Slides by Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

Iterators

Introduction to iterators

- An iterator allows you to traverse a container
- We've seen something like this before with arrays

```
int a[SIZE]; // fill a
for (int i=0; i < SIZE; ++i)
  cout << a[i] << endl;
```

- How would you do this for a linked list, like our `List` type?

Introduction to iterators

- So far, we've looked at several different kinds of abstractions:
  1. Procedural abstraction (functions, function pointers)
  2. Data abstraction (Abstract Data Types /ADTs)

- Today, we will use procedural abstraction and data abstraction to iterate over a container

Where we are going

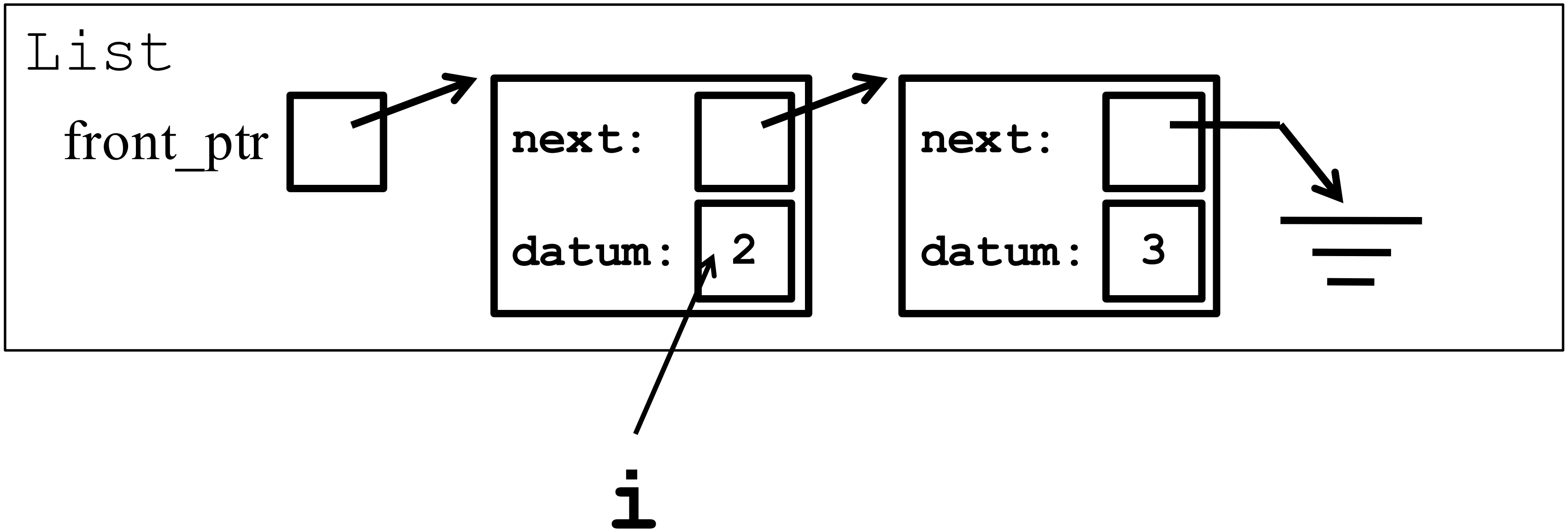
- In the end, our iterator will work a lot like using a pointer to traverse an array

```
int a[SIZE]; // fill a
for (int *p=a; p < a+SIZE; ++p)
   cout << *p << endl;

List<int> l; // fill l
for (List<int>::Iterator i=l.begin();
     i != l.end(); ++i)
   cout << *i << endl;
```

You can think of an iterator as a class pretending to be a pointer.

Where we are going




```
List<int> l; // fill l
for (List<int>::Iterator i=l.begin();
      i != l.end(); ++i)
  cout << *i << endl;
```

```
./a.out
2 3
```

Introduction to iterators

- Each item in the container will be returned exactly once
- In general, an iterator makes no guarantee about the order in which objects are returned, though it is possible to define specific iterators with such a guarantee (like when working with sorted lists).
- Usually, for each container type, there is also (at least) one iterator type.

Iterator functions

1. **Create** an iterator with a constructor
2. **Get** the `T` at the iterator's current position
3. **Move** the iterator to the next position
4. **Compare** two iterators

```
List<int> l; // fill l
for (List<int>::Iterator i=l.begin();
       i != l.end(); ++i)
   cout << *i << endl;
```

Review: operator overloading

- *Operator overloading* lets us customize what happens when we use a built-in symbol
- Example: we overloaded the assignment operator in our IntSet

```
int main() {
  IntSet is1(3);
  IntSet is2(6);
  is2 = is1;
}
```

```
class IntSet {
  //...
  //EFFECTS: assignment operator does
  // a deep copy
  IntSet & operator= (const IntSet &rhs);
};
```

- Here, we changed what the equals "=" sign does, by doing a deep copy instead of a shallow copy
- We can also overload other operators

Implementation

- So, for this example, we'll have two classes:
  - One `List` class, that holds `T`'s.
  - One `Iterator` class, that returns each `T` in the `List`

```
template <typename T>
class List {
  // ...
};

class Iterator {
  // ...
};
```

Implementation

- This brings us to two problems

1. For the `Iterator` to do its job, it needs to have access to the `Node` type, which is a private type inside `List`
2. The `Iterator` needs return a type `T`, which should be the *exact same type `T`* that is inside the `List`

Implementation

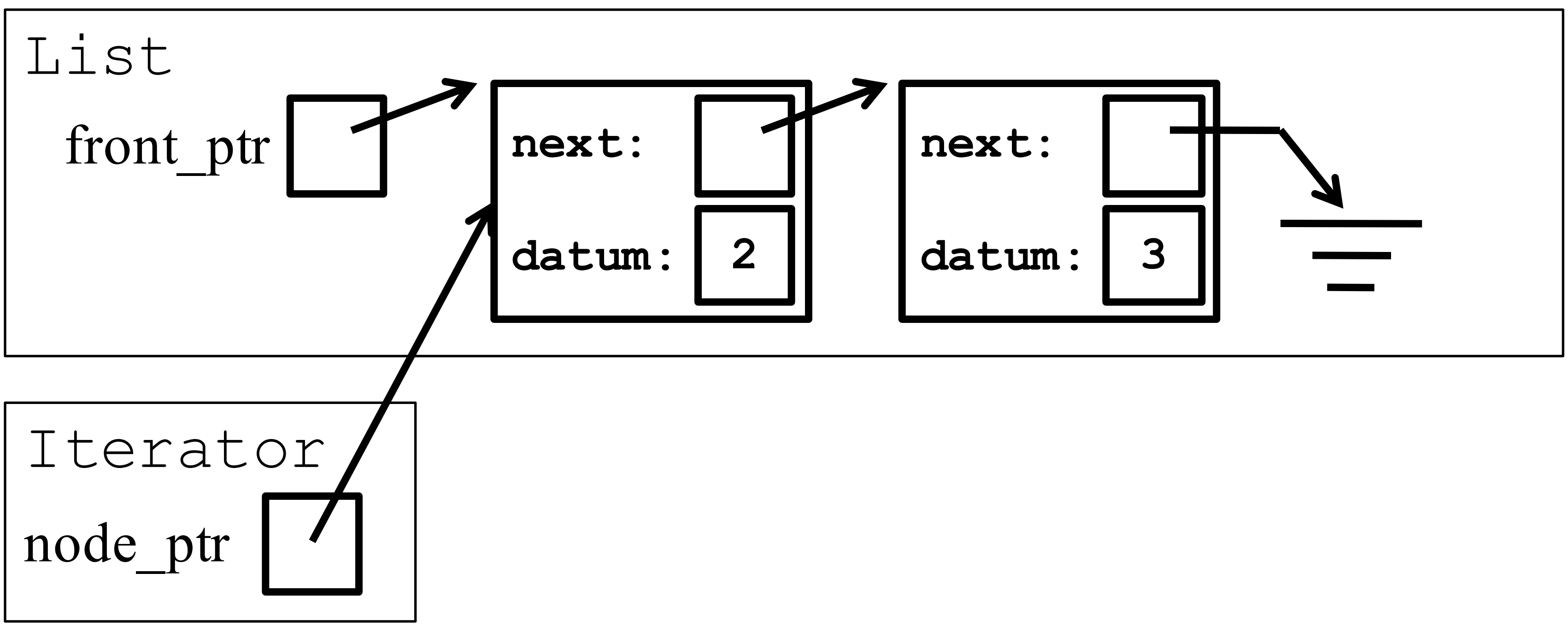
- We can solve both problems by declaring the `Iterator` class inside the `List` class

```
template <typename T>
class List {
  // ...
public:
  class Iterator {
    // ...
  };
};
```

From the outside, we can refer to our new `Iterator` class by specifying the scope: `List<int>::Iterator`

Implementation

- Now, let's choose a concrete representation
- The only thing the `Iterator` has to do is keep track of is the current node
- We can use a `Node*` pointer to do this



Implementation

- `Iterator` has only one member variable: `node_ptr`
- The invariant on `node_ptr` is that it points to the current node in the underlying `List`, and `0` (AKA `NULL`) otherwise

```
template <typename T>
class List {
 struct Node {  // same as before
    Node *next;
    T datum;
  };
  // ...
public:
  class Iterator {
    Node* node_ptr;
    // ...
  };
};
```

List does not need to change!

