EECS 280 – Lecture 17

Iterators

3/16/2022

Recall: Traversal by Pointer

```
int const SIZE = 5;
int arr[SIZE] = { 1, 2, 3, 4, 5 };
```

- Traversal by Pointer
  - Walk a **pointer** across the array elements.
  - When you want an element, just dereference the pointer!

Notice that "end" is really "one past the end"

Continue until pointer at end.

```
int *end = arr + SIZE;
for (int *ptr = arr; ptr != end; ++ptr) {

  cout << *ptr << endl;
}
```

Pointer starts at beginning of the array.

Increment pointer.

Dereference pointer to current element.

Can we use this for a `std::vector`?

```
vector<int> v = { 1, 2, 3, 4, 5 };
```

- Let's try it…
  - What parts don't work?

```
int *end = v + SIZE;
for (int *ptr = v; ptr != end; ++ptr) {

  cout << *ptr << endl;
}
```

Can we use this for a `std::vector`?

```
vector<int> v = { 1, 2, 3, 4, 5 };
```

- Ask the container for the endpoints!
  - `begin()` and `end()` member functions

```
for (int *ptr = v.begin(); ptr != v.end(); ++ptr) {
  cout << *ptr << endl;
}
```

Can we use this for a Linked List?

```
List<int> list;
// Assume we add 1, 2, 3, 4, 5 to the list
```

- Let's try it…
  - What parts don't work?

```
for (int *ptr = v.begin(); ptr != v.end(); ++ptr) {

  cout << *ptr << endl;

}
```

Iterating Through a `List`

- Here's one way to do it…

```
int main() {
  List<int> list;
  int arr[3] = { 1, 2, 3 };
  fillFromArray(list, arr, 3);

  for (List<int>::Node *np = list.first; np != nullptr; np = np->next) {
    cout << np->datum << endl; // print each element
  }
}
```

- Problems:
  - This breaks the interface of the `List`. `Nodes` are an implementation detail we don't want to mess with here.
  - The `Node` type is `private`, so this won't even compile.

The Iterator Interface

- Iterators provide a common interface for iteration.
  - A generalized version of traversal by pointer.
  - An iterator "points" to an element in a container and can be "incremented" to move to the next element.

- Iterators[1] support these operations:
  - Dereference – access the current element.
    `*it`
  - Increment – move forward to the next element.
    `++it`
  - Equality – check if two iterators point to the same place.
    ```
    it1 == it2
    it1 != it2
    ```

1 There are many different kinds of iterators. These operations are specifically required for *input iterators*.

Traversal by Iterator

- The big picture:
  - Walk an **iterator** across the elements.
  - When you want an element, just dereference the iterator!
  - We'll look at how to get the beginning and end iterators in just a bit…

```
Iterator end = list.end();
for (Iterator it = list.begin(); it != end; ++it) {

  cout << *it << endl;
}
```

Notice that “end” is really “one past the end”

Continue until iterator at end.

Iterator starts at beginning of the container.

Increment iterator.

Dereference iterator to current element.

What is an Iterator?

- An iterator is an object that "works like a pointer".
- This can be implemented by a class that overloads the appropriate operators (*, ++, ==, !=).

```
class Iterator {
public:
  ___ & operator*() const;

  Iterator & operator++();

  bool operator==(Iterator rhs) const;

  bool operator!=(Iterator rhs) const;
  ...
};
```

Element type omitted for now.

Dereference – access the current element.

