Name: ____

Today's Date: ____

Today's Goals

- Prepare to *implement* an Iterator
- Prepare to *implement* a linked list in C++

Today's Question(s)

Why should every data structure we write have a corresponding `Iterator` defined?

Lingering Questions

Iterators

(Class worksheet)

Linked Lists

(`IntList` specification)

using_barn.cpp

```
#include <iostream>
#include <cstdlib>
#include <string>
#include "barn.hpp"
using std::cout, std::endl;

int main()  {
    Barn b;
    b.addCow("bessie");
    b.addCow("mabel");

    Barn c;
    c.addCow("mabel");
    c.addCow("bessie");

    if (b == c) {
        cout << "Uh-oh! These should not be equal." << endl;
    }

    for (Barn::iterator i = b.begin(); i != b.end(); ++i) {
        cout << *i << endl;
    }

    Barn::iterator i = b.findCow("bessie");
    if (i != b.end()) {
        cout << "Found " << i->getName() << " in the barn!" << endl;
    }
}
```

cow.hpp

```
#ifndef COW_HPP_INCLUDED
#define COW_HPP_INCLUDED

#include <iostream>
#include <string>

class Cow {
 public:
    Cow() = default;
    ~Cow() = default;
    explicit Cow(const std::string& cowName);

    std::string getName() const;
    bool operator==(const Cow& other) const;
    bool operator!=(const Cow& other) const;
    friend std::ostream& operator<<(std::ostream& output, const Cow& p);
 private:
    std::string name_;
};
#endif  // cow_hpp_included
```

barn.hpp

```
#ifndef BARN_HPP_INCLUDED
#define BARN_HPP_INCLUDED

#include <string>
#include "cow.hpp"

class Barn {
 private:
    class Iterator;
 public:
    using iterator = Iterator;

    Barn();
    Barn(const Barn& otherBarn) = delete;
    Barn& operator=(const Barn&  otherBarn) = delete;
    ~Barn();
    bool operator==(const Barn& other) const;
    bool operator!=(const Barn& other) const;
    iterator begin() const;
    iterator end() const;
    iterator addCow(const std::string& cowName);
    iterator findCow(const std::string& cowName) const;
 private:
    Cow** cows_;
    size_t size_;
    size_t capacity_;

    class Iterator {
     public:
        using value_type = Cow;
        using reference = value_type&;
        using pointer = value_type*;
        using difference_type = ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;

        Iterator() = default;
        Iterator(const Iterator& other) = default;
        ~Iterator() = default;
        Iterator& operator=(const Iterator& other) = default;
        Iterator& operator++();
        reference operator*() const;
        bool operator==(const Iterator& other) const;
        bool operator!=(const Iterator& other) const;
        pointer operator->() const;
     private:
        friend class Barn;
        Cow** here_;
        explicit Iterator(Cow** here);
    };
};
#endif  // BARN_HPP_INCLUDED
```

barn.cpp

```
#include <string>
#include "barn.hpp"
#include "cow.hpp"

using std::string;

Barn::Barn() : cows_{new Cow*[4]}, size_{0}, capacity_{4}  {
    // nothing (else) to do
}

Barn::~Barn() {
    for (size_t i = 0; i < size_; ++i) {
        delete cows_[i];
    }
    delete[] cows_;
}

Barn::iterator Barn::addCow(const string& cowName) {
    if (size_ == capacity_) {
        capacity_ *= 2;
        Cow** oldcows = cows_;
        cows_ = new Cow*[capacity_];
        for (size_t i=0; i < size_; ++i) {
            cows_[i] = oldcows[i];
        }
        delete[] oldcows;
    }
    cows_[size_] = new Cow{cowName};
    ++size_;
    return Iterator{cows_ + size_-1};
}

Barn::iterator Barn::findCow(const string& cowName) const {
    for (iterator i = begin(); i != end(); ++i) {
        if ((*i).getName() == cowName) {
            return i;
        }
    }
    return end();
}

bool Barn::operator==(const Barn& other) const {
    /*
     How can we determine if two Barns are equal
     to each other? How does having an Iterator
     for the Barn class help us determine if the
     Barns are equal?
    */
}

bool Barn::operator!=(const Barn& other) const {
    return !(operator==(other));
}

Barn::iterator Barn::begin() const {
    return Iterator{cows_};
}

Barn::iterator Barn::end() const {
    return Iterator{cows_ + size_};
}

Barn::Iterator::Iterator(Cow** here)
  : here_{here} { }

Barn::Iterator& Barn::Iterator::operator++() {
    ++here_;
    return *this; }

Barn::iterator::reference Barn::Iterator::operator*() const {
    return **here_;
}

bool Barn::Iterator::operator==(const Iterator& other) const {
    return here_ == other.here_;
}

bool Barn::Iterator::operator!=(const Iterator& other) const {
    return !(operator==(other));
}

Barn::iterator::pointer Barn::Iterator::operator->() const {
    return *here_;
}
```

Overview

This file describes the interfaces and encodings for the `IntList` and `IntList::Iterator` classes. Your implementation must support all the elements of the interfaces, and they must have the specified complexity. You may not change the names of anything in the interfaces, nor may you change the encodings. However, you are free (and encouraged!) to add private, helper member functions.

The provided code already contains declarations for the interfaces and encodings.

