

# Ve 280

## Programming and Elementary Data Structures

### **Template; Container**

#### **Learning Objectives:**

Understand what is a template and why it is useful.

Understand what is a container, why a container of pointers is useful, what is a polymorphic container.

Know how to implement templated containers, containers of pointers and polymorphic containers.

# Outline

- Templates
- Container of Pointers
- Polymorphic Container

# 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 to 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".

Note: 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>  
bool 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

Add `<T>` every time `List` is used as a class name

- The function header of the constructor is

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

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

Must have `<T>!`

No `<T>!`

Have `<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<T> &l)`**

Must have `<T>!`

Have `<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

- Templates
- Container of Pointers
- Polymorphic Container

# Container of Pointers

## Introduction

- So far, we've inserted and removed elements **by value**.
- In other words, we **copy** the things that 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.



# Select the Correct Answer

- **Question:** suppose we had a list of `BigThings`. When you call `insert()`, how many copy-related operations on `BigThing` will be done?
  - A. 0.
  - B. 1.
  - C. 2.
  - D. It depends on the compiler.

```
foo.insert(A_Big_Thing);

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



# Container of Pointers

## Introduction

- **Question**: suppose we had a list of `BigThings`. When you call `insert()`, how many copy-related operations on `BigThing` 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->value = 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

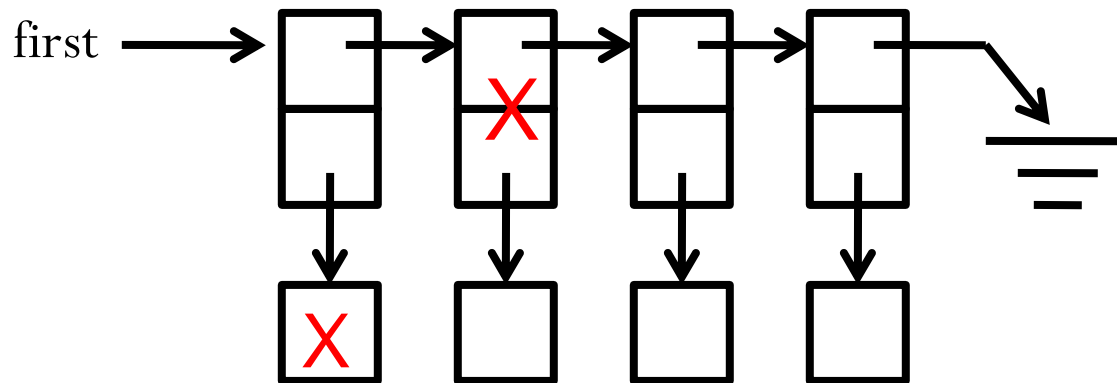
```
List<BigThing> ls;
```

```
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:
  - Using an object after it has been deleted
  - Leaving an object **orphaned** by **never** deleting it



