Ve 280

Programming and Introductory Data Structures

Linked List; Template; Container of Pointers

Announcement

- Project 4 announced
  - On abstract data type, inheritance, and interface
  - Due by the midnight of July 24^{th}.

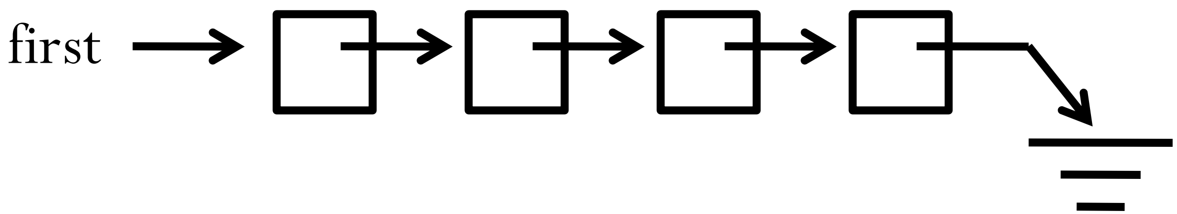
Outline

- Double-Ended Linked Lists
- Templates
- Container of Pointers

Linked Lists

Double-ended list

- What if we wanted to insert something at the end of the list?
- Intuitively, with the current representation, we need to walk down the list until we found "the last element", and then insert it there.



- That's not very efficient, because we have to go through every element to insert something at the tail.
- Instead, we'll change our concrete representation to track both the front and the back of our list.

Linked Lists

Double-ended list

- The new representational invariant has **two** node pointers:

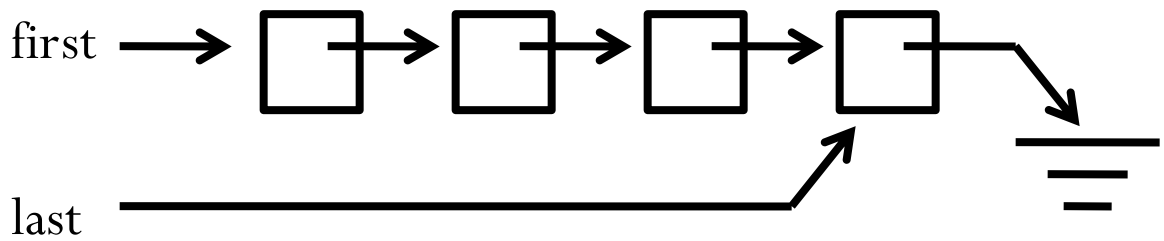
```
class IntList {
    node *first;
    node *last;
  public:
    ...
};
```

- The invariant on `first` is unchanged.
- The invariant on `last` is:
  - `last` points to the last node of the list if it is not empty, and is NULL otherwise.

Linked Lists

Double-ended list

- So, in an empty list, both `first` and `last` point to NULL.
- However, if the list is non-empty, they look like this:



- Question: Adding this new data member, what methods should be changed?
  - Answer: remove, insert, and default/copy constructor should be re-written
- In lecture, we'll only write a new method, `insertLast`, which inserts a node at the tail of the linked list

Linked Lists

Double-ended list

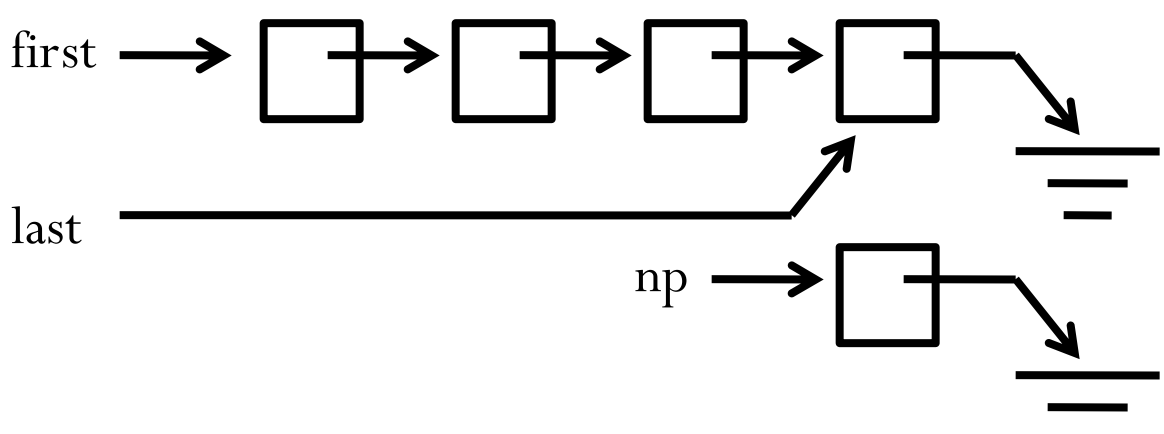
- First, we create the new node, and establish its invariants:

```
void IntList::insertLast(int v) {
  node *np = new node;
  np->next = NULL;
  np->value = v;
  ...
}
```

