

Today's announcements

- MP3 is out, **E/C** is due on Tuesday, February 19 @11:59 PM
MP3 is due on Friday, February 26 @11:59 PM
- **lab_gdb** is due on Tuesday, February 16 at 11:59 PM

Abstract Base Classes: - a class that contains a "pure virtual function"

```
class flower {  
public:  
    flower();  
    virtual void drawBlossom() = 0;  
    virtual void drawStem() = 0;  
    virtual void drawFoliage() = 0;  
    ...  
};
```

flower.h

↓
pure virtual
function

↓
"it won't
be implemented
in flower
class"

```
void daisy::drawBlossom() {  
    // whatever  
}  
void daisy::drawStem() {  
    // whatever  
}  
void daisy::drawFoliage() {  
    // whatever  
}
```

daisy.cpp

```
class daisy:public flower {  
public:  
    virtual void drawBlossom();  
    virtual void drawStem();  
    virtual void drawFoliage();  
    ...  
private:  
    int blossom; // number of petals  
    int stem; // length of stem  
    int foliage // leaves per inch  
};
```

daisy.h

```
flower f;  
daisy d;  
flower * fptr;
```


Concluding remarks on inheritance:

Polymorphism: objects of different types can employ methods of the same name and parameterization.

```
animal ** farm;  
  
farm = new animal*[5];  
farm[0] = new dog;  
farm[1] = new pig;  
farm[2] = new horse;  
farm[3] = new cow;  
farm[4] = new duck;  
  
for (int i=0; i<5;i++)  
    farm[i]->speak();
```

Inheritance provides DYNAMIC polymorphism—type dependent functions can be selected at run-time. Wikipedia: Polymorphism in OOP