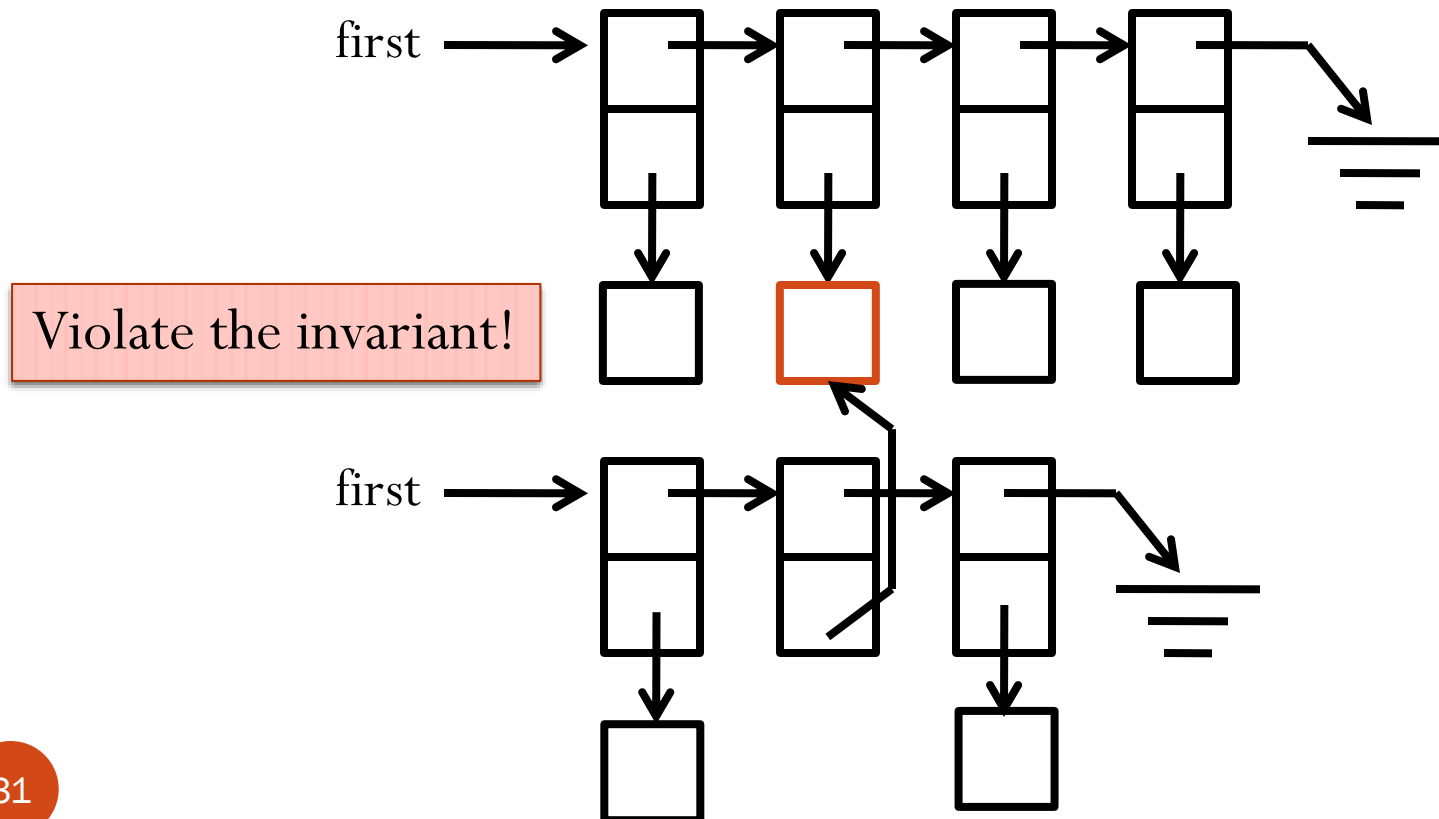
# Container of Pointers

## 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. It can only be modified through the methods of the container.
  - 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.



# Existence Rule

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

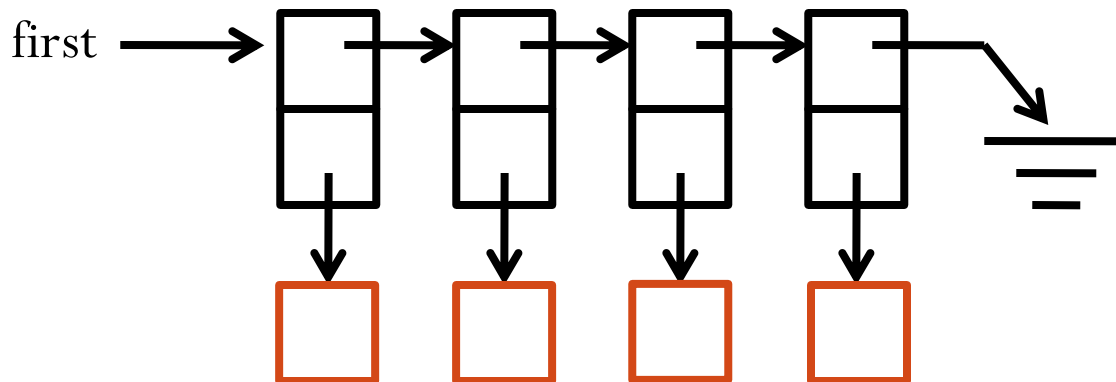
```
void foo(List<BigThing> &l) {  
    // l: container of pointer  
    BigThing b;  
    l.insert(&b) ; ✗  
}
```

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



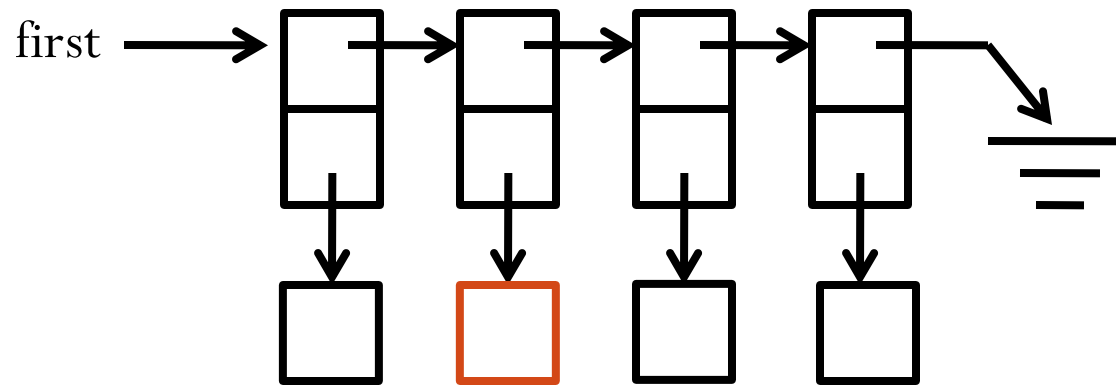
# Ownership Rule

- Once a pointer to an object is inserted, that object becomes the property of the container. It can only be modified through the methods of the container
- Because others may break the representation invariants, e.g., container of positive integers



# 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.



# Which Invariant/Rule Is Violated?

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();
    }
}
```

int \*



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

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

Select **the** correct answer.

- A. At-most-once invariant
- B. Existence rule
- C. Ownership rule
- D. 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;
    }
}
```