Linked Lists

Double-ended list

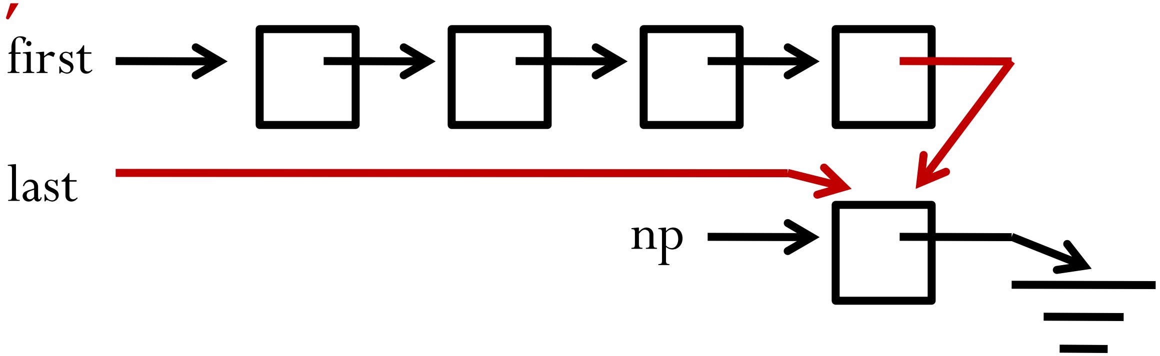
- To actually insert, there are two cases:
  - If the list is empty, we need to reestablish the invariants on `first` **and** `last` (the new node is both the first and last node of the list)
  - If the list is **not** empty, there are two broken invariants. The "old" `last->next` element (incorrectly) points to NULL, and the `last` field no longer points to the last element.



Linked Lists

Double-ended list

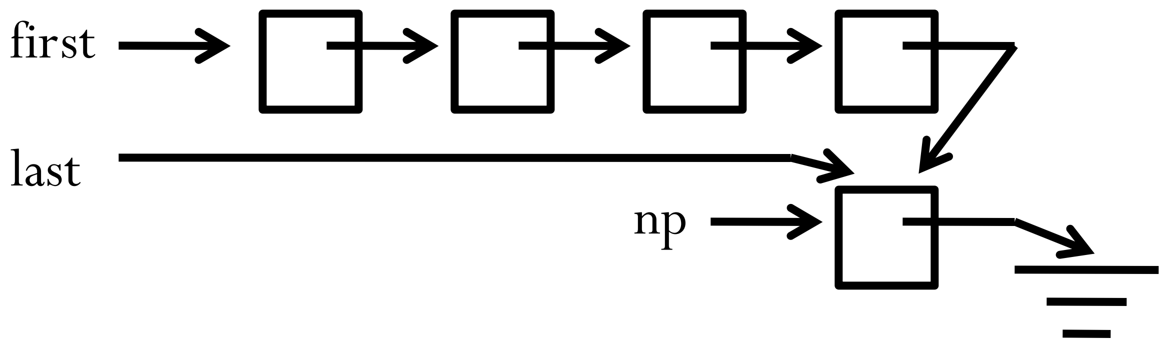
```
void IntList::insertLast(int v) {
  node *np = new node;
  np->next = NULL;
  np->value = v;
  if (isEmpty()) {
    first = last = np;
  }
  else {
    last->next = np;
    last = np;
  }
}
```



