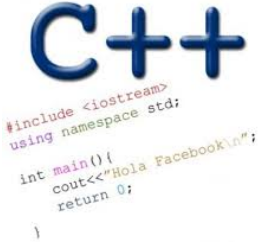
BIG FOUR AND THE RULE OF THREE LINKED LISTS

Problem Solving with Computers-II



The Big Four (review)

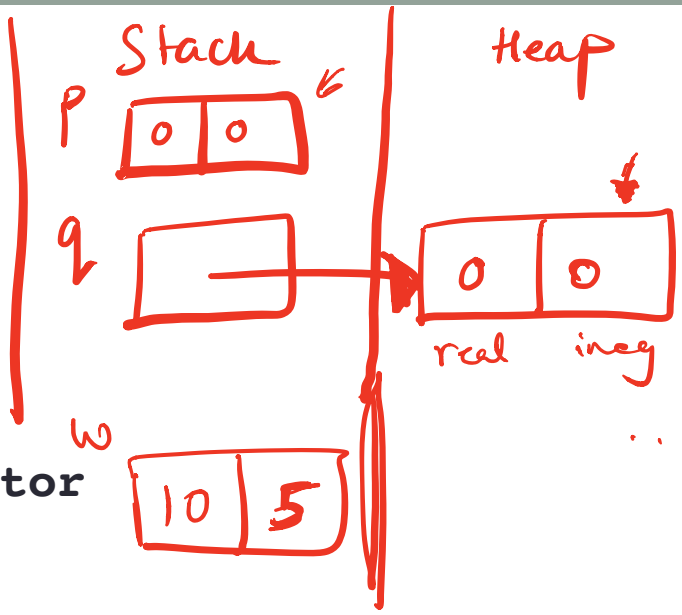
C++ provides default implementations

1. Constructor : Create a new object
2. Destructor : Called Right before an object is destroyed
3. Copy Constructor :
4. Copy Assignment : c1 = c2;

override: you write your version

Constructor (review)

```
void foo(){
    Complex p; // default constructor
    Complex* q = new Complex;
    Complex w{10, 5};
}
```

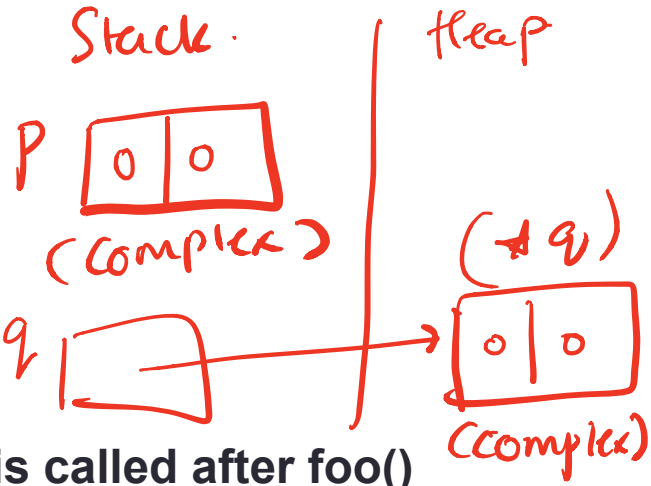


How many times is the constructor called in the above code?

A. Never
B. Once
C. Two times
D. Three times
E. Four times

Destructor (review)

```
void foo(){
     Complex p;
     Complex *q = new Complex;
} //delete q; (destroys *q)
```



The destructor of which of the objects is called after foo() returns?