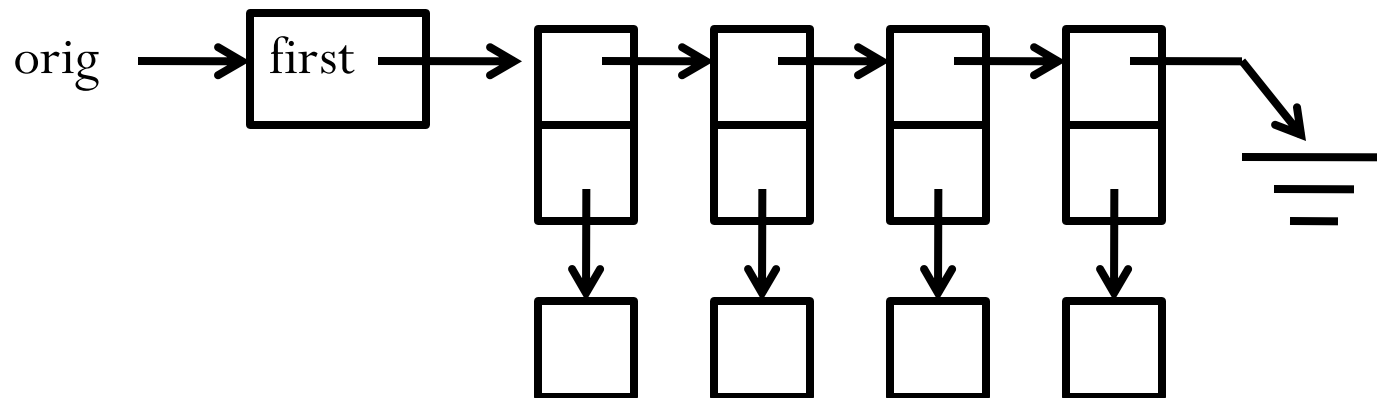


This keeps conservation rule.

# Container of Pointers

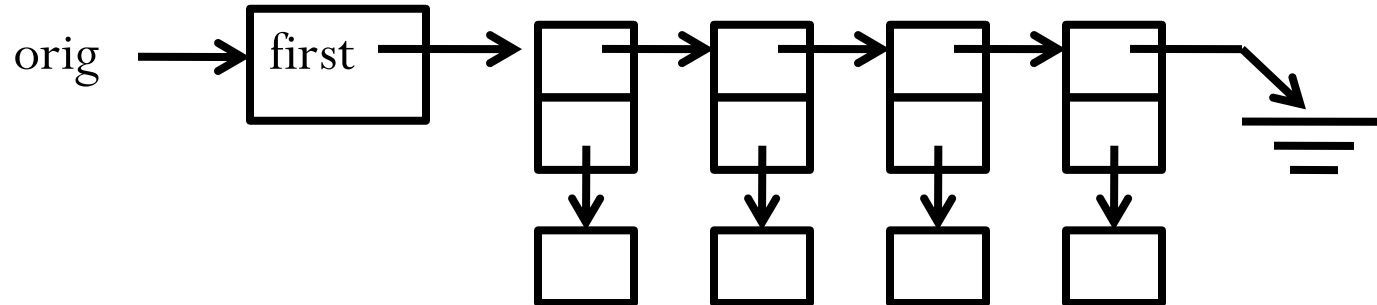
## Copy

- Copy is also tricky for container of pointers.
- Here is the original singly-linked list of  $T^*$ s :





# Which Invariant/Rule Is Violated?



- Here is the old copy constructor and utility function:

```
template <class T>
List<T>::List(const List<T> &l) {
    first = NULL;
    copyList(l.first);
}
```

```
template <class T>
void List<T>::copyList(node *list) {
    if(!list) return;
    copyList(list->next);
    insert(list->value);
}
```

**T \* type**

Select **the** correct answer.

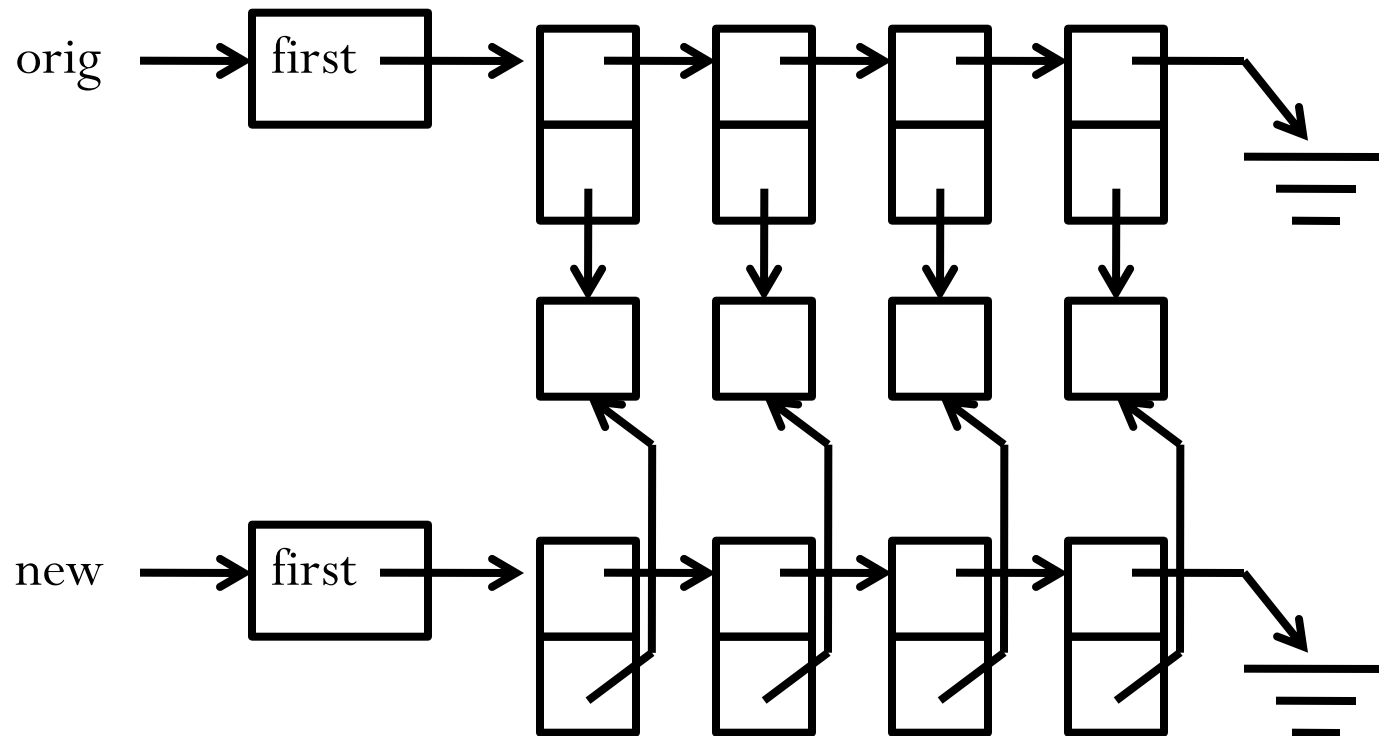
- A. At-most-once invariant
- B. Existence rule
- C. Ownership rule
- D. Conservation rule



