CS221: Algorithms and Data Structures

Quick Review of Pointers

Alan J. Hu

(Borrowing some slides from Steve Wolfman)

Learning Goals

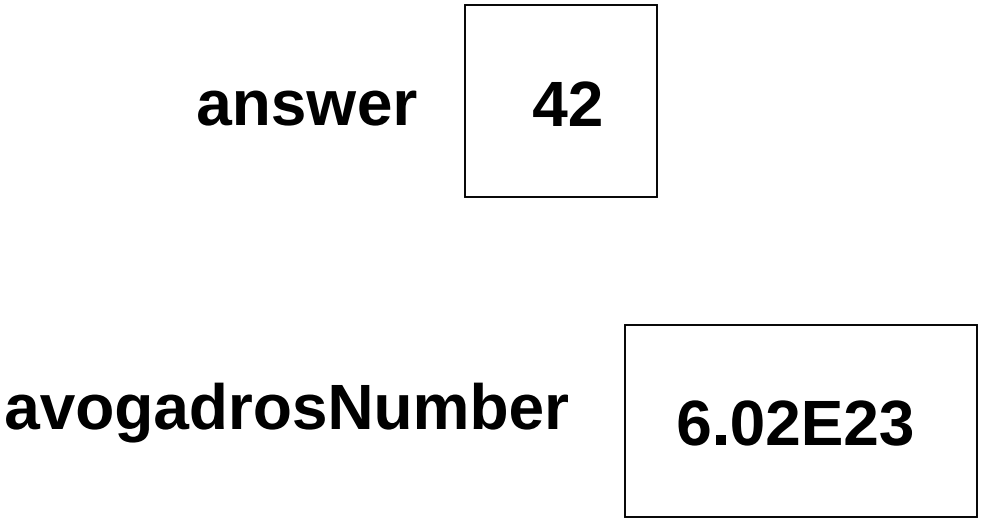
- Get comfortable with C++ pointers, understand the * and & operators.
- Draw diagrams to help understand code that manipulates pointers.

Review of Java References

- Java has “references” which are basically the same as C++ pointers.
  - Most of what you’ve learned already applies! ☺
- C++ pointers are more general and give you more control.
  - In some ways, they are more consistent and logical.
  - But you have to do more work and be more careful.

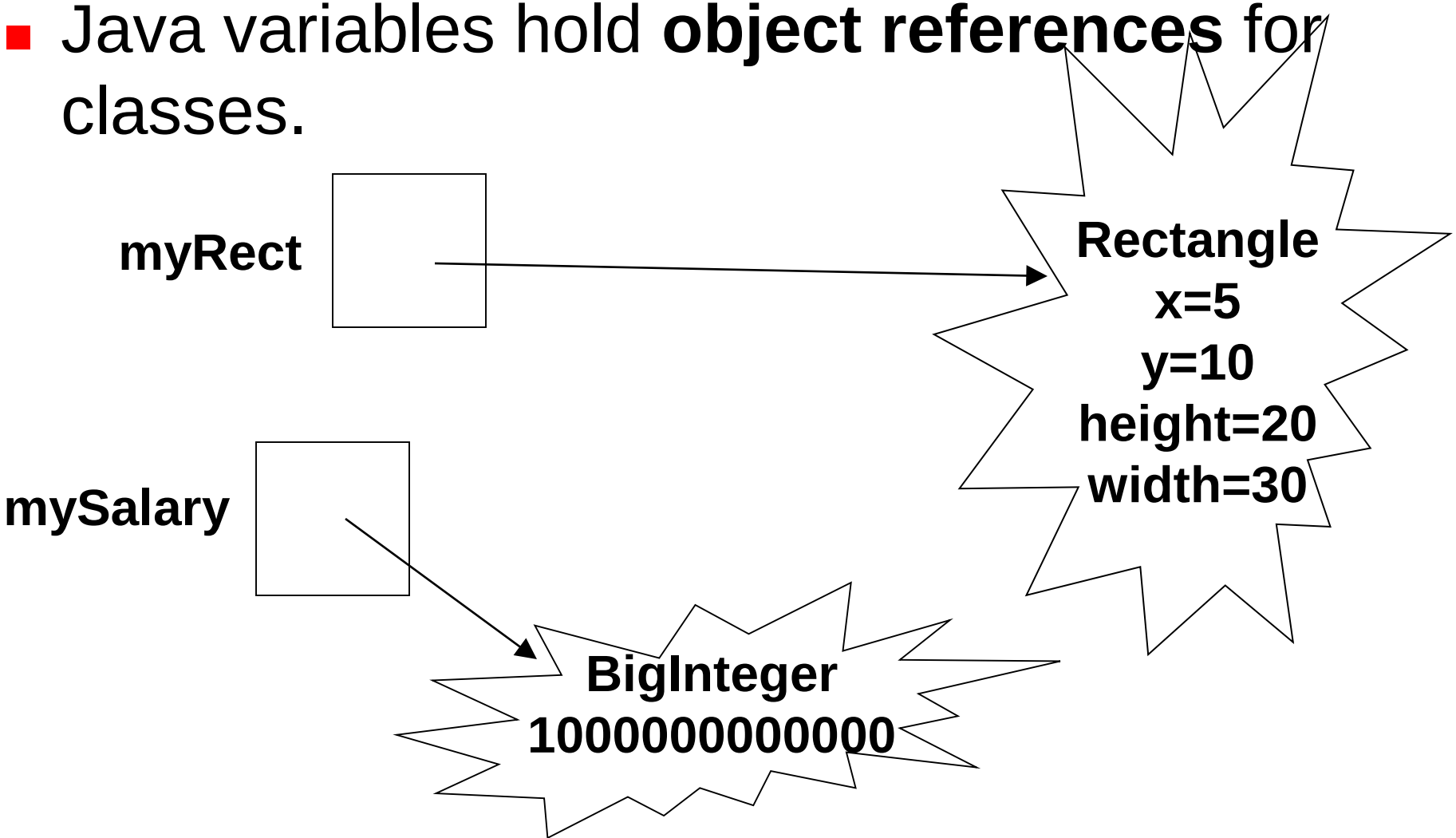
Java Primitive Types: Variables Hold Values

- Java variables hold **values** for primitive types.



