Exam 1 Next Tuesday

• In class, 1 hour and 50 minutes.
• Closed-book, closed-notes
• Covers material up through C++ strings and their implementation
• Questions may include:
  – A few definitions
  – Predicting output from code segments
  – Declaring classes
  – Writing functions, member functions, and short code segments
• Syntax is important, although we will try not to penalize minor syntax mistakes.
• Review notes, labs, homeworks. Redo examples on your own.
• Use Prof. Kettnaker’s homeworks for extra practice.

Review from Tuesday’s Lecture

• Applications of the C++ string class
• C strings vs. C++ strings.
• Defining the string class:
  – Constructors and destructors
  – Deep vs. shallow copying: copy constructors and assignment operators.
  – String operations
Today’s Class

• An abstract view of lists and list operations

• Linking objects

• Fundamental linked list operations:
  – Stepping through the list
  – Appending to the end
  – Inserting an item
  – Removing an item

• SetOfInt revisited

Reading

Carrano and Prichard, pages 173-195. This contains an excellent discussion of the basics of linked list manipulations.

Lists

• We have defined an abstract Set (of integers) class.

• We can define an abstract List class, as has been done in the Carrano and Prichard text.

• Lists can be thought of ordered sets that can contain repeated objects. They include the following operations
  – Insert at a specified location
  – Remove item from a specified location
  – Access item at specified location

• Arrays are the only implementation mechanism that we have discussed so far, but that is about to change.

• Our new mechanism is through dynamic allocation of objects and objects having pointers to other (dynamically allocated) objects.

• Sophisticated data structures can be built using this idea. For now, though, we consider only linear linking to form linked lists.
Objects with Pointers, and Linking Objects

The two fundamental mechanisms of linked lists are

- to create objects with pointers as one of the member variables, and
- to make these pointers point to other objects of the same type.

These mechanisms are illustrated in the following short program.

```cpp
#include <iostream>
using namespace std;

class Node {
public:
    int value;
    Node* ptr;
};

void main()
{
    Node* ll; // ll is a pointer to a (non-existent) Node.
    ll = new Node; // Create a Node and assign its memory address to ll
    ll->value = 6; // Note, this is the same as (*ll).value = 6;
    ll->ptr = 0; // The value 0 indicates a "null" pointer.

    Node* q = new Node;
    q->value = 8;
    q->ptr = 0;

    ll->ptr = q; // ll’s ptr member variable now has the same value
                 // at the pointer variable q

    cout << "1st value: " << ll->value << "\n"
         << "2nd value: " << ll->ptr->value << endl;
}
```

The following picture illustrates the structure of memory at the end of the program.
Definition: A Linked List

A linked list is a data structure where

- There is a pointer, usually called the head pointer, to the first node.
- Each node is a class object, and each node contains a pointer to the next node in the list.
- The last node contains a null pointer.
- If the list is empty, the head pointer contains 0 as its value.

Visualizing Linked Lists

- The head variable is drawn with its own box. It is an individual variable.
- The objects (nodes) that have been dynamically allocated and stored in the linked lists are shown as boxes, with arrows drawn to represent pointers.
  - Note that this is a conceptual view only. The memory locations could be anywhere.
- The last node must have a 0 for its pointer value.
• You should make a habit of drawing pictures of linked lists to figure out how to do the operations.

Basic Mechanisms: Stepping Through
Goal: access every value in the linked list.
• We will illustrate by adding the values.
• We will think through the code in class.
• In all examples, we will assume that we have a head pointer variable.

Basic Mechanisms: Append
Goal: place a new node at the end of the list.
• We must step to the end of the linked list, remembering the pointer to the last node.
• We must create a new node and attach it to the end.
• We must remember to update the head variable’s value if the list is initially empty.

Basic Mechanisms: Inserting a Node
• There are two parts to this: finding the location where the insert must take place, and doing the insert operation.
• The find operation depends on the ordering (if any) of the list. We will see an example where the list is ordered in SetOfInt.
• The insert operation itself requires that we have a pointer to the location before the insert location.
• Inserting at the beginning, at the end, and to an empty list are all important special cases. At least you must be check that your code works on these. You might also need to write special case code to handle them.
Basic Mechanisms: Removing a Node

- There are two parts to this: finding the node to be removed and doing the remove operation.
- The remove operation itself requires a pointer to the node **before** the node to be removed.
- Removing the first node is an important special case.

Basic Mechanisms: Common Mistakes

- Allocating a new node to step through the linked list; only a pointer variable is needed.
- Confusing the `. ` and the `->` operators.
- Not setting the pointer from the last node to 0 (null).
- Not considering special cases of inserting / removing at the beginning or the end of the linked list.
- Deleting a node before it is removed. Delete should be done after all pointer manipulations are completed.
- Pointer manipulations that are out of order. These can ruin the structure of the linked list.

SetOfInt Revisited

We have provided a new implementation of the SetOfInt class where the public interface has not changed, but the implementation now uses a linked list.

- The linked list is ordered by increasing element now.
- The only member variable is the pointer to the head of the internally represented linked list.
- There is an auxiliary SetOfIntNode class with all public members variables.

  - Auxiliary classes like this are the only case where public member variables are appropriate.
• The insert and remove functions are provided and will be discussed in class. We will concentrate on finding the right location and then doing the pointer operations.

• Simple versions of union and intersection are also provided. We will look at faster, more sophisticated versions next week.

• You will have to implement many of the other functions in next week’s lab.

Summary of Major Ideas

• ADT set and ADT list

• Pointers to class objects, and class objects with pointers.

• Linked list structure

• Fundamental operations: stepping through the list, appending a node, inserting a node, removing a node.

• Common mistakes.

• SetOfInt class, where the head pointer is the only member variable and the linked list operations are hidden inside the member functions.