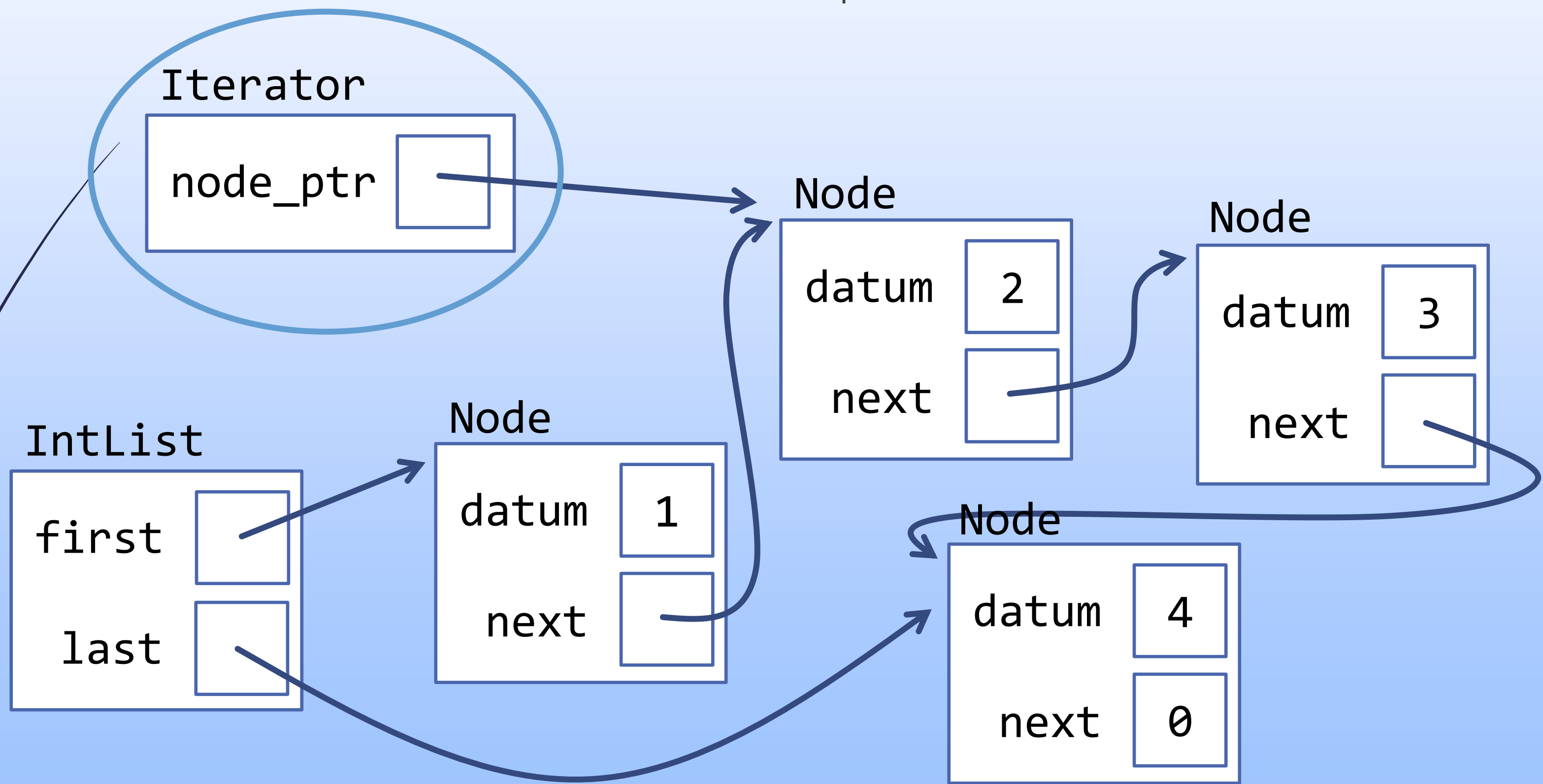
Increment – move forward to the next element.

Equality – check if two iterators point to the same place.

We'll discuss each of these in more detail in the coming slides...