`IntList` Interface

Your linked-list class must be named `IntList` and must support the following operations:

- A default constructor that creates an empty list.
- A copy constructor that copies all the integer values into a new list.
- An assignment operator.†
- A destructor.
- A swap operation.†
- An O(1) `push_front` function that inserts a single integer at the head of the list.
- An O(1) `pop_front` function that removes and returns a single integer from the head of a non-empty list.
- An O(1) `push_back` function that inserts a single integer at the tail of the list.
- An O(1) `size` function returning the number of elements in the list. †
- An O(1) `empty` function returning `true` if the list is empty.
- An equality test (`operator==`).
- An inequality test (`operator!=`). †
- A `using` statement, defining the type `iterator`. †
- A `begin` function that returns an iterator that refers to the start of the list.
- An `end` function that returns an invalid/past-the-end iterator. Note: it is undefined behavior to dereference `end()`. We will make life easier for ourselves and the users of our `IntList` class by having a failed `affirm` if we try to dereference `end()`.
- A O(1) `insert_after` function that inserts a single integer after the position indicated by a given iterator. For this member function, you should have a failed `affirm` if the list is empty or the iterator is `end()`.

† These functions have been implemented for you; you don't need to modify them.

`IntList::Iterator` Interface

The `IntList::Iterator` class must provide at least the following operations:

- A default constructor. The resulting iterator doesn't have to be valid (i.e., referring to anything), just overwritable via assignment. †
- A copy constructor (either written or intentionally chosen as the default copy constructor). †
- An assignment operator (either written or intentionally chosen as the default copy constructor). †
- A destructor (either written or intentionally chosen as the default copy constructor). †
- An equality test (`operator==`) and an inequality test (`operator!=`). When comparing two iterators, we can assume that the iterators refer to the same `IntList` object. We therefore only need to check that the iterators are also refering to the same element.
- A preincrement operator (`operator++`)
- An `operator*` that returns an `int&` (so that the integer in the current position can be modified if necessary)

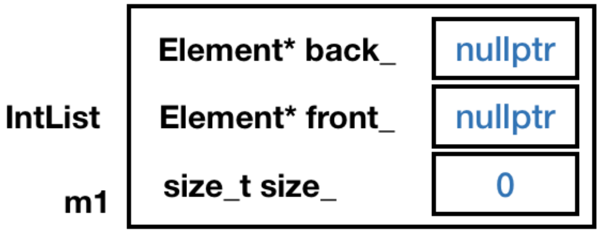
It should also be an STL-friendly iterator, which means the class must include appropriate type definitions (see the header file for details).

† These functions have been implemented for you; you don't need to modify them.

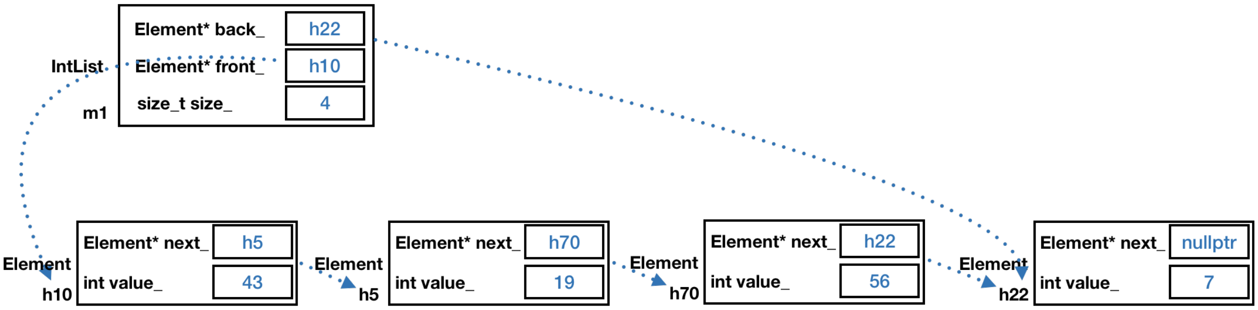
IntList Encoding

The IntList data structure is similar to a linked list. The IntList object will be relatively small, and all the data will be in objects of class IntList::Element on the heap. The IntList object will contain pointers to the first and last data element (if any), and a count of the object's size. These extensions let the class provide efficient push_back and size operations.

An empty IntList:



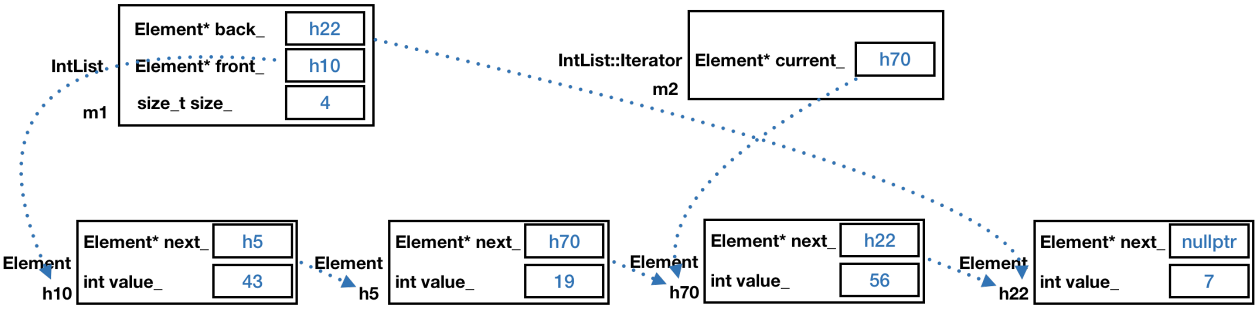
A nonempty IntList:



IntList::Iterator encoding

The list's iterator provides a way to access an element of a list, and it is encoded as a pointer to an Element.

An iterator that refers to the third element of a list:


Element* back_
h22
IntList
Element* front_
h10
m1
size_t size_
4
IntList::Iterator
m2
Element* current_
h70
Element
h10
Element* next_
h5
int value_
43
Element
h5
Element* next_
h70
int value_
19
Element
h70
Element* next_
h22
int value_
56
Element
h22
Element* next_
nullptr
int value_
7