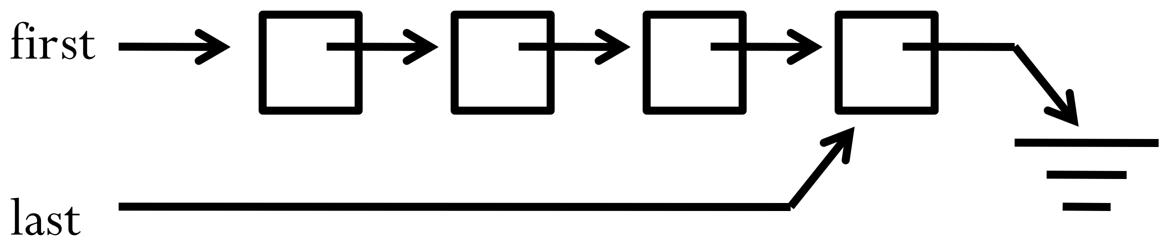
Linked Lists

Double-ended list

- This is efficient, but only for insertion.



- **Question**: Is removal **from the end** efficient or not? Why?



Linked Lists

Double-ended list

- To make removal from the end efficient, as well, we have to have a **doubly-linked** list, so we can go forward **and** backward.
- To do this, we're going to change the representation again.
- In our new representation, a node is:

```
struct node {
  node *next;
  node *prev;
  int   value;
};
```

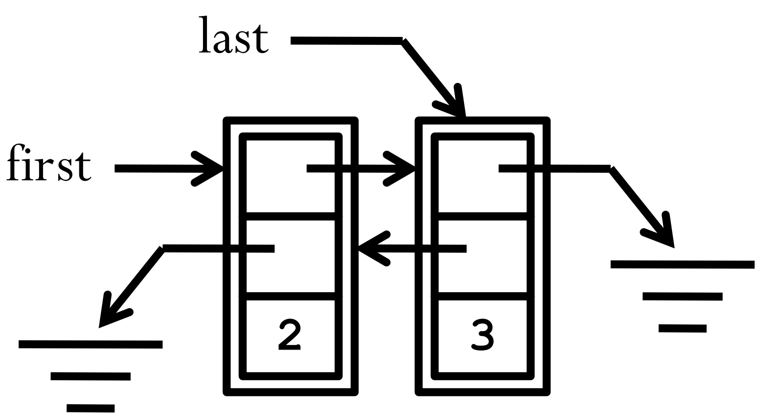
- The `next` and `value` fields are the same as before.
- The `prev` field's invariant is:
  - The `prev` field points to the previous node in the list, or NULL if no such node exists (e.g., the current node is the first node).

Linked Lists

Double-ended list

- With this representation, an empty list is unchanged: both "first" and "last" are NULL.

- While the list (2, 3) would look like this:



- We will implement each method in project five.

Outline

- Double-Ended Linked Lists
- Templates
- Container of Pointers

Containers

Introduction

- Things like `IntSet` and `IntList` are often called **containers** or **container classes**.
- Their purpose in life is to "**contain**" other objects, and they generally have no intrinsic meaning on their own.

- **Question**: how can we write a `CharList`?
  - **Answer**: we have to write almost **exactly** the same code, changing each instance of `int` to `char`.

Containers

Introduction

- IntList versus CharList

```
struct node {
   node *next;
   int    v;
};

class IntList {
    node *first;
  public:
    void insert(int v);
    int remove();
    ...
};
```

```
struct node {
    node *next;
    char  v;
};

class CharList {
    node *first;
  public:
    void insert(char v);
    char remove();
    ...
};
```

Containers

Polymorphism

- It turns out we need write the code **only once**, and can reuse it for each different type we want to use it for.

- Reusing code for **different types** is called **polymorphism** or **polymorphic** code:
  - "poly" meaning "many" and "morph" meaning "forms".

- One way to achieve polymorphism in C++ is **templated containers**.

Containers

Templating

- Often, any **single** container needs to contain only **one type** of object.

- If this is the case, then you can use a C++ mechanism called "**templates**" to write the container code only once.

- You can then use that single implementation to realize any container of any **single** type.

