

Ve 280

Programming and Introductory Data Structures

Container of Pointers; Polymorphic Containers;
Operator Overloading

Outline

- Container of Pointers
- Polymorphic Containers
- Operator Overloading

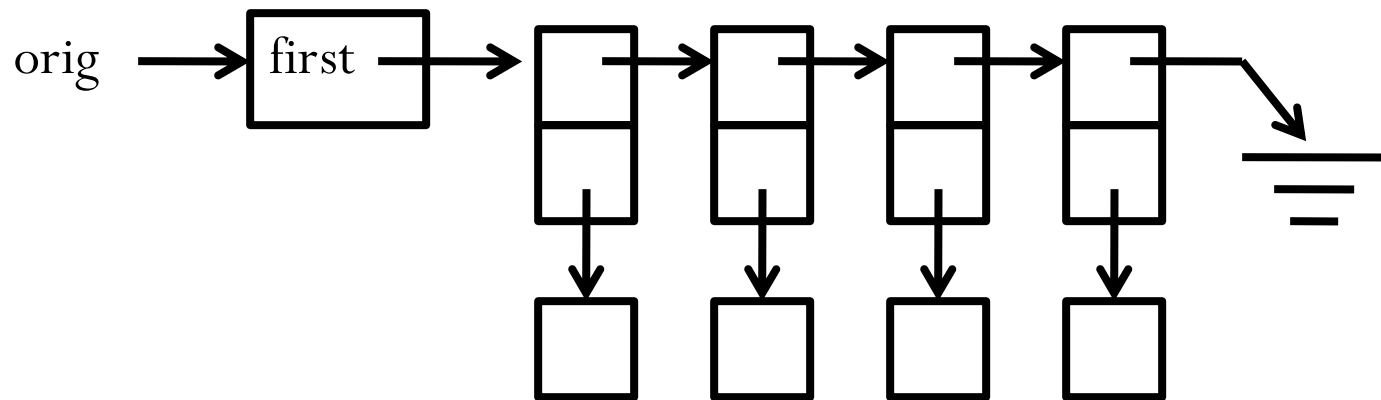
Review

- Container of Pointers
- Subject to bugs. To avoid bugs
 - At-most-once invariant
 - Existence rule
 - Ownership rule
 - Conservation rule
- Due to conservation rule, we should rewrite destructor

Container of Pointers

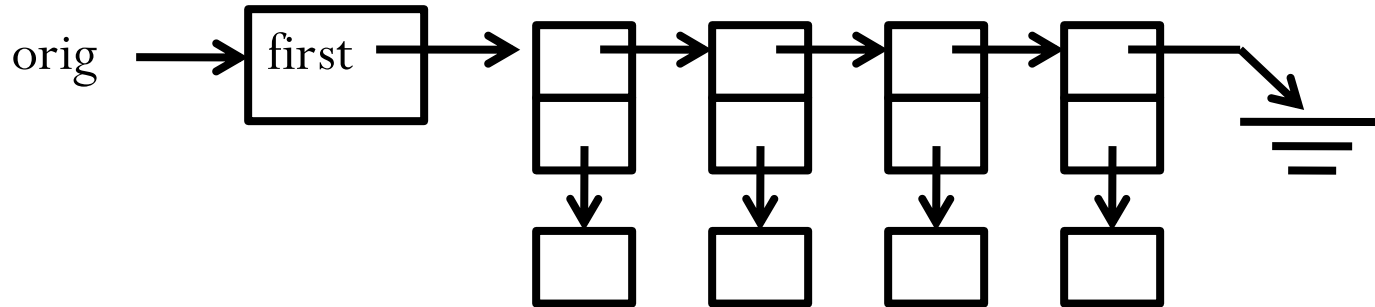
Copy

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



Container of Pointers

Copy



- Here is the old copy constructor and utility function:

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

Question: Does it violate any invariant or rules (Existence, Ownership, Conservation)? Which? Why?