# Container of Pointers

## Copy

- The list we would end up with is:



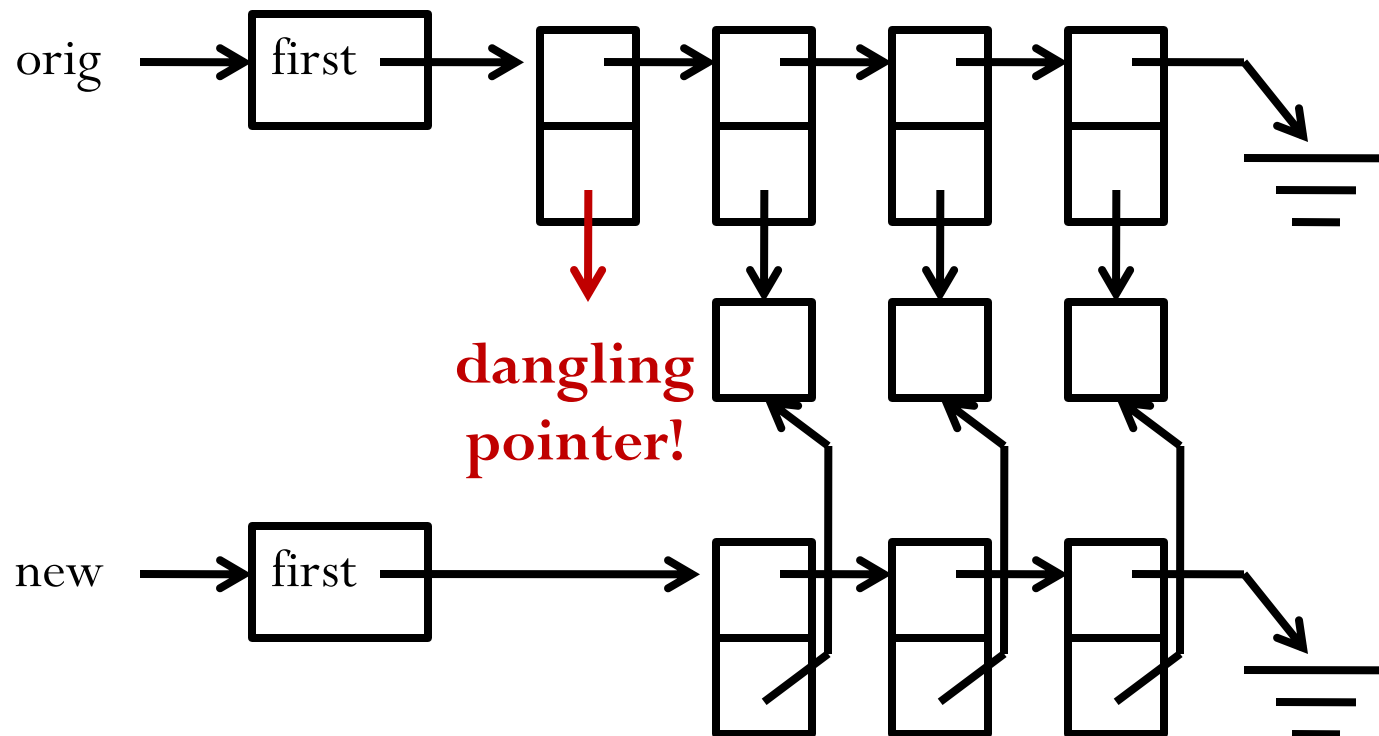
This violates the at-most-once invariant