Java Classes: Variables Hold (Java) References

- Java variables hold **object references** for classes.



Java Object References

- This is a bit of Java weirdness:
  - For primitive types, variables hold the value.
  - For classes, variables hold reference to object
- Declaration creates variable that can hold a primitive value or an object reference.
- Constructor creates the object itself.

```
BigInteger mySalary =
        new BigInteger("1000000000");
```

Why Care About References?

- You go skiing with a friend. You split a granola bar with him. He eats his half. Does it affect yours?
- You make a copy of your lecture notes for a friend. Her dog chews up her copy. Does it affect yours?

Why Care About References?

- You go skiing with a friend. You have the hotel make a copy of your hotel key for your friend, so he can leave some stuff there. He trashes the room. Does it affect your room?
- Your parents get an extra credit card for you, on their account. You go wild on a shopping spree. Does this affect your parents' credit?

Why Care About References?

- Sometimes it can matter.
- Just like in real life, it can matter if:
  - There are more than one reference to the object. (This is called ***aliasing***.)

    AND
  - The object can be modified/changed. (This is called being ***mutable***.)

(Java) What does this print?

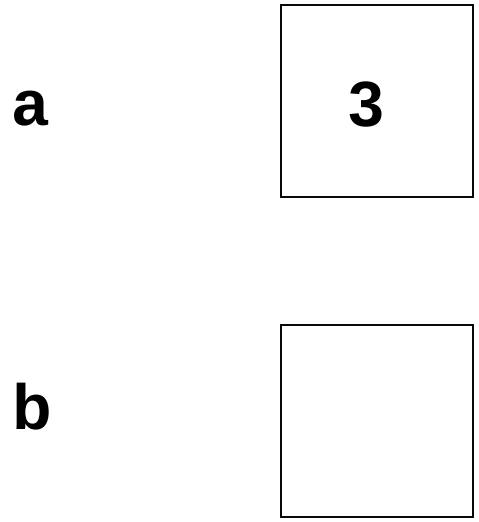
```
int a;
int b;

a = 3;
b = a;
b = b+1;
System.out.println(“a = “ + a + “ and b = “ + b);
```

(Java) What does this print?

```
Rectangle a;
Rectangle b;


a = new Rectangle(3,3,0,0);
b = a;
b.translate(1,1); // add 1 to x and y coordinates
System.out.println(“a = “ + a + “ and b = “ + b);
```

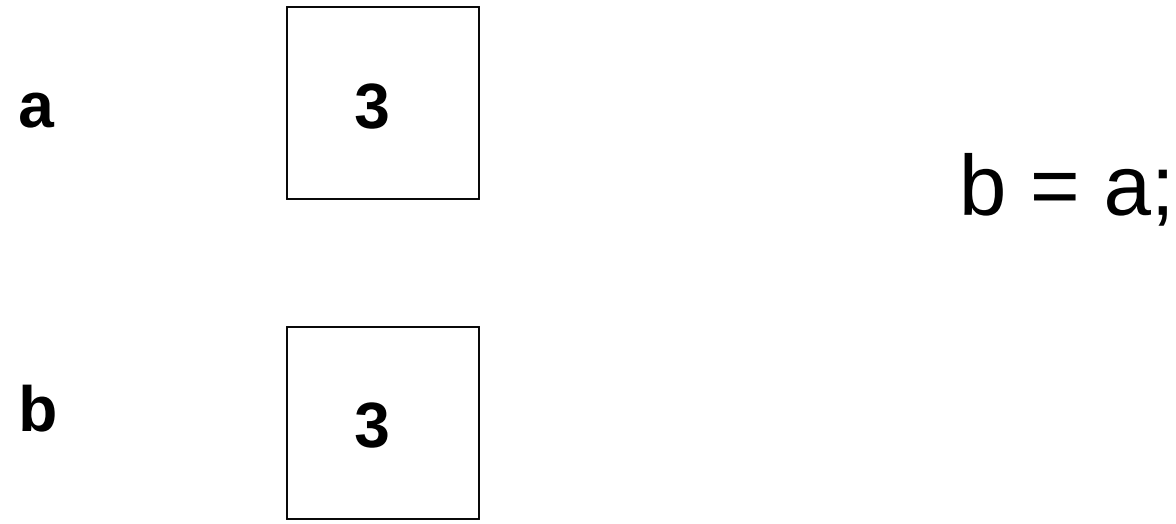
For Java Primitive Types, Variables Hold Values

- Java variables hold **values** for primitive types. (Therefore, can't have aliasing.)



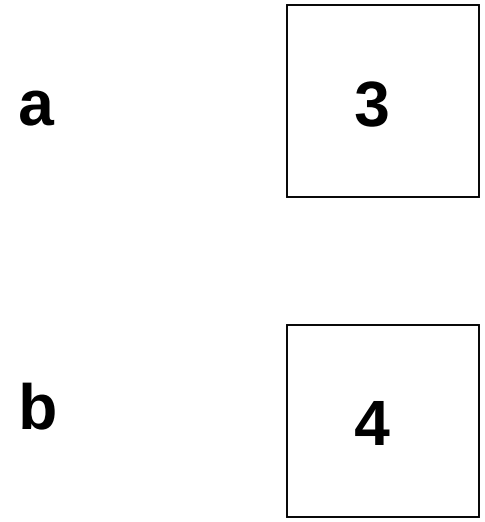
For Java Primitive Types, Variables Hold Values

- Java variables hold **values** for primitive types. (Therefore, can't have aliasing.)