A. p ← (Stack)
B. q
C. *q
D. None of the above
E. Both p and *q

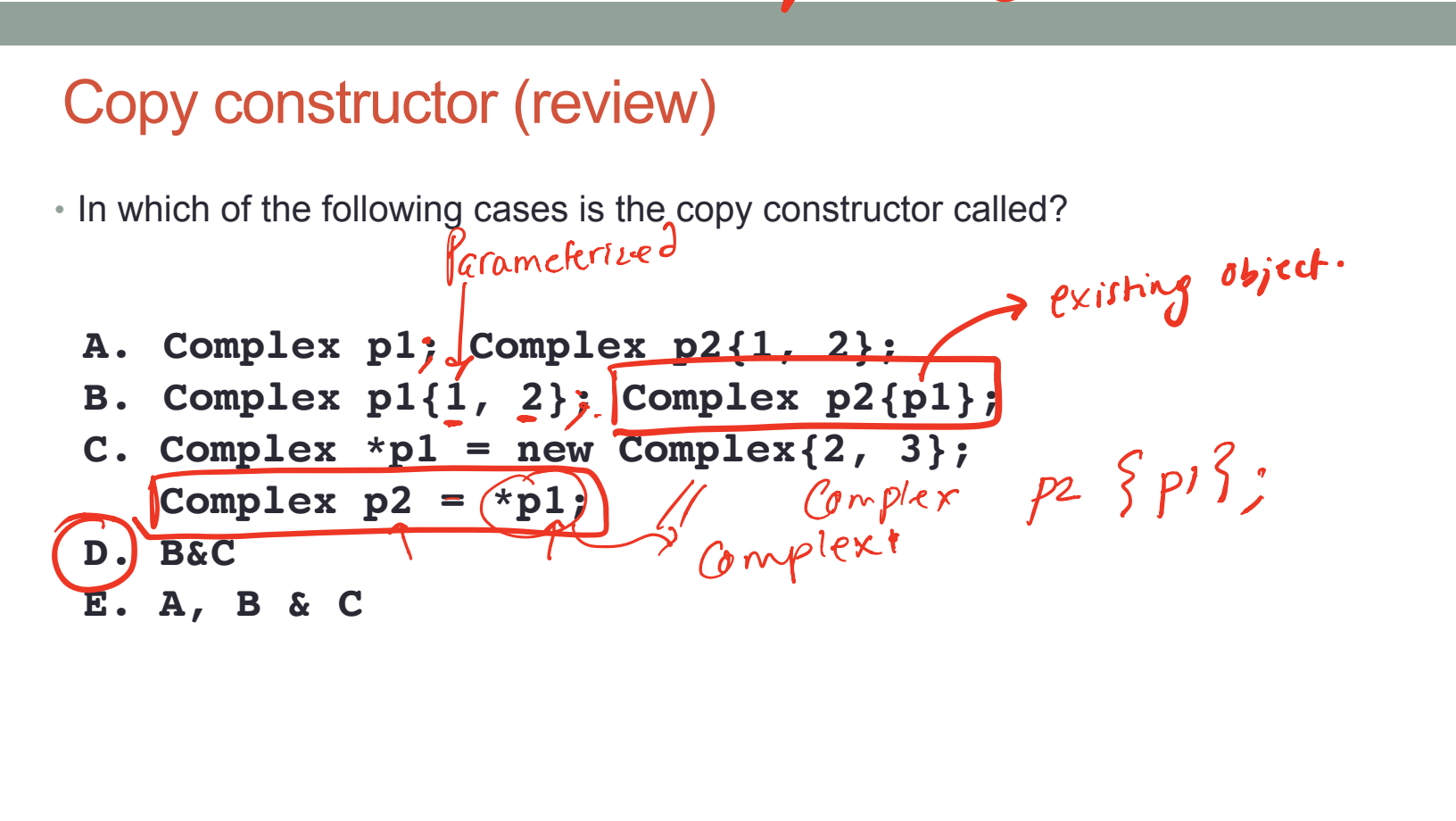
(1) When foo returns the stack objects are destroyed
p, q

(2) p is a type Complex
q is a type Complex*

Copy constructor (review)

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

```
A. Complex p1; Complex p2{1, 2};
B. Complex p1{1, 2}; Complex p2{p1};
C. Complex *p1 = new Complex{2, 3};
   Complex p2 = *p1;
D. B&C
E. A, B & C
```



```
double foo(Complex p){
    return p.conjugate(10);
}
int main(){
     Complex q{1, 2};
     foo(q); // Pass parameter q (Pass by value)
}
```

Which of the following special methods is called when passing parameters to foo()?

A. Parameterized constructor

B. Copy constructor

C. Copy assignment

D. Destructor

Linked Lists

Array List

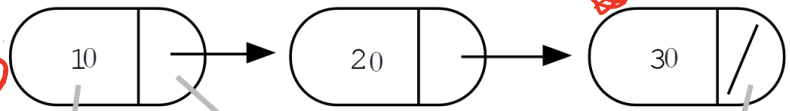
Stack

Heap

head tail

The overall list is built by connecting the nodes together by their next pointers. The nodes are all allocated in the heap.