Implementation

- Now we can declare our member functions, one for each of the four behaviors we talked about

```
class Iterator {
  Node* node_ptr;
public:
  Iterator();                  // Create
  T& operator* () const;       // Get
  Iterator& operator++ ();     // Move
  bool operator!= (Iterator rhs) const;//Compare
};
```

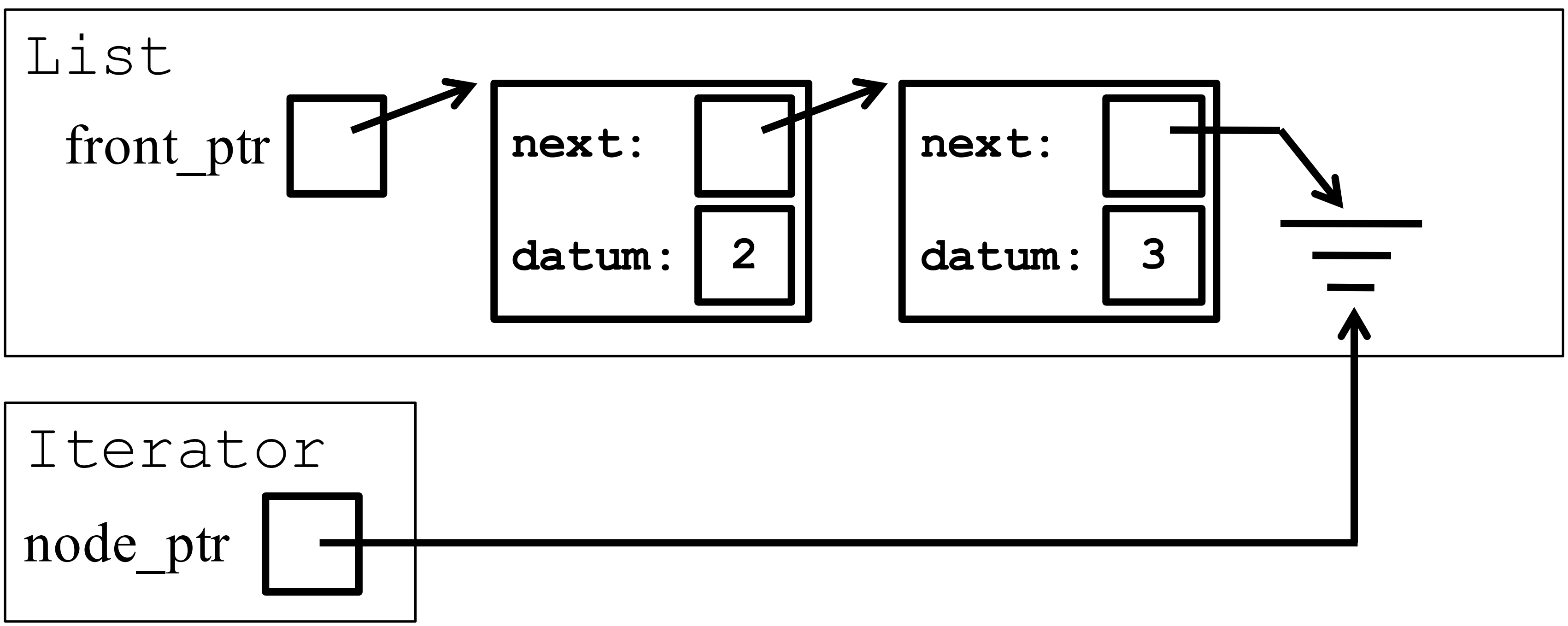
Implementation

```
Iterator();                   // Create new Iterator
T& operator* () const;        // Get T from node
Iterator& operator++ ();// Move to next node
bool operator!= (Iterator rhs) const;
                              // Compare two Iterators
```

```
int main() {
  List<int> l; // fill l
  for (List<int>::Iterator i=l.begin();
       i != l.end(); ++i)
    cout << *i << endl;
}
```

Create new iterator: `Iterator()`

- Now we can define (implement) each `Iterator` method.
- The constructor establishes the invariant on `node_ptr`
- Think of this as an `Iterator` "past the end" of the `List`



Create new iterator: `Iterator()`

- Now we can define (implement) each `Iterator` method.
- The constructor establishes the invariant on `node_ptr`
- Think of this as an `Iterator` "past the end" of the `List`
- For short functions (~1 line), it's OK to implement them directly in the class declaration. This is called an *in line* implementation.

```
class List {
  // ...
  class Iterator {
    Node* node_ptr;
   public:
    Iterator() : node_ptr(0) {}
    // "past the end" by default
    // ...
  };
};
```

Get `T` at the iterator's current position

- The dereference operator returns a reference to the `datum` at the current `Iterator` position
- Write this function

```
class List {
 struct Node {  //same as before
    Node *next;
    T datum;
  };
  // ...
  class Iterator {
    Node* node_ptr;
   public:
    T& operator* () const {
      /* your code here */
    }
    // ...
  };
};
```

```
int main() {
  List<int> l; // fill l with ( 1 2 )
  for (List<int>::Iterator i=l.begin();
       i != l.end(); ++i)
    cout << *i << endl;
}
```

Get `T` at the iterator's current position

- The dereference operator returns a reference to the `datum` at the current `Iterator` position

```
class List {
  // ...
  class Iterator {
    Node* node_ptr;
   public:
    T& operator* () const {
      assert(node_ptr);
      return node_ptr->datum;
    }
    // ...
  };
};
```

Pro tip: use `assert()`

```
T& operator* () const {
  assert(node_ptr); //OR assert(node_ptr != 0)
  return node_ptr->datum;
}
```

- Pro tip: use `assert()` to help find bugs in your program
- Stops execution if you try to use an iterator "past the end"

```
Assertion failed: (node_ptr), function operator*, file
20_List_with_Iterator.h, line 83.
Abort trap: 6
```

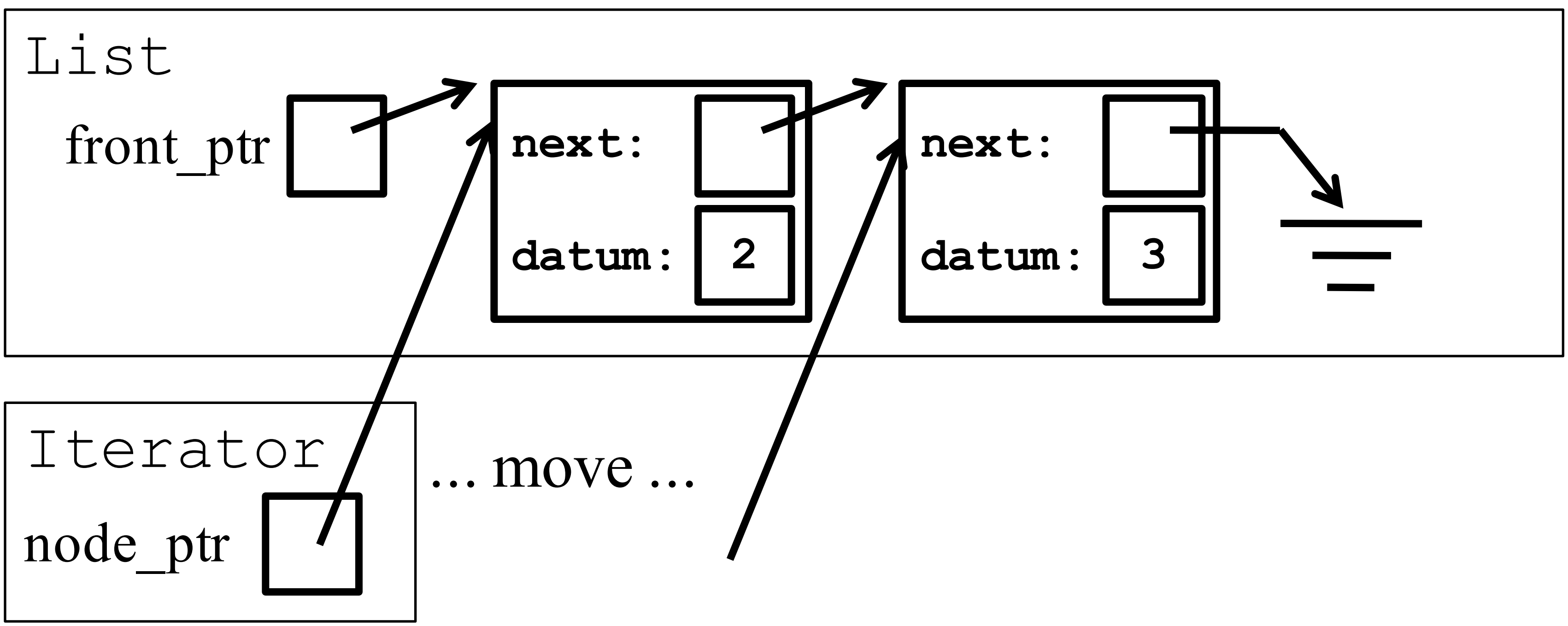
Get `T` at the iterator's current position

```
T& operator* () const {
  assert(node_ptr);
  return node_ptr->datum;
}
```

- Question: `node_ptr` points to a `List<T>::Node` that was created by the `List` implementation. Why can an `Iterator` function access it and its data members?
- Answer: the `Iterator` has a pointer to the `Node` created by the `List`, and the `Node`'s data members are all public because it is a `struct`.

Move to next position: `operator++`

- The prefix increment operator moves to the next `List` node



Move to next position: `operator++`

- The prefix increment operator moves to the next `List` node
- Write this function and use assert to check for a zero pointer

```
class List {
  // ...
  class Iterator {
    Node* node_ptr;
  public:
    Iterator& operator++ () {
      /* your code here */
    }
    // ...
  };
};
```

Move to next position: `operator++`

- The prefix increment operator moves to the next `List` node

```
class List {
  // ...
  class Iterator {
    Node* node_ptr;
  public:
    Iterator& operator++ () {
      assert(node_ptr);
      node_ptr = node_ptr->next;
      return *this;
    }
    // ...
  };
};
```

Prefix increment: `operator++`

```
Iterator& operator++ () {
  assert(node_ptr);
  node_ptr = node_ptr->next;
  return *this;
}
```

- Moves `Iterator` to the next node
- REQUIRES that `Iterator` is not “past the end”
- Why does it `return *this` ?
  - So you can do cool things like
    ```
    ++++i;
    Iterator j = ++i;
    ```