For Java Primitive Types, Variables Hold Values

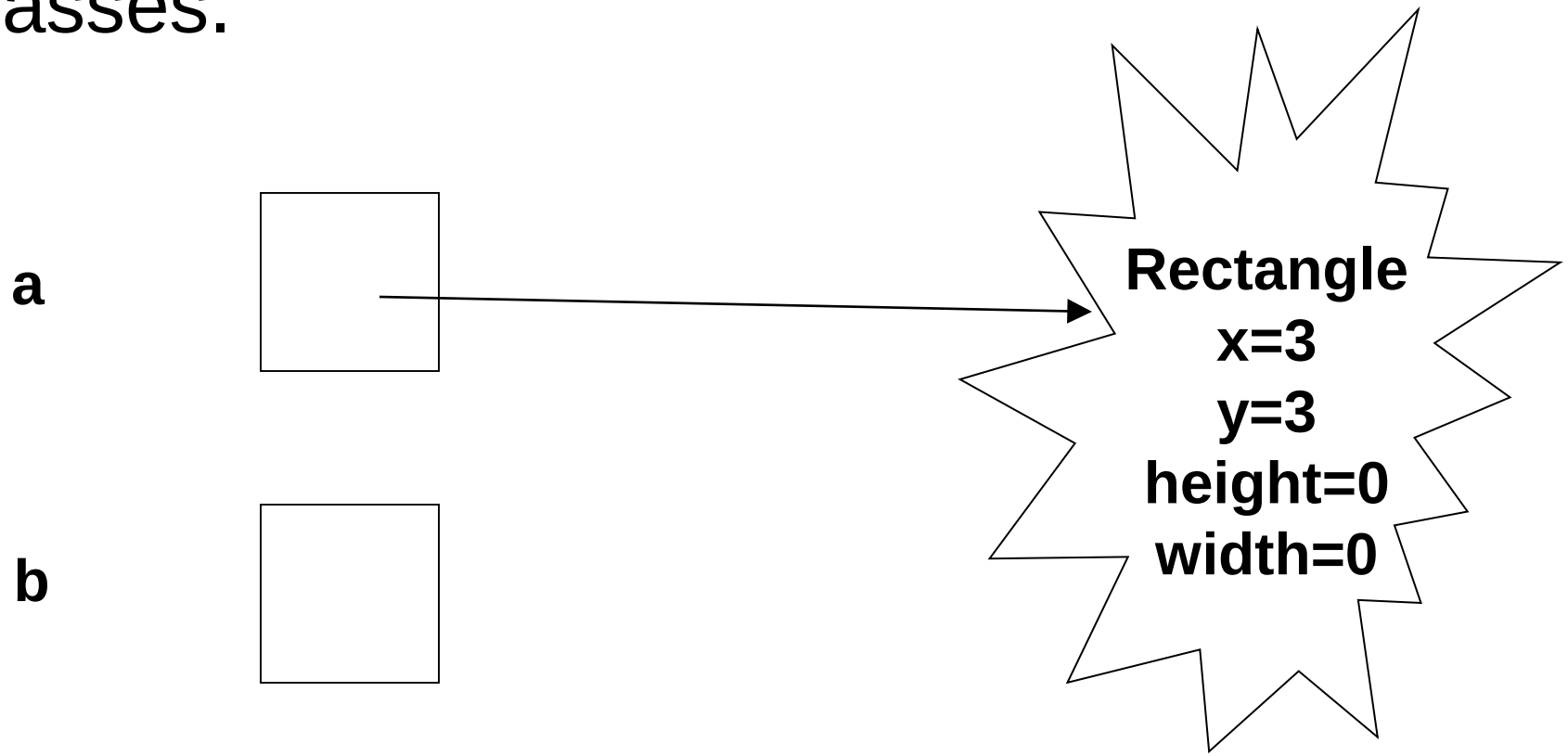
- Java variables hold **values** for primitive types. (Therefore, can't have aliasing.)



b = b+1;

For Java Objects, Variables Hold References

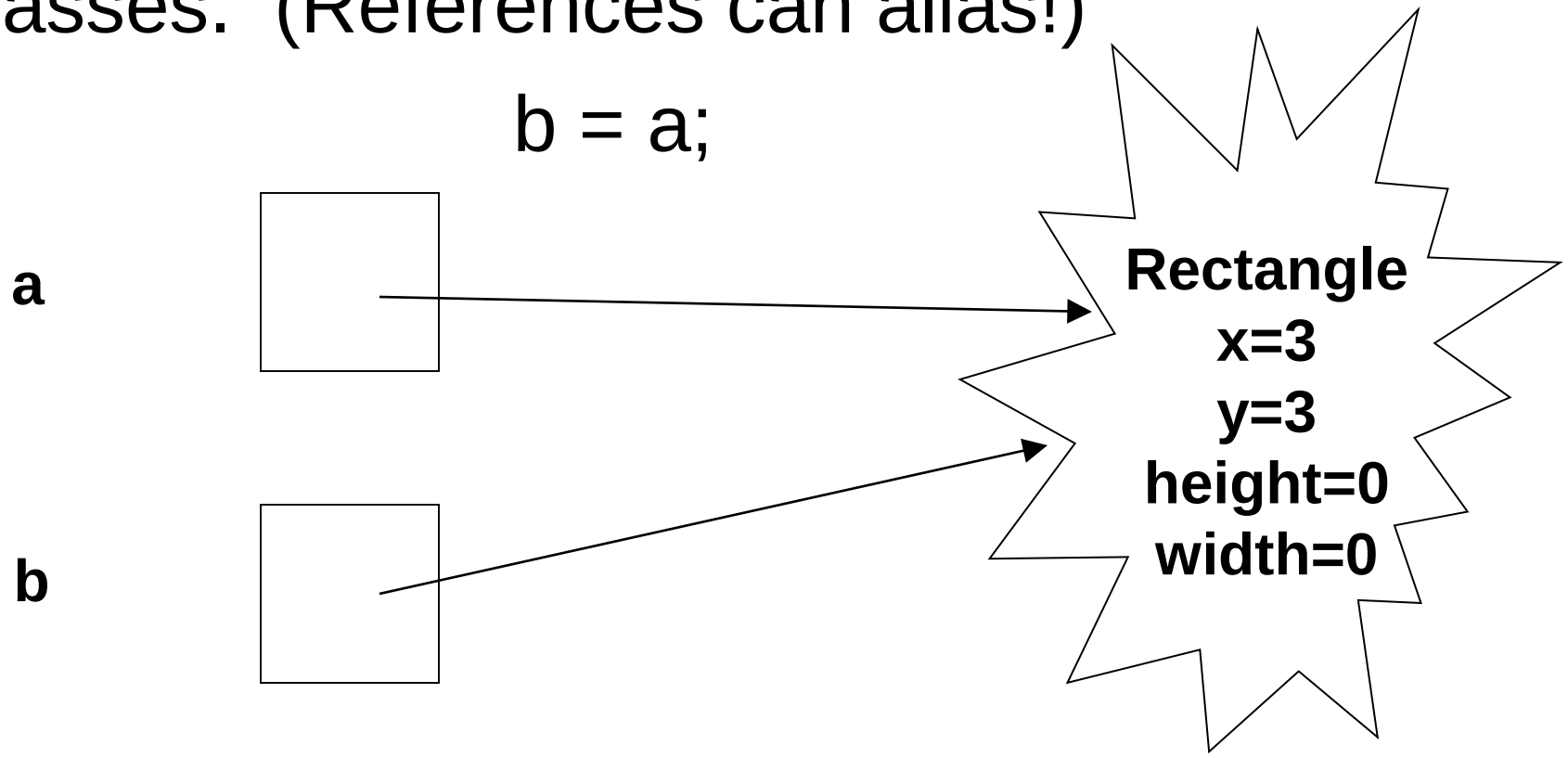
- Java variables hold **object references** for classes.