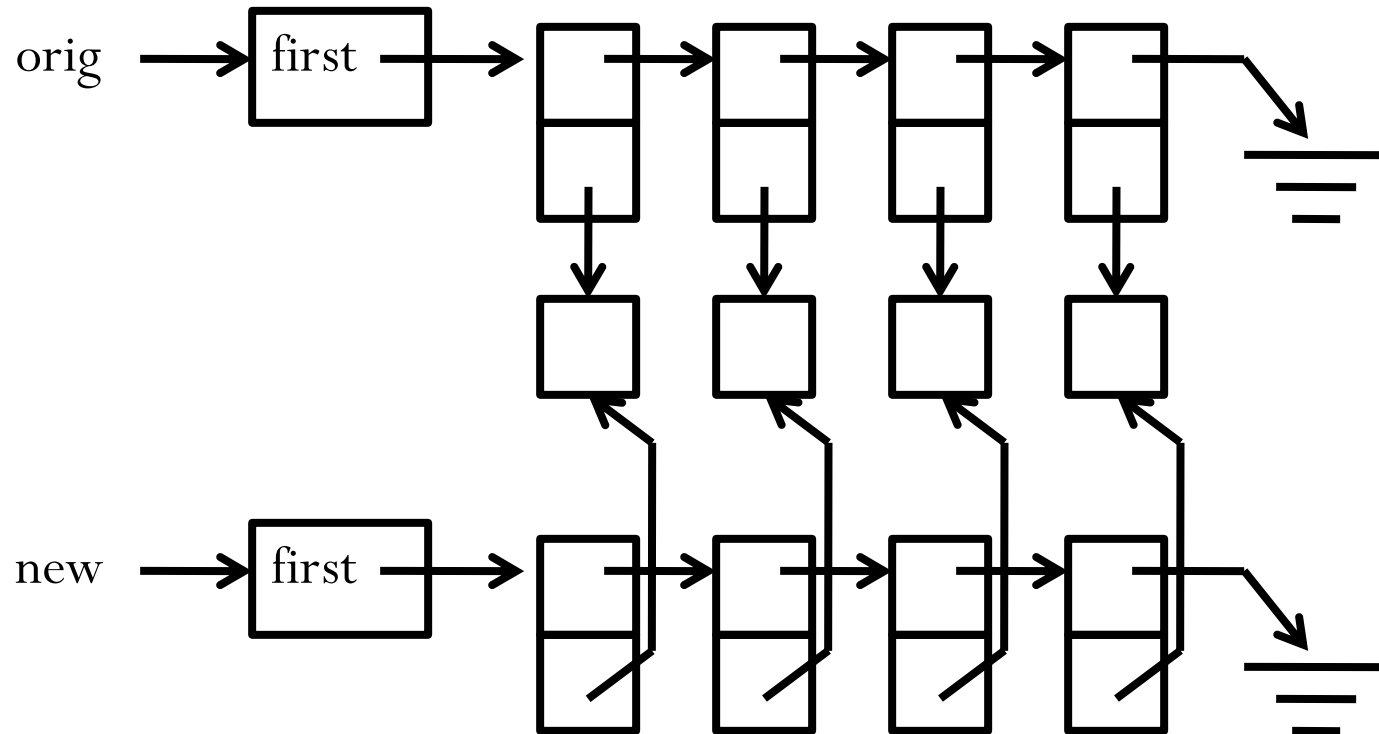
```
template <class T>
void List<T>::copyList(node *list) {
    if(!list) return;
    copyList(list->next);
    insert(list->value);
}
```

T * type

Container of Pointers

Copy

- The list we would end up with is:

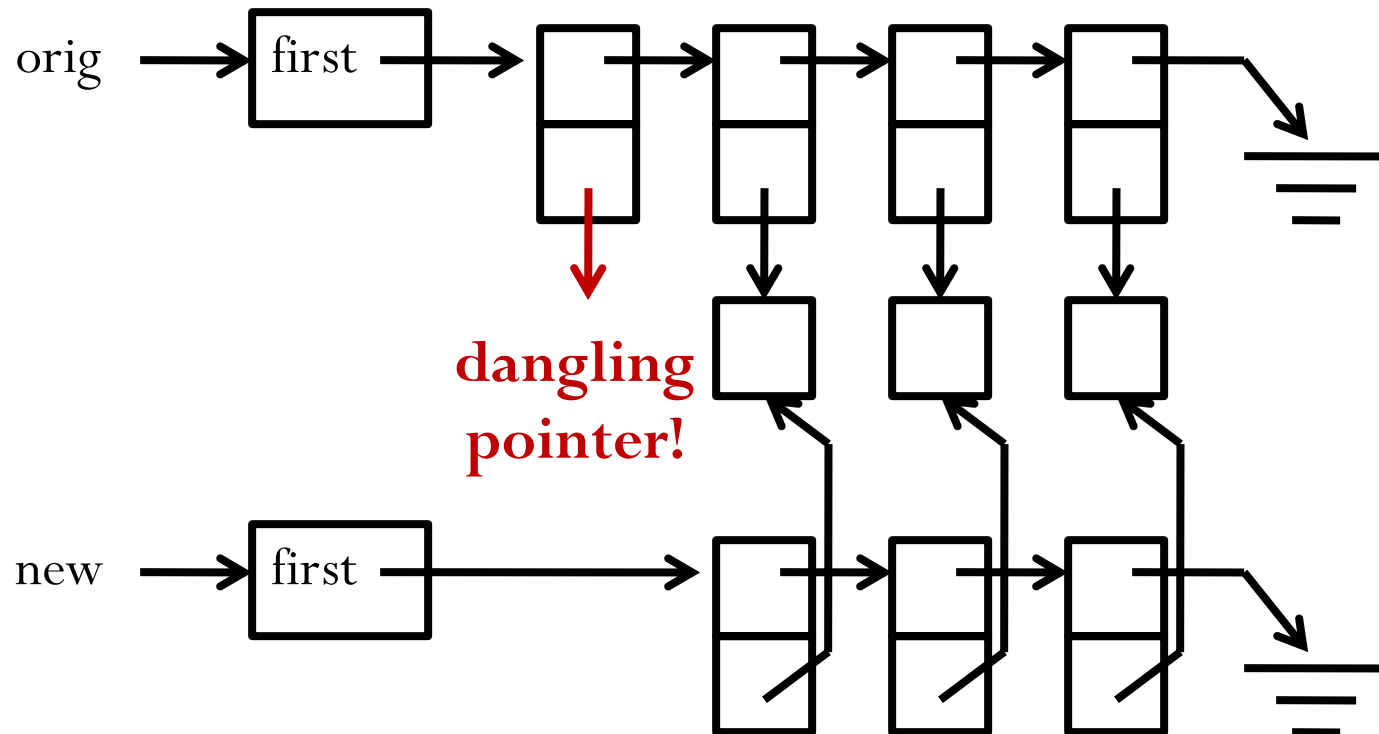


This violates the at-most-once invariant

Container of Pointers

Copy

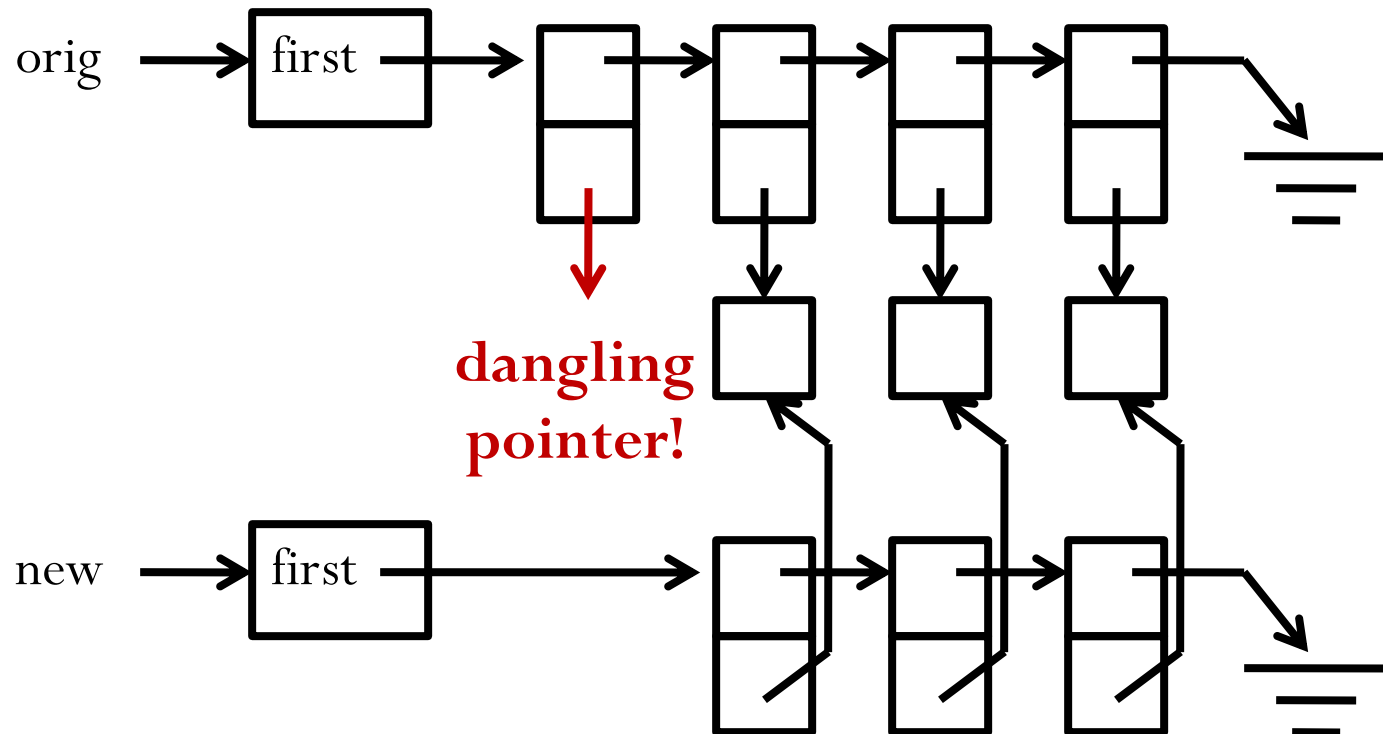
- Now, if we remove the first item of the new list, we delete the first node, and return a pointer to the item.
- The client will use the item and delete it (Why?).
- Leaving us with this:



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 *

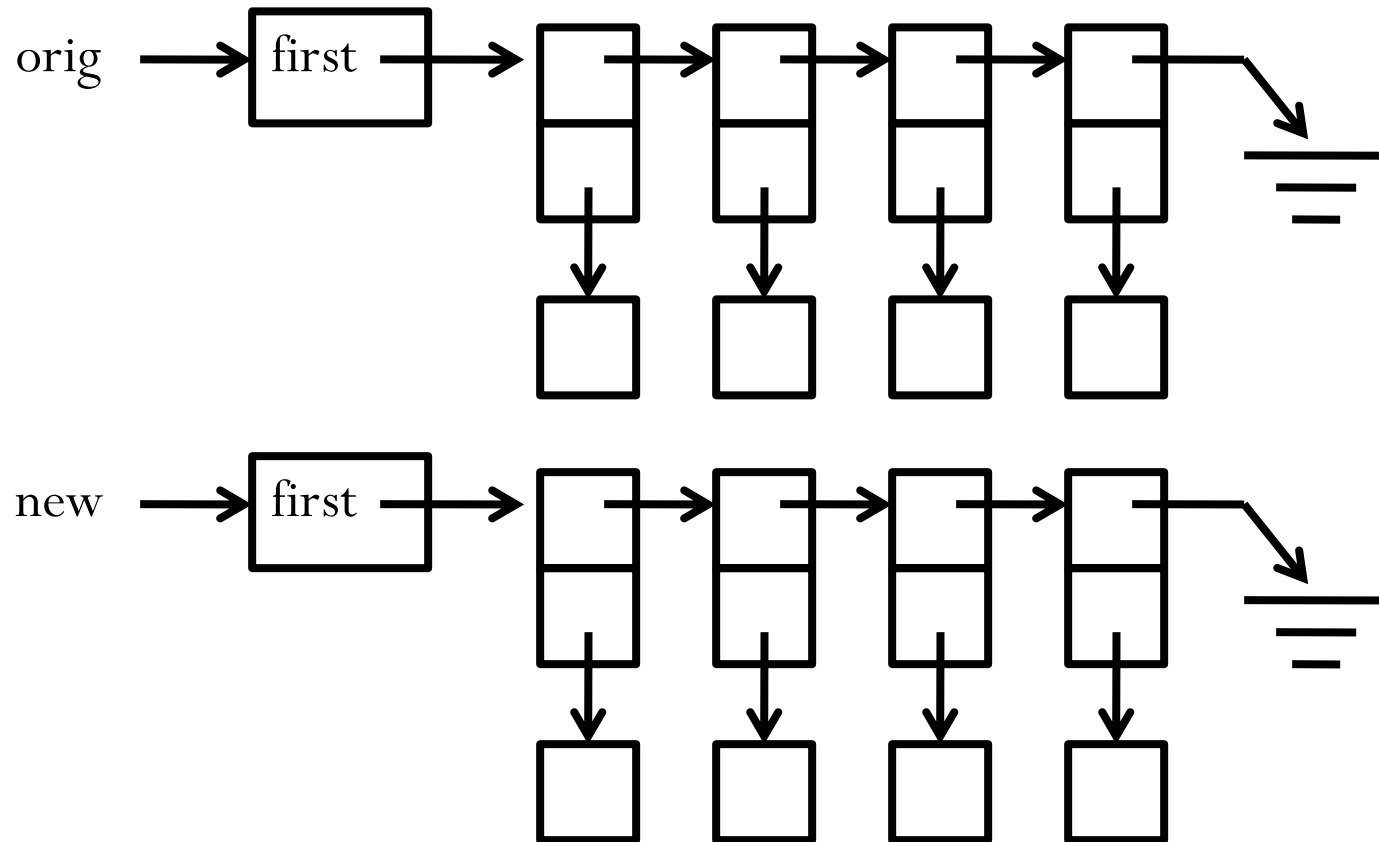
What does the blue statement mean?

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`:

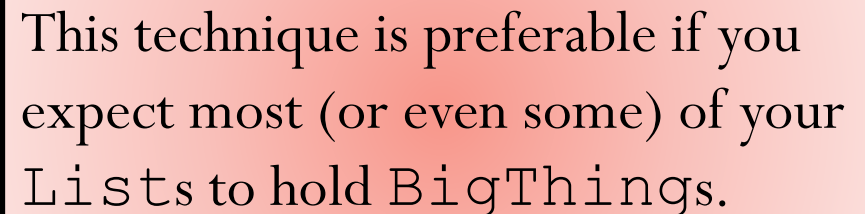
```
ValList<BigThing> vbl;  
PtrList<BigThing> pbl;
```

- 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

- Container of Pointers
- Polymorphic Containers
- Operator Overloading

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 subtype  
// 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 = remove();  
bp = dynamic_cast<BigThing *>(op);  
assert(bp);
```

Note: This only works for classes which have one or more virtual methods. That's okay, because `BigThing` 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);  
}
```

