RECURSION ON LINKED LISTS
C++ RULE OF THREE

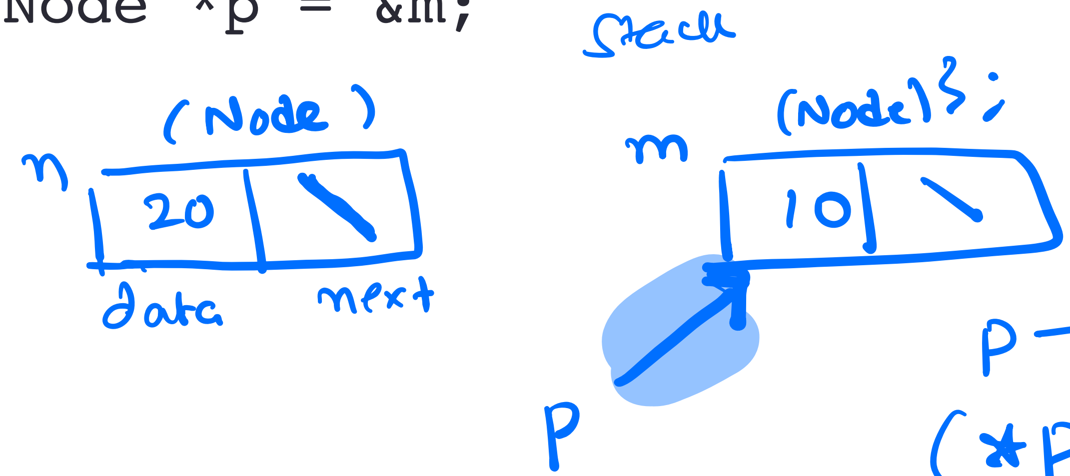
Problem Solving with Computers-II

Read the syllabus. Know what's required. Know how to get help.

Review: Accessing structs using pointers

```
Node n {20, nullptr};
Node m {10, nullptr};
Node *p = &m;
```

```
struct Node {
    int data;
    Node* next;
};
```

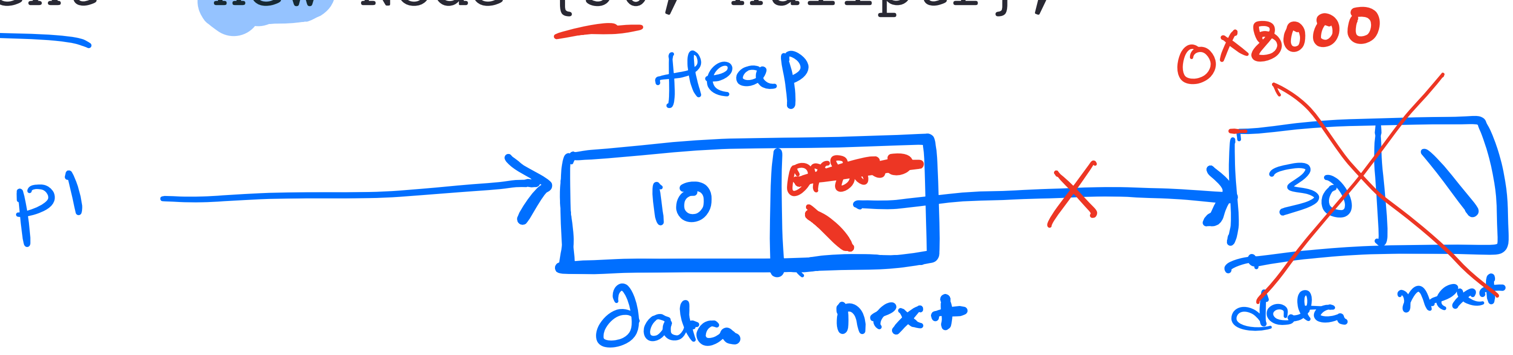



```
cout<< n.data;  // 20
n.next;  // nullptr
```

```
p->data
(*p).data;
(*p).next;
p->next;
```

Review: Dynamic memory (new and delete)

```
Node* p1 = new Node {10, nullptr};
p1->next = new Node {30, nullptr};
```



p1 → data // 10

delete p1 → next ;