Containers

Templating

- Consider the following fragments defining a **list-of-int** and a **list-of-char**:

```
struct node {
   node *next;
   int   v;
};

class List {
    node *first;
  public:
    void insert(int v);
    int remove();
    ...
};
```

```
struct node {
    node *next;
    char  v;
};

class List {
    node *first;
  public:
    void insert(char v);
    char remove();
    ...
};
```

Containers

Templating

- It's like someone took the list-of-int definition and **replaced** each instance of `int` with an instance of `char`.
- Templates are a mechanism to do exactly that.

```
struct node {
   node *next;
   int   v;
};

class List {
    node *first;
  public:
    void insert(int v);
    int remove();
    ...
};
```

```
struct node {
    node *next;
    char  v;
};

class List {
    node *first;
  public:
    void insert(char v);
    char remove();
    ...
};
```

Containers

Templates

- The intuition behind templates is that they are code with the "**type name**" left as a **(compile-time) parameter**.
- So, they are another form of **parametric generalization** except this time, the **parameter is a type**, not a variable.

- To start, you first need to declare that something will be a template:

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

`T` stands for "the name of the type contained by this `List`".

By convention, we always use `T` for the name of the "type" over which the template is parameterized.

Containers

Templates

- The intuition behind templates is that they are code with the "**type name**" left as a **(compile-time) parameter**.
- So, they are another form of **parametric generalization** except this time, the **parameter is a type**, not a variable.

- To start, you first need to declare that something will be a template:

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

C++ uses "class" to mean "type" here, but that doesn't mean only class names can serve as "T". Any valid type such as int and double can.

Containers

Templates

```
template <class T>
class List {
 public:
  bool isEmpty();
  void insert(T v);
  T remove();

  List();
  List(const List &l);
  List &operator=(const List &l);
  ~List();

 private:
  ...
};
```

Now, you write the definition of the List, using T where you mean "the type of thing held in the list".

<u>Note</u>: For this example, we put the public part first, and the private part after

Containers

Templates

```
template <class T>
class List {
 public:
  bool isEmpty();
  void insert(T v);
  T remove();

  List();
  List(const List &l);
  List &operator=(const List &l);
  ~List();

 private:
  ...
};
```

Note: The only thing different between this definition and the `IntList` one is that we've used `T` rather than `int` to name objects held in this list.

This will work for any type.

Containers

Templates

- We also have to pick a representation for the node contained by this List, and that representation must also be parameterized by T.
  - The "node" type has to have an element of type T.

- We do this by creating a **private** type, which is part of this class definition:

```
private:
  struct node {
    node *next;
    T     v;
  };
```

Containers

Templates

```
template <class T>
class List {
 public:
  // methods

  // constructors/destructor

 private:
  struct node {
    node *next;
    T     v;
  };
  ...
};
```

So, this type "node" is only available to implementations of this class' methods.

On the other hand, this node will hold only objects of the appropriate type.

Containers

Templates

```
template <class T>
class List {
 public:
  // methods/constructors/destructor
 private:
  struct node {
     node *next;
     T     v;
  };
  node *first;
  void removeAll();
  void copyList (node* np);
};
```

The rest of the class definition is just what you expect

Containers

Templates

- All that is left is to define each of the method bodies.
- Each **method** must also be declared as a "**templated**" method and we do that in much the same way as we do for the class definition.
- Each function begins with the "template declaration":

```
template <class T>
```

- And each method name must be put in the "`List<T>`" namespace:

```
template <class T>
void List<T>::isEmpty() {
  return (first == NULL);
}
```

Containers

Templates

- `isEmpty()` isn't that interesting, since it doesn't use any `T`'s.
- Here is a more interesting one:

```
template <class T>
void List<T>::insert(T v) {
    node *np = new node;
    np->next = first;
    np->v = v;
    first = np;
}
```

- The argument, `v`, is of type T which is exactly the same type as `np->v`.

Containers

Templates

- The `#include` and compiling of templates are a little bit different.
- You should put your class member function definition also in the .h file, following class definition. So, there is no .cpp for member functions

list.h

```
template <class T>
class List {
    ...
};
template <class T>
void List<T>::insert(T v) {
    ...
}
```

Containers

Templates

- The function header of the constructor is

```
List<T>::List()
List<T>::List(const List &l)
```

Must have <T>!

No <T>!

No <T>!

- The function header of the destructor is

```
List<T>::~List()
```

Must have <T>!

No <T>!

- The function header of the assignment operator is

```
List<T> &List<T>::operator=(const List &l)
```

Must have <T>!

No <T>!