Next topic: “templates” are C++ implementation of static polymorphism, where type dependent functions are chosen at compile-time.


```

class sphere {
public:
    sphere & operator+(const double & a) {
        rad +=a; return *this;
    }
    double getRadius() {return rad;}
    void setRadius(double r) {rad = r;}
private:
    double rad;
};

template <class T, class U>
T addEm(T a, U b) {
    return a+b;
}

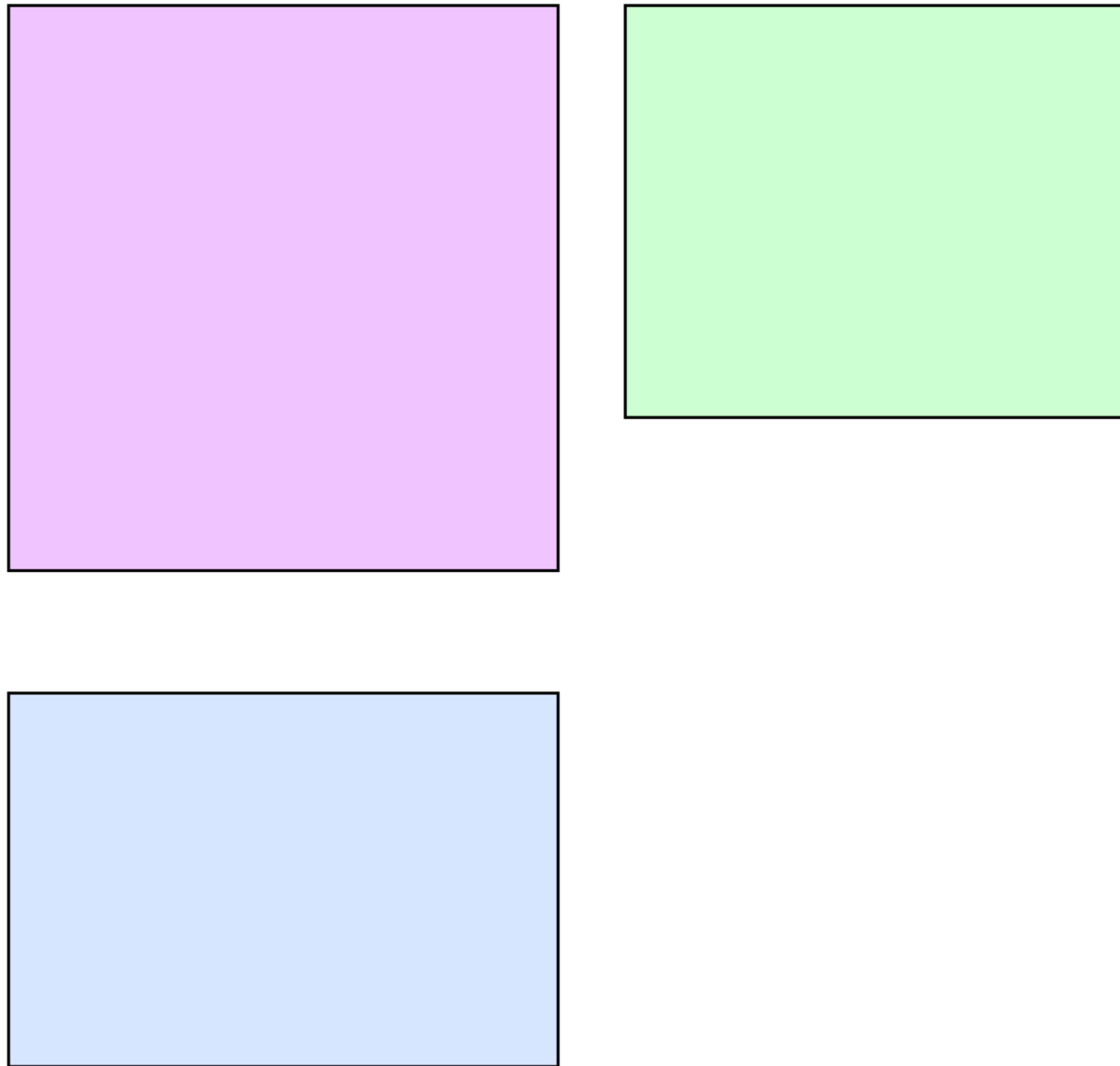
int main () {
    cout << addEm<int,int>(3, 4) << endl;
    cout << addEm<double,int>(3.2, 4)<< endl;;
    cout << addEm<int,double>(4, 3.2)<< endl;;
    cout << addEm<string,string>("7.2", "")<< endl;;
    cout << addEm<string,int>("7.", 2)<< endl;;

    sphere a;
    a.setRadius(3);
    cout << (addEm<sphere,double>(a, 4.2)).getRadius() << endl;
    cout << (addEm<double,sphere>(4.2, a)).getRadius() << endl;
    return 0;
}

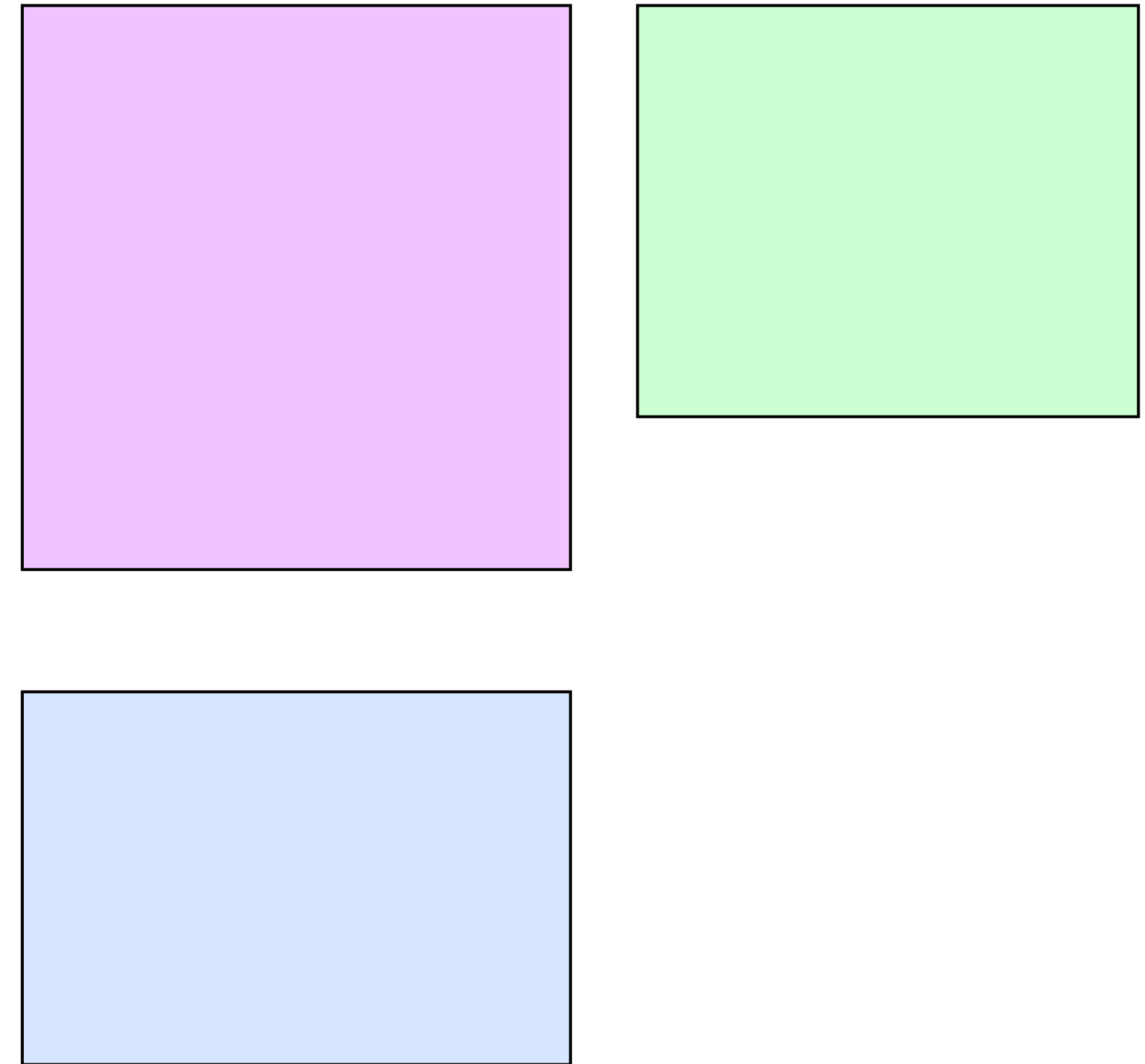
```


