An example

- Let's define a linked list of integers
- What does it look like, abstractly?
- How does that look physically, in C?
- What operations on linked lists, abstractly?
  - e.g. addFirst, addLast, findItem
- How do they look physically, in C?

Data structure declarations

```
struct Link {
  int data;     // [why not int*?]
  Link* next;   // [why not Link?]
};

Link* emptyList = NULL;
```


An operation

```
Link* addFirst(Link* list, int data) {
  Link* newLink = new Link;
  // C: ... = (Link*) malloc(sizeof(Link))
  newLink->data = data;
  newLink->next = list;
  return newLink;
}
```


Why not this?

```
Link* addFirst(Link* list, int data) {
  Link newLink;   // faster: no heap alloc!
  newLink.data = data;
  newLink.next = list;
  return &newLink;
}
```


Another operation

```
Link* addLast(Link* list, int data) {
  List* lastLink = findLastLink(list);
  if (lastLink == NULL) {  // empty list
      return addBefore(list, data);
  } else {  // non-empty list
      addAfterLastLink(lastLink, data);
      return list;
  }
}
```


A helper

```
void addAfterLastLink(Link* lastLink, int data) {
  Link* newLink = new Link;
  newLink->data = data;
  newLink->next = NULL;
  assert(lastLink->next == NULL);
  lastLink->next = newLink;
}
```

Another helper

```
Link* findLastLink(Link* list) {
  if (list == NULL) {  // empty list
      return NULL;
  } else if (list->next == NULL) {  // last link
      return list;
  } else {
      return findLastLink(list->next);
  }
}
```


A non-recursive version

```
Link* findLastLink(Link* list) {
  if (list == NULL) {  // empty list
      return NULL;
  } else {
      while (list->next != NULL) {
          list = list->next;
      }
      return list;
  }
}
```


Another operation

```
List* findItem(List* list, int data) {
  if (list == NULL) {
      return NULL;  // NULL == not found
  } else if (list->data == data) {
      return list;  // found it
  } else {
      return findItem(list->next, data);
  }
}
```


A non-recursive version

```
List* findItem(List* list, int data) {
  for (;;) {
      if (list == NULL) {
          return NULL;  // NULL == not found
      } else if (list->data == data) {
          return list;   // found it
      }
      list = list->next;
  }
}
```


An improvement: list header

- Add an extra structure that points to the first and last Links in the list, for faster addLast behavior

```
struct List {
  Link* first;
  Link* last;
};
```


Revised operation

```
List* addLast(List* list, int data) {
  if (list == NULL) {  // empty list
      return addFirst(list, data);
  } else {  // non-empty list
      addAfterLastLink(list->last, data);
      list->last = list->last->next;  // [why?]
      return list;
  }
}
```

Another revised operation

```
List* addFirst(List* list, int data) {
   Link* newLink = new Link;
   newLink->data = data;
   if (list == NULL) {  // need to create the list
        list = new List;
        list->first = NULL;  list->last = newLink;
   }
   newLink->next = list->first;
   list->first = newLink;
   return list;
}
```


Doubly-linked lists

- Extend with a previous link

```
struct DLink {
        int data;
        DLink* prev;
        DLink* next;
};
```

- An exercise for the reader…
  - Lots of fun pointer surgery & splicing!

Assignment

- Consider x = y
- In Java, this makes x refer to whatever y refers to
  - x and y **share** the object
- In C, this **shallow-copies** y to x
  - if x & y are numbers, they're copied
  - if x & y are pointers, then the pointer is copied, but not what's pointed to
  - if x & y are structs, then the whole struct is copied, but not anything pointed to by that struct

An example

- List list1;
- List list2;
- … // a bunch of operations to build list1
- list2 = list1; // what does this do?
- … // a bunch of ops to extend list1
- // now what's the state of list1? list2?

A variation

- List* list1;
- List* list2;
- … // a bunch of operations to build list1
- list2 = list1; // what does this do?
- … // a bunch of ops to extend list1
- // now what's the state of list1? list2?

Tips

- Watch out for assignments doing (partial) copies behind your back
  - Using pointers to non-trivial data structures avoids this problem
- It's good to define your own (deep) copy functions that copy exactly what you want copied to duplicate the *abstract* state of your data structure