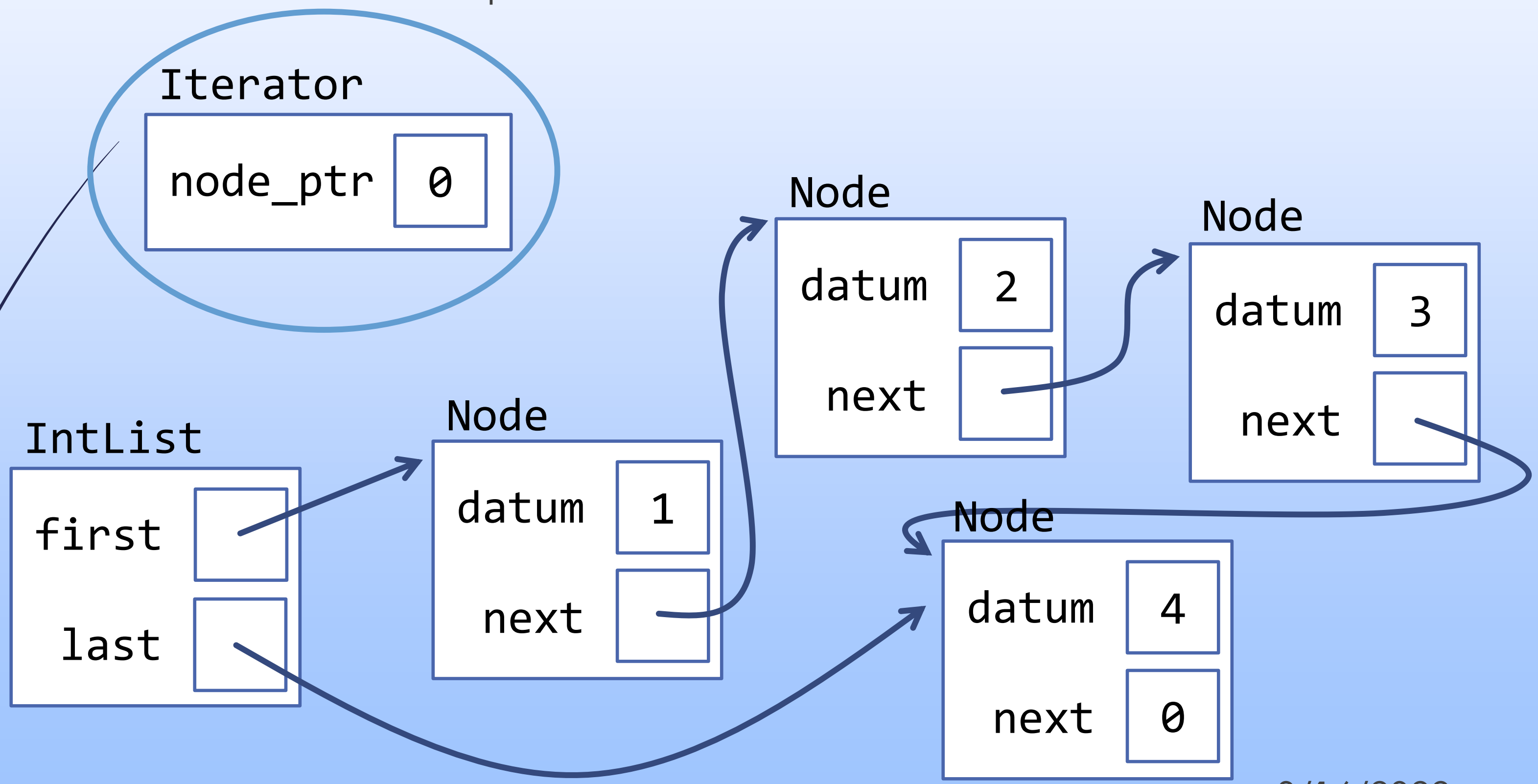
List Iterator: Data Representation

- Keep track of the **Node** holding the current element.
  - Element access through the **datum** member.
  - Move forward via the **next** pointer.



List Iterator: Data Representation

- How do we represent an end iterator?
  - "One past the end"
  - Use a null pointer as a sentinel value.



Implementing a `List` Iterator

- Data representation
  - Store a pointer to the node holding the current element.
- The `Iterator` class is defined inside the `List` class.
  - This gives `Iterator` access to the private section of `List`, including the `Node` struct.
  - `Iterator` can also use the same template parameter `T` so that the * operator returns the correct element type.
  - Each different instantiation of the `List` template will have its own corresponding `Iterator`.