X ~~delete p1 → next;~~

p1 → next = nullptr

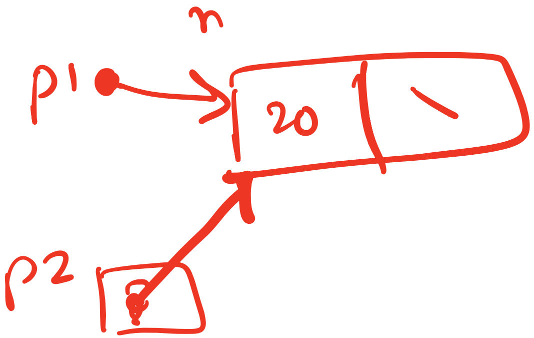
A. delete p1

B. delete p1 → next

C. delete p1 → next → next

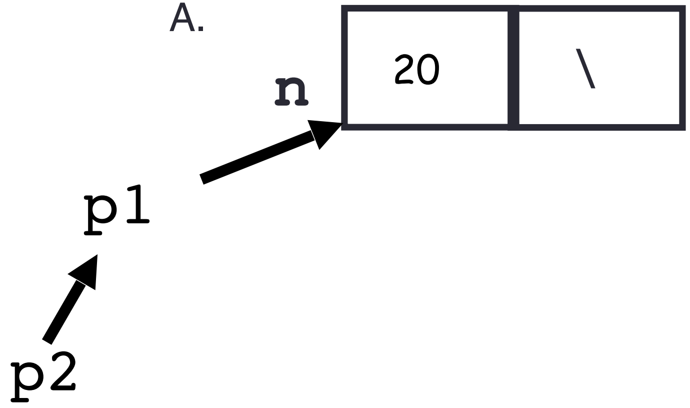
Review: Pointer assignment

```
Node* p1, *p2;
Node n {20, nullptr};
p1 = &n;
p2 = p1;
```

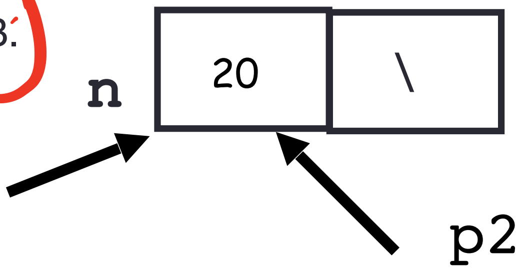


Q: Which of the following pointer diagrams best represents the outcome of the above code?

A.



B.



C. Neither, the code is incorrect

Today's learning goals:

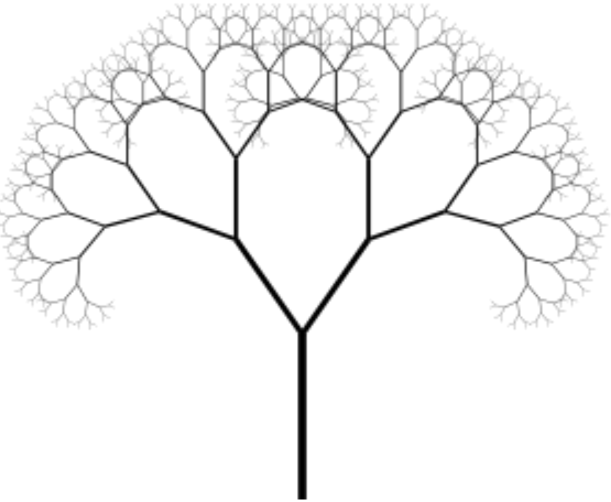
Recursion and its application to linked list operations

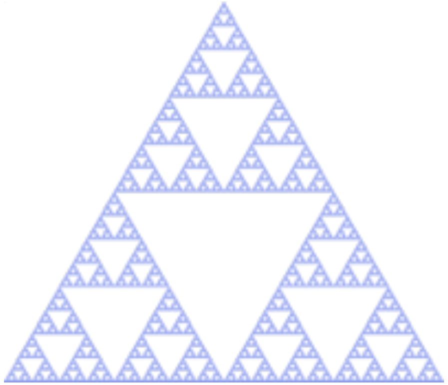
Dynamic memory and common errors

We want to understand the what, why, and how of the C++ Big Three:

