CSE 143 SUMMER 1998

Pointer-Based Data Structures
[8.1-8.6, 8.8]



## Boxes and Arrows

- A convenient (and important) visual notation for manipulating objects and pointers
- A box represents a value in memory
- If a box is named by a variable, that box will have a name tag attached to it
- Boxes contain data: numbers, strings, even pointers to other boxes!
- An arrow starts inside one box and points to some other box



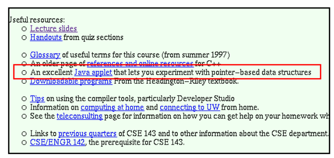
## Example of Boxes and Arrows

```
struct Employee {
    int    id;
    double salary;
};

Employee e1;
e1.id = 1234;
e1.salary = 6.75;

Employee *e2 = new Employee;
e2->id = 9667;
e2->salary = 9.25;

Employee *e3 = &e1;
e2->salary = e3->salary + 2.00;

delete e2;
e2 = NULL;
```



## Learning More

- Follow this link:


Useful resources:
- Lecture slides
- Handouts from quiz sections
- Glossary of useful terms for this course (from summer 1997)
- An older page of references and online resources for C++
- An excellent Java applet that lets you experiment with pointer–based data structures
- Downloadable programs From the Headington–Riley textbook.
- Tips on using the compiler tools, particularly Developer Studio
- Information on computing at home and connecting to UW from home.
- See the teleconsulting page for information on how you can get help on your homework wh
- Links to previous quarters of CSE 143 and to other information about the CSE department.
- CSE/ENGR 142, the prerequisite for CSE 143.



## Pointers Inside Structures

- A **struct** or **class** certainly can't contain another instance of itself as a member:

```
struct Employee {
    int      id;
    double   salary;
    Employee manager;
};
```

- But it can contain a *pointer* to another instance!

```
struct Employee {
    int      id;
    double   salary;
    Employee *manager;
};
```



## Using Pointers to Other Instances

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

Node *n = new Node;

n->data = 15;
n->link = new Node;

n->link->data = 192;
n->link->link = new Node;

n->link->link->data = -99;
n->link->link->link = NULL;
```

## Putting it Into a Function

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

Node *cons( int data, Node *rest )
{
    Node *ret = new Node;
    ret->data = data;
    ret->link = rest;
    return ret;
}

Node *n = NULL;

n = cons( 53, n );
n = cons( 59, n );
n = cons( 61, n );
```



## Putting it Into a Class (I)

```
#ifndef __COLLECTION_H__
#define __COLLECTION_H__

struct Node {
    Node( int d, Node *l );

    int   data;
    Node  *link;
};
```

```
class Collection {
public:
            Collection();

    bool    noFirst();

    void    cons( int item );
    int     unCons();
    int     getFirst();

private:
    Node    *first;
};

#endif // __COLLECTION_H__
```



## Putting it Into a Class (II)

```
#include "collection.h"

Node::Node( int d, Node *l )
    : data( d )
    , link( l )
{}

Collection::Collection()
    : first( NULL )
{}

bool Collection::noFirst()
{
    return first == NULL;
}
```

```
void Collection::cons( int item )
{
    first = new Node( item, first );
}

int Collection::unCons()
{
    Node *tmp = first;
    first = first->link;
    int ret = tmp->data;
    delete tmp;
    return ret;
}

int Collection::getFirst()
{
    return first->data;
}
```



## Back to Lists

- Now that we've got the power of dynamic memory, it's time to go back and reimplement lists
- We'll implement the List ADT using the **linked list** data structure
  - Remember that the ADT is not the same as the data structure



```
#ifndef __LIST_H__
#define __LIST_H__

class IntList
{
public:
    IntList();

    bool isEmpty();
    bool isFull();
    int getSize();

    void start();
    void advance();
    bool atEnd();
    int getData();

    void insertBefore( int item );
    void insertAfter( int item );
    void deleteItem();
private:

};

#endif // __LIST_H__
```



```
#include "list.h"

IntList::IntList()
{

}

bool IntList::isEmpty()
{

}
```

```
void IntList::start()
{
}
void IntList::advance()
{
}
bool IntList::atEnd()
{
}
int IntList::getData()
{
}
```



```
void IntList::insertAfter( int item )
{
}
```



## Deleting From a List

- Why can't we just use **`cursor`** to delete from the list?
- How can we solve this problem?
- Note that **`insertBefore`** will also need to use this solution



```
void IntList::deleteItem()
{
}
```



## Doubly-Linked Lists

- Searching from start of list to find element before cursor is slow
- Idea: each list node contains pointers in two directions: forwards and backwards

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

- Moving forwards or backwards now easy, but more work is necessary to update pointers



## Arrays vs. Linked Lists

- Advantages of array-based implementation
  - Fixed upper bound on size
  - Low overhead for each list item
  - Possibility of direct access in implementation
- Advantages of linked-list implementation
  - Size can grow without bounds
  - More efficient insertion and deletion possible
- Best choice can depend on application

## Summary

- Boxes and arrows help to visualize linked data structures
- Structures (and classes) can contain links to other instances of themselves
- Linked implementation of stack ADT
- Linked implementation of List ADT
- Doubly-linked lists