For Java Objects, Variables Hold References

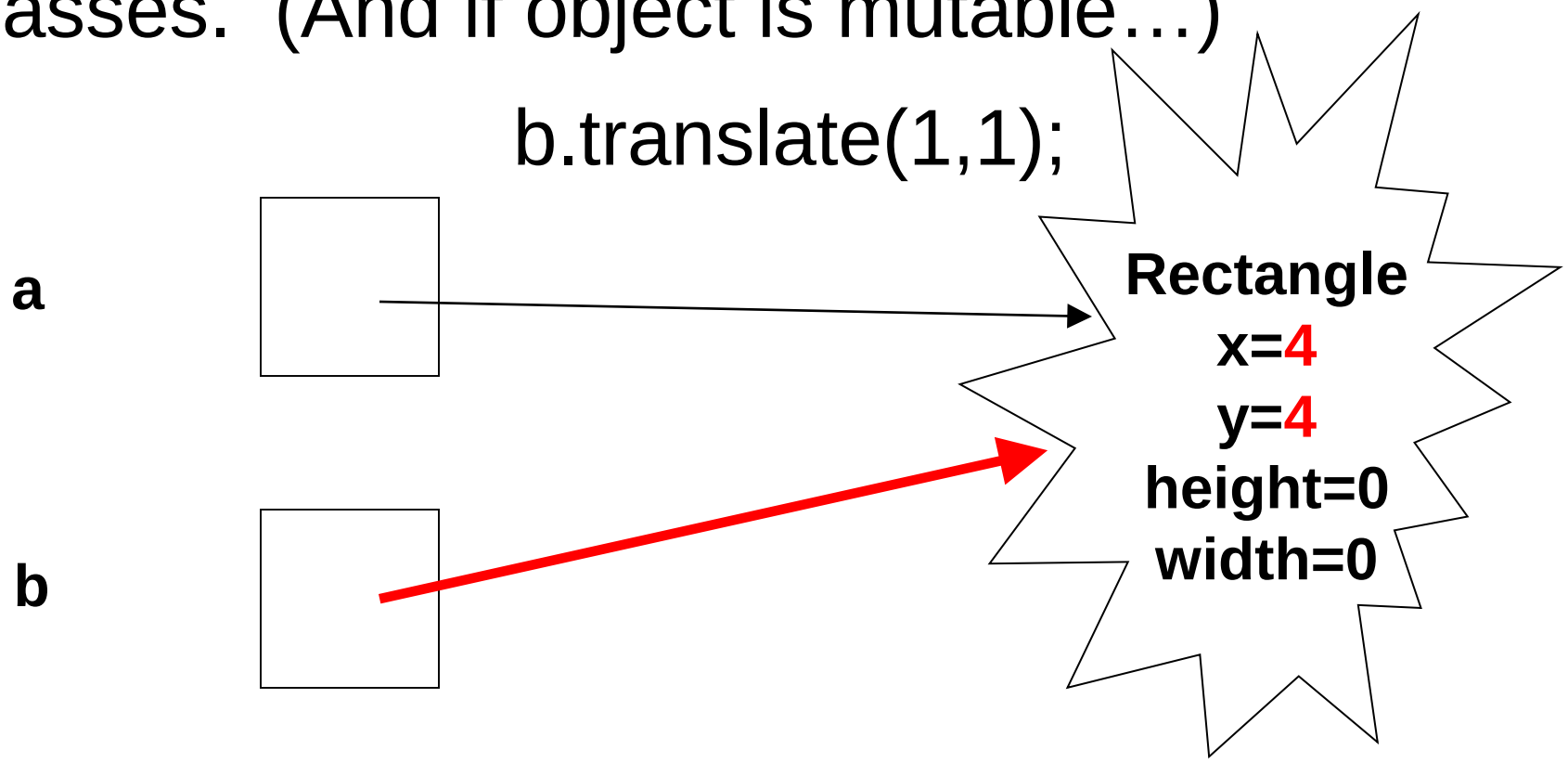
- Java variables hold **object references** for classes. (References can alias!)

b = a;



For Java Objects, Variables Hold References

- Java variables hold **object references** for classes. (And if object is mutable…)



Java References vs. C++ Pointers

- What Java calls a “reference” is basically the same as what C++ calls a “pointer”. (C++ has something different called a “reference” that we will learn later.)
- However, in Java, you never declare a reference/pointer explicitly:
  - Variables for primitive types are always values, never pointers.
  - Variables for objects are always references, never the objects themselves.
- In C++, you can do whatever you want:
  - Variables can hold primitive values or entire objects.
  - You can make pointer variables to anything you want.