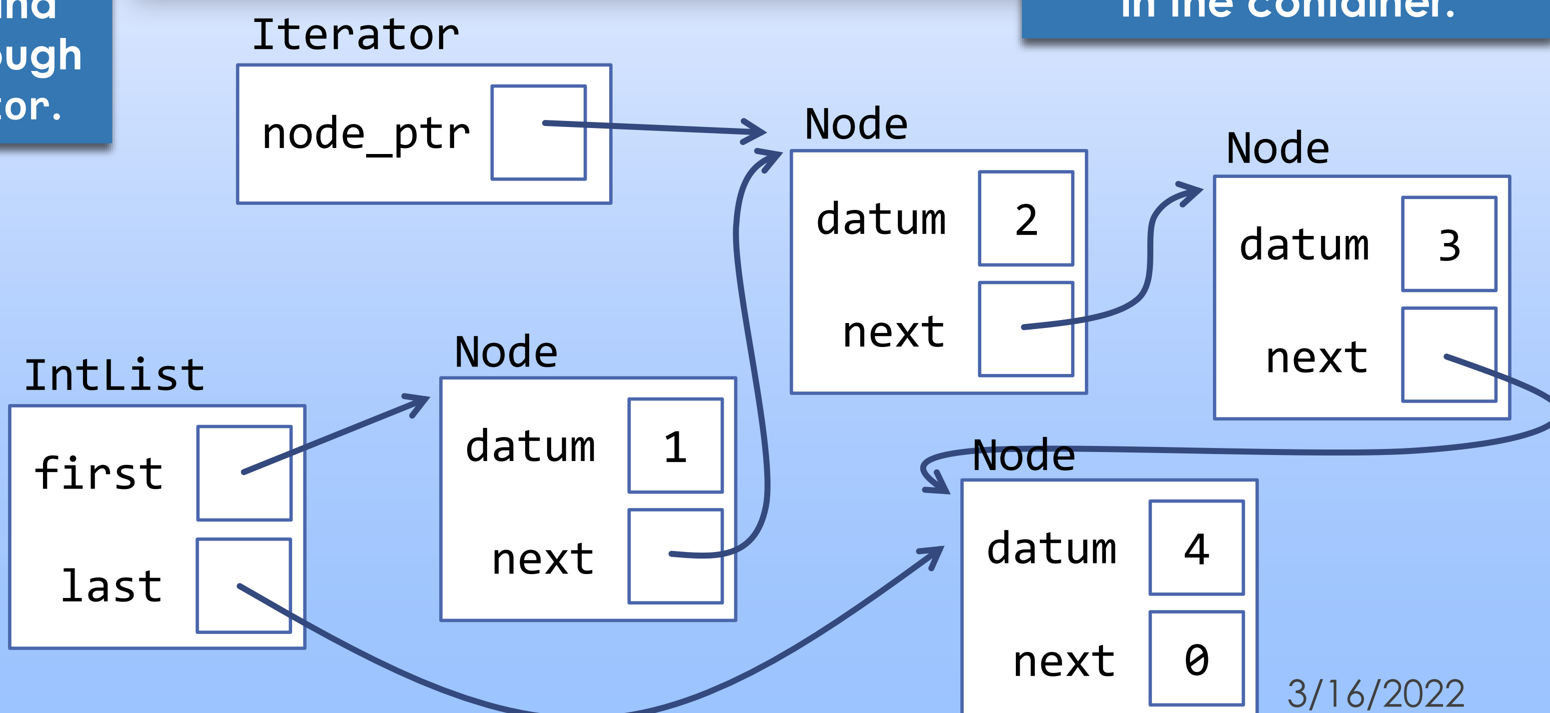
```
template <typename T>
class List {
private:
  struct Node {
    T datum;
    Node *next;
  };
  Node *first;
  Node *last;
  ...
public:
  ...
  class Iterator {
  public:
    T & operator*() const;
    ...
  private:
    Node *node_ptr;
  };
  ...
};
```

List Iterator: The * operator

```
// REQUIRES: this is a dereferenceable iterator
// EFFECTS:  Returns the element this iterator points to.
template <typename T>
T & List<T>::Iterator::operator*() const {
  assert(node_ptr);
  return node_ptr->datum;
}
```

We return by reference to allow both reading and writing through an `Iterator`.

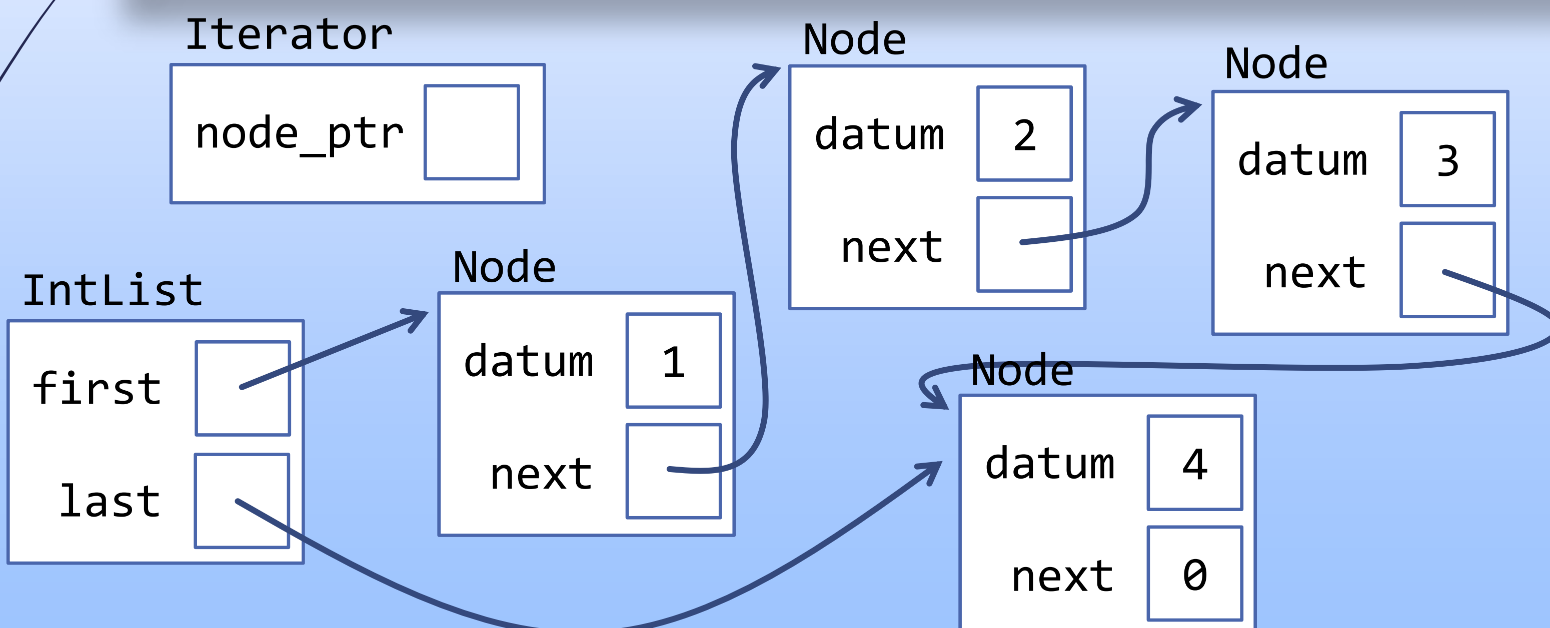
An iterator is <u>dereferenceable</u> if it points to some element in the container.