Template compilation:

Old:

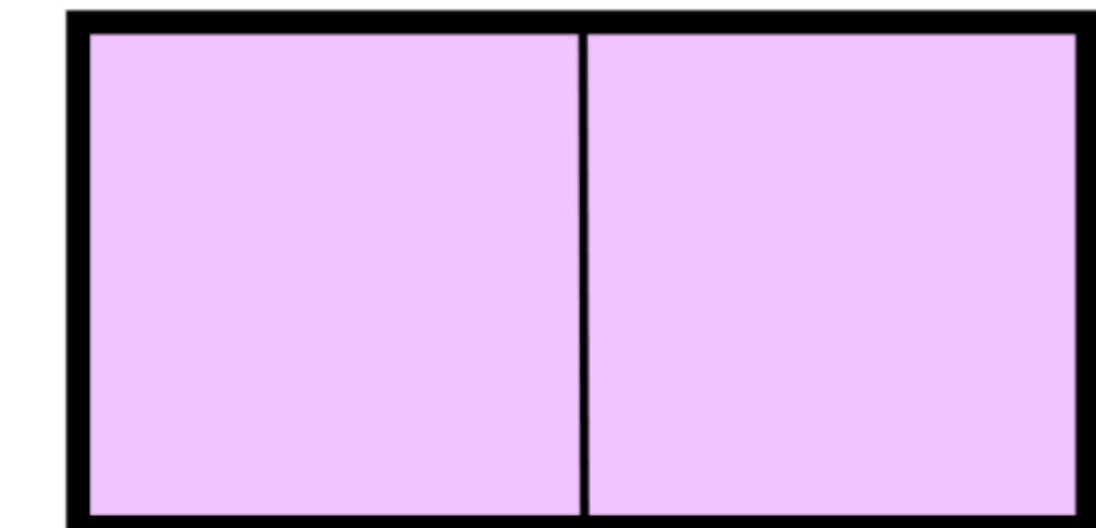


New:



