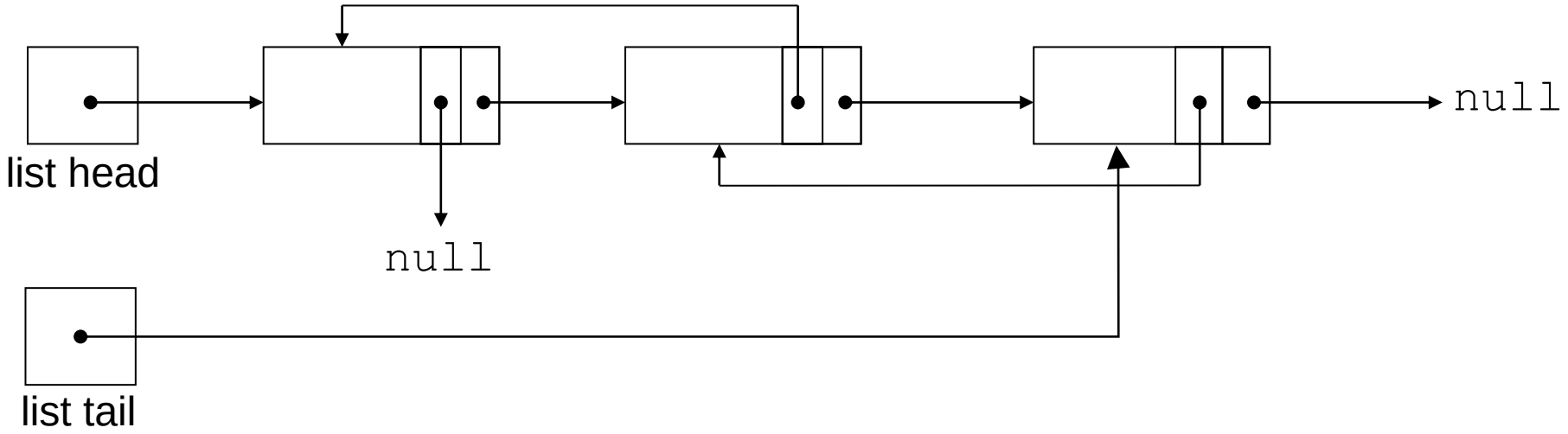
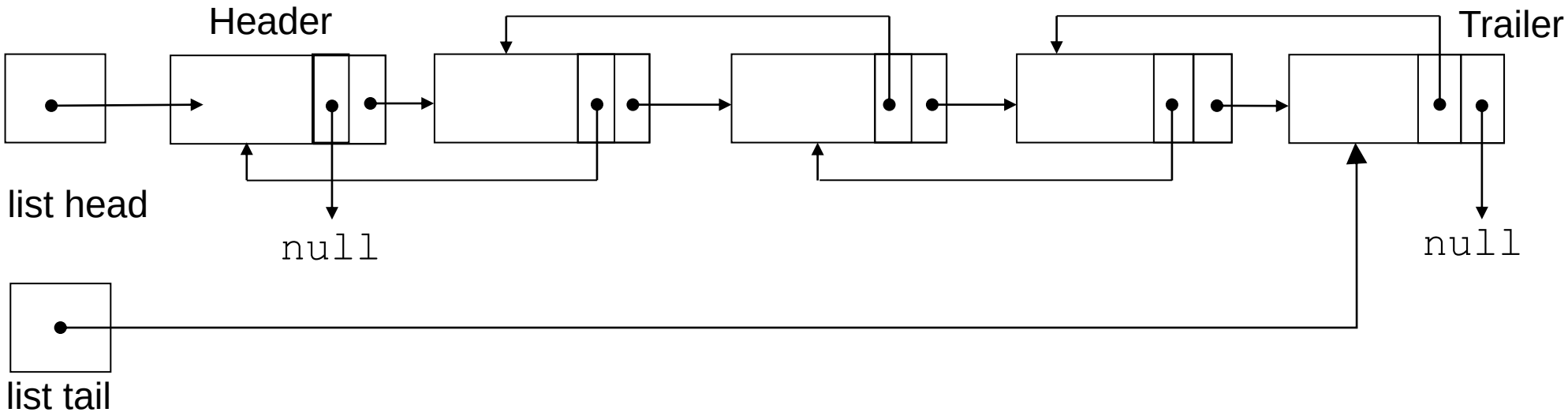
Doubly linked lists



- Each node contains two pointers: one to the next node in the list, one to the previous node in the list

Doubly linked lists with sentinels



- **Sentinel nodes** may be used in order to have more uniform operations, i.e. no special handling of empty lists.
  - In the lab, we'll add two sentinel nodes; a header node + a trailer node.