Linked List



Node

Each node stores one data element (int in this example).

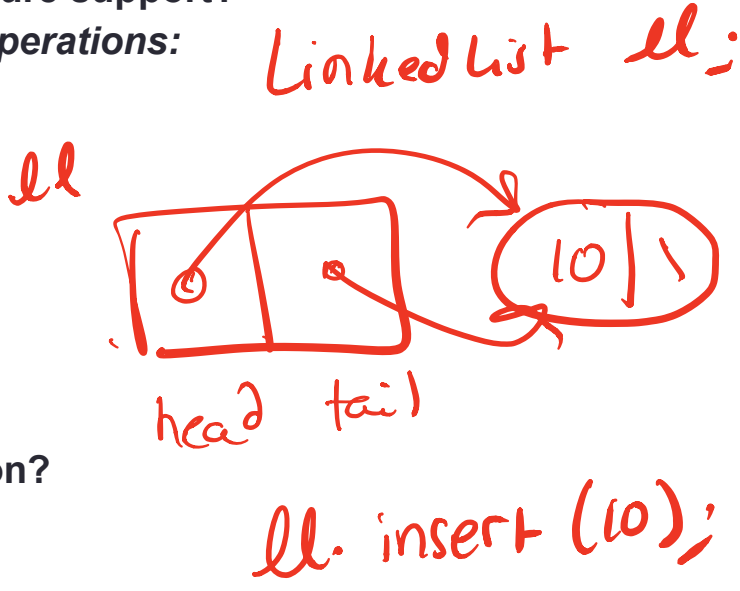
Each node stores one next pointer.

The next field of the last node is NULL.

What is the key difference between the two?

Questions you must ask about any data structure:

- **What operations does the data structure support?**
  A linked list supports the following operations:
  1. Insert (a value to the head)
  2. Append (a value to the tail)
  3. Delete (a value)
  4. Search (for a value)
  5. Min
  6. Max
  7. Print all values
- **How do you implement each operation?**
- **How fast is each operation?**

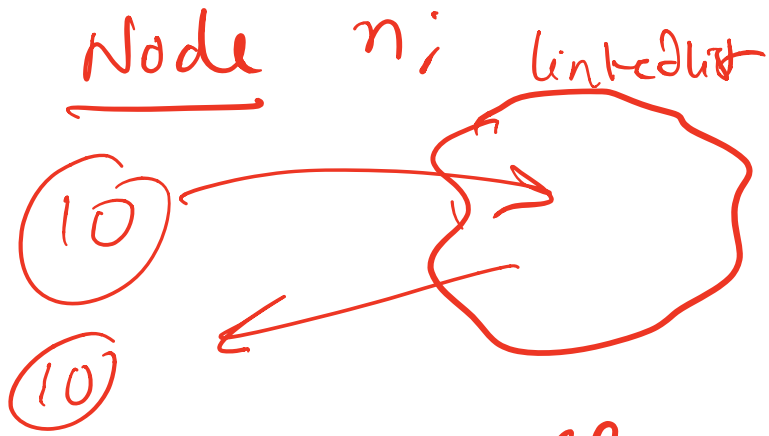


Linked-list as an Abstract Data Type (ADT)

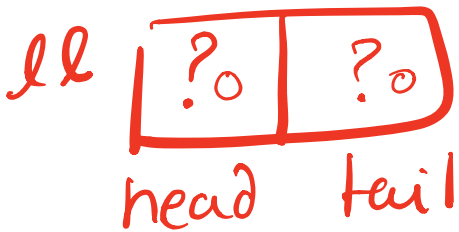
```
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    // other public methods
    void append(int value);

private:
    struct Node {
        int info;
        Node* next;
    };
    Node* head;
    Node* tail;
};
```