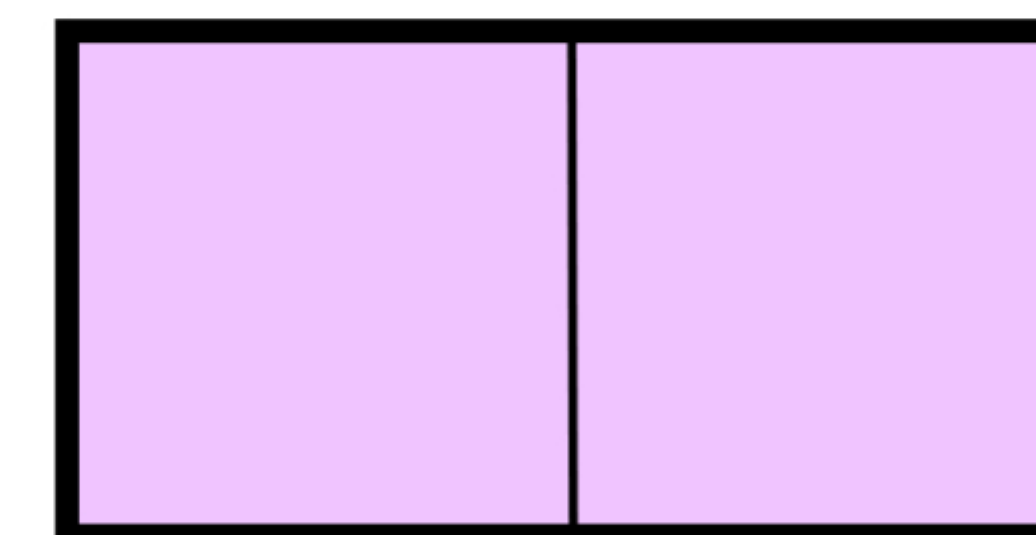
Toward a new memory model:

```
struct listNode {  
    LIT data;  
    listNode * next;  
    listNode(LIT newData) : data(newData), next(NULL) {}  
};
```

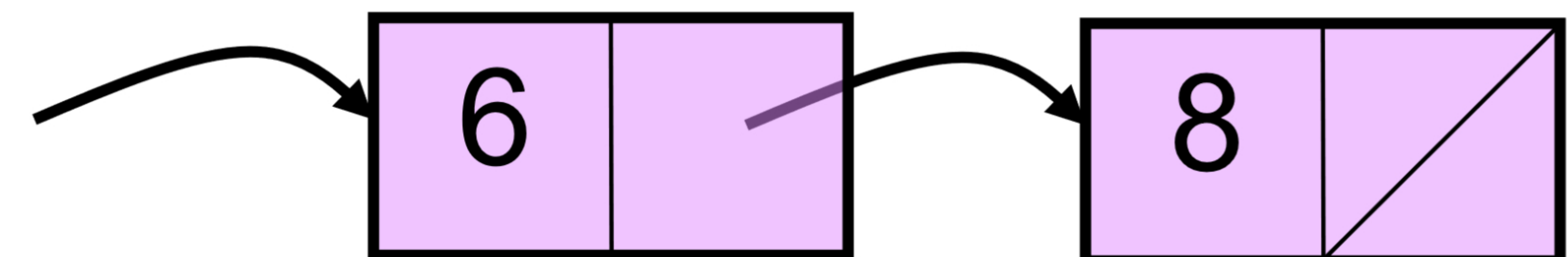
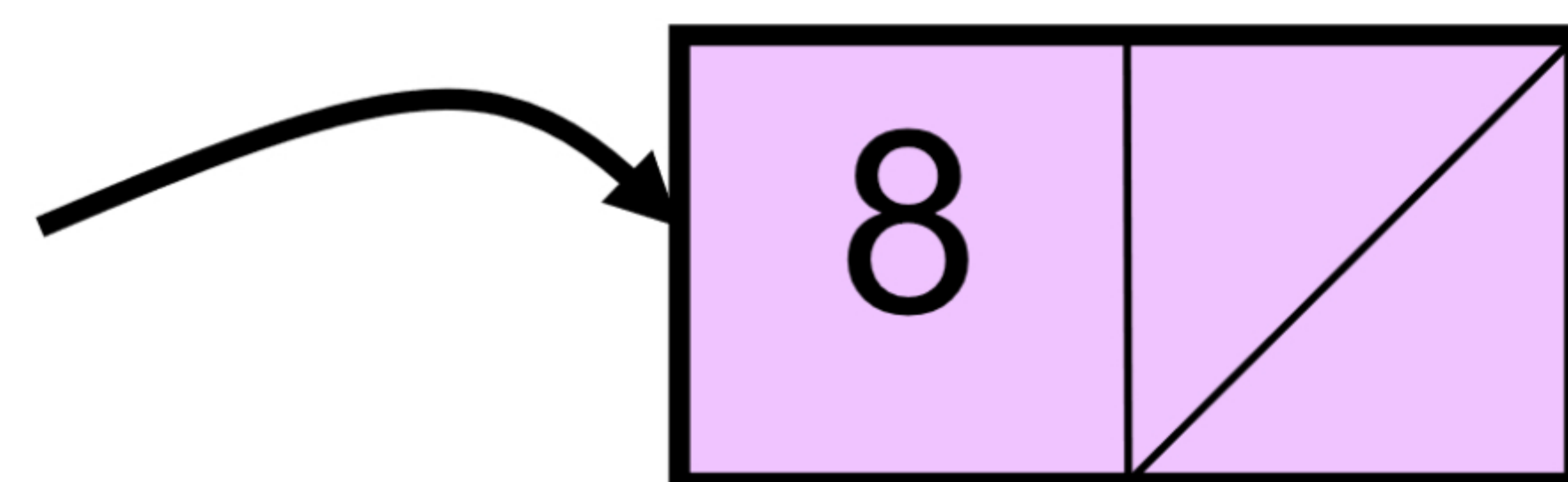