List Iterator: The ++ operator (prefix[1])

```
// REQUIRES: this is a dereferenceable iterator
// EFFECTS:  Increments this iterator to point to the next
//           element. Returns this iterator by reference.
template <typename T>
typename List<T>::Iterator & List<T>::Iterator::operator++() {
  assert(node_ptr);
  node_ptr = node_ptr->next;
  return *this;
}
```

The typename keyword is required here.



1 The postfix increment operator can also be overridden.

The **typename** Keyword

- The **typename** keyword is required when naming a type nested inside another type that depends on a template parameter.

```
template <typename T>
void func() {

  IntList::Iterator it1;

  List<int>::Iterator it2;

  typename List<T>::Iterator it3;

}
```

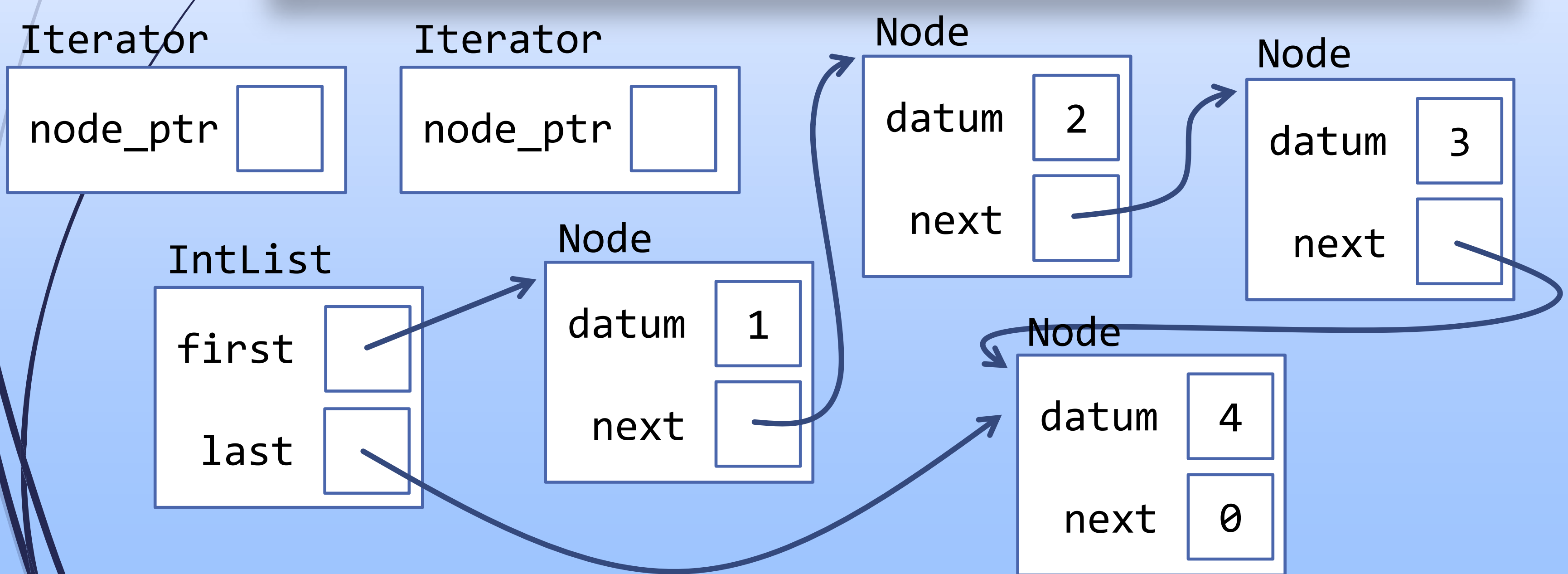
No typename required, since IntList does not depend on the parameter T.