Comparing iterators: `operator!=`

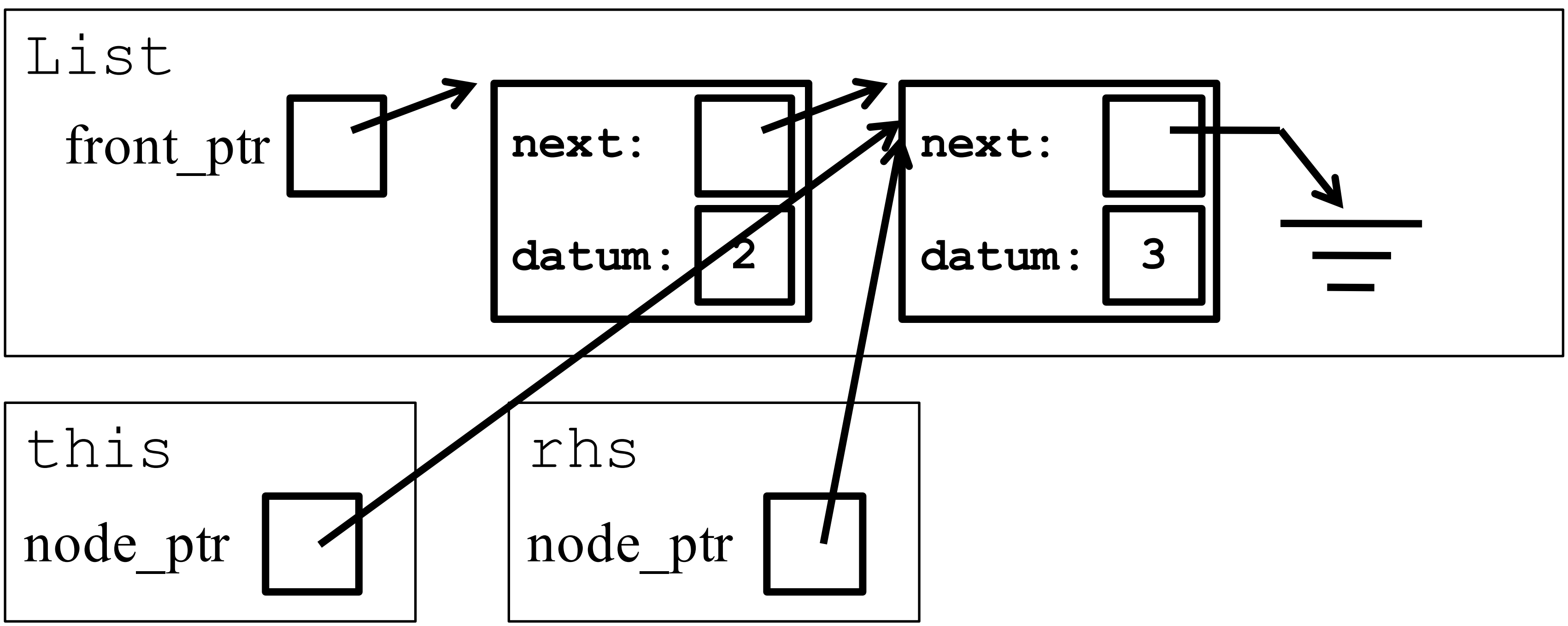
- Now, we need a way to compare two iterators
- Overload `operator!=` to achieve this

```
class List {
  // ...
  class Iterator {
    Node* node_ptr;
   public:
    bool operator!= (Iterator rhs) const {
      return node_ptr != rhs.node_ptr;
    }
    // ...
  };
};````
```

Comparing iterators: `operator!=`

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

- Compares "`this`" `Iterator` with `rhs` `Iterator`



Comparing iterators: `operator!=`

```
// in List.h :
bool operator!= (Iterator rhs) const {
  return node_ptr != rhs.node_ptr;
}

// main.cpp :
#include "List.h"
int main() {
  List<int> l; // fill l
  for (List<int>::Iterator i=l.begin();
       i != l.end(); ++i)
    cout << *i << endl;
}
```

The missing piece

- We can now
  1. **Create** an iterator with a constructor (`Iterator()`)
  2. **Get** the `T` at the iterator's current position (`operator*`)
  3. **Move** the iterator to the next position (`operator++`)
  4. **Compare** two iterators (`operator!=`)

```
List<int> l; // fill l
for (List<int>::Iterator i=l.begin();
     i != l.end(); ++i)
  cout << *i << endl;
```

The missing piece

- How do we know where to start? Where to end?
- Need to ask the list!
- Typically implemented as member functions called `begin()` and `end()` inside the container, not in the iterator.

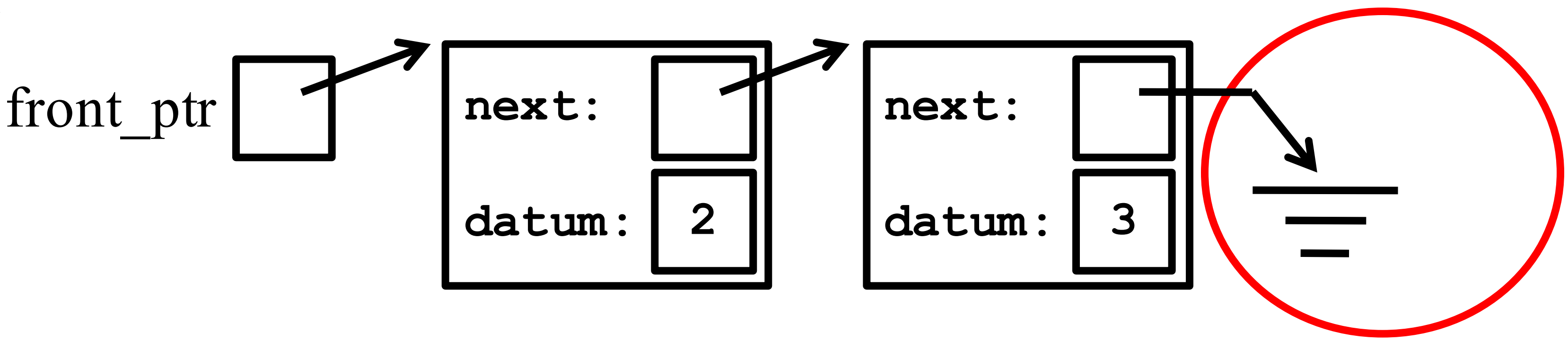
```
List<int> l; // fill l
for (List<int>::Iterator i=l.begin();
      i != l.end(); ++i)
  cout << *i << endl;
```

Implementing `end()`

- `end()` is a `List` member function
- Returns a default `Iterator` object, "past the end" position

```
class List {
  // ...
  class Iterator { /*...*/ };
  Iterator end() const {
    return Iterator();
  }
};
```

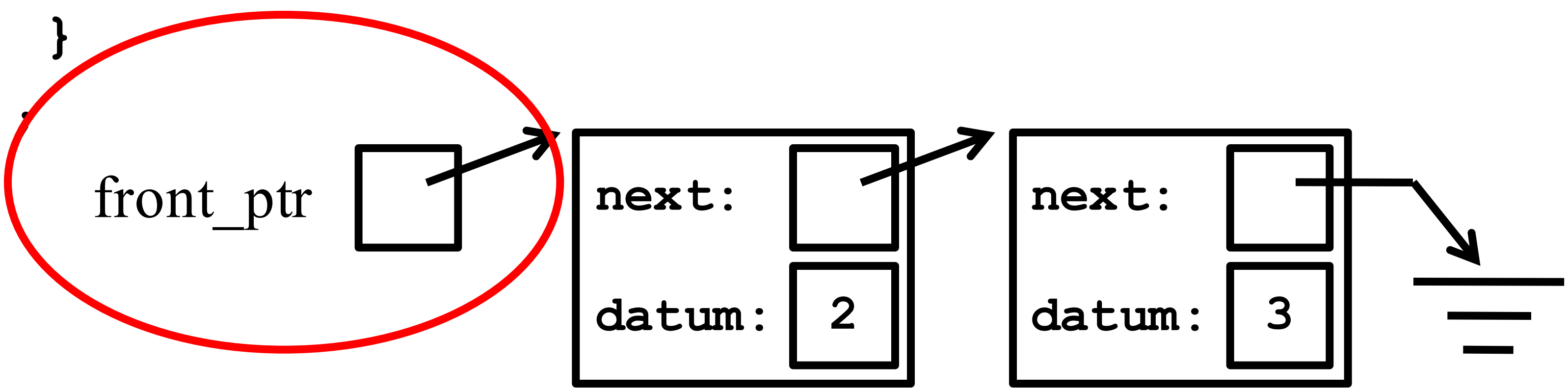
```
Iterator()
: node_ptr(0) {}
```



Implementing `begin()`

- `begin()` is also a `List` member function, just like `end()`
- Returns an `Iterator` object pointing to first `List` position

```
class List {
  Node *front_ptr; // points to first list Node
  // ...
  class Iterator { /*...*/ };
  Iterator begin() const {
    return Iterator(front_ptr);
  }
};
```



Implementing `begin()`

- Now we need another `Iterator` constructor with a `Node*` input
- No one outside the `List` class should see this, so make it `private`

```
class List {
  // ...
  class Iterator {
  // ...
  private:
    Iterator(Node* p) : node_ptr(p) {}
  };

  Iterator begin() const {
    return Iterator(front_ptr);
  }
};
```

Implementing `begin()`

- Now we have another problem
  - `Iterator(Node* p)` is private to the `Iterator` class
  - `begin()` is a member function of `List`, not `Iterator`
  - Therefore, `begin()` cannot access `Iterator(Node* p)`
- Bad solution: make it public.
  - This exposes the representation to clients, leaving the implementation unsafe.
- Good Solution: `friend` classes

`friend` classes

- The `friend` declaration allows you to expose the private members of one class to another class (and only that class).
- The methods inside `List` now have access to the private members of `Iterator`

```
class List {
  // ...
  class Iterator {
    // ...
    friend class List;
  };
};
```

`friend` classes

```
class List {
  // ...
  class Iterator {
  // ...
  private:
    Iterator(Node* p)
      : node_ptr(p) {}
    friend class List;
  };

  Iterator begin() const {
    return Iterator(front_ptr);
  }
};
```

Now, `begin()` can access `Iterator(Node* p)`, but clients of `List` cannot.

Understanding that friendship is something given, not taken, will help you remember that "`friend class List;`" goes inside `Iterator`, not the other way around.

Putting it all together: recap

```
class List {
  // ...
  class Iterator {
    Node* node_ptr;
    // ...
  };
};
```

`Iterator` goes inside `List`

Store a pointer to the current `List` position

Putting it all together: recap

```
class List {
  // ...
  class Iterator {
    // ...
    Iterator() : node_ptr(0) {}

    T& operator* () const {
      assert(node_ptr);
      return node_ptr->datum;
    }
    // ...
  };
};
```

Construct a default `Iterator`, which points to "past the end"

Return the datum (of type `T`) at the current `Iterator` position

Putting it all together: recap

```
class List {
  // ...
  class Iterator {
    // ...
    Iterator& operator++ () {
      assert(node_ptr);
      node_ptr = node_ptr->next;
      return *this;
    }
    bool operator!= (Iterator rhs) const {
      return node_ptr != rhs.node_ptr;
    }
    // ...
  };
};
```

Move to the next `List` node

Compare two `Iterators`

Putting it all together: recap

```
class List {
  // ...
  class Iterator {
    // ...
  private:
    Iterator(Node* p) : node_ptr(p) {}
    // ...
  };
};
```

Construct an `Iterator` at a specific position

This method is only for internal use, so it is private.