What is the result of this declaration?

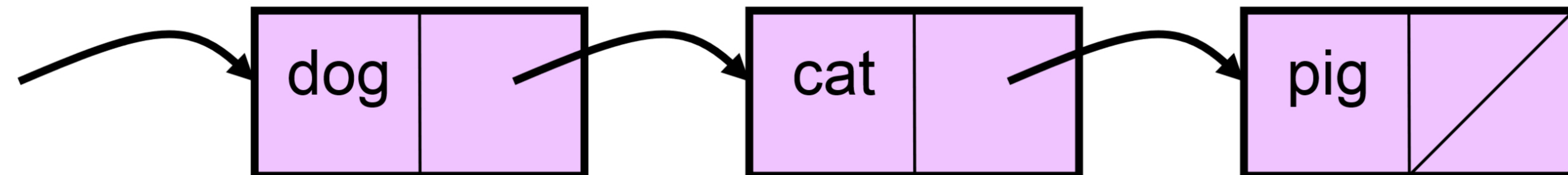
```
listNode<int> nln(5);
```



Write code that would result in each of these memory configurations?



Example 1: `insertAtFront<farmAnimal>(head, cow);`



`void insertAtFront(listNode * curr, LIT e) {`

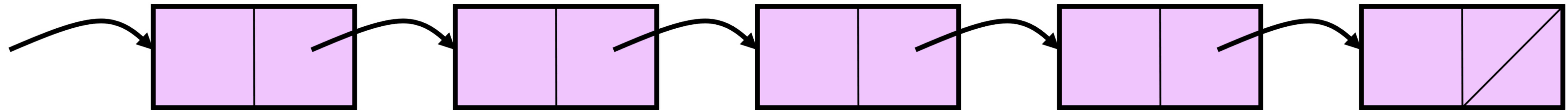
}

Running time?

```
struct listNode {
    LIT data;
    listNode * next;
    listNode(LIT newData) : data(newData), next(NULL) {}
}
```


Example 2:

8 4 2 6 3



*void printReverse(listNode * curr) {*

}
Running time?

```
struct listNode {  
    LIT data;  
    listNode * next;  
    listNode(LIT newData) : data(newData), next(NULL) {}  
}
```