Containers

Templates

- To use templates, you specify the type T when creating the container object.

```
// Create a static list of integers
List<int> li;
// Create a dynamic list of integers
List<int> *lip = new List<int>;
// Create a dynamic list of doubles.
List<double> *ldp = new List<double>;
```

- Thereafter, you just use these normally.

Outline

- Double-Ended Linked Lists
- Templates
- Container of Pointers

Container of Pointers

Introduction

- So far, we've inserted and removed elements **by value**.
- In other words, we **copy** the things we insert into/remove from the container.
- Copying elements by value is fine for types with "small" representations.
  - For example, all of the built-in types.
- This is **not** true for "large" types – any nontrivial struct or class would be expensive to pass by value, because you'll spend a lot of your time copying.

Container of Pointers

Introduction

- **Question**: suppose we had a list of `BigThings`. When you call `insert()`, how many copy-related operations will be done?
- Answer: Twice
  - First time as an argument to `insert()`, and
  - Second time when you store the item in the list node.

```
foo.insert(A_Big_Thing);

void List::insert(BigThing v) {
  node *np = new node;
  np->value = v;
  np->next = first;
  first = np;
}
```

This is unacceptable!

Container of Pointers

Introduction

- Instead of copying large types by value, we usually insert and remove them **by reference**.
  - The container stores **pointers-to-`BigThing`** instead.

```
struct node {
  node *next;
  BigThing *value;
};
```

- So, if we have a `BigThing` list, its `insert` and `remove` methods have the following type signatures.

```
void insert(BigThing *v);
BigThing *remove();
```

Container of Pointers

Introduction

```
struct node {
  node *next;
  BigThing *value;
};

void ListBigThing::insert(BigThing *v) {
  node *np = new node;
  np->next = first;
  np->v = v;
  first = np;
}
```

Templated Container of Pointers

Practice: when we define templated container of pointers, we do **NOT**

- define a template on **object**
- and define

```
List<BigThing *> ls;
```

```
template <class T>
class List {
  public:
    ...
    void insert(T v);
    T     remove();
  private:
    struct node {
        node *next;
        T     o;
    };
    ....
};
```

Templated Container of Pointers

Instead, we

- define a template on **pointer**
- and define

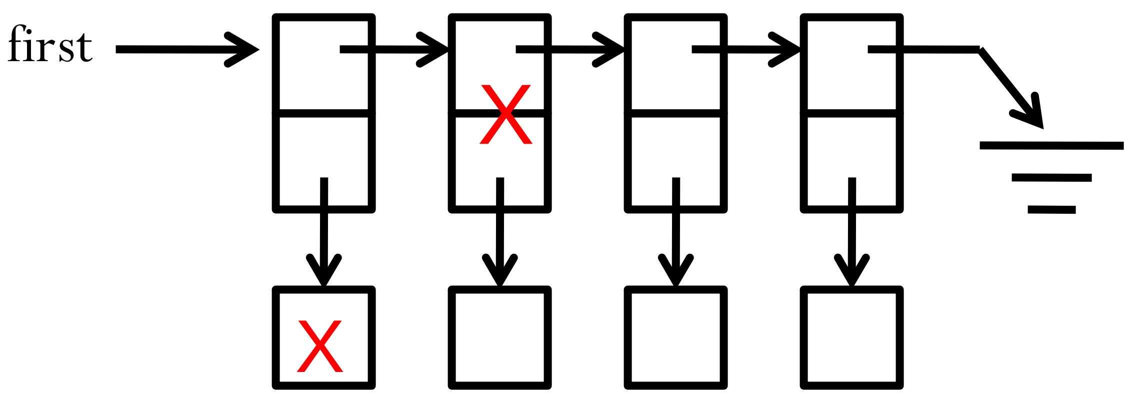
  `List<BigThing> ls;`

```
template <class T>
class List {
  public:
    ...
    void insert(T *v);
    T    *remove();
  private:
    struct node {
        node *next;
        T    *o;
    };
    ....
};
```

Container of Pointers

Templates

- Containers-of-pointers are subject to two broad classes of potential bugs:
  1. Using an object after it has been deleted
  2. Leaving an object **orphaned** by **never** deleting it