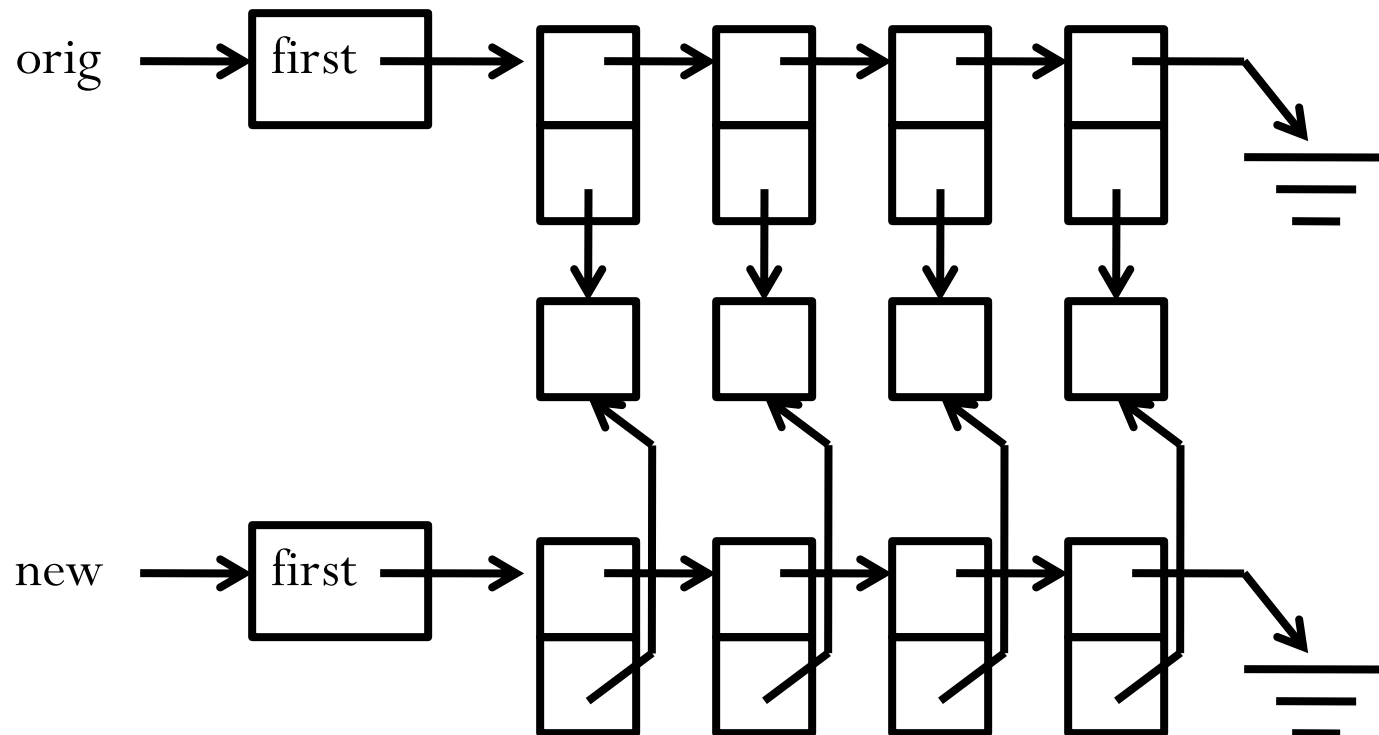
```
void List::copyList(node *list) {  
    if (list != NULL) {  
        copyList(list->next);  
        insert(list->value);  
    }  
}
```

