Segmentation fault
- C. Memory leak
- D. A & B
- E. A, B and C

Assume:

- * **User-defined** destructor
- * **Default copy constructor**
- * **Default copy assignment**

Behavior of default copy assignment

```
void test_default_assignment_3(){  
    IntList x;  
    x.push_front(10);  
    x.push_front(20)  
    IntList y(x);  
    y.push_front(30);  
    y.push_front(40);  
    y = x;  
    y.print()  
}
```

What is the result of running the above code?

- A. Prints 20, 10
- B. Segmentation fault
- C. Memory leak
- D. A & B
- E. A, B and C

Assume:

- * **User-defined** destructor
- * **User-defined** copy constructor
- * **Default** copy assignment

RULE OF THREE

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

1. Destructor
2. Copy constructor
3. Copy assignment