C++ Basic Pointer Operations

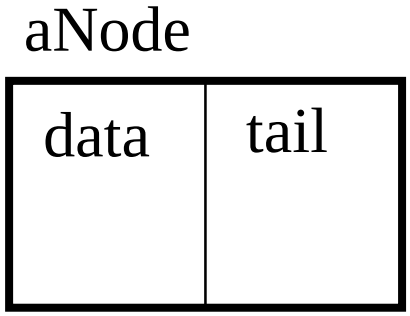
- If foo is any variable, then &foo gives you a pointer to that variable. (Think of this as the "address of foo" or an arrow pointing to foo.)
- If foo is any pointer, then *foo gives you whatever foo points to. (Think of this as giving you the data at address foo, or following the arrow where foo points.
- If foo is an object, then foo.bar gives you the member variable named "bar" in object foo.
- NOTE! In C++, you'll usually have a pointer to an object instead of the object itself, so you'd have to write `(*foo).bar` instead of `foo.bar`
  - This is so common that C++ has special syntax for this:
    `foo->bar` is exactly the same as `(*foo).bar`

Practice with Pointers

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

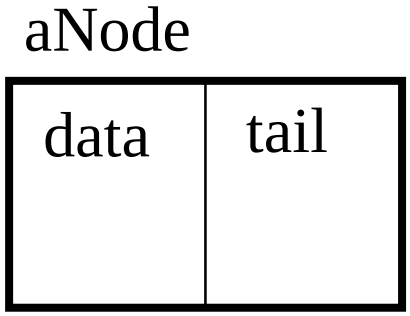
Practice with Pointers

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

Node aNode;
```



Practice with Pointers

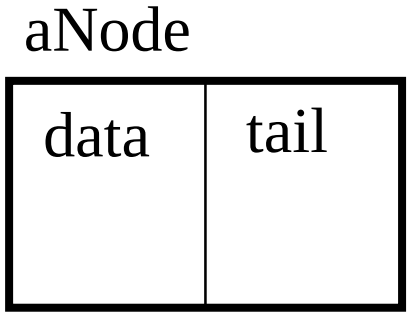
```
struct Node {
   int data;
   Node *tail;
}

Node aNode;
```

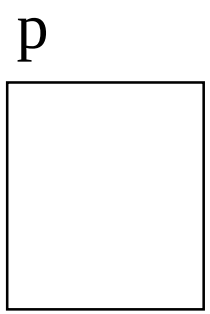


In C++, this actually creates the object.
In Java, it would create only a “reference”/pointer.

Practice with Pointers

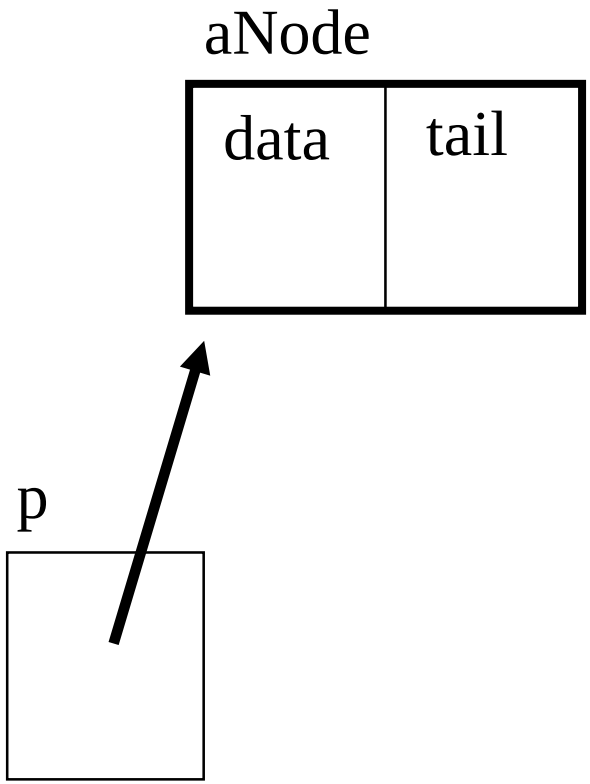
```
struct Node {
   int data;
   Node *tail;
}

Node aNode;
Node *p;
```





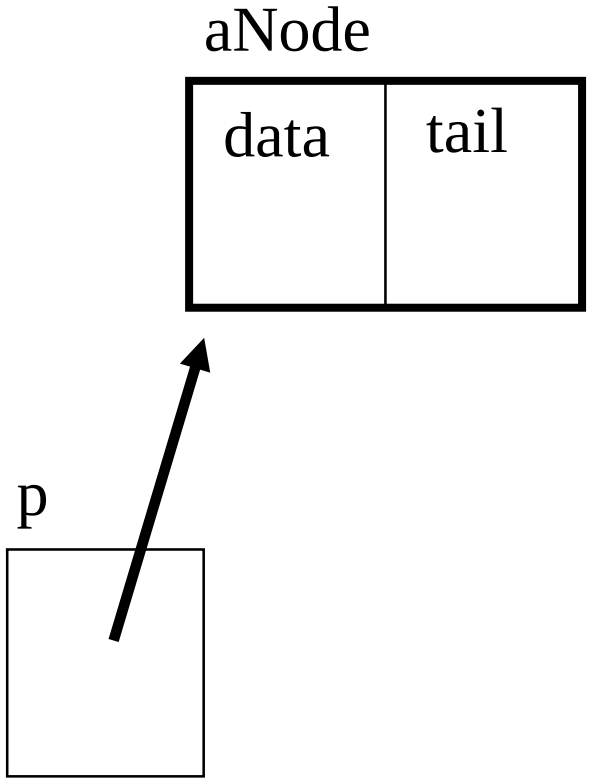
Practice with Pointers

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

Node aNode;
Node *p = &aNode;
```



Practice with Pointers

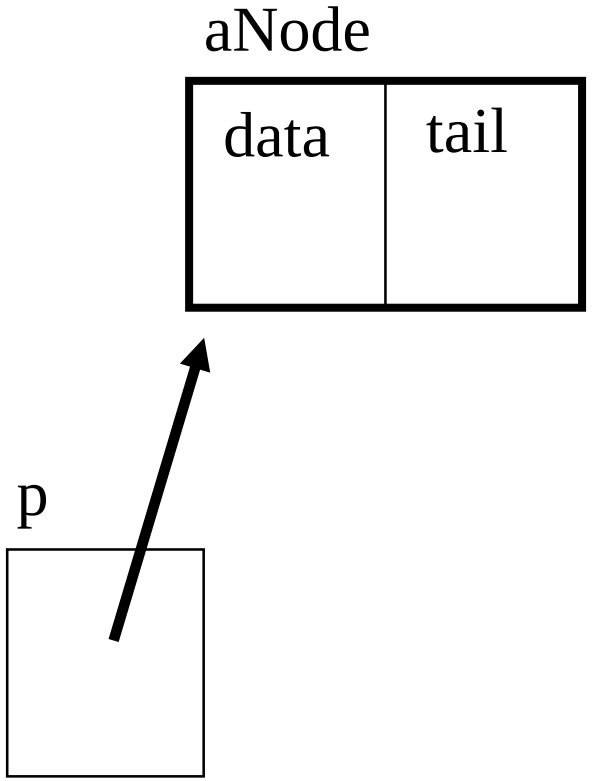
```
struct Node {
    int data;
    Node *tail;
}

Node aNode;
Node *p = &aNode;
```



&foo gives you a pointer to foo.
You can also think of this as the (starting) address of foo.
You can draw it as an arrow pointing to foo.

