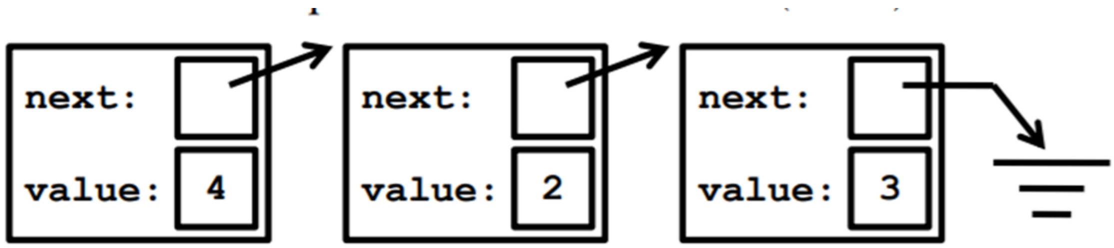
VE280 2022FA RC8

Hint: Very difficult and important two lectures!

L19 Linked-List

What is linked list?



Linked-List class

```
class IntList {
    struct node {
        node *next;
        int value;
    };
    node *first;
public:
    bool isEmpty();
    void insert(int v);
    int remove();
    IntList(); // default ctor
    IntList(const IntList& l); // copy ctor
    ~IntList(); // dtor
    // assignment
    IntList &operator=(const IntList &l);

    //helper functions
    void removeAll();
    void copyList(node *list);
```

```
};
```

Invariant: the pointer of the first node the linked list. Always remember to maintain the invariant!

Hint: The node is dynamically allocated. For this type of class, always write the three methods: **copy constructor**, **assignment operator overload** and **default destructor**.

Methods Definition

```
bool IntList::isEmpty() {
    return !first;
}

void IntList::insert(int v) {
    node *np = new node; //Dynamic allocated
    np->value = v;
    np->next = first;
    first = np; //Invariant maintain
}

int IntList::remove() {
    node *victim = first;
    int result;
    if (isEmpty()) {
        listIsEmpty e;
        throw e;
    }
    first = victim->next;
    result = victim->value;
    //What is the small trick here to maintain the invariant?
    delete victim;
    return result;
}

IntList::IntList(): first(nullptr) {}

void IntList::removeAll(){
    while (!isEmpty()){
        remove();
    }
}

IntList::~IntList() {
    removeAll();
}

void IntList::copyList(node *list) {
    if (!list) return; // Base case
    copyList(list->next);
    insert(list->value);
    //Why use recursion here? What is the trick?
}
```

Doubled Ended Linked List

A new invariant need to be maintained.

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

Questions: What is the benefits and drawbacks of the double linked list?

L20 Template; Container

Template

Intro: Another very very important feature of c++ to support polymorphism.

Polymorphism: Reusing code for different types.

Motivation: For the linked list we define before, we want our class to support both the **char** and **int** data types.

```
template <class T> //class and typename both are ok
class List {
//Use T to displace the place where we need a type name or class name before.
private:
    struct node {
        node *next;
        T v;
    };

public:
    bool isEmpty();
    void insert(T v);
    T remove();
    List();
    List(const List &l);
    List &operator=(const List &l);
    ~List();
private:
...
};
```

Syntax of Template

Method Definition: Always add template before the definition before you implement a function outside of the class field.

For example:

```
template <class T> //You need to declare this before all methods.
void List<T>::insert(T v) { //List<T>
    node *np = new node;
    np->next = first;
    np->v = v;
    first = np;
}

//Error-prone
List<T>::List(const List<T> &l);
List<T> &List<T>::operator=(const List<T> &l);
```

Generate Instance:

```
// Every time you generate the instance of the template class, the template is always needed!
// 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>
```

Container

Motivation: Still consider the templated linked list we define before. What if the template class T is itself a very large data type containing a very large memory. We *do not want to directly make a list for those large things*. However, what we want is to make a list for the **addresses**.

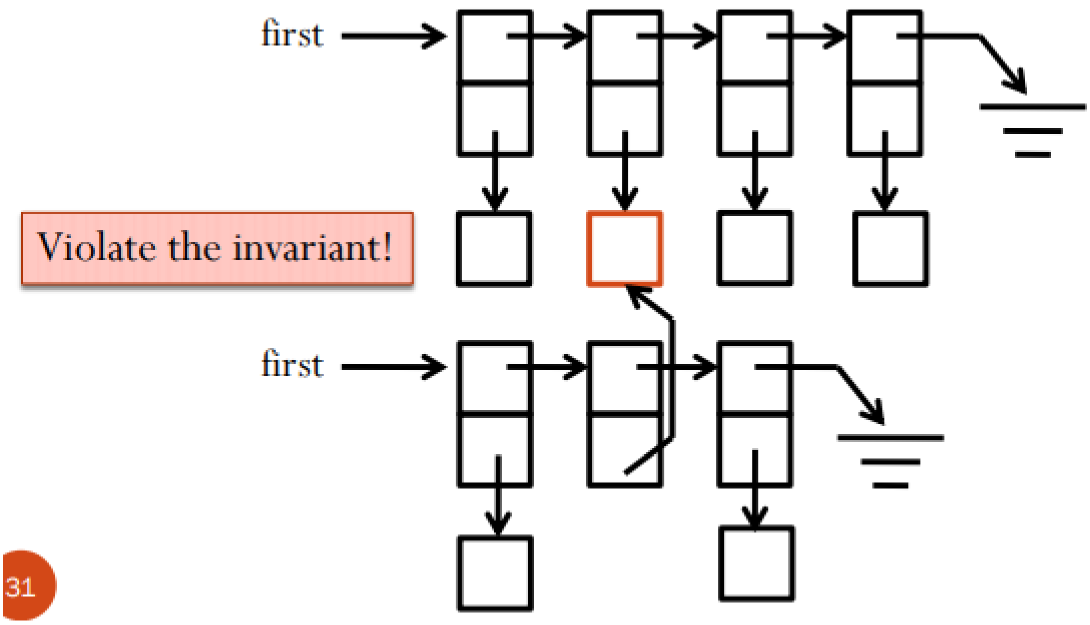
```
//The "pointer" version of our templated linked list.
template <class T>
class List {
public:
    ...
    void insert(T *v);
    T *remove();
private:
    struct node {
        node *next;
        T *o;
        //Two dynamically allocated objects.
    };
    ....
};
//Since you use pointer, you should be very careful about the memory!
```

Three rules and One invariant (almost to appear in the exam)

- **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** after using.

One Invariant Violation:



Existence Violation:

```
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 Violation: As the description said.

Conservation Violation: As the description said.

Besides: For destroying a container, the objects contained in the container should also be deleted.

How can we modify the following functions?

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

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;
}

/////////////////////////////////////////////////

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

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

```
    insert(list->value); //?
}
```

Polymorphic Containers

Intro: Use the derived class to implement the "polymorphic" functions.

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

class BigThing : public Object {
...
};

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

//However, we cannot substitute the derived class with the father class. So use the dynamic_cast.
Object *op;
BigThing *bp;
op = l.remove();
bp = dynamic_cast<BigThing *>(op);

//Why the following cannot work? How to solve?
void List::copyList(node *list) {
    if(!list) return;
    copyList(list->next);
    Object *o = new Object(*list->value);//?
    insert(o);
}
```

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

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

//Continue to use the feature that when the base class is needed, a derived class can always take place.
Object *BigThing::clone() {
```

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