Announcements

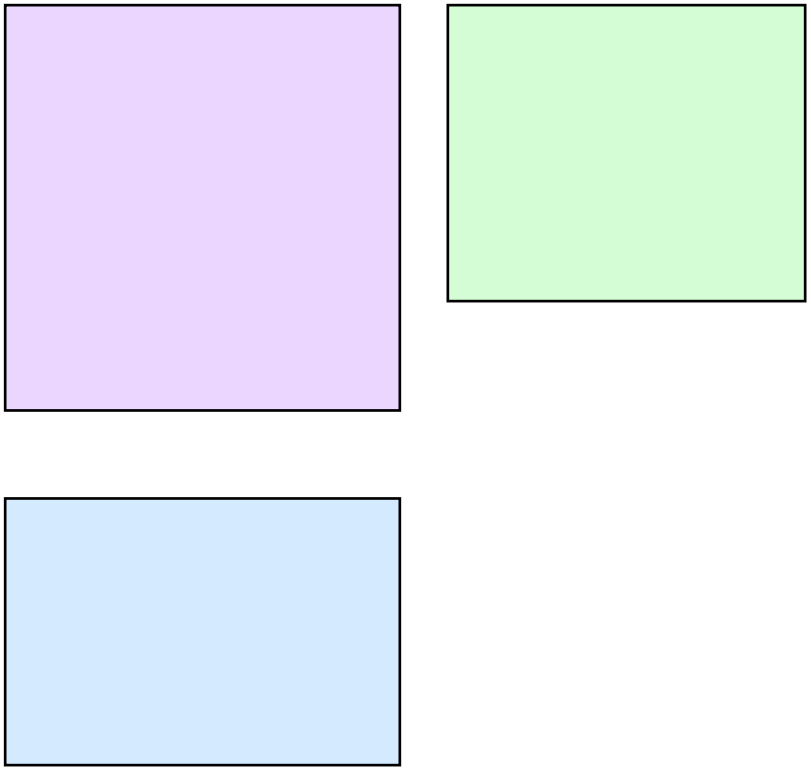
MP3 available, due 10/2, 11:59p. EC due 9/25, 11:59p.

Exam 1: 9/30, 7-10p in rooms TBA

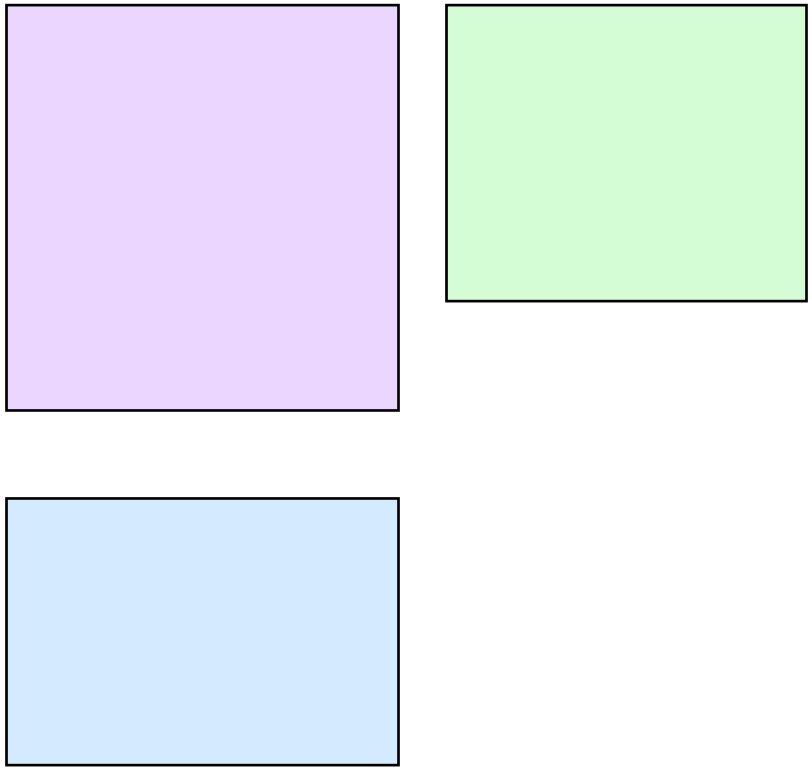
```
template <class T, class U>
T addEm(T a, U b) {
   return a + b;
}

int main() {
   addEm<int,int>(3,4);
   addEm<double,int>(3.2,4);
   addEm<int,double>(4,3.2);
   addEm<string,int>("hi",4);
   addEm<int,string>(4,"hi");
}
```