Object * type

Containers

Polymorphic containers

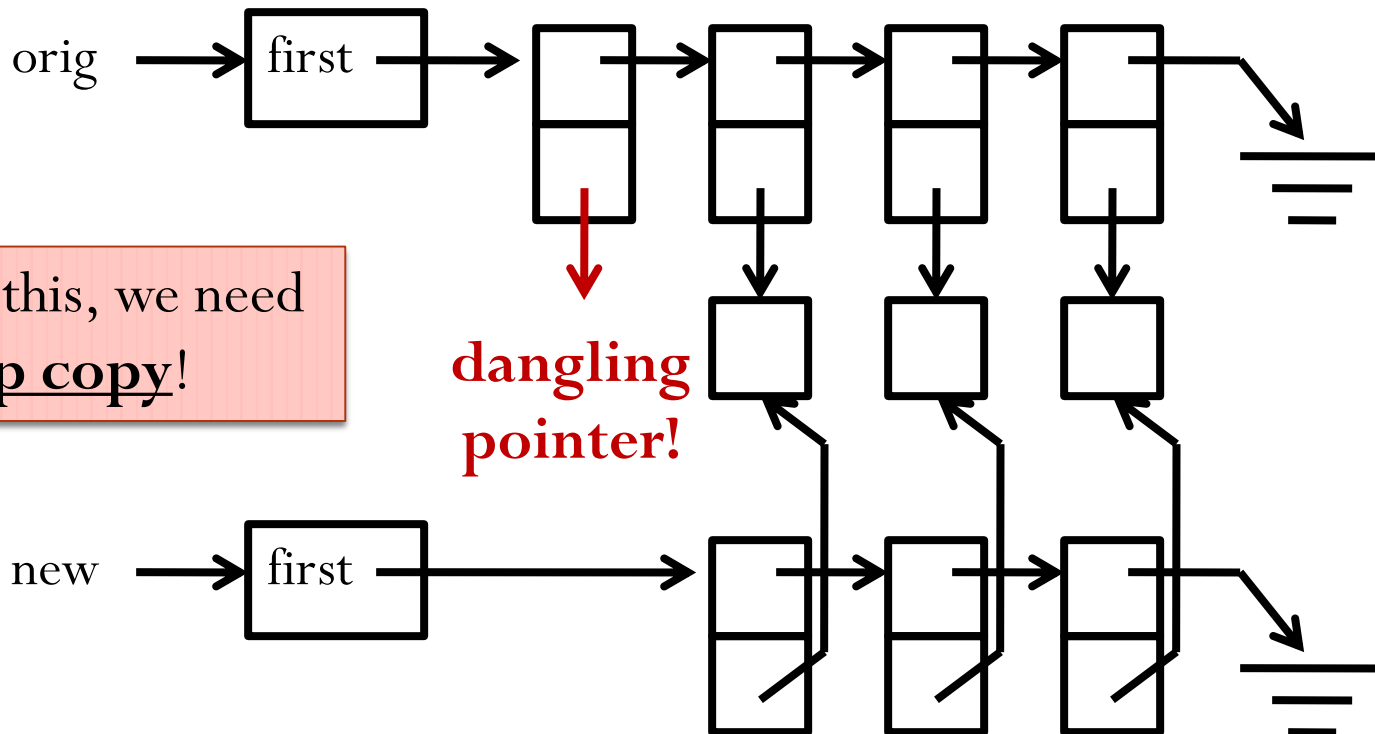
- Using the previous `copyList()`, the list we copied will be:



Containers

Polymorphic containers

- Now, if we remove the first item of the `new` list, we delete the first node, and return a pointer to it.
- The client, after removing that `Object`, will use it and 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 != NULL) {  
        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 != NULL) {  
        Object *o;  
        copyList(list->next);  
        o = list->value->clone();  
        insert(o);  
    }  
}
```

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

Outline

- Container of Pointers
- Polymorphic Containers
- Operator Overloading

Operator Overloading

Introduction

- C++ lets us **redefine** the meaning of the operators when applied to objects of **class type**.
- This is known as **operator overloading**.
- We have already seen the overloading of the assignment operator.
- Operator overloading makes programs much easier to write and read:

```
IntSet is;  
int x = is[5]; // overload [] operator  
           // access the IntSet element by index  
cout << is << endl; // overload << operator  
           // print all the IntSet elements
```


Operator Overloading

Basics

- Overloaded operators are functions with special names: the keyword **operator** followed by the symbol (e.g., +, -, etc.) of the operator being redefined.
- Like any other function, an overloaded operator has a return type and a parameter list.

```
A operator+(const A &l, const A &r) ;
```

Operator Overloading

Basics

- Most overloaded operators may be defined as ordinary **nonmember** functions or as class **member** functions.