# Container of Pointers

## Copy

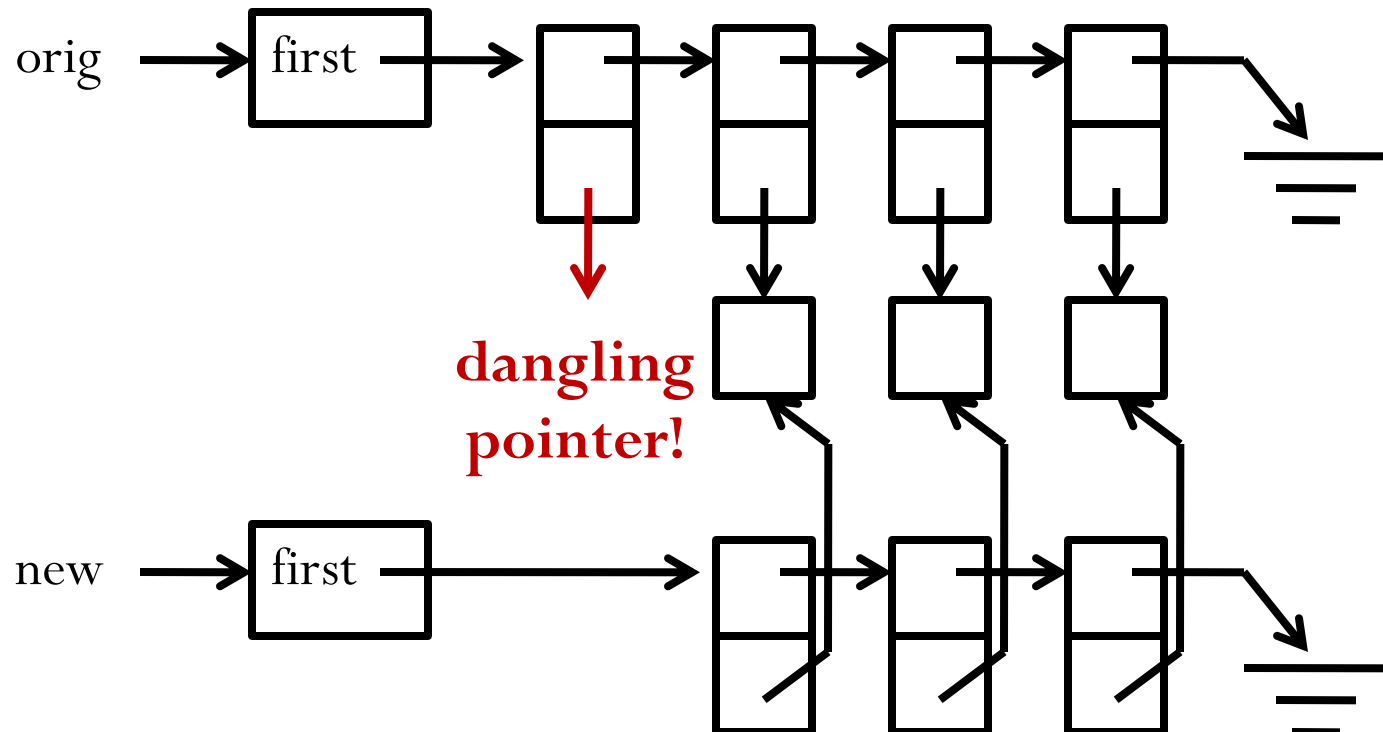
- Now, if we remove the first item of the new list, we delete the first node, and return a pointer to the item.
- The client, after using it, will delete it (Why?).
- Leaving us with this:



# Container of Pointers

## Copy

- Clearly, this is not a good thing because we aren't doing a "full" **deep copy**.
- The list nodes are deeply copied, but the Ts are not since we are copying the pointers, but **not** the objects they point to.



# Container of Pointers

Copy

- Fix:

```
template <class T>
void List<T>::copyList(node *list) {
    if (!list) return;
    copyList(list->next);
    T *o = new T(*list->value);
    insert(o);
}
```

-> binds tighter than \*

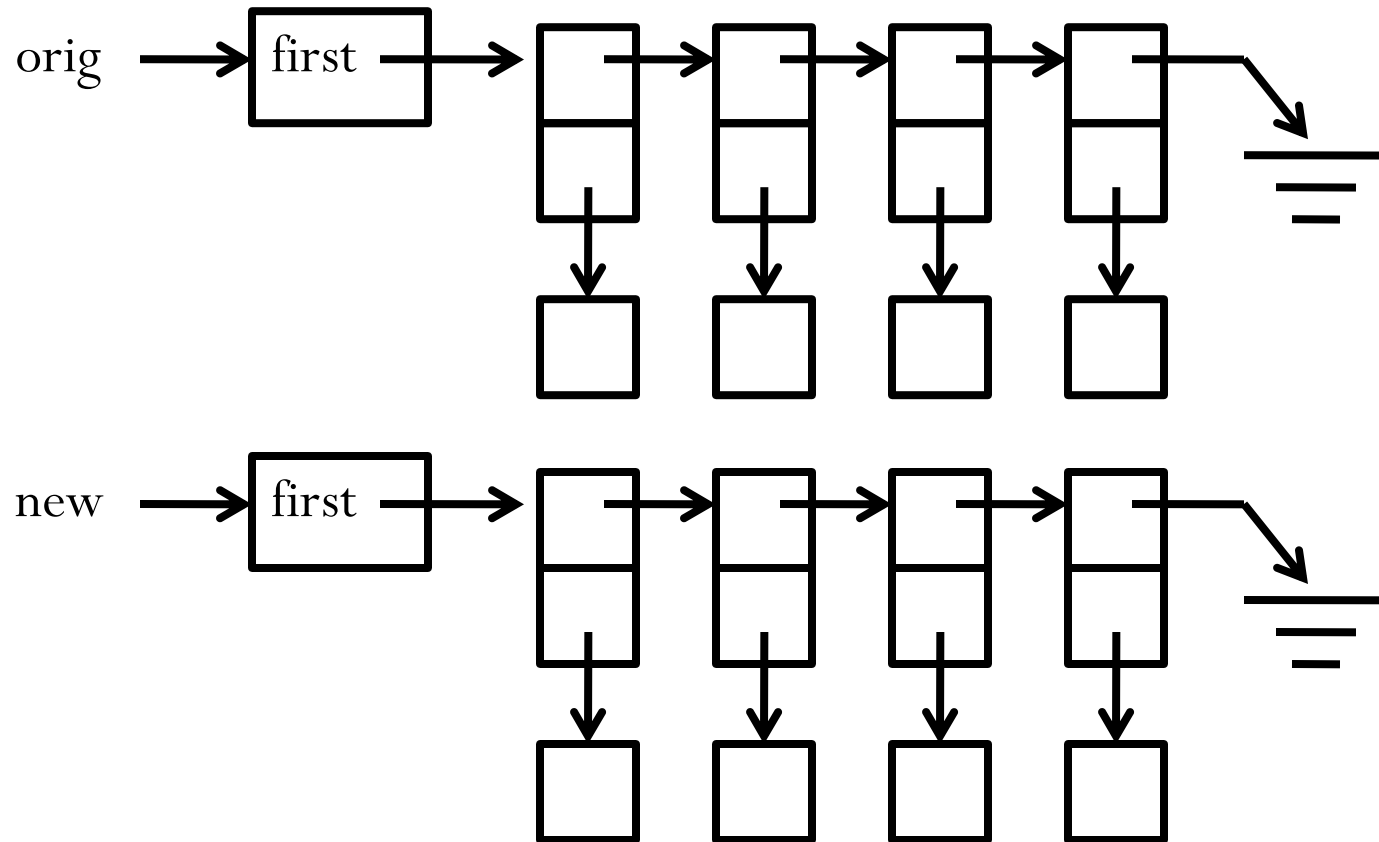
What does the blue statement mean?

