ACCESSIBILITY

• We’ve had classes where everything is accessible to anyone using our classes. This is not ideal because it means that individuals will likely use our classes in ways we did not intend.

• In other words, we need to be able to ensure that our invariant is always true.

• We can restrict what someone outside of the class can see using the private and public keywords:

```cpp
struct Node {
    int val;
    private:
        Node* next;
};
```
• Everything after private: and before public: will only be accessible within the class, i.e., within methods. Everything which is public will be accessible to everyone as before.

• C++ has the class keyword, which is like a struct, but the default accessibility is private:

```cpp
class Node {
    Node* next;
    public:
        int val;
};
```

• The accessibility restriction can be bypassed with the friend keyword
class LinkedList {

    ... // Pretty prints the LinkedList

    // the friend keyword allows access of LinkedList's private fields
    friend std::ostream &operator<<(std::ostream &out, const LinkedList &rhs);
}

In CS 247, you should not use the friend keyword unless instructed otherwise or we provide you code that uses friend. In real life, you should restrict the use of the friend keyword as much as possible.
NESTED CLASSES

• We may want to create a class which doesn’t make sense to exist on its own. An example is implementing a wrapper class for some structure to restrict others’ ability to alter the class.
• A good example of this would be creating a wrapper class around the Node class we implemented.
class LinkedList {
    struct Node {
        int val;
        Node* next;
    }
    int numNodes;
    Node *head;
    Node *tail;

public:
    LinkedList();
    LinkedList(int amount, int what); // fill constructor
    void insertHead(int value);
    void insertTail(int value);
    void remove(int index);
};
Since Node is declared within LinkedList, when we refer to the Node struct in our source code, we will refer to it as LinkedList::Node. This states that we are using the Node struct which is part of the LinkedList class.

Note that since the Node struct is private within the LinkedList class, we will not be able to create instances of Nodes outside of the LinkedList class. Our Nodes are safe from others tampering with our LinkedList Nodes.

* Scope Resolution operator
An iterator provides a way to iterate (i.e., loop through) the elements of a collection of objects.

- Doesn’t expose the underlying representation of the collection.
- Moves responsibility for access and traversal order into a separate “iterator” object.

**Motivation:**

- Want to apply a function (e.g., `transform()`) to all items in collection.
- Want to be able to have multiple, simultaneous traversals of the collection.
Pre-order

In Order

Post-order
ITERATORS

• Iterators are used to traverse containers in some order.
• Such order can be specified by the definition of the iterator, or not specified in any order.
• In C++, iterators are usually implemented with the following functions with in the container class (Container) and the nested iterator class (Container::Iterator):
  • – Container::begin() — returns the iterator that represents the beginning of the iteration sequence.
  • – Container::end() — returns the iterator that represents the end of the iteration sequence (which is NOT included in the elements being iterated)
• – Container::Iterator::operator++() (prefix) — increment the iterator by one, and return the incremented iterator.

• – Container::Iterator::operator!=(const Iterator &other) — returns true if two iterators does not represent the same element, false otherwise.

• – Container::Iterator::operator*(()) (unary) — return a reference of the element being represented by the iterator. It could also be a const reference, but what would you lose?

• The above is crucial to support the range-based for loop for objects:

  // Assume that Container iterator iterates through objects of type T Container c;
  for (T &v : c) { // You can change v here. v.modify(); cout << v << endl; }

• See code material for implementation of iterators for linked lists
ITERATOR DESIGN PATTERN

• Who Controls the Iteration?
• Who defines the traversal algorithm?
• How robust is the Iterator?
• Iteration operations ( ++ , != , * )
• Additional Iterator operations – First, Next, IsDone, CurrentItem
• * Iterators may have privileged access