Practice with Pointers

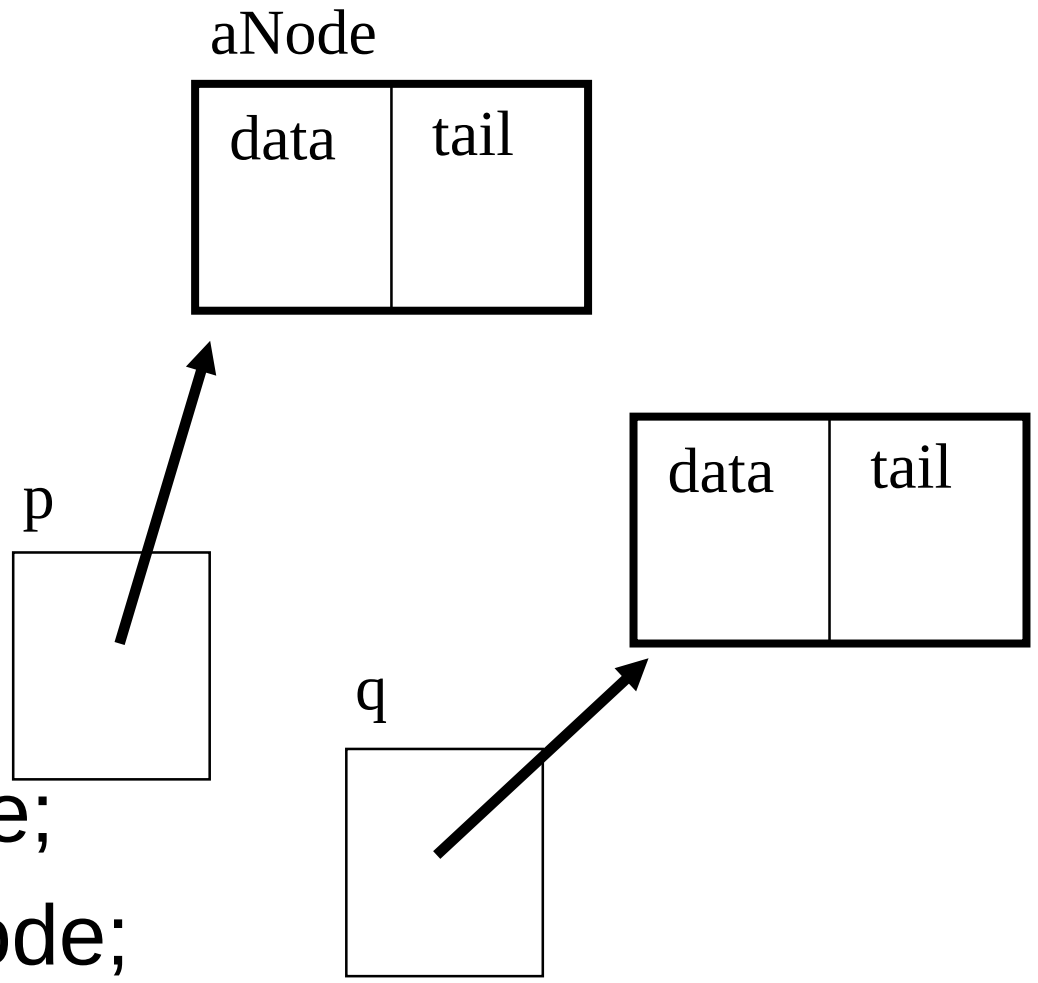
```
struct Node {
    int data;
    Node *tail;
}

Node aNode;
Node *p = &aNode;
```



*foo gives whatever foo points to. (Hopefully, foo is a pointer.)
You can also think of this as whatever is at address foo.
When you draw a diagram, it means following the arrow.