Putting it all together: recap

```
class List {
  // ...
  class Iterator {
    //...
    friend class List;
  };

  Iterator begin() const {
    return Iterator(front_ptr);
  }

  Iterator end() const {
    return Iterator();
  }
};
```

`friend` declaration so that `List::begin()` can access private constructor

Return an `Iterator` to the first position in the `List`

Return an `Iterator` "past the end" of the `List`

Putting it all together: example

- We now have a `List` and `Iterator` that work just like STL!

```
List<int> l;
l.push_front(3);
l.push_front(2);
l.push_front(1);

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

```
./a.out
1 2 3
```

Using iterators

- This allows us to write any number of algorithms that must examine every entry of a list, without:
  - Needing to understand the mechanics of how `Lists` actually work.
  - Requiring that the designer of the `List` class knew everything that the client wanted in advance.
- In other words, we have successfully created a function abstraction to access a container

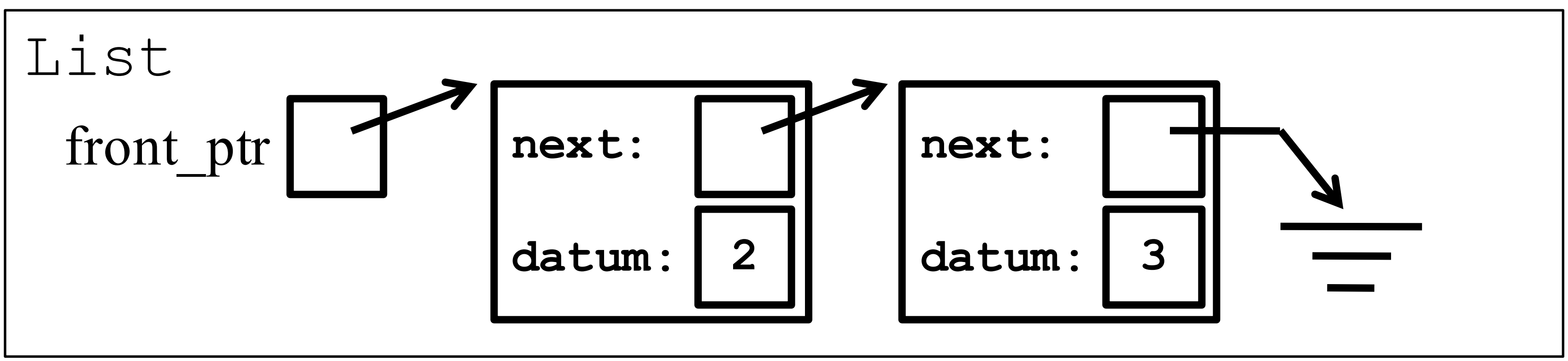
Multiple iterators

- Since the `Iterator` is an object, we can have more than one pointing to the same `List`

```
List<int> l; //fill with ( 2 3 )
List<int>::Iterator i = l.begin();
List<int>::Iterator j = i;
++j;
cout << "*i = " << *i << endl;
cout << "*j = " << *j << endl;
```

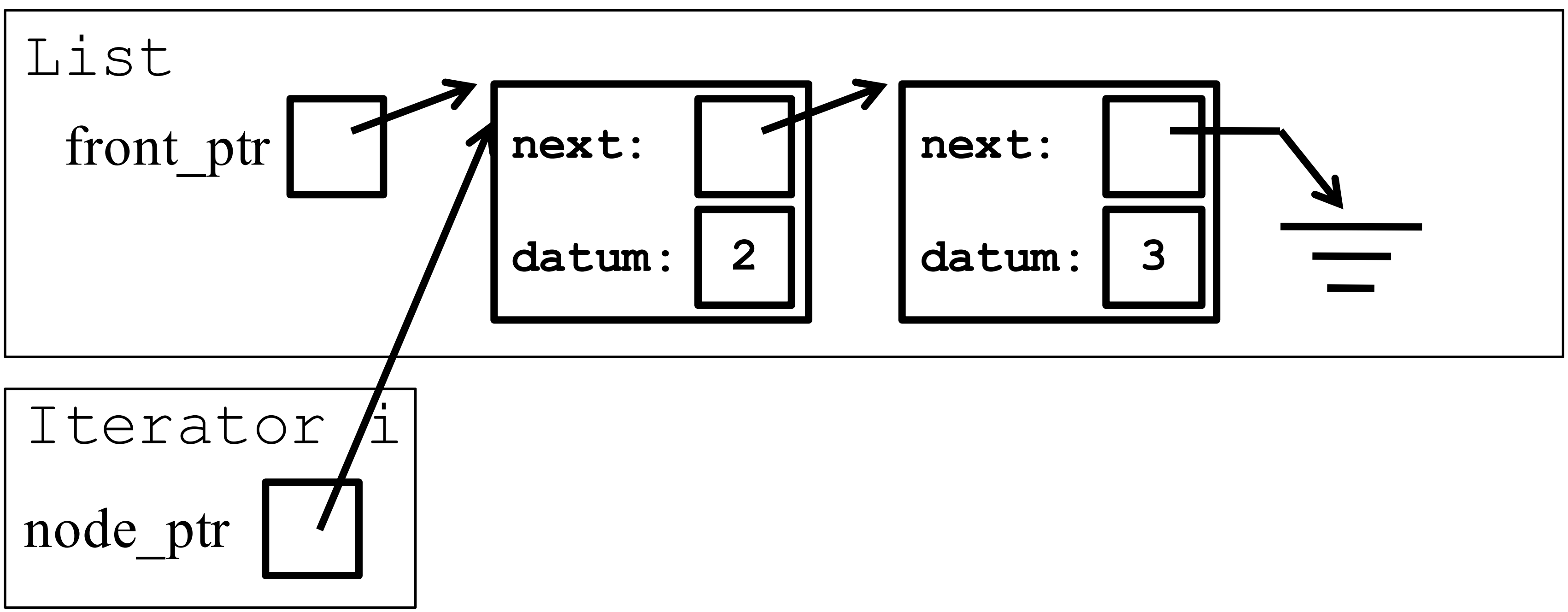
Multiple iterators

```
List<int> l; //fill with ( 2 3 )
List<int>::Iterator i = l.begin();
List<int>::Iterator j = i;
++j;
cout << "*i = " << *i << endl;
cout << "*j = " << *j << endl;
```



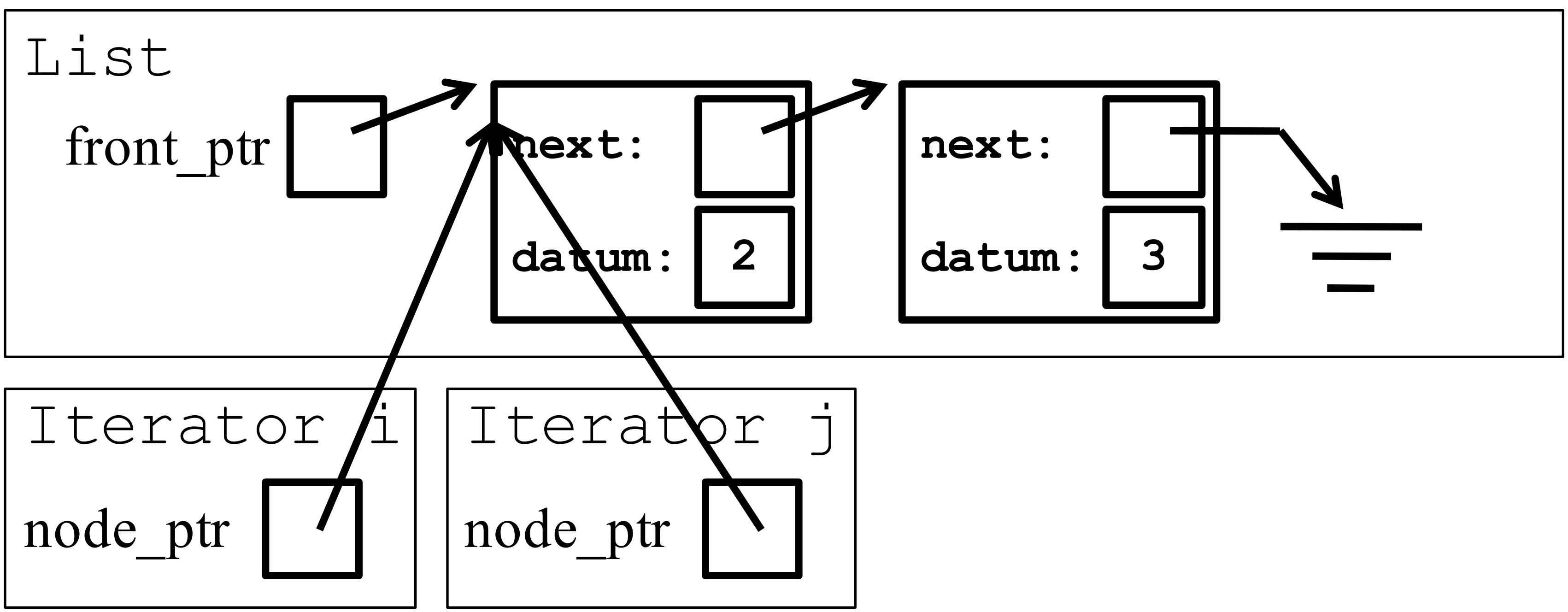
Multiple iterators