Doubly linked lists with sentinels

Ranged for support

```
for (const int* iter = vec.begin(); iter != vec.end(); ++iter){
    int val = *iter;
    cout << val << ' ';
}

for (int val : vec)
    cout << val << ' ';
```

Ranged for support – cont.

To support ranged for loops

- The class must provide begin() and end() functions that return an iterator (a pointer in our case)
  - One version that is for getter (returns a const int*)
  - Another version that is a setter (returns int*)

- The iterator must be able to support the following functions:
  - Increment and decrement the iterator
  - Compare operations
  - Dereference operations

Ranged for support

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

class Vector {
public:
    // default constructor
    Vector() : data(nullptr), theSize(0), theCapacity(0) {}

    explicit Vector(size_t howMany, int val = 0) {
        theSize = howMany;
        theCapacity = howMany;
        data = new int[howMany];
        for (size_t i = 0; i < theSize; ++i) {
            data[i] = val;
        }
    }
    ////////////////////////
    // Vector Copy Control //
    ////////////////////////
        // Destructor
    ~Vector() {
        delete[] data;
    }
    // Copy constructor
    Vector(const Vector& rhs) {
        theSize = rhs.theSize;
        theCapacity = rhs.theCapacity;
        data = new int[theCapacity];
        for (size_t i = 0; i < theSize; ++i) {
            data[i] = rhs.data[i];
        }
    }
    // Assignment operator
    Vector& operator=(const Vector& rhs) {
        if (this != &rhs) {
            // Free up the old (destructor)
            delete[] data;
            // Allocate new
            data = new int[rhs.theCapacity];
            // Copy all the stuff
            theSize = rhs.theSize;
            theCapacity = rhs.theCapacity;
            for (size_t i = 0; i < theSize; ++i) {
                data[i] = rhs.data[i];
            }
        }
        // Return ourselves
        return *this;
    }
```

Ranged for support – simple pointer iterators

```
 void push_back(int val) {
        // Handle if we have run out of space.
        if (theSize == theCapacity) {
            // Do we have a zero coapacity vector?
            // Special case becasue doubling zero wouldn't help.
            if (theCapacity == 0) {
                theCapacity = 1;
                data = new int[theCapacity];
            }
            else {
                // Remember the old array
                int* oldData = data;
                // Allocate a new array with twice the capacity
                theCapacity *= 2;
                data = new int[theCapacity];
                // Copy over the POINTERS. This is NOT a deep copy.
                for (size_t i = 0; i < theSize; ++i) {
                    data[i] = oldData[i];
                }
                // Free up the old arrayl
                delete[] oldData;
            }
        }
        // Now we know there is enoubh space.
        // theSize is already the index for the new entry
        data[theSize] = val;
        // And bump theSize now that we have a new entry added
        ++theSize;
    }
    size_t size() const { return theSize; }
    void pop_back() { --theSize; }
    void clear() { theSize = 0; };
    int operator[](size_t index) const {
        return data[index];
    }
    int& operator[](size_t index) {
        return data[index];
    }
    int* begin() { return data; }
    int* end() { return data + theSize; }
    const int* begin() const { return data; }
    const int* end() const { return data + theSize; }

private:
    int* data;
    size_t theSize;
    size_t theCapacity;
};
```

- Note that overloading is based on the const

Does not allow modification

Does allow modification

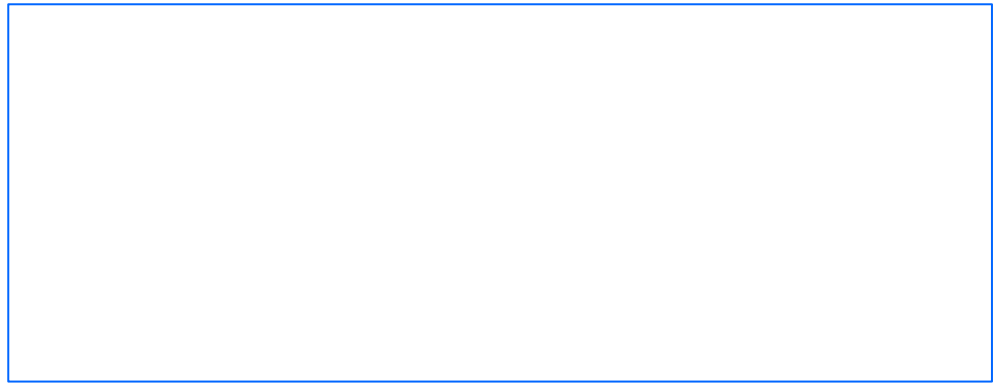
- Iterators allow modification of the Vector
- const_Iterator does not allow modification

Ranged for support – simple pointer iterators