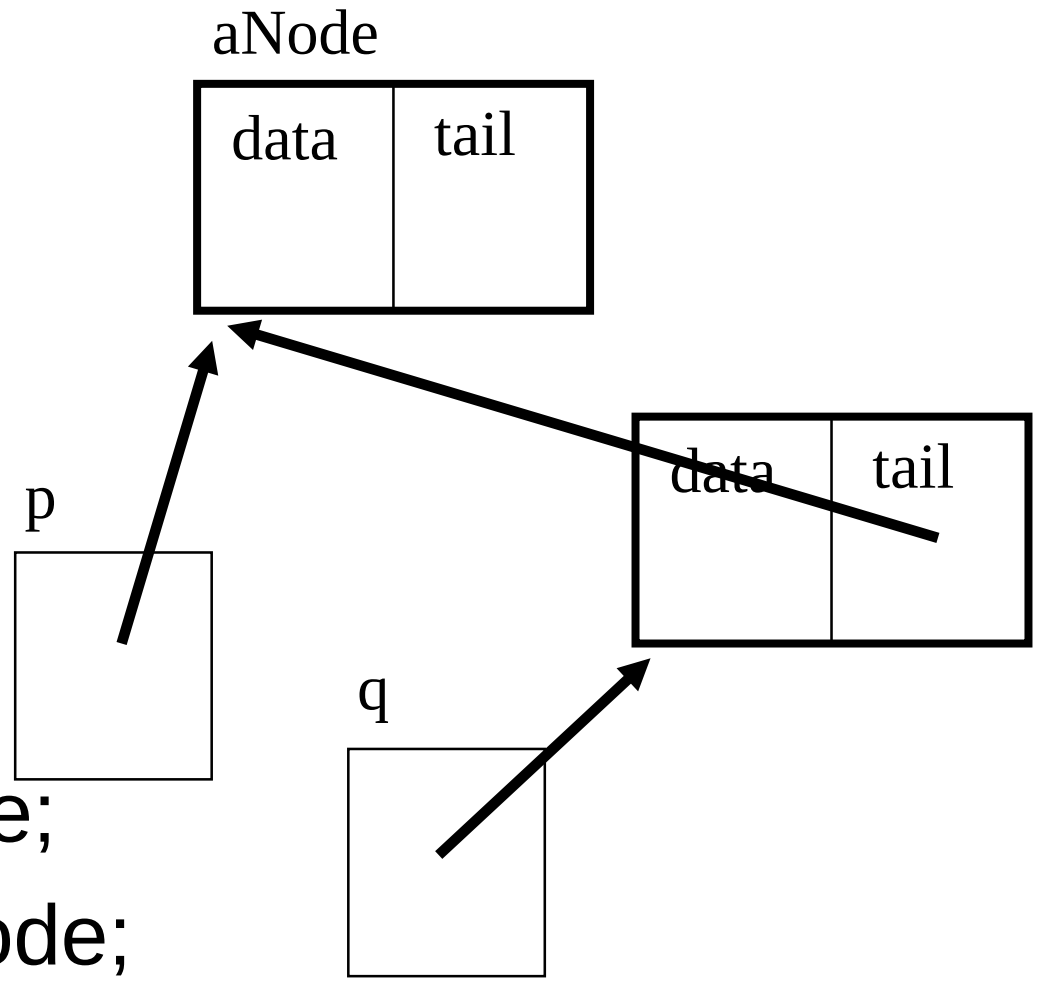
Practice with Pointers

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

Node aNode;
Node *p = &aNode;
Node *q = new Node;
```



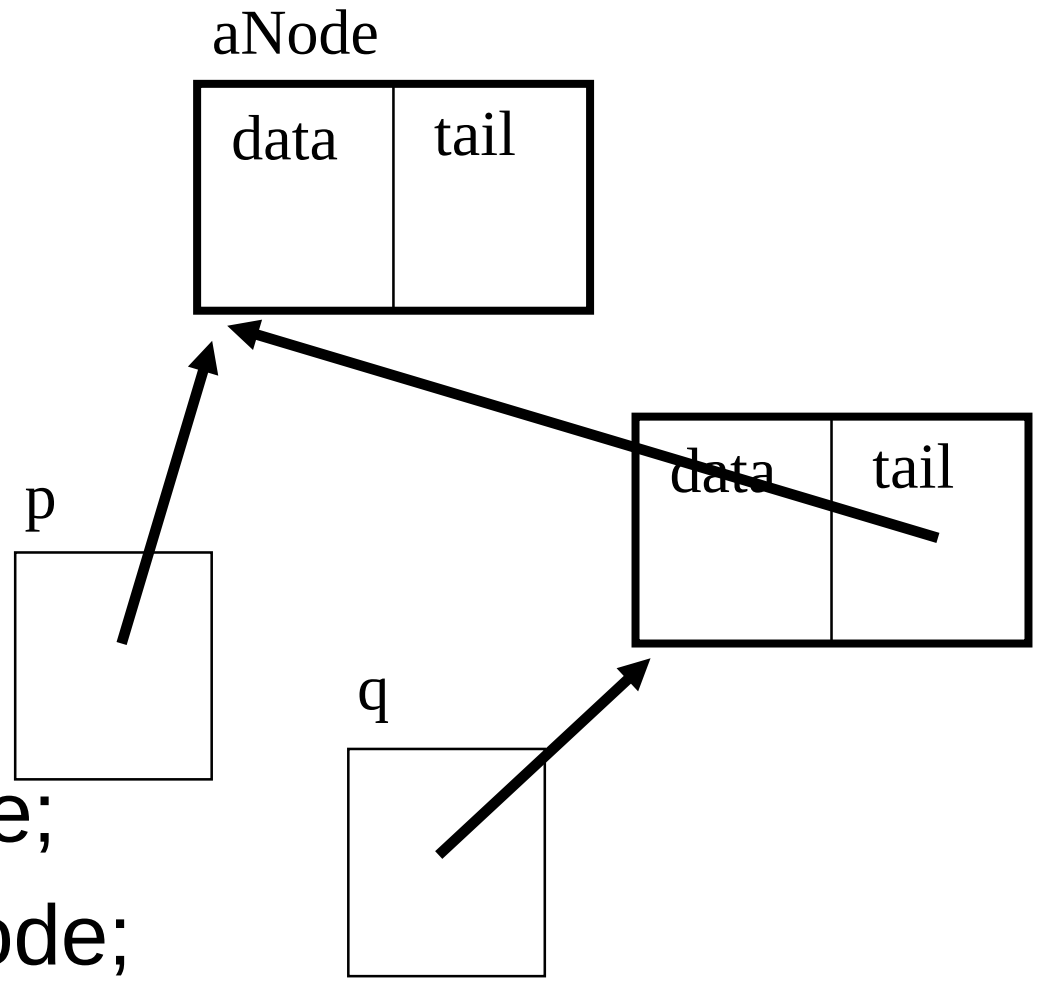
Practice with Pointers

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

Node aNode;
Node *p = &aNode;
Node *q = new Node;
q->tail = p;
```