```
List<int> l; //fill with ( 2 3 )
List<int>::Iterator i = l.begin();
List<int>::Iterator j = i;
++j;
cout << "*i = " << *i << endl;
cout << "*j = " << *j << endl;
```



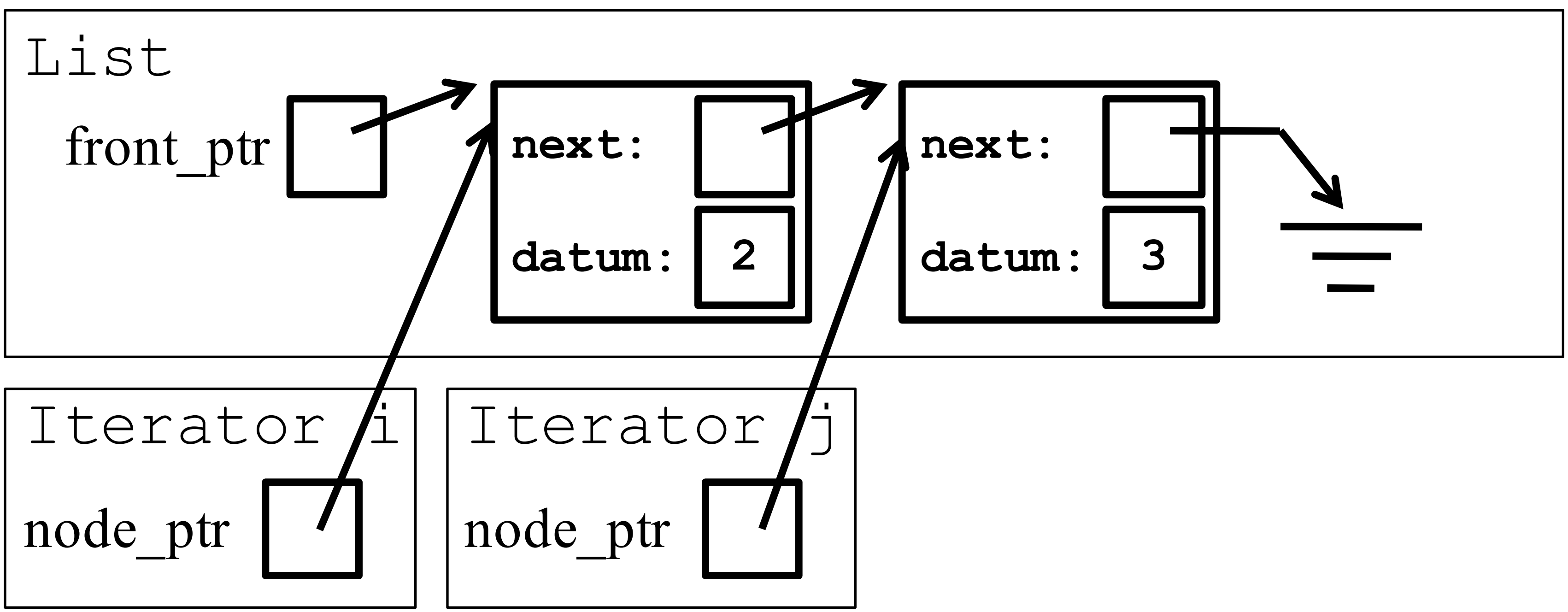
Multiple iterators

```
List<int> l; //fill with ( 2 3 )
List<int>::Iterator i = l.begin();
List<int>::Iterator j = i;
++j;
cout << "*i = " << *i << endl;
cout << "*j = " << *j << endl;
```



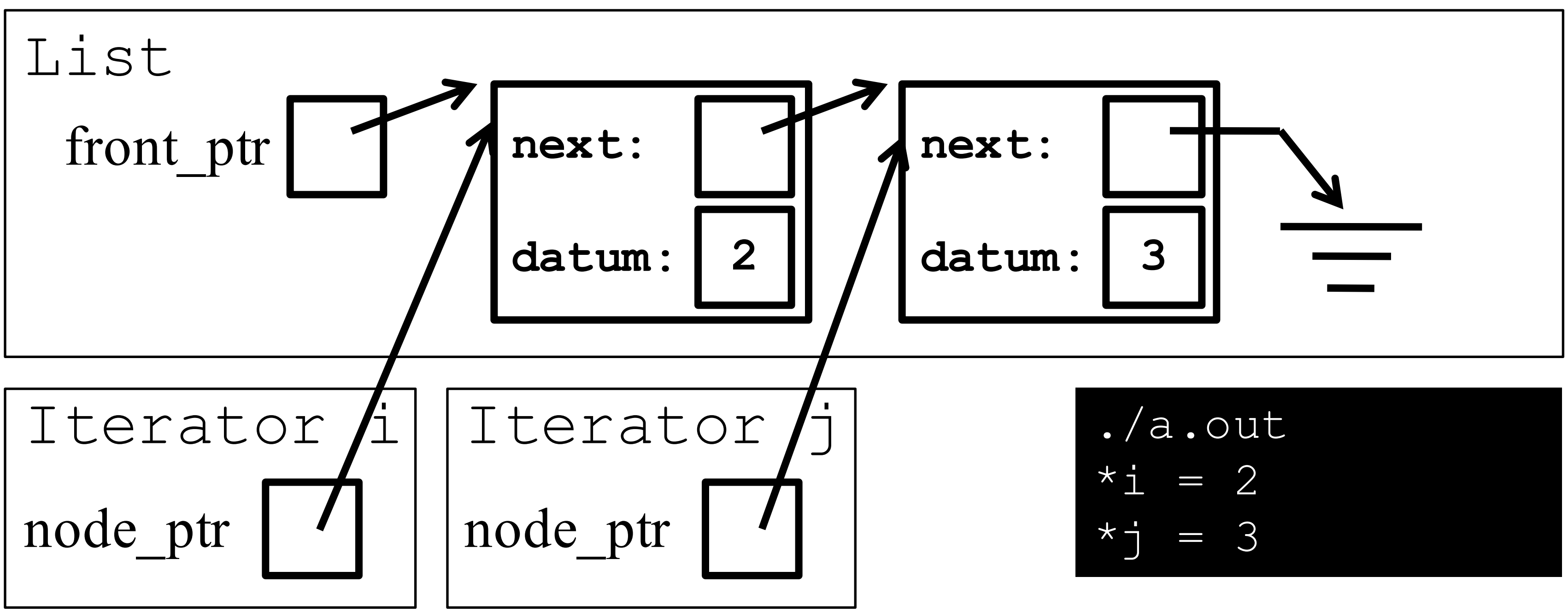
Multiple iterators

```
List<int> l; //fill with ( 2 3 )
List<int>::Iterator i = l.begin();
List<int>::Iterator j = i;
++j;
cout << "*i = " << *i << endl;
cout << "*j = " << *j << endl;
```



Multiple iterators

```
List<int> l; //fill with ( 2 3 )
List<int>::Iterator i = l.begin();
List<int>::Iterator j = i;
++j;
cout << "*i = " << *i << endl;
cout << "*j = " << *j << endl;
```




```
./a.out
*i = 2
*j = 3
```

Exercise: multiple iterators

- Write a function to check a list for duplicates
- Here is code that works for an array

```
//EFFECTS: returns true if a contains no duplicates
bool no_duplicates(int a[], int size) {
  for (int i=0; i<size; ++i) {
    for (int j=i+1; j<size; ++j) {
      if (a[i] == a[j]) return false;
    }
  }
  return true;
}
```

- Write a new version of `no_duplicates` for a `List<int>`

Discussion

- Our Iterator class has one member variable "`Node *node_ptr`" which is a pointer to dynamically allocated memory.
- Question: do we need to implement the Big Three? Why?

Iterator invalidation

- Once an iterator is created, if the underlying container is modified, the iterator *may* become invalid
  - This is dependent on the container

- If the iterator is invalidated, its behavior is undefined, much like an invalid pointer

- The intuition behind this is that the iterator depends on the representation of the container – if that changes, the iterator is likely to miss an element or return an element that no longer exists

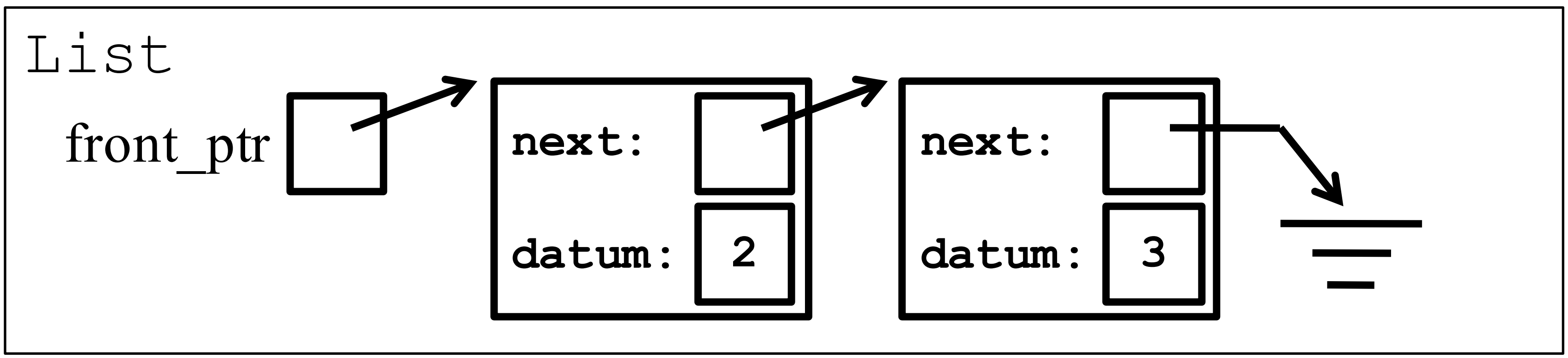
Iterator invalidation

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  ++i; ++i; ++i;
}
```

- Draw a picture of the `List`, `Node`, and `Iterator` objects
- What's wrong with this code?