```
A operator+(const A &l, const A &r);  
// returns l "+" r
```

```
A A::operator+(const A &r);  
// returns *this "+" r
```

- Overloaded functions that are members of a class may appear to have **one fewer** parameter than the number of operands.
 - Operators that are member functions have an implicit **this** parameter that is bound to the **first operand**.

Operator Overloading

Basics

- An overloaded **unary** operator has **no** (explicit) parameter if it is a member function and **one** parameter if it is a nonmember function.
- An overloaded **binary** operator would have **one** parameter when defined as a member and **two** parameters when defined as a nonmember function.

Example

- Overload **operator+=** for a class of complex number.

```
class Complex {  
    // OVERVIEW: a complex number class  
    double real;  
    double imag;  
public:  
    Complex(double r=0, double i=0); // Constructor  
    Complex &operator += (const Complex &o);  
    // MODIFIES: this  
    // EFFECTS: adds this complex number with the  
    // complex number o and return a reference  
    // to the current object.  
};
```

Example

```
Complex &Complex::operator += (const Complex &o)
{
    real += o.real;
    imag += o.imag;
    return *this;
}
```

Example

- `operator+=` is a member function.
- We can also define a nonmember function that adds two numbers.

```
Complex operator + (const Complex &o1,  
                  const Complex &o2)  
{  
    Complex rst;  
    rst.real = o1.real + o2.real;  
    rst.imag = o1.imag + o2.imag;  
    return rst;  
}
```

- However, there is a problem with this. What is it?
- Since `operator+` is a nonmember function, it cannot access the private data members.

Friend

- So, we'll need some other mechanism to make the function as a "**friend**".
- The "friend" declaration allows you to expose the **private** state of one class to another function (and only that function) explicitly.

```
class foo {  
    friend void baz();  
    int f;  
};  
void baz() { ... }
```

The function **baz** has access to **f**, which would otherwise be private to class **foo**.

Friend

- So, we'll need some other mechanism to make the function as a "**friend**".
- The "friend" declaration allows you to expose the **private** state of one class to another function (and only that function) explicitly.

```
class foo {  
    friend void baz();  
    int f;  
};  
void baz() { ... }
```

Note: a friend function is **NOT** a member function; it is an ordinary function.

Note: NOT `void foo::baz() { ... }`

Friend

- So, we'll need some other mechanism to make the function as a "**friend**".
- The "friend" declaration allows you to expose the **private** state of one class to another function (and only that function) explicitly.

```
class foo {  
    friend void baz();  
    int f;  
};  
void baz() { ... }
```

Note: "friend void baz ();" goes inside foo. It means foo gives friendship to function baz ().

Friend

- Besides function, we can also declare a class to be friend.

```
class foo {  
    friend class bar;  
    int f;  
};  
class bar {  
    ...  
};
```

Then, objects of class `bar` can access private member `f` of `foo`.

Friend

```
class foo {  
    friend class bar;  
    friend void baz();  
    int f;  
};  
class bar { ... };  
void baz() { ... }
```

Friendship of both
class and function.

- Note: Although “**friendship**” is declared inside **foo**, **bar** and **baz()** are not the members of **foo**!
- “**friend**” declaration may appear anywhere in the class.
 - It is a good idea to **group** friend declarations **together** either at the beginning or end of the class definition.

Example

- In our example of complex number class, we will declare **operator+** as a friend:

```
class Complex {  
    // OVERVIEW: a complex number class  
    double real;  
    double imag;  
public:  
    Complex(double r=0, double i=0);  
    Complex &operator += (const Complex &o);  
    friend Complex operator+(const Complex &o1,  
                             const Complex &o2);  
};
```

Its implementation is the same as before.

Reference

- **C++ Primer (4th Edition)**, by *Stanley Lippman, Josee Lajoie, and Barbara Moo*, Addison Wesley Publishing (2005)
 - Chapter 12.5 **Friends**
 - Chapter 14 **Overloaded Operations and Conversions**