# Container of Pointers

## Copy

- The list we would end up with is:

**deep copy!**



# Templated Container of Pointers

- Given container of pointers, the `List` template **must know** whether it is something that holds `T`'s or “pointers to `T`”.
- The former **cannot** delete the values it holds, while the latter **must** do so.
- So, if we want to write a template class that holds pointer-to-`T`, we should provide a version based on pointer.

# Containers

## Templates

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

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

# Containers

## Templates

- This means that if we create two lists of `BigThings`:

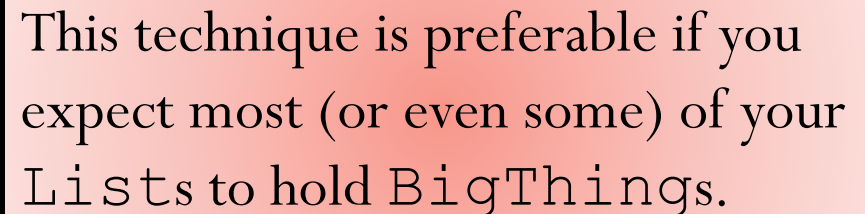
```
ValList<BigThing> vbl;  
PtrList<BigThing> pbl;
```

- Then the first list takes `BigThings` by value:

```
BigThing b;  
vbl.insert(b) ;
```

- But the second list takes them as pointers:

```
BigThing *bp = new BigThing;  
pbl.insert(bp) ;
```



This technique is preferable if you expect most (or even some) of your `Lists` to hold `BigThings`.

# Containers

## Templates

- This means that if we create two lists of `BigThings`:

```
ValList<BigThing> vbl;  
PtrList<BigThing> pbl;
```

- Then the first list takes `BigThings` by value:

```
BigThing b;  
vbl.insert(b) ;
```

- But the second list takes them as pointers:

```
BigThing *bp = new BigThing;  
pbl.insert(bp) ;
```

However, it is **impossible** to have only a **single** implementation of `List` that can correctly contain things either as pointer or by value.



# Outline

- Templates
- Container of Pointers
- Polymorphic Container

# Containers

## Polymorphic containers

- Templates are checked at compile time, but when used straightforwardly, they cannot hold more than one kind of object at once, and sometimes this is desirable.
- There is another kind of container, called a "**polymorphic**" container, that **can** hold more than one type at once.
- The intuition behind polymorphic containers is that, because the container must contain **some** specific type, we'll manufacture a **special "contained" type**, and every real type will be a **subtype** of this contained type.

# Containers

## Polymorphic containers

- We are going to use derived class mechanism

```
class bar: public foo {  
    ...  
};
```

- Recall: a `bar*` can always be used where a `foo*` is expected, but not the other way around.

```
bar b;  
foo *pf = &b;
```

# Containers

## Polymorphic containers

- We can take advantage of this by creating a "dummy class", called `Object`, that looks like this:

```
class Object {  
    public:  
        virtual ~Object() { }  
};
```

- This defines a single class `Object` with a virtual destructor.
- Remember that if a method is virtual, it is also virtual in all derived classes.
- Why we need this? Because when a base-class pointer to a derived-class object is deleted (for example, in function **`removeAll()`**), it will call the destructor of the derived class.

