Announcements
- Program due Sunday by midnight
- Next program is posted by due Tuesday the 12th
Recap of Vectors:

Idea: extend arrays so that they grow when needed

But keep things efficient
Running times

Constructor: $O(1)$
Operator []: $O(1)$
Destructor: $O(1)$
Insert: $O(N)$ if $N$ elements in vector
Remove: $O(N)$
Push_back: $O(N)$
Proposition: The running time of making $N$ push-back operations in an empty array is $O(N)$.

Initially

Proof: (called Amortized analysis)

Virtual dollars make every function call "pay" for its time.

If we don't double the array, each call will cost $1.

How much do the doubling calls take?
Instead of $1, I'm going to charge $3.

Bank account: $3 \cdot 2^{i+1} - 1 \cdot 2^{i-1} = 2 \cdot 2^{i+1} = 2^i$

2^{i+1} \rightarrow \text{2\textsuperscript{i+1} insertions to fill this array}$

2^i \rightarrow \text{2\textsuperscript{i} insertions to fill this array}$
Linked lists

Motivation: The running time of insert in a vector is awful.

Idea: If we know where an element should go, inserting should be faster.
Doubly Linked List: Insert

Operations
new
4 pointers
O(1) time
Problem: What do we need the user to have in order to implement `insert`?

Need to specify a node.
But the user can't know about Nodes!

Solution: Iterator
An iterator will give the user a “pointer”, but with a heavily controlled structure (so they can’t manipulate the nodes directly).

Compromise between hiding the underlying data & allowing the user to specify a pointer directly.
template <typename ItemType>

class List {

protected:

struct Node {
    ItemType _data;

    Node* _prev;
    Node* _next;

    Node (const ItemType& data, Node* next, Node* prev) {
        _data = data;
        _next = next;
        _prev = prev;
    }

};
**Iterator class**: What should we code?

```cpp
public: // in list class

class iterator {

private:

    Node* _current;

public:

    iterator(): _current(NULL) {}  // defaulted constructor

    iterator(const iterator & other): _current(other._current) {} // copy constructor

};
```
// takes an iterator `it` points to front of the list

```cpp
void front (iterator& it) {
    it = _current = _front;
}
```

```cpp
const ItemType& operator*() const {
    return _current->_data;
}
```

```cpp
iterator operator++ () {
    _current = _current->_next;
    return *this;
}
```