Template compilation:

Old:

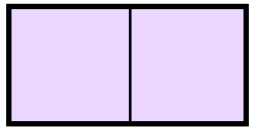


New:



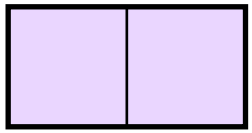
Toward a new memory model:

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

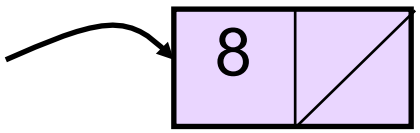


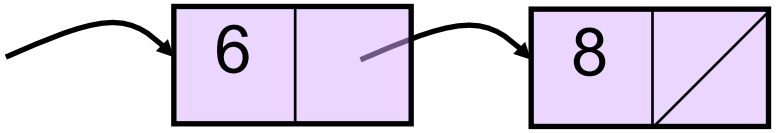
What is the result of this declaration?

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

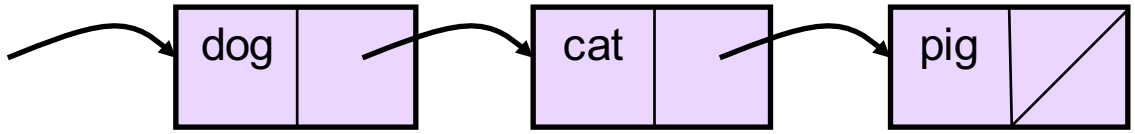


Write code that would result in each of these memory configurations?





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



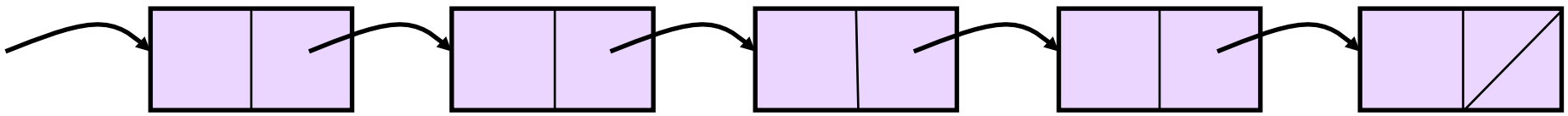

```
void insertAtFront(listNode *   curr, LIT e) {



}
```

Running time?

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

8 4 2 6 3

Example 2:

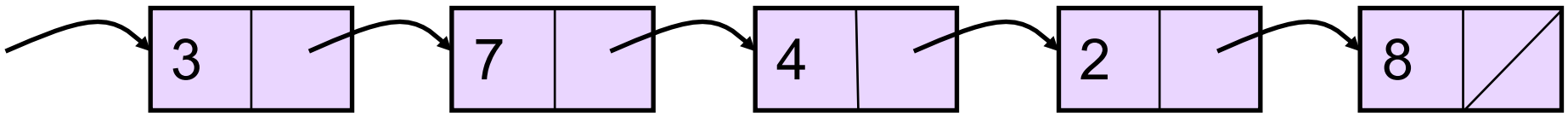


```
void printReverse(listNode *   curr) {



}
```

Running time?