Practice with Pointers

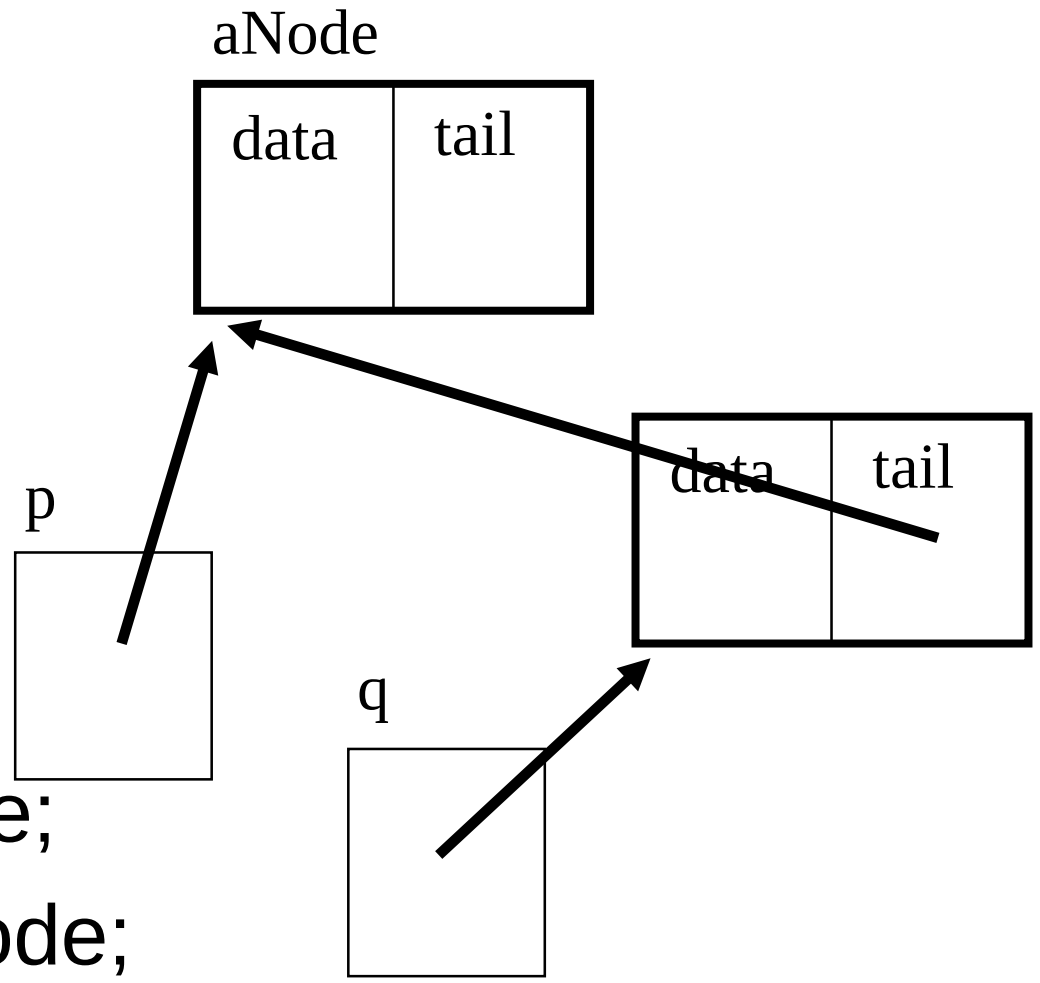
```
struct Node {
   int data;
   Node *tail;
}

Node aNode;
Node *p = &aNode;
Node *q = new Node;
q->tail = p;
```



A copy of a pointer is an arrow to the same place.

Practice with Pointers

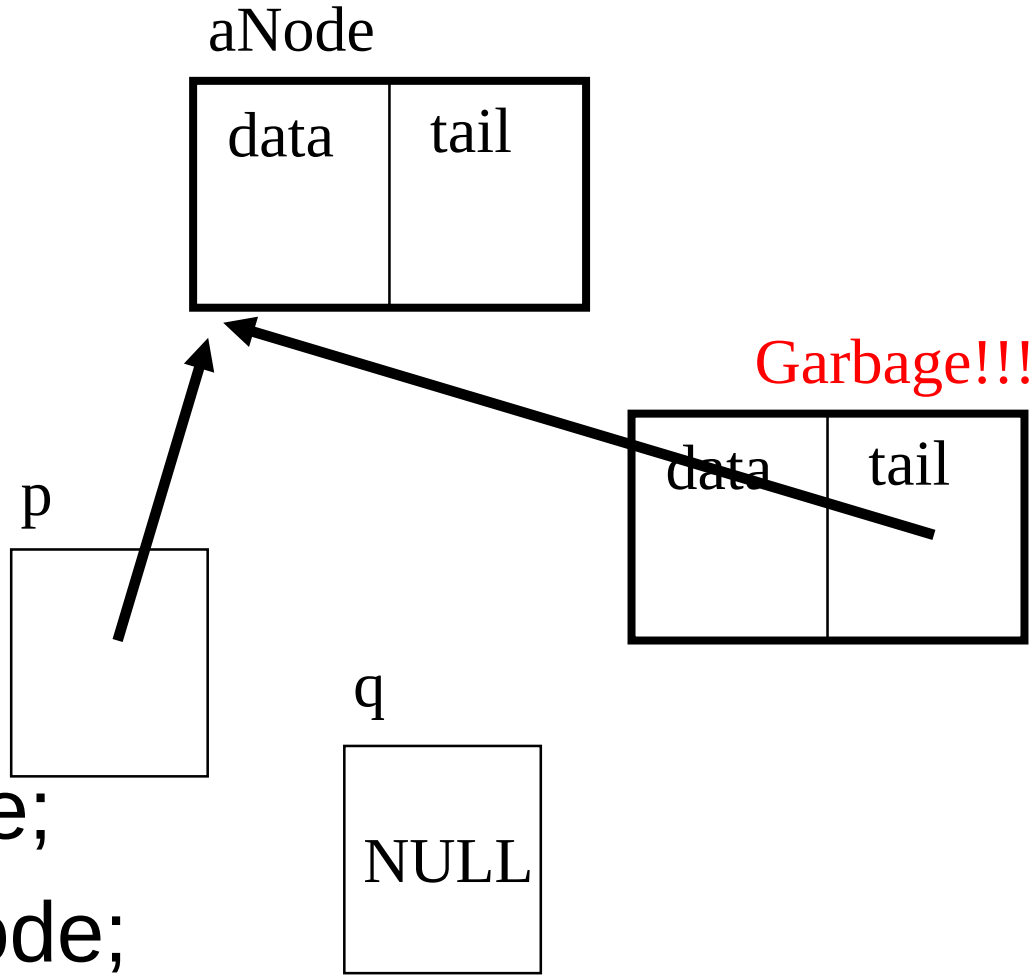
```
struct Node {
   int data;
   Node *tail;
}

Node aNode;
Node *p = &aNode;
Node *q = new Node;
q->tail = p;
```



Could have written instead (*q).tail = p;

Practice with Pointers

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

Node aNode;
Node *p = &aNode;
Node *q = new Node;
q->tail = p;
q = NULL;
```



Practice with Pointers

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

Node aNode;
Node *p = &aNode;
Node *q = new Node;
q->tail = p;
delete q;  // Important in C++ to not leak mem!
```