# Containers

## Polymorphic containers

- Now, we can write a `List` that holds `Objects`:

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

```
class Object {  
public:  
    virtual ~Object() {};  
};
```

```
class List {  
    ...  
public:  
    void    insert(Object *o) ;  
    Object  *remove() ;  
    ...  
};
```

# Containers

## Polymorphic containers

- To put `BigThings` in a `List`, you define the class so that it is derived from `Object`:

```
class BigThing : public Object {  
    ...  
};
```

- By the derived class rules, a `BigThing*` can always be used as an `Object*`, but not the other way around.
- So the following works without complaint:

```
BigThing *bp = new BigThing;  
l.insert(bp); // Legal due to  
              // substitution rule
```

# Containers

## Polymorphic containers

- However, the compiler complains about the following because `remove()` returns an `Object *`; we cannot use a base class pointer when a derived class pointer is expected:

```
BigThing *bp;  
bp = l.remove();
```

- However, we can do this:

```
Object *op;  
BigThing *bp;  
  
op = l.remove();  
bp = dynamic_cast<BigThing *>(op);  
...
```

# Containers

## Polymorphic containers

- The `dynamic_cast` operator does the following:

```
dynamic_cast<Type*>(pointer) ;  
// EFFECT: if pointer's actual type is either  
// pointer to Type or some pointer to derived  
// class of Type, returns a pointer to Type.  
// Otherwise, returns NULL;
```

- So, after this cast, we `assert()` that the pointer is valid:

```
Object *op;  
BigThing *bp;  
op = l.remove() ;  
bp = dynamic_cast<BigThing *>(op) ;  
assert(bp) ;
```

**Note:** This only works when the **apparent type** of pointer has one or more virtual methods. That's okay, because `Object` will always have at least a virtual destructor.



# Containers

## Polymorphic containers

- Even with this, there is still one problem.
- This is a **container of pointers**, so we need **deep copy** for copy constructor and assignment operator
- The copyList() below just does shallow copy

```
List::List(const List &l) {  
    first = NULL;  
    copyList(l.first);  
}
```

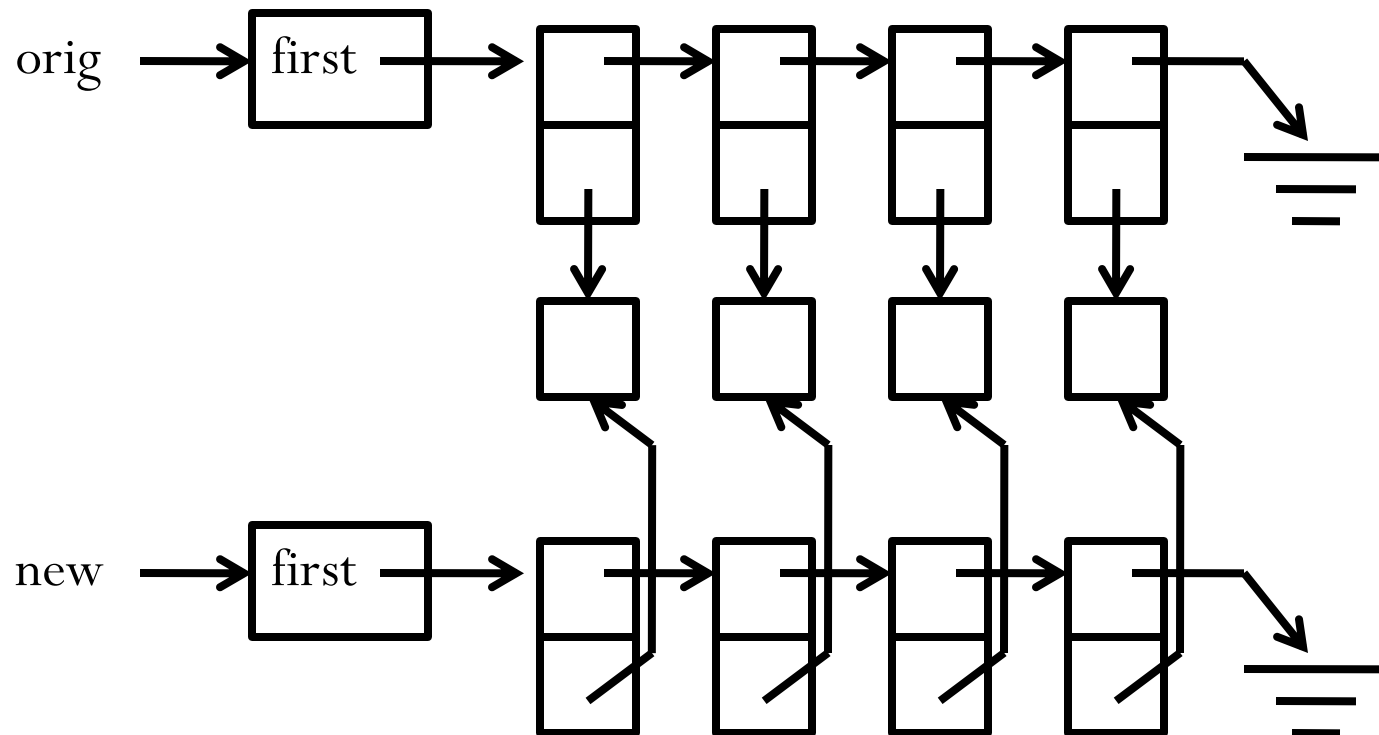
```
void List::copyList(node *list) {  
    if(!list) return;  
    copyList(list->next);  
    insert(list->value);  
}
```