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

Example 3: Find kth position (we'll need this later)

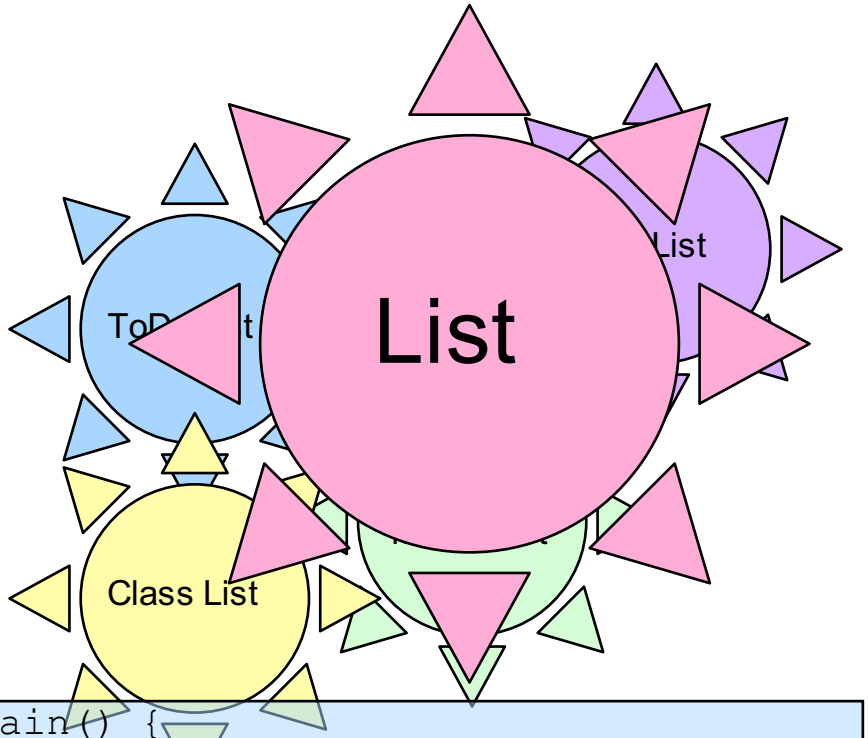



```
//returns pointer to node k steps forward from *curr
listNode * findKth(listNode *   curr, int k) {



}
```

Analysis:

Find kth in array:

Abstract Data Types (an example):




```
template<class LIT>
class List {
public:
    List();
    //~List();
    int getSize() const;
    void insert(int loc, LIT e);
    void remove(int loc);
    LIT const & getItem(int loc) const;
private:
    //my little secret
};
```

```
int main() {
    List<int> myList;
    myList.insert(1,4);
    myList.insert(1,6);
    myList.insert(1,8);
    myList.insert(3,0);
    myList.insert(4,myList.getItem(2));
    cout << myList.getSize() << endl;
    myList.remove(2);
    cout << myList.getItem(3) << endl;
    return 0;
}
```

ADT List, implementation 1:

```
template<class LIT>
class List {
public:
    List():size(0){}
    //~List();
    int getSize() const;
    void insert(int loc, LIT e);
    void remove(int loc);
    LIT const & getItem(int loc) const;
private:
    LIT items[8];
    int size;
};
```

0	1	2	3	4	5	6	7

```
template<class LIT>
int List<LIT>::getSize() const {
   return size;
}
template<class LIT>
void List<LIT>::insert(int loc, LIT e){
if ((size + 1) < 8) {
   LIT go = e;
   int it = loc-1;
   while (it < size+1){
     LIT temp = items[it];
     items[it] = go;
     go = temp;
     it ++;
   }
   size ++;
}}
template<class LIT>
void List<LIT>::remove(int loc) {
if (size > 0) {
   int it = loc-1;
   while (it < size){
     items[it] = items[it+1];
     it ++;
   }
   size --;
}}
template<class LIT>
LIT const & List<LIT>::getItem(int loc)
const {return items[loc -1];}
```

Don't look at this code!

Don't look at this code!