Container of Pointers

Use

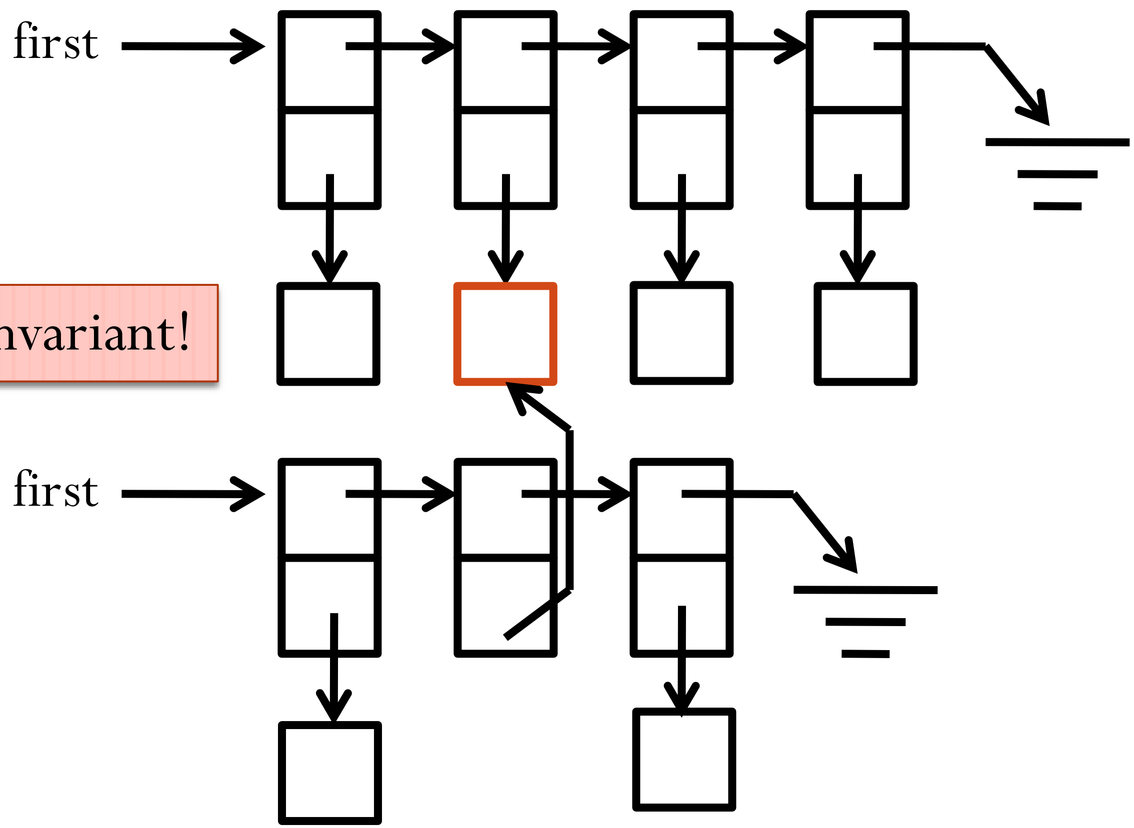
- To avoid the bugs related to container of pointers, one usual "pattern" of using container of pointers has an **invariant**, plus three **rules** of use:
  - **At-most-once invariant**: any object can be linked to at most one container at any time through pointer.

1. **Existence**: An object must be **dynamically allocated** before a pointer to it is inserted.
2. **Ownership**: Once a pointer to an object is inserted, that object becomes the property of the container. No one else may use or modify it in any way.
3. **Conservation**: When a pointer is removed from a container, either the pointer must be inserted into **some** container, or its referent must be **deleted**.

At-most-once Invariant

- Any object can be linked to at most one container at any time through pointer.

Violate the invariant!