```
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) os << val << ' ';
    return os;
}


void printVec(const Vector& vec) {
    // Requires const versions of begin and end.
    cerr << "printVec: displaying using ranged for:\n";
    for (int val : vec) cout << val << ' ';
    cout << endl;

    cerr << "printVec: displaying using const int*\n";
    for (const int* iter = vec.begin(); iter != vec.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```



Ranged for support – simple pointer iterators

```
int main() {

    Vector v;

    v.push_back(17);
    v.push_back(42);
    v.push_back(6);
    v.push_back(28);
    cerr << "Using indices to access Vector v contents:\n";
    for (size_t i = 0; i < v.size(); ++i) {
        cout << v[i] << ' ';
        //cout << v.operator[](i) << endl;
    }
    cout << endl;

    cerr << "Modifying v[0]\n";
    v[0] = 100;

    cerr << "Using copy constructor to initialize v2 from v\n";
    Vector v2 = v;
    cerr << "Using indices to access Vector v2 contents:\n";
    for (size_t i = 0; i < v2.size(); ++i) {
        cout << v2[i] << ' ';
    }
    cout << endl;
}
```

```
Using indices to access Vector v contents:
17 42 6 28
Modifying v[0]
Using copy constructor to initialize v2 from v
Using indices to access Vector v2 contents:
100 42 6 28
```

Ranged for support – simple pointer iterators

```
int main() {
    Vector v;
    v.push_back(17);
    v.push_back(42);
    v.push_back(6);
    v.push_back(28);
    Vector v2 = v;
    Vector v3;
    v3 = v;

    //v3 = 17;
    v3 = Vector(17);

    cerr << "============\n";
    cerr << "Using int* to access Vector v contents:\n";
    for (int* iter = v2.begin(); iter != v2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
    cerr << "============\n";
    cerr << "Using ranged for to access Vector v2 contents:\n";
    for (int val : v2) cout << val << ' ';
    cout << endl;
    cerr << "============\n";

    //    int* beginning = v2.begin();
    cerr << "Appending a value to v2\n";
    v2.push_back(17);

    cerr << "Calling printVec for v2\n";
    printVec(v2);

    cerr << "============\n";
    cerr << "Using operator<< to display v2:\n";
    cout << v2 << endl;
}
```

```
============
Using int* to access Vector v contents:
17 42 6 28
============
Using ranged for to access Vector v2 contents:
17 42 6 28
============
Appending a value to v2
Calling printVec for v2
printVec: displaying using ranged for:
17 42 6 28 17
printVec: displaying using const int*
17 42 6 28 17
============
Using operator<< to display v2:
17 42 6 28 17
```

Iterators

- With our “Vector” class, a simple pointer sufficed as an iterator
  - Because the underlying data is stored as an array.

- What about if we have a more complex data structure, whose underlying data is not stored as a contiguous array?
  - e.g. a tree or a linked list
  - → A **pointer won’t suffice**!
  - → Use iterators instead.

Iterators – cont.

An iterator object should allow us to:

- Access elements of a container by dereferencing.
  - i.e. not only vectors, but also linked lists, maps, etc.

- Iterate (move) over the container to access next, previous, first or last elements.
  - Incrementing the iterator needs to point to the next node
  - Vice versa for decrementing the iterator

- Compare iterators to assess if they are referring to the same element within a container.

Iterators

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

class Iterator {

    // Not needed, but we usually implement != in terms of ==
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return lhs.ptr == rhs.ptr;
    }
public:
    // Used by begin / end
    Iterator(int* ptr=nullptr) : ptr(ptr) {}

    // pre-increment. This is what the ranged for needs
    // Could certainly also implement post-increment
    Iterator& operator++() {
        ++ptr;
        return *this;
    }

    // dereference operator. Allows modification of the Vector
    // but not of the Iterator (that's what the const is there to say).
    int& operator*() const { return *ptr; }
private:
    int* ptr;
};
```

Iterator required functionality:

- constructor(s)
- != operator (implemented as non-friend using ==
- increment operator
- derefence operator

Post increment prototype :

Iterator& operator++(int dummy);

const iterators

```
class Const_Iterator {
    // Not needed, but we usually implement != in terms of ==
    friend bool operator==(const Const_Iterator& lhs, const Const_Iterator& rhs) {
        return lhs.ptr == rhs.ptr;
    }
public:
    // Used mostly by begin() const / end() const
    Const_Iterator(int* ptr=nullptr) : ptr(ptr) {}

    // pre-increment. This is what the ranged for needs
    // Could implement post-increment also
    Const_Iterator& operator++() {
        ++ptr;
        return *this;
    }

    int operator*() const { return *ptr; }
private:
    // Not mandatory that this be a pointer to const but it does
    // help say what we mean.
    const int* ptr;
};
```

The const on this method is only to, in principle, prevent the function from modifying the iterator itself, not the Vector

dereference operator.

- Does NOT allow modification of the Vector. The return by value of the int prevents changes.
  - Could also have been return by constant reference.

Iterators – cont.