- Destructor
- Copy constructor
- Copy assignment operator

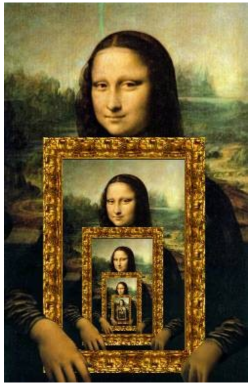
Recursion



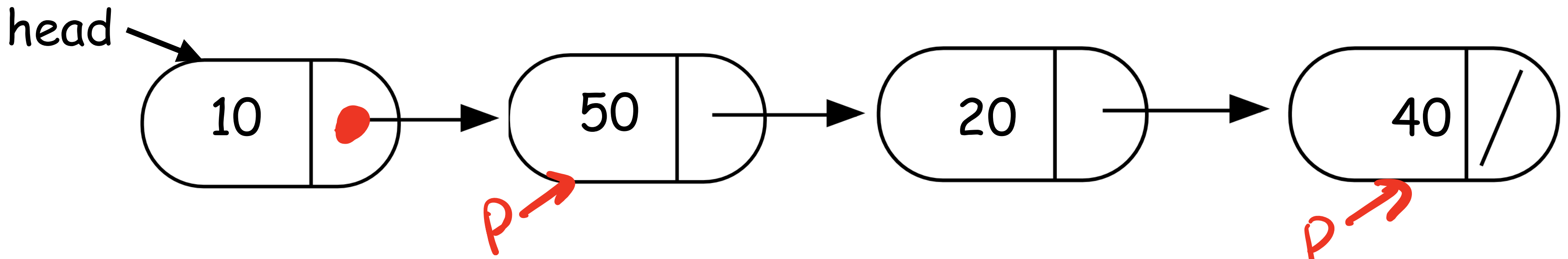


Sierpinski triangle

Zooming into a Koch's snowflake



Using recursion to implement operators involving a linked list




```
int IntList::sum(){

    //return the sum of the sequence
 }
```

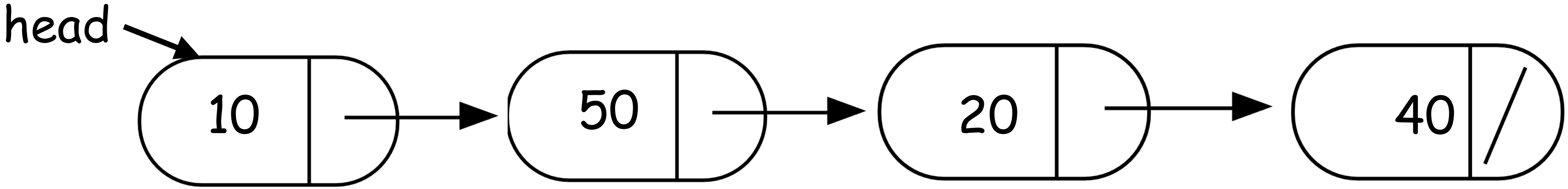
10 + 50 + 20 + 40 + 0

Helper functions

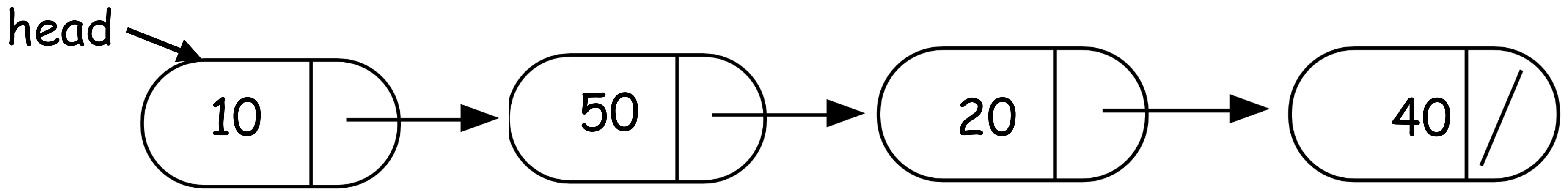
- Sometimes your functions takes an input that is not easy to recurse on
- In that case define a new function with appropriate parameters: This is your helper function
- Call the helper function to perform the recursion
- Usually the helper function is private

For example

```
Int IntList::sum(){

   return sum(head);
    //helper function that performs the recursion.

}
```




```
int IntList::sum(Node* p){




 }
```



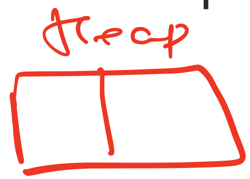

```
bool IntList::clear(Node* p){


}
```

Dynamic Memory: common errors

- Memory Leak: Program does not free memory allocated on the heap.

```
void foo() {
    Node *p = new Node;
}
```

Memory leak



- Segmentation Fault: Code tries to access an invalid memory location

Constructor and Destructor

Every class has the following special methods:

- Constructor: Called right AFTER an object is created in memory
- Destructor: Called right BEFORE an object is deleted from memory

The compiler automatically generates default versions, but you can provide user-defined implementations

```
void foo(){
    Complex p(1, 2);
    Complex* q = new Complex(3, 4);
}
```

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

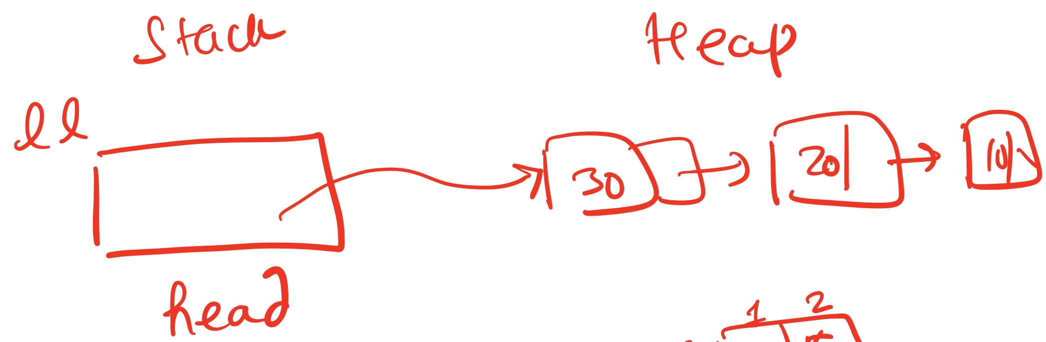
What is the output?