constructor

destructor



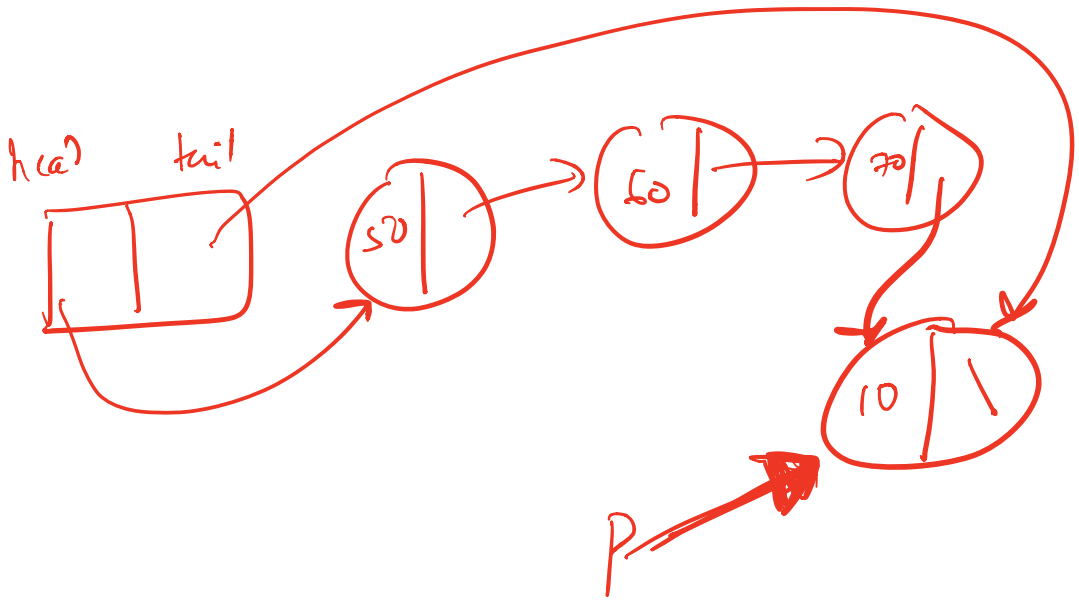
LinkedList ll;



① If we were to use the default constructor that c++ provides then, head & tail will have junk values

```
LinkedList :: LinkedList ( ):
        head {0},
        tail {0}
{}
```

Initializer list



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

The questions we ask are:

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

```
void test_append_0(){
      LinkedList ll;
      ll.append(10);
       ll.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. Prints 10
D. None of the above

Behavior of default copy constructor

l1 : 1 -> 2- > 5 -> null

```
void test_default_copy_constructor(LinkedList& l1){
    // Use the copy constructor to create a copy of l1




}
```

* What is the default behavior?
* Is the default behavior correct ?
* How do we change it?

Assume:

* **Overloaded destructor**
* **Default copy constructor**
* **Default copy assignment**

Behavior of default copy assignment

l1 : 1 -> 2- > 5 -> null

```
void default_assignment_1(LinkedList& l1){
   LinkedList l2;
   l2 = l1;
}
```

* What is the default behavior?

Assume:

* **Overloaded destructor**
* **Default copy constructor**
* **Default copy assignment**

Behavior of default copy assignment

```
void test_default_assignment_2(){
    LinkedList l1, l2;
    l1.append(1);
    l1.append(2)
    l2.append(10);
    l2.append(20);
    l2 = l1;
    l2.print()
}
```

What is the result of running the above code?
A. Segmentation fault
B. Prints 1 , 2
C. Both A and B
D. None of the above

Assume:

* **Overloaded destructor**
* **Default copy constructor**
* **Default copy assignment**

Behavior of default copy assignment

```
void test_default_assignment_2(){
   LinkedList l1;
   l1.append(1);
   l1.append(2)
   LinkedList l2{l1};
   l2.append(10);
   l2.append(20);
   l2 = l1;
   l2.print()
}
```

What is the result of running the above code?
A. Segmentation fault
B. Memory leak
C. Both A and B
D. None of the above

Assume:

* **Overloaded destructor**
* **Overloaded copy constructor**
* **Default copy assignment**

Overloading Binary Comparison Operators

We would like to be able to compare two objects of the class using the following operators

==

!=

and possibly others

```
void isEqual(const LinkedList & lst1, const LinkedList &lst2){
    if(lst1 == lst2)
          cout<<“Lists are equal”<<endl;
     else
          cout<<“Lists are not equal”<<endl;

}
```

Next time

- Linked Lists contd.
- GDB