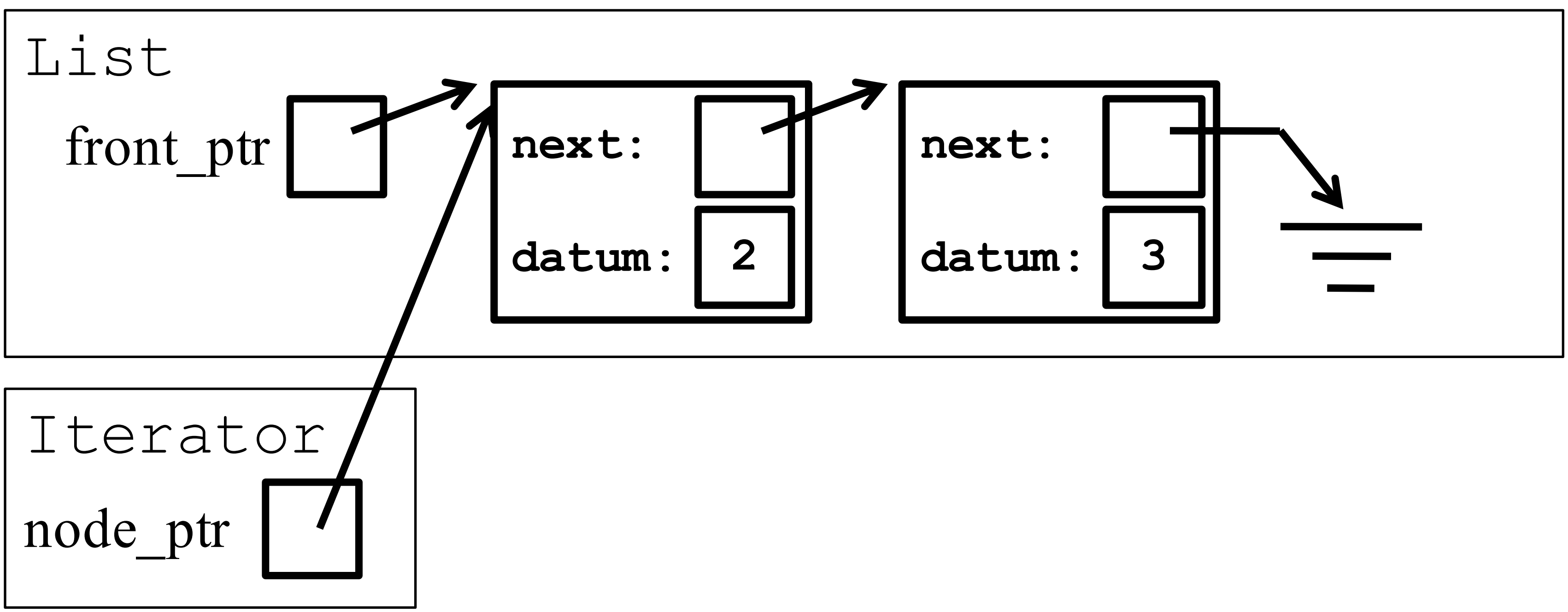
Iterator invalidation

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  ++i; ++i; ++i;
}
```



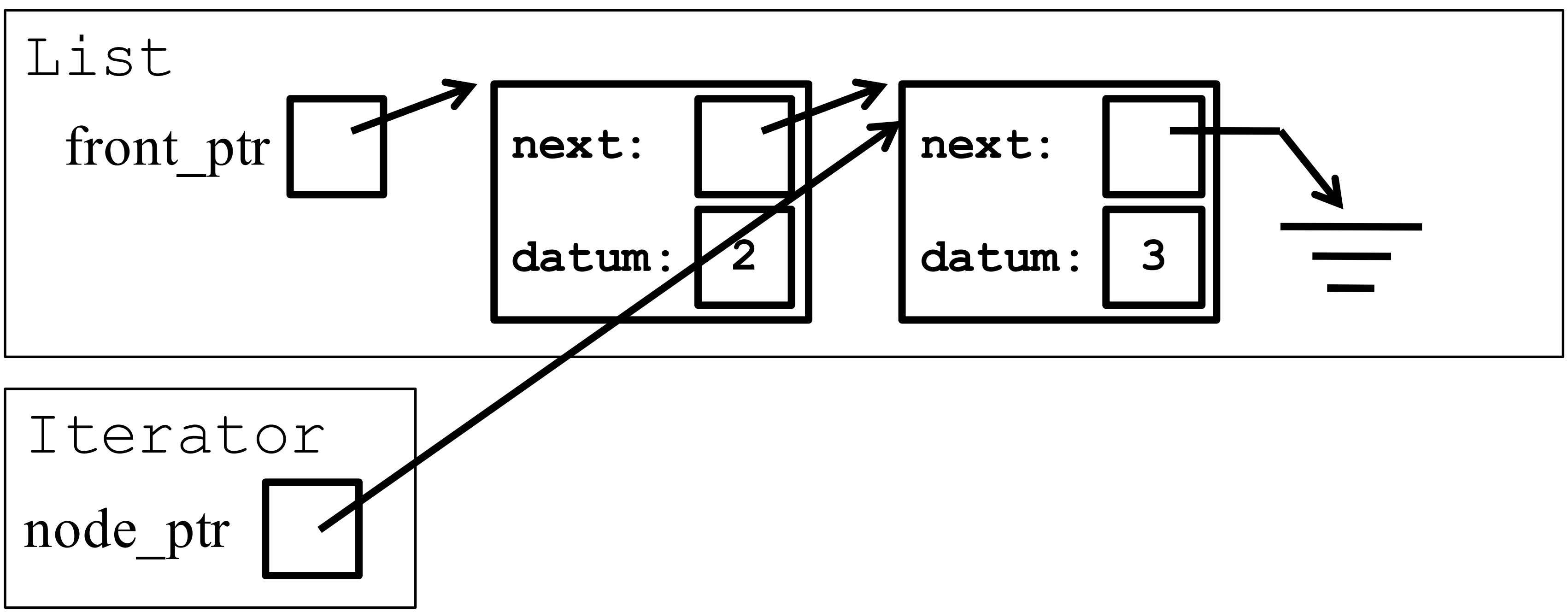
Iterator invalidation

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  ++i; ++i; ++i;
}
```



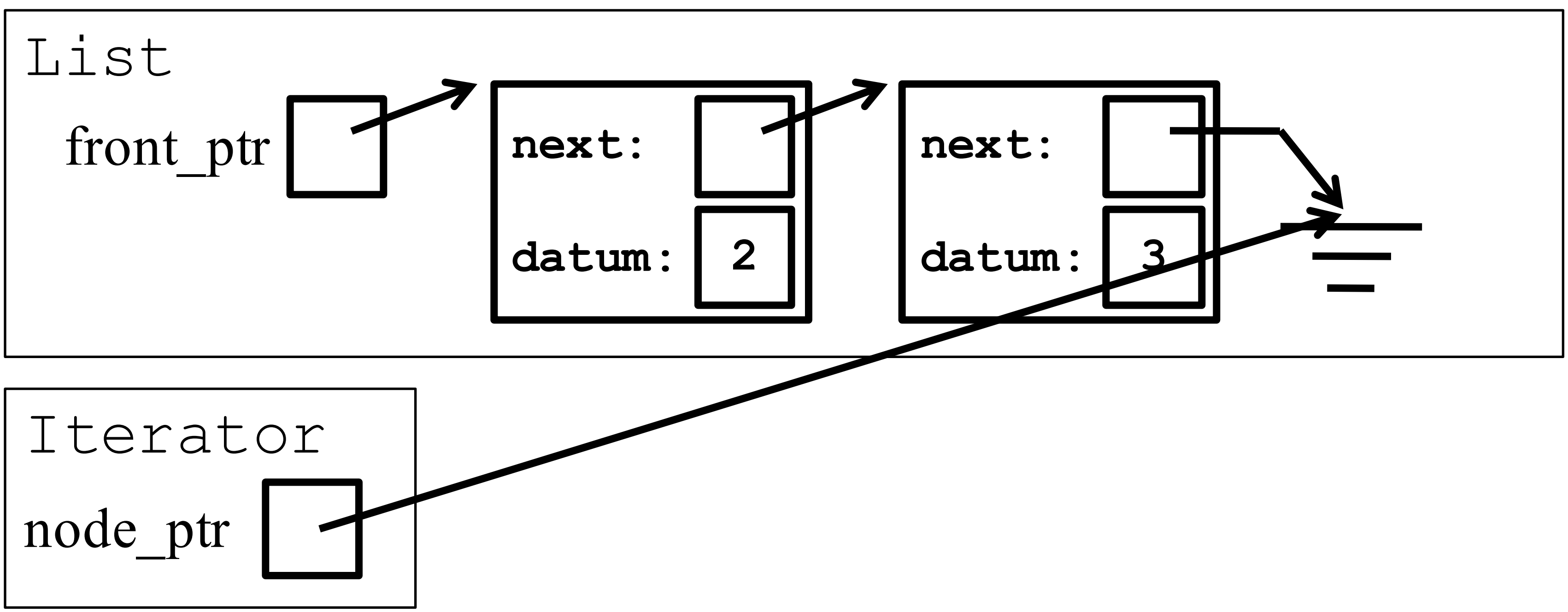
Iterator invalidation

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  ++i; ++i; ++i;
}
```



Iterator invalidation

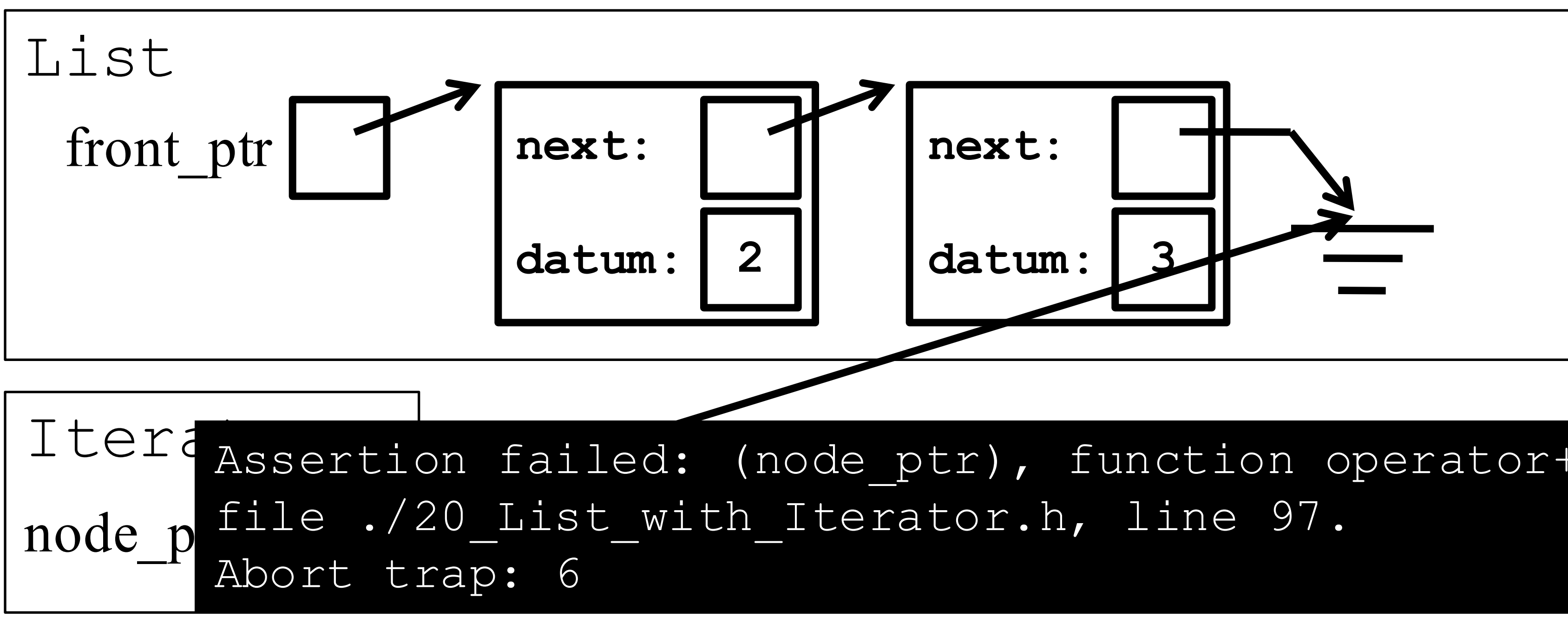
```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  ++i; ++i; ++i;
}
```



Iterator invalidation