No typename required, since List<int> does not depend on the parameter T.

typename required, since List<T> depends on the parameter T.

List Iterator: The == operator[1]

```
// EFFECTS: Returns whether this and rhs are pointing to
//          the same place.
// NOTE:    The result is only meaningful if both are
//          pointing into the same underlying container.
template <typename T>
bool List<T>::Iterator::operator==(Iterator rhs) const {
  return node_ptr == rhs.node_ptr;
}
```



1 The != operator is defined analogously.

Creating Iterators

- We'll provide two constructors for `Iterator`.

```
class Iterator {
public:
  // Public constructor. Creates an end Iterator
  Iterator()
    : node_ptr(nullptr) { }
  ...

private:
  // Private constructor. Creates an Iterator pointing
  // to the specified Node.
  Iterator(Node *np)
    : node_ptr(np) { }

  Node *node_ptr;
};
```

There's no need for the outside world to use this one. Node itself is `private`, after all.

Getting Iterators for a Container

- The missing piece from earlier was how to get the iterators for a container…

```
List<int> list;
int arr[3] = { 1, 2, 3 };
fillFromArray(list, arr, 3);


List<int>::Iterator end = ______;
for (List<int>::Iterator it = ______; it != end; ++it) {
  cout << *it << endl;
}
```

How do we get beginning and end iterators?

- We'll implement `begin()` and `end()` functions for the `List` class that construct these iterators for us.

begin() and end()

```
template <typename T>
class List {
public:
  ...
  class Iterator {
  public:
    Iterator() : node_ptr(nullptr) { }
    ...
  private:
    Iterator(Node *np) : node_ptr(np) { }
    Node *node_ptr;
  };

  Iterator begin() { return Iterator(first); }

  Iterator end() { return Iterator(); }
  ...
private:
  Node *first;
  ...
};
```

Question

What's wrong with this code?

A) Memory Leak

B) Missing const

C) Use of first

D) Use of ctor

The begin() function uses the constructor to create an iterator pointing to the first element.

The end() function uses the default constructor to create and return a "past the end" iterator.

Friend Declarations

```
template <typename T>
class List {
public:
  ...
  class Iterator {
    friend class List;
  public:
    Iterator() : node_ptr(nullptr) { }
    ...
  private:
    Iterator(Node *np) : node_ptr(np) { }
    Node *node_ptr;
  };

  Iterator begin() { return Iterator(first); }

  Iterator end() { return Iterator(); }
  ...
private:
  Node *first;
  ...
};
```

We use a `friend` declaration to give `List` special privileges to access the private members of `Iterator`.

`List` member functions, like `begin()`, can now access the private constructor.

It's easy to get this backwards. Remember that "friendship is given, not taken."

Traversal by Iterator

- We now have all the pieces to implement and use the traversal by iterator pattern.

```
List<int> list;
int arr[3] = { 1, 2, 3 };
fillFromArray(list, arr, 3);


List<int>::Iterator end = list.end();
for (List<int>::Iterator it = list.begin(); it != end; ++it) {
  cout << *it << endl;
}
```

