L14: Subtypes; Inheritance; Virtual Functions

Subtypes

Definition: S is called the subtype of T whenever the instance of an object of type T is expected, an object of type S can be supplied.

Question: Whether the following type showing in left is the subtype of that is showing right?

```
double, int
ifstream, istream
```

How to create subtypes?

1. Add one or more operations.

   Example: `IntSet` and `MaxIntSet`, where `MaxIntSet` has an additional function `max()`. So, `MaxIntSet` is the subtype of `InSet`.

2. Strengthen the postcondition of one or more operations.

   Remark: What are called postcondition here are "EFFECTS" and return type.

   Example:

   ```
   int A::f();
       //EFFECTS: return an int.
   int B::f();
       //EFFECTS: return a positive int.
   int main(){
       int integer = A::f();
       //Since the variable need A() to return an int, it is obviously legal to return a positive int. So B::f() can replace A::f() here.
       //However, if we actually need a positive integer, B::f() can not be replaced by A::(), since an integer is not always a positive integer.
   }
   ```

3. Weaken the precondition of one or more operations.

   Remark: What are called precondition here are "REQUIRES" and argument type.

   Example:

   ```
   void A::f(string a);
       //REQUIRES: the input string a should be a non-empty string.
   void B::f(string b);
       //No specific limitation to b
   int main(){
       string str("lkz");
       A::f("lkz");
       //Since the input of A::f() should be non-empty, it will always work if we replace A::f() with B::f(), since B::f() does not have such limitations.
       //So B is the subtype of A.
   }
   ```

Creating Subtypes by C++

Inheritance: Create a subclass of the base class

```
//Syntax
class son : public father{};
//use ":" to show the inheritance relationship.
//use "public", "private", "protected" to specify the type of inheritance
```

Base Class Member Type	Inheritance Type	Derived Class Member Type
private	public, private, protected	No permission
public	public, private, protected	public, private, protected respectively
protected	public, protected	protected
protected	private	private

Remark: If you want the member of the base class to be accessed by the derived class, but you do not want the outsider to access the member of the bass class, then use protected.

Note: Subclasses may not be subtypes.

Virtual Functions

Motivation: Consider the following situations:

```
//base class father has three children (subclasses)
class father{
    public:
    void speak(){
        cout<<"I am Eddard Stark"<<endl;
    }
};
class son1: public father{
    void speak(){
        cout<<"I am John Snow"<<endl;
    }
}
class son2: public father{
    void speak(){
        cout<<"I am Arya Stark"<<endl;
    }
}
class son3: public father{
    void speak(){
        cout<<"I am Robb Stark"<<endl;
    }
}
```

```
//Then we want to use the pointer of the father type to represent all the three
subtypes as the function input.
void loudspeak(father * role){
    *role.speak();
    //In this domain, however, the role can only be seen as an instance of
"father", but you really want to use function of speak() of son1, son2 or son3.
}

int main(){
    son1 John;
    son2 Arya;
    son3 Robb;
    loudspeak(&John);//This will work due to the replacement principle of
subtype.
    loudspeak(&Arya);