```
class Vector {
public:
    // default constructor
    Vector() : data(nullptr), theSize(0), theCapacity(0)  {}

    explicit Vector(size_t howMany, int val=0)
    {
        theSize = howMany;
        theCapacity = howMany;
        data = new int[howMany];
        for (size_t i = 0; i < theSize; ++i) {
            data[i] = val;
        }
    }
    //
    // Vector Copy Control
    //
    // Destructor
    ~Vector() {
        delete [] data;
    }
    // Copy constructor
    Vector(const Vector& rhs) {
        theSize = rhs.theSize;
        theCapacity = rhs.theCapacity;
        data = new int[theCapacity];
        for (size_t i = 0; i < theSize; ++i) {
            data[i] = rhs.data[i];
        }
    }
    // Assignment operator
    Vector& operator=(const Vector& rhs) {
        if (this != &rhs) {
            // Free up the old (destructor)
            delete [] data;
            // Allocate new
            data = new int[rhs.theCapacity];
            // Copy all the stuff
            theSize = rhs.theSize;
            theCapacity = rhs.theCapacity;
            for (size_t i = 0; i < theSize; ++i) {
                data[i] = rhs.data[i];
            }
        }
        // Return ourselves
        return *this;
    }
```

Iterators – cont.

```
 void push_back(int val) {
// Handle if we have run out of space.
        if (theSize == theCapacity) {
        // Do we have a zero coapacity vector?
        // Special case becasue doubling zero wouldn't help.
            if (theCapacity == 0) {
                theCapacity = 1;
                data = new int[theCapacity];
            } else {
                // Remember the old array
                int* oldData = data;
                // Allocate a new array with twice the capacity
                theCapacity *= 2;
                data = new int[theCapacity];
                // Copy over the POINTERS. This is NOT a deep copy.
                for (size_t i = 0; i < theSize; ++i) {
                    data[i] = oldData[i];
                }
                // Free up the old arrayl
                delete [] oldData;
            }
        }
        // Now we know there is enoubh space.
        // theSize is already the index for the new entry
        data[theSize] = val;
        // And bump theSize now that we have a new entry added
        ++theSize;
    }

    size_t size() const { return theSize; }

    void pop_back() { --theSize; }

    void clear() { theSize = 0; };
```

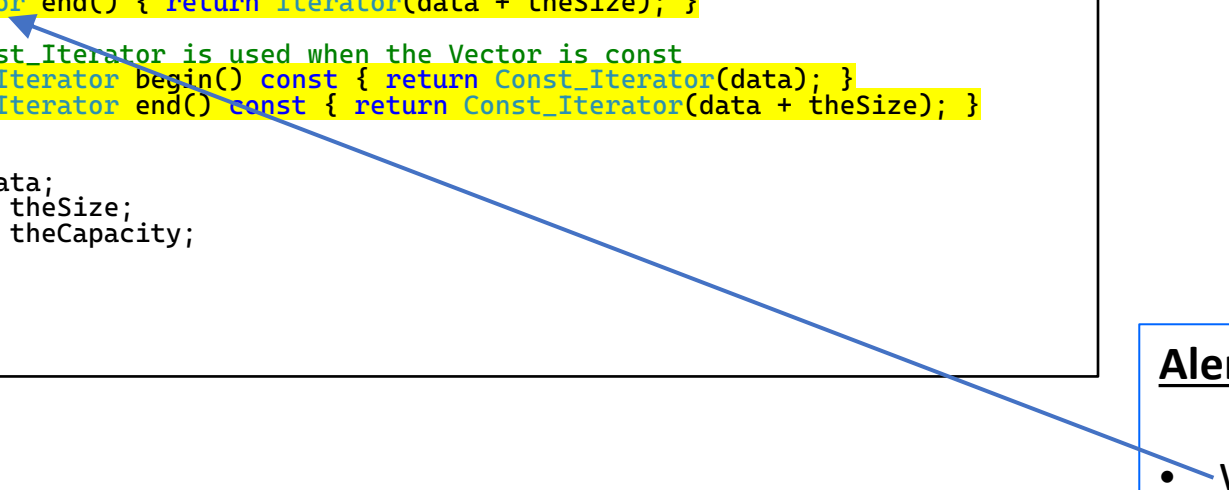
Iterators – cont.