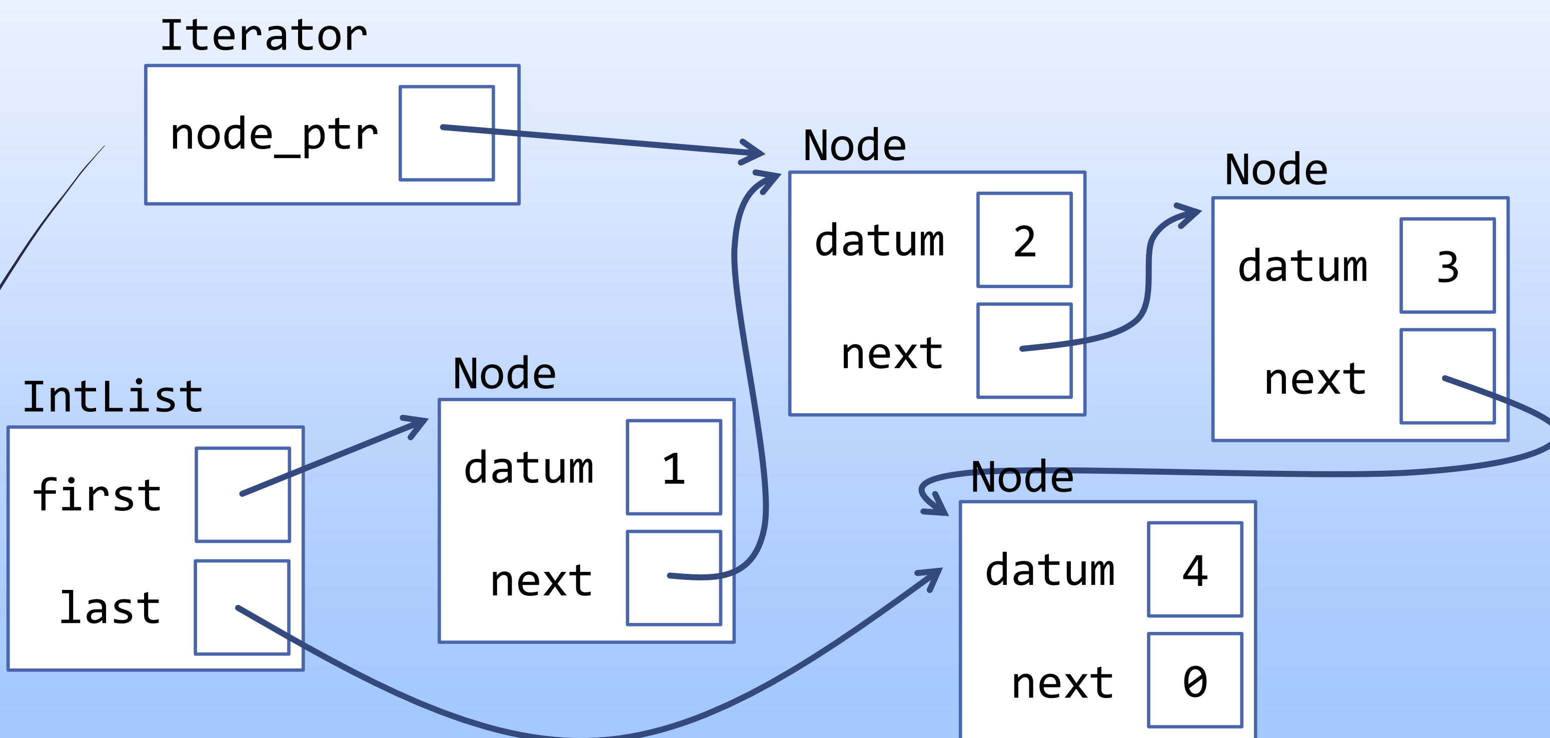
Ask the List for iterators that define the sequence of elements.

Break Time!

We’ll start again in five minutes.

Iterator Big Three?

- Do we need custom versions of the Big Three for the `Iterator` class? Let's do an exercise…



Exercise


```
int main() {
  List<int> list;
  // Add to list so it contains 1, 2, 3
  List<int>::Iterator it1 = list.begin();
  ++it1;
  List<int>::Iterator it2 = it1;
  ++it2;
  // Draw memory at this point
}
```

Exercise


```
int main() {
  List<int> list;
  // Add to list so it contains 1,
  List<int>::Iterator it1 = list.b
  ++it1;
  List<int>::Iterator it2 = it1;
  ++it2;
  // Draw memory at this point
}
```

Question

If we add the dtor below:

A) Memory Leak

B) Use a dead object

C) Double Free

D) Dereference a null pointer

E) No errors occur

```
class Iterator {
    friend class List;
  public:
    Iterator() : node_ptr(nullptr) { }
    ~Iterator() { delete node_ptr; } // Should we add this???
    ...
  private:
    Iterator(Node *np) : node_ptr(np) { }
    Node *node_ptr;
  };
};
```

The Iterator Interface

- Iterators provide a common interface for traversing a sequence of elements.
- They allow us to reuse the same code to work with many different kinds of containers as long as they provide an iterator interface.
- The STL containers work this way. For example:

```
vector<int> vec;
// Fill vec with numbers

vector<int>::iterator end = vec.end();
for (vector<int>::iterator it = vec.begin(); it != end; ++it) {
  cout << *it << endl;
}
```

Iterators Generic Functions

- A key strength of iterators is that we can write functions to work with iterators, rather than with a particular container.
- This allows the same function to be used with many different containers!
- The STL contains many functions, like `std::sort`, that work this way.

```
int main() {
  vector<int> vec; // fill with numbers
  sort(vec.begin(), vec.end());
}
```

Example: max_element

```
template <typename Iter_type>
Iter_type max_element(Iter_type begin, Iter_type end) {

  Iter_type maxIt = begin;

  for (Iter_type it = begin; it != end; ++it) {
    if (*it > *maxIt) {
      maxIt = it;
    }
  }
  return maxIt;
}

int main() {
  vector<int> vec; // fill with numbers
  cout << *max_element(vec.begin(), vec.end()) << endl;
}
```

Start by assuming first element is the max.

Use traversal by iterator to check each element.

If we find a larger element, update maxIt to point to it.

Dereference returned iterator to get the element itself.

Using max_element

- As long as we are working with a container that supports iterators, we don't ever have to write that maximum-finding loop again!

```
int main() {
  vector<int> vec; // fill with numbers
  cout << *max_element(vec.begin(), vec.end()) << endl;

  List<int> list; // fill with numbers
  cout << *max_element(list.begin(), list.end()) << endl;

  List<Card> cards; // fill with Cards
  cout << *max_element(cards.begin(), cards.end()) << endl;

  int const SIZE = 10;
  double arr[SIZE]; // fill with numbers
  cout << *max_element(arr, arr + SIZE) << endl;
}
```

Pointers also work as iterators!

Exercise

Question

Which of these is a correct generic length function?

A

```
template <typename Iter_type>
int length(Iter_type begin,
           Iter_type end) {
  int count = 0;
  List<int>::iterator it = begin;
  while(it != end) {
    ++count;
    ++it;
  }
  return count;
}
```

B

```
template <typename Iter_type>
int length(Iter_type begin, Iter_type end) {
  int count = 0;
  for(Iter_type it = begin; it < end; ++it) {
    ++count;
  }
  return count;
}
```

C

```
template <typename Iter_type>
int length(Iter_type begin, Iter_type end) {
  int count = 0;
  while(begin != end) {
    ++count;
    ++begin;
  }
  return count;
}
```

D

```
template <typename Iter_type>
int length(Iter_type begin, Iter_type end) {
  return end - begin;
}
```

```
int main() { // EXAMPLE
  std::vector<Card> v; // assume it's filled with some cards
  cout << length(v.begin(), v.end()) << endl;
}
```

Take Home Exercise: `no_duplicates`

- Write a function template that takes in begin and end iterators and determines whether the given range contains any duplicate elements. For example:

```
int main() { // EXAMPLE
  List<int> list; // assume it's filled with some numbers
  cout << no_duplicates(list.begin(), list.end()) << endl;
}
```

- Use this code for an array of `int`s as an example:

```
bool no_duplicates(int arr[], int size) {
  for (int i = 0; i < size; ++i) {
    for (int k = i + 1; k < size; ++k) {
      if (a[i] == a[k]) {
        return false; // If any duplicates, return false
      }
    }
  }
  return true; // If we got here, no duplicates
}
```

Solution: no_duplicates

- Write a function template that takes in begin and end iterators and determines whether the given range contains any duplicate elements.

```
template <typename Iter_type>
bool no_duplicates(Iter_type begin, Iter_type end) {
  for (Iter_type it1 = begin; it1 != end; ++it1) {
    Iter_type it2 = it1;
    ++it2;
    for (; it2 != end; ++it2) {
      if (*it1 == *it2) {
        return false; // If any duplicates, return false
      }
    }
  }
  return true; // If we got here, no duplicates
}
```

```
int main() { // EXAMPLE
  List<int> list; // assume it's filled with some numbers
  cout << no_duplicates(list.begin(), list.end()) << endl;
}
```

Iterator Invalidation

```
int main() {
  List<int> list;
  list.push_back(1);
  list.push_back(2);

  List<int>::Iterator it = list.begin();
  List<int>::Iterator it2 = list.begin();
  cout << *it << endl; // OK
  cout << *it2 << endl; // OK

  list.erase(it);

  cout << *it << endl; // EXPLODE
  cout << *it2 << endl; // ALSO EXPLODE
}
```

Iterator Invalidation

- Invalidated iterators are like dangling pointers – it's no longer safe to dereference them and try to access the object they point to.
- Seemingly innocuous operations on a container can result in iterator invalidation.
  - For example, iterators pointing into a vector are invalidated if an operation causes a grow.
- A function's documentation should specify which iterators, if any, it may invalidate.