Linked-list of ints

```
// ListNode.h
#include <iostream>

class ListNode {
public:
    ListNode(int val, ListNode *nxt)
      : data(val), next(nxt) {}

//private:
    //usually private but public for this example
    int data;
    ListNode *next;
};
```

Linked-list of ints

```cpp
// ListNodeMain.cpp

#include <iostream>
#include <string>
#include "ListNode.h"

int main() {
    ListNode l3(3, nullptr);
    ListNode l2(2, &l3);
    ListNode l1(1, &l2);

    //Run through all items in list, output them one by one
    for (ListNode* cur = &l1; cur != nullptr; cur = cur->next) {
        std::cout << cur->data << " ";
    }
    return 0;
}
```

```
$ g++ -o ListNodeMain ListNodeMain.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./ListNodeMain
```

```
1 2 3
```

class MyVector

```
// MyVector.h
#include <iostream>
#include <string>

class MyVector {
public:
    MyVector(): data(new int[5]), capacity(5), num_elts(0) { }
    void add(int item);

//private:
    //but public for this example
    int* data;
    int capacity;
    int num_elts;
};

void MyVector::add(int item) {
    if (num_elts >= capacity) {
        /* then double the size of the array - code not shown */
    }
    data[num_elts++] = item;
}
```

class MyVector

```cpp
// MyVectorMain.cpp

#include <iostream>
#include "MyVector.h"

int main() {
    MyVector v = MyVector();
    v.add(1);
    v.add(2);
    v.add(3);

    //Run through all items in list, output them one by one
    for (int i = 0; i != v.num_elts; i++) {
        std::cout << v.data[i] << " ";
    }
    return 0;
}
```

```
$ g++ -o MyVectorMain MyVectorMain.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./MyVectorMain
```

```
1 2 3
```

Iterators

- In both `class`es, we needed to loop over all elements in the "container"
  - In our example, we printed items, but we might have been, say, searching for a value
- Code to "run through all elements" looks very different (`cur` pointer that advances through linked list vs. for loop over integer indices of vector)
- C++ iterators unify these different code segments
  - Regardless of the container specifics, an iterator feels like a pointer to successive individual elements, that we can easily advance
- Iterators encapsulate the iteration logic. As a user, we don't need to care about how the iteration is done.

Iterators

There are different iterators:

- We use an `iterator` over a container to traverse elements in the container in order from beginning to end
- A `reverse_iterator` can be used to traverse elements in a backwards direction
- A `const_iterator` is an iterator which promises not to modify individual elements as it progresses through them

They are provided for the container `class`es in STL.

- Suppose we write a new container class from scratch to represent, say, a deck of cards.
  - It would be nice to have an iterator for the deck!
- Let's write one...

Define our own iterator

- Can we just use a pointer as our iterator?
  - A pointer might work for a container where elements are laid out contiguously in memory, e.g. for an array
  - But a pointer doesn't work well for say, std::map. How would ++it advance properly?
- Instead, we actually define an entirely new class to represent an iterator...
- We can write our own iterator (or const_iterator or reverse_iterator) as a **nested class** inside the container class
- A nested class sits inside another class definition, and has access to the members of the enclosing class, including private members
  - For our purposes, we don't need access to the private members; each iterator class simply wraps a layer of operator overloads around a pointer

Usage of an iterator

Suppose we want to output the elements in MyContainerType c, we use the iterator in this way:

```
for (MyContainerType::iterator it = c.begin(); it != c.end(); ++it) {
    //*it can now be used to refer to each successive element
    std::cout << *it << " ";
}
```

Therefore, we at least need to overload:

- inequality operator (operator!=)
- dereference operator (operator*)
- preincrement operator (operator++)

A real-world iterator might also overload:

- equality operator (operator==)
- arrow operator / class member access operator (operator->)

Define our own `iterator`

- In addition to overload these operators, the enclosing (container) class (`MyContainerType`) should also define methods named `begin` and `end`, which return iterators to the first item in the collection, and the just-past-last element in the collection,respectively
- If you are defining a `const_iterator`, then it should have `cbegin` and `cend`. Similarly for `reverse_iterator`, it should have `rbegin` and `rend` defined.
- When defining a `const_iterator`, it should have a different overloaded `operator*`
- When defining a `reverse_iterator`, it should have a different overloaded `operator++`