Existence Rule

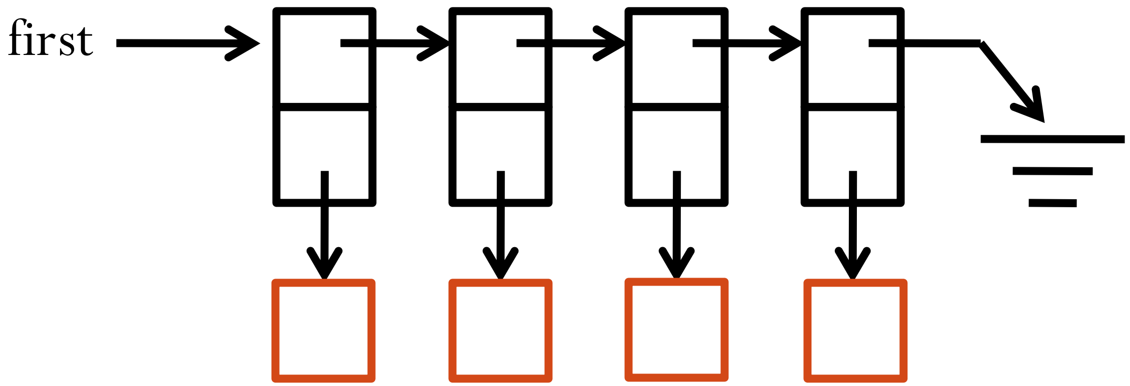
- An object must be **dynamically allocated** before a pointer to it is inserted

```
List<BigThing> l;
// l: container of pointer
BigThing b;
l.insert(&b); ✗
```

```
List<BigThing> l;
// l: container of pointer
BigThing *pb = new BigThing;
l.insert(pb); ✓
```

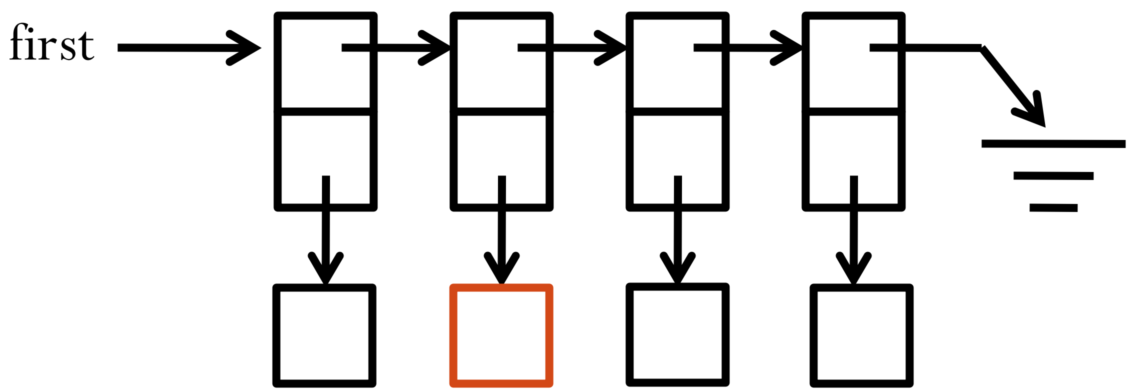
Ownership Rule

- Once a pointer to an object is inserted, that object becomes the property of the container. No one else may use or modify it in any way.



Conservation Rule

- When a pointer is removed from a container, either the pointer must be inserted into **some** container, or its referent must be **deleted**.



- Either be inserted into another container
- Or delete the object

Container of Pointers

Templates

- These three rules have an important implication for any method that **destroys** an existing container.
  - When a container is destroyed, the objects contained in the container should also be deleted!

- There are (at least) two such methods that could destroy a container:
  1. **The destructor**: Destroys an existing instance.
  2. **The assignment operator**: Destroys an existing instance before copying the contents of another instance.

Container of Pointers

Templates

- Consider the following implementation of the destructor for a singly-linked list, using the interface we've discussed so far:

```
template <class T>
List<T>::~List() {
  while (!isEmpty()) {
    remove();
  }
}
```

```
template <class T>
T* List<T>::remove() {
  node *victim = first;
  if(isEmpty()) {
    listIsEmpty e;
    throw e;
  }
  T* result = victim->value;
  first = victim->next;
  delete victim;
  return result;
}
```

```
struct node {
  node *next;
  T* value;
};
```

`int *`

- **<u>Question</u>**: Note that this list stores things **by pointer**. This implementation violates one of the three rules. Which one is violated, and how?

The conservation rule!

Containers

Destructor

- To fix this, we **must** handle the objects we remove:

```
template <class T>
List<T>::~List() {
   while (!isEmpty()) {
      T *op = remove();
      delete op;
   }
}
```

This keeps conservation rule.

Reference

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 13.1 Nodes and Linked Lists
  - Chapter 17 Templates