A. 1 + 2j

B. 3 + 4j

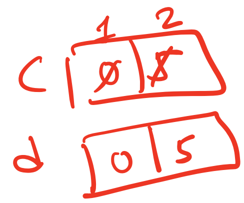
C. 1 + 2j
   3 + 4j

D. None of the above



① Destructor :

Complex c(0,5);



→ Complex d(c); // copy constructor: use c to initialize d

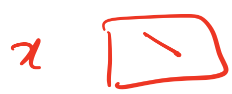
②

Complex e(1,2);

c = e

③ assignment operator : Assign the member variables of e to c
The value of each member variable is
copied from e to c.

IntList x;

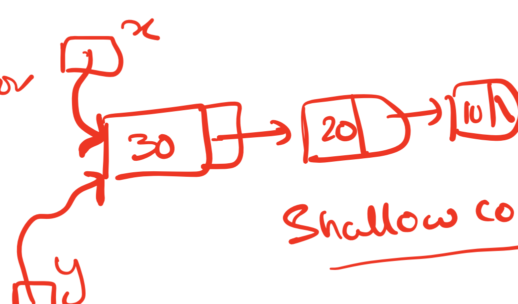


// push some value

// 10 20 30

This statement calls the copy constructor of IntList:

IntList y(x);



Shallow copy

The default behavior of the copy constructor for a linked list results in a shallow copy — which is problematic!

Big Three: Destructor, copy constructor, copy assignment operator

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

Assume:

* Default destructor
* Default copy constructor
* Default copy assignment

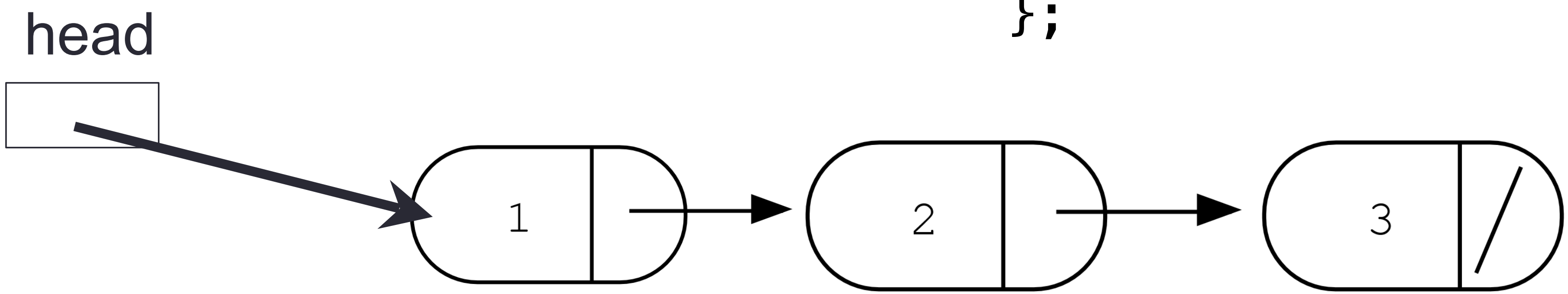
What is the result of running the above code?

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

Concept Question

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

```
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

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

We answered the following questions for the Complex class:

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

Copy constructor

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

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. 10 20

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()
}
```

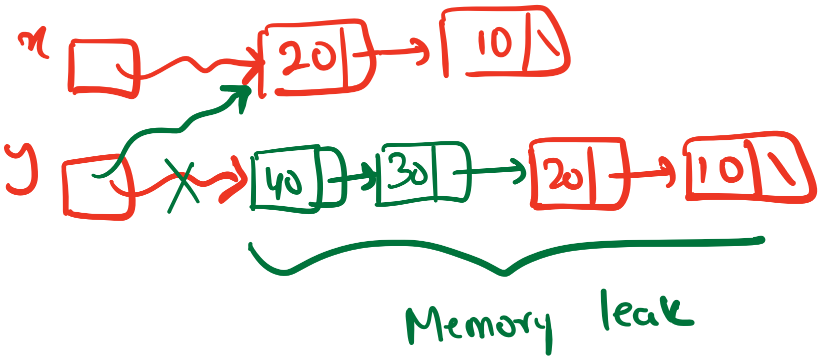
What is the result of running the above code?
A. Prints 20, 10
B. Segmentation fault
C. Memory leak
D. A &B
E. A, B and C

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 ~~10,~~ 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