```
Iterator& operator++ () {
  assert(node_ptr);
  node_ptr = node_ptr->next;
  return *this;
}
```

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  ++i; ++i; ++i;
}
```

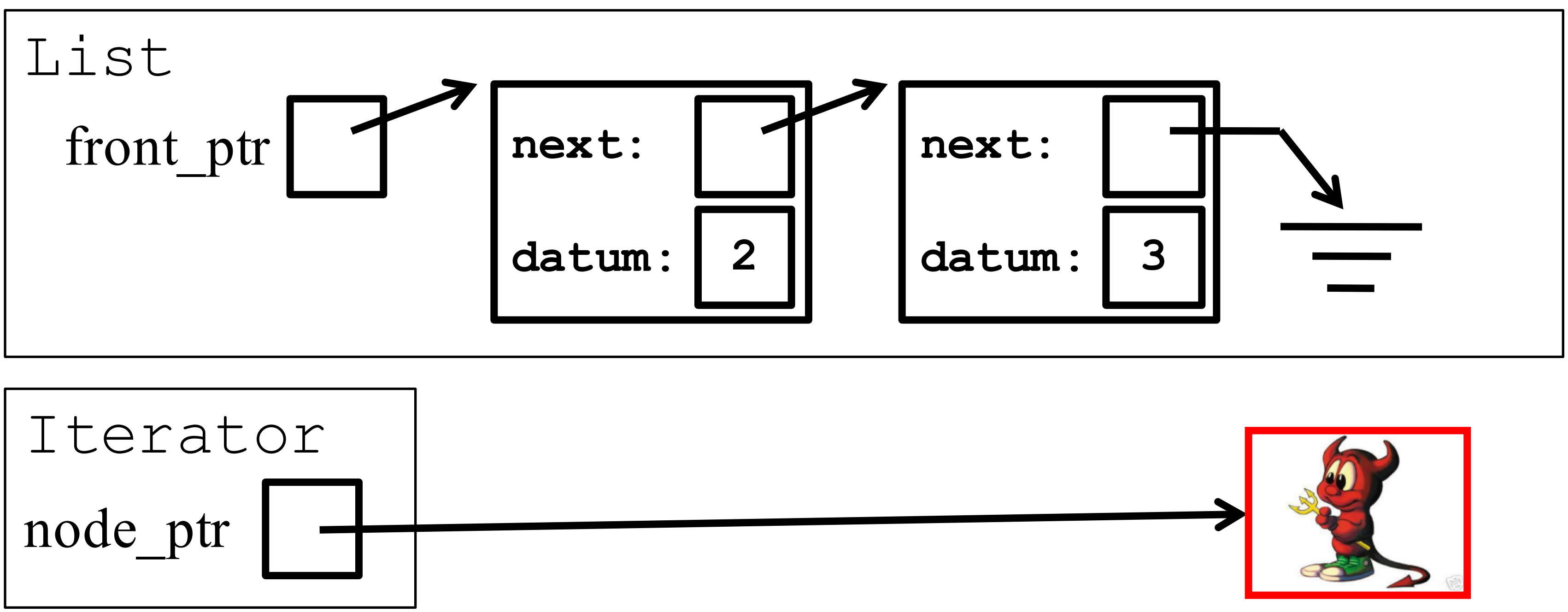



```
Assertion failed: (node_ptr), function operator++,
file ./20_List_with_Iterator.h, line 97.
Abort trap: 6
```

Iterator invalidation

```
Iterator& operator++ () {
  assert(node_ptr);
  node_ptr = node_ptr->next;
  return *this;
}
```

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  ++i; ++i; ++i; //Technically, it's undefined,
}                //e.g., if we #define NDEBUG
```



Iterator invalidation

```
Iterator& operator++ () {
  assert(node_ptr);
  node_ptr = node_ptr->next;
  return *this;
}
```

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  ++i; ++i; ++i; //Technically, it's undefined
}
```

- It's *your* responsibility to keep your `Iterator` within the bounds of the container

```
Assertion failed: (node_ptr), function operator++,
file ./20_List_with_Iterator.h, line 97.
Abort trap: 6
```

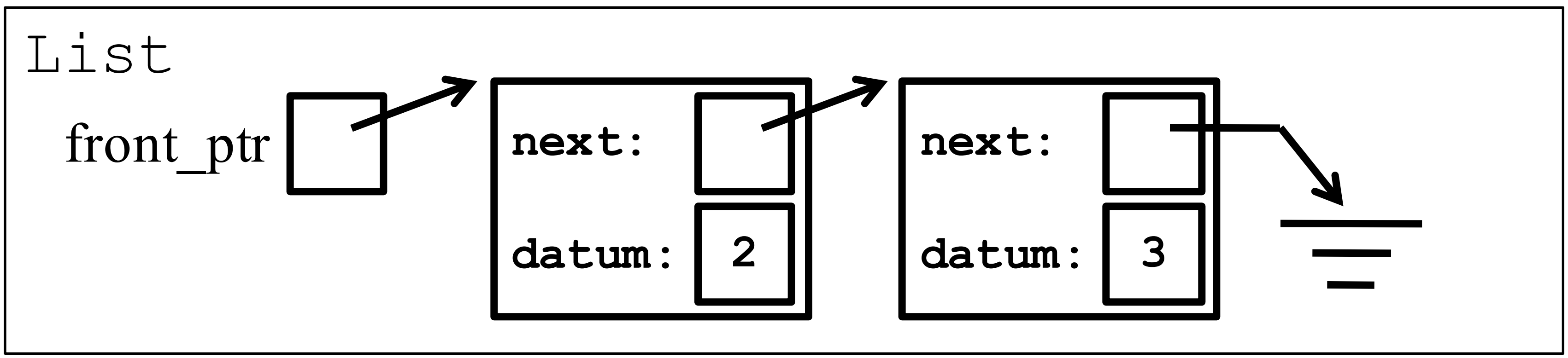
Iterator invalidation

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  l.pop_front();
  cout << *i;
}
```

- Draw a picture of the `List`, `Node`, and `Iterator` objects
- What's wrong with this code?

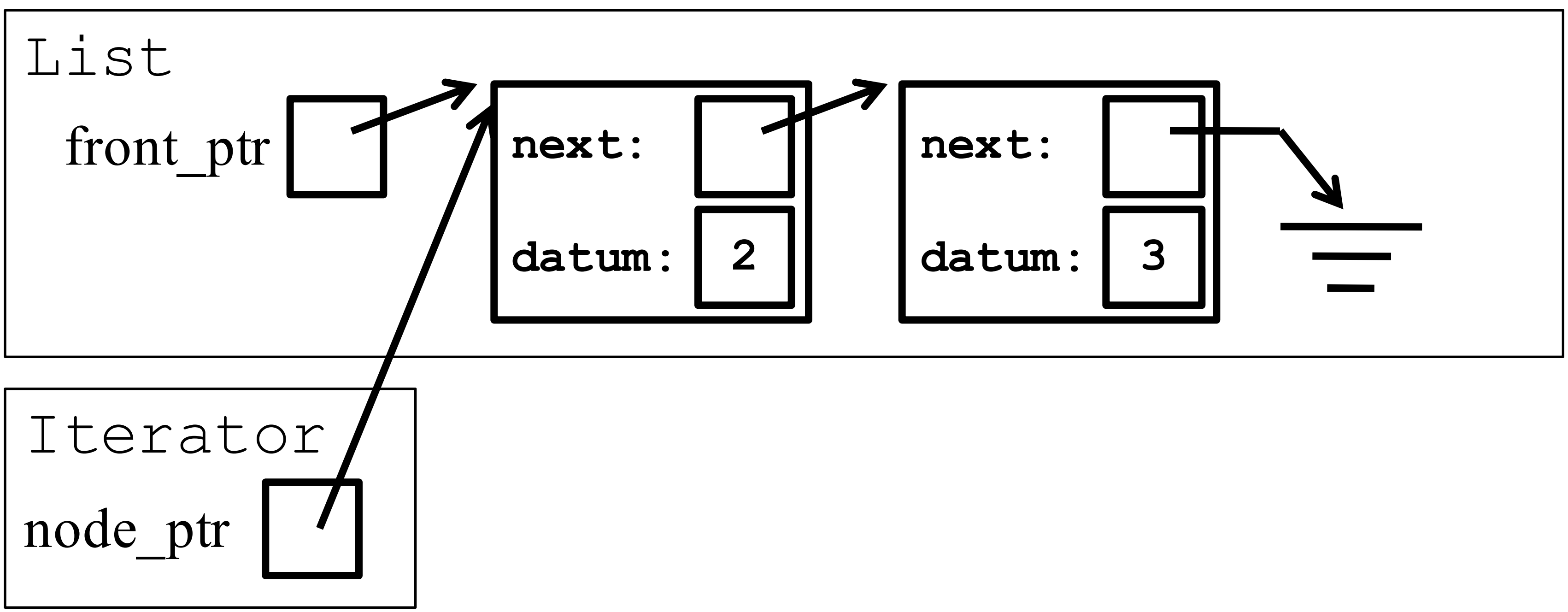
Iterator invalidation

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  l.pop_front();
  cout << *i;
}
```