```
 // Square bracket operators. Note that overloading is based on the const
    // op[] that does not allow modification
    int operator[](size_t index) const {
        return data[index];
    }
    // op[] that does allow modification
    int& operator[](size_t index) {
        return data[index];
    }

    // Iterators allow modification of the Vector
    Iterator begin() { return Iterator(data); }
    Iterator end() { return Iterator(data + theSize); }

    // Const_Iterator is used when the Vector is const
    Const_Iterator begin() const { return Const_Iterator(data); }
    Const_Iterator end() const { return Const_Iterator(data + theSize); }

private:
    int* data;
    size_t theSize;
    size_t theCapacity;
};
```



Alertness Test:

- Why not return Iterator& or const_Iterator& ??

Iterators – cont.

```
// This is what ranged for needs.
// Implemeting it in terms of op==, as is common.
bool operator!=(const Iterator& lhs, const Iterator& rhs) {
    return !(lhs == rhs);
}

// This is what ranged for needs.
// Implemeting it in terms of op==, as is common.
bool operator!=(const Const_Iterator& lhs, const Const_Iterator& rhs) {
    return !(lhs == rhs);
}

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) os << val << ' ';
    return os;
}

//
// User code
//

void printVec(const Vector& vec) {
    // Requires const versions of begin and end.
    cerr << "printVec: displaying using ranged for:\n";
    for (int val : vec) cout << val << ' ';
    cout << endl;

    cerr << "printVec: displaying using Const_Iterator:\n";
    for (Const_Iterator iter = vec.begin(); iter != vec.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

Which overload is picked?
Why?

Iterators – cont.

```
int main() {

    // Not templated.  Our Vector class can only hold ints.
    Vector v;

    v.push_back(17);
    v.push_back(42);
    v.push_back(6);
    v.push_back(28);
    cerr << "Using indices to access Vector v contents:\n";
    for (size_t i = 0; i < v.size(); ++i) {
        cout << v[i] << ' ';
        //cout << v.operator[](i) << endl;
    }
    cout << endl;

    cerr << "Modifying v[0]\n";
    v[0] = 100;

    //    v[0];
    //Vector v2(v);
    cerr << "Using copy constructor to initialize v2 from v\n";
    Vector v2 = v;
    cerr << "Using indices to access Vector v2 contents:\n";
    for (size_t i = 0; i < v2.size(); ++i) {
        cout << v2[i] << ' ';
    }
    cout << endl;

    Vector v3;
    v3 = v;

    //v3 = 17;
    v3 = Vector(17);

    cerr << "============\n";
    cerr << "Using Iterator to access Vector v contents:\n";
    for (Iterator iter = v2.begin(); iter != v2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
    cerr << "============\n";
    cerr << "Using ranged for to access Vector v2 contents:\n";
    for (int val : v2) cout << val << ' ';
    cout << endl;
    cerr << "============\n";

    //    int* beginning = v2.begin();
    cerr << "Appending a value to v2\n";
    v2.push_back(17);

    cerr << "Calling printVec for v2\n";
    printVec(v2);

    cerr << "============\n";
    cerr << "Using operator<< to display v2:\n";
    cout << v2 << endl;
}
```

```
Using indices to access Vector v contents:
17 42 6 28
Modifying v[0]
Using copy constructor to initialize v2 from v
Using indices to access Vector v2 contents:
100 42 6 28
============
Using Iterator to access Vector v contents:
100 42 6 28
============
Using ranged for to access Vector v2 contents:
100 42 6 28
============
Appending a value to v2
Calling printVec for v2
printVec: displaying using ranged for:
100 42 6 28 17
printVec: displaying using Const_Iterator:
100 42 6 28 17
============
Using operator<< to display v2:
100 42 6 28 17
```

Iterators – cont.

Question:

- If you wish to create an iterator class, what will be the name you would use intuitively?

Iterators – cont.

Question:

- If you wish to create an iterator class, what will be the name you would use intuitively?

```
Iterator
```

Iterators – cont.

Question:

- If you wish to create an iterator class, what will be the name you would use intuitively?

```
Iterator
```

- Okay, what happens if we have multiple containers that we wish to use, and each has decided to use the same intuitive name `Iterator`? What is the solution?