**Object \* type**

# Containers

## Polymorphic containers

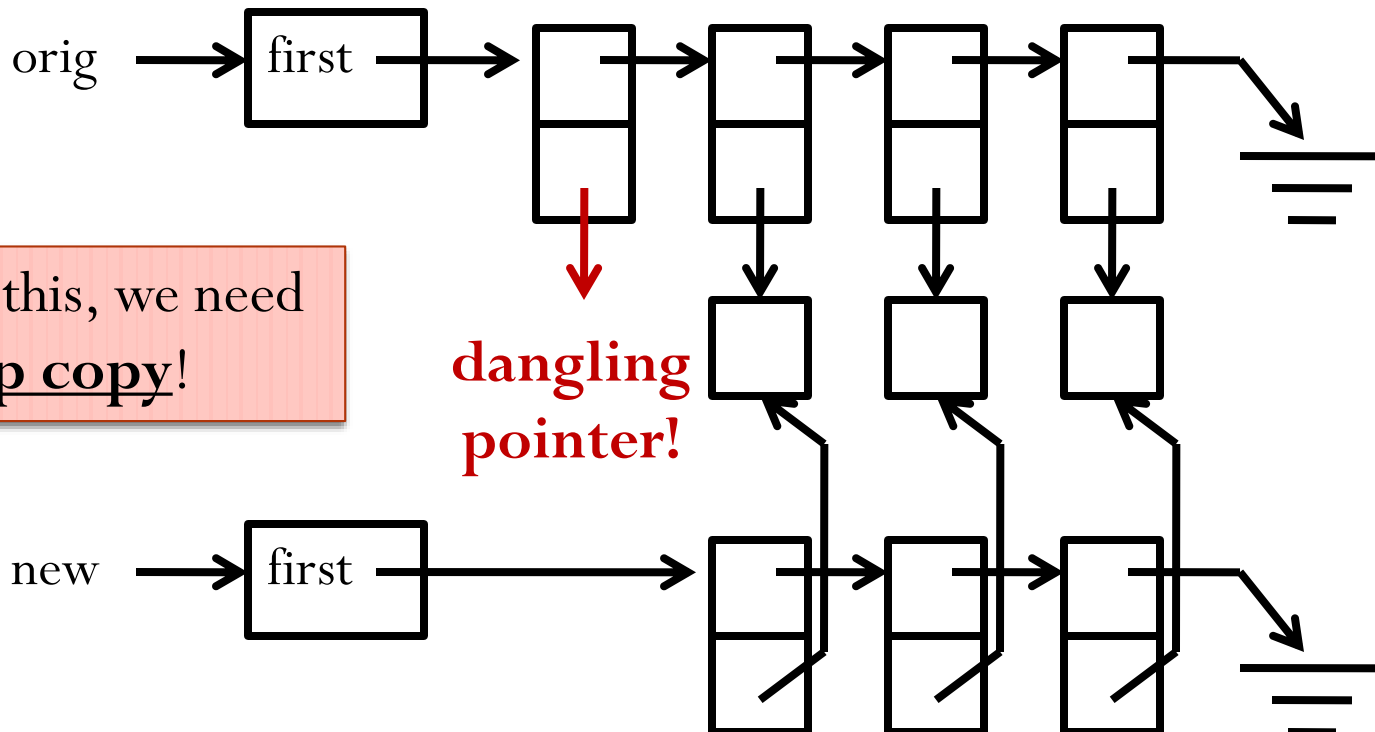
- Using the previous `copyList()`, the list we copied will be:



# Containers

## Polymorphic containers

- Now, if we remove the first item of the new list, we delete the first node, and return a pointer to its `Object`.
- The client, after using that `Object`, will delete it.
- Leaving us with this:



To fix this, we need  
a deep copy!

# Containers

## Polymorphic containers

- To fix this, we might be tempted to rewrite the `copyList` function to create a copy of the `Object`, as follows:

```
void List::copyList(node *list) {  
    if(!list) return;  
    Object *o;  
    copyList(list->next);  
    o = new Object(*list->value);  
    insert(o);  
}
```

A `BigThing` object

- Unfortunately, this won't work, because `Object` does not have a constructor that takes `BigThing` as an argument.

# Containers

## Polymorphic containers

- The way to fix this is to use something called the “**named constructor idiom**”.
  - **named constructor**: A method that (by convention) copies the object, **returning a pointer to the "generic" base class**.
- The name of this method (again, by convention) is usually “clone”.

# Containers

## Polymorphic containers

- Modify the definition of `Object` to include a pure virtual `clone()` method:

```
class Object {  
    public:  
        virtual Object *clone() = 0;  
        // EFFECT: copy this, return a pointer to it  
        virtual ~Object() { }  
};
```

- Declare that method `clone()` in `BigThing`, which **also** has a **copy constructor**:

```
class BigThing : public Object {  
    ...  
    public:  
        Object *clone();  
    ...  
        BigThing(const BigThing &b);  
}
```

# Containers

## Polymorphic containers

- `BigThing::clone()` can then call the correct copy constructor directly, and return a "generic" pointer to it:

```
Object *BigThing::clone() {  
    BigThing *bp = new BigThing(*this);  
    return bp;    // Legal due to substitution  
                  // rule  
}
```

# Containers

## Polymorphic containers

- With this, we can finally rewrite copyList to use clone:

```
void List::copyList(node *list) {  
    if(!list) return;  
    Object *o;  
    copyList(list->next) ;  
    o = list->value->clone() ;  
    insert(o) ;  
}
```

- This gives us a true **deep copy** 😊



# Reference

- **Problem Solving with C++ (8<sup>th</sup> Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 17 **Templates**
  - Chapter 18.2 **Containers**