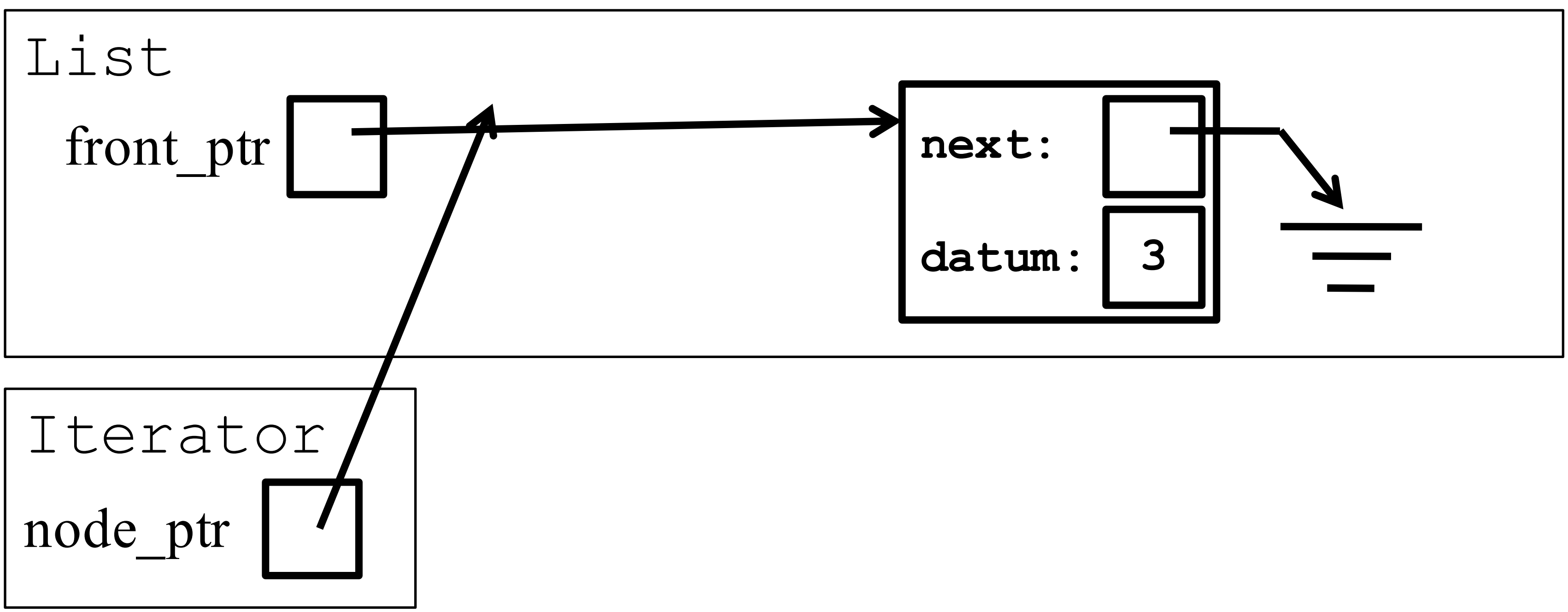
Iterator invalidation

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  l.pop_front();
  cout << *i;
}
```



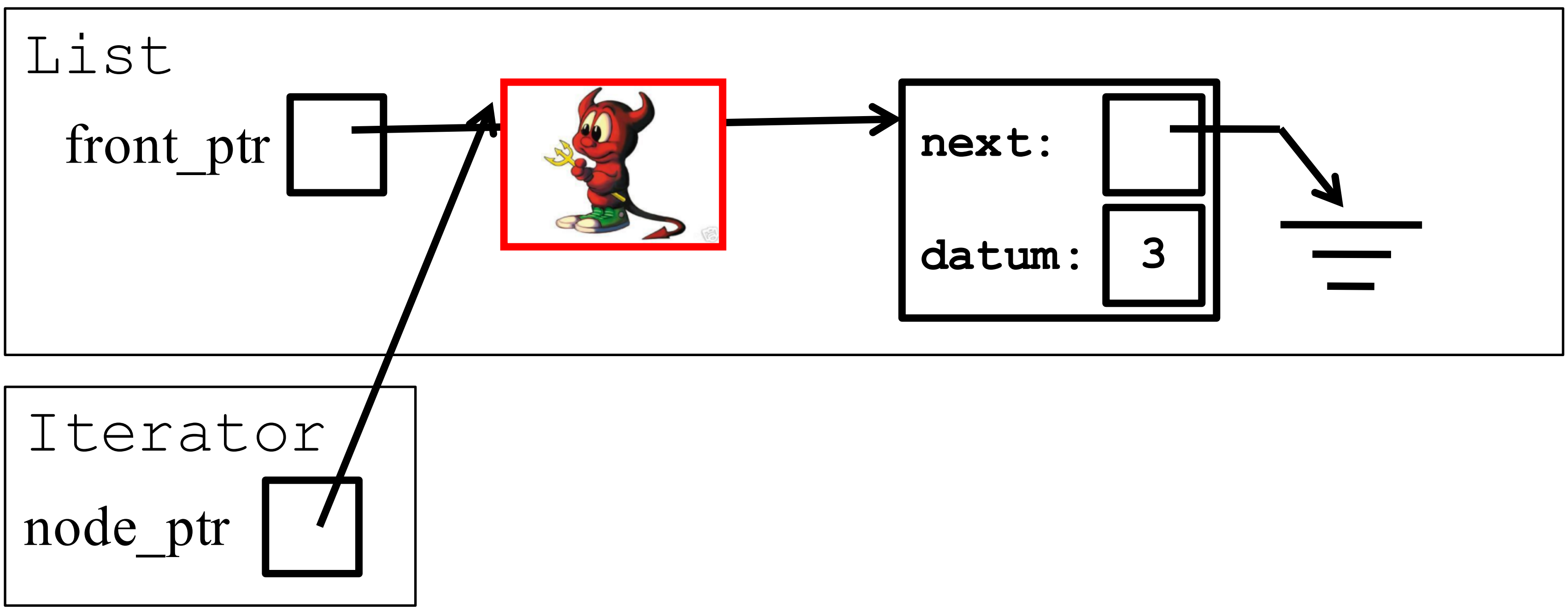
Iterator invalidation

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  l.pop_front();
  cout << *i;
}
```



Iterator invalidation

```
#include "List.h"
int main() {
  List<int> l; //fill with ( 2 3 )
  List<int>::Iterator i = l.begin();
  l.pop_front();
  cout << *i; //Undefined!
}
```



Putting it all together

- Recall last lecture:

```
class Gorilla {
  // OVERVIEW a really big class :)
  string name;
public:
  Gorilla(const string &name_in) : name(name_in) {
    cout << "Gorilla ctor: " << name << "\n";
  }

  ~Gorilla() {
    cout << "Gorilla dtor: " << name << "\n";
  }

   string get_name() { return name; } // new
};
```

```
Gorilla ctor: Colo
Gorilla ctor: Koko
```

Putting it all together

```
List<Gorilla*> zoo;
zoo.push_front(new Gorilla("Colo"));
zoo.push_front(new Gorilla("Koko"));
```

- Destructors haven’t run yet, since we’re still coding ☺

Putting it all together

```
Gorilla ctor: Colo
Gorilla ctor: Koko
Koko
Colo
```

```
List<Gorilla*> zoo;
zoo.push_front(new Gorilla("Colo"));
zoo.push_front(new Gorilla("Koko"));

for (List<Gorilla*>::Iterator i=zoo.begin();
     i != zoo.end(); ++i) {
  Gorilla *g = *i;
  cout << g->get_name() << endl;
}
```

dereferences `Iterator`, returning a `Gorilla*`

dereferences `Gorilla*`, calling member function `get_name()`

Putting it all together

```
Gorilla ctor: Colo
Gorilla ctor: Koko
Koko
Colo
```

```
List<Gorilla*> zoo;
zoo.push_front(new Gorilla("Colo"));
zoo.push_front(new Gorilla("Koko"));

for (List<Gorilla*>::Iterator i=zoo.begin();
      i != zoo.end(); ++i)
   cout << (**i).get_name() << endl;
```

dereferences `Iterator`, returning a `Gorilla*`

calls `Gorilla` member function

dereferences `Gorilla*`, returning a `Gorilla&`

Putting it all together

```
Gorilla ctor: Colo
Gorilla ctor: Koko
Koko
Colo
Gorilla dtor: Koko
Gorilla dtor: Colo
```

```
List<Gorilla*> zoo;
zoo.push_front(new Gorilla("Colo"));
zoo.push_front(new Gorilla("Koko"));

for (List<Gorilla*>::Iterator i=zoo.begin();
     i != zoo.end(); ++i)
  cout << (**i).get_name() << endl;

for (List<Gorilla*>::Iterator i=zoo.begin();
     i != zoo.end(); ++i) {
  delete *i; *i=0;
}
cout << endl;
```

Need to clean up after yourself: if you call `new`, you must call `delete`.

If we have time ...

- Let's take a sneak peak at C++11
- C++11 added new features to the C++ language
- C++14 fixed some of the bugs in C++11

- Compile C++11 code on OSX like normal
- Compile on Windows (Visual Studio) like normal
- Compile on Linux with `g++ -std=c++11`
  - If it's CAEN Linux, run this command first: `module load gcc`

Sneak peak at C++11 / C++14

- This is a lot of typing!

```
for (List<Gorilla*>::Iterator i=zoo.begin();
     i != zoo.end(); ++i)
  cout << (**i).get_name() << endl;
```

- Same code, using C++11 features

```
for (auto i:zoo) {
  cout << i->get_name() << endl;
}
```

Sneak peak at C++11 / C++14

```
for (auto i:zoo) {
  cout << i->get_name() << endl;
}
```

- The `auto` specifier automatically figures out the type
- Uses *dereferenced* `zoo.begin()` for the type of `i`
  - `*zoo.begin()`
  - `zoo.begin()` returns `List<Gorilla*>::Iterator`
  - Dereferencing `List<Gorilla*>::Iterator` gives us a `Gorilla*` type

Sneak peak at C++11 / C++14

```
for (auto i:zoo) {
  cout << i->get_name() << endl;
}
```

- *Range for* automatically starts at `zoo.begin()` and ends before `zoo.end()`
- *Range for* works with any container that has a "standard" iterator
  - Like our `List`

Sneak peak at C++11 / C++14

```
for (auto i:zoo) {
  cout << i->get_name() << endl;
}
```

- `i` is an access by value, so it's a copy of a `Gorilla*`
- We can dereference the pointer and call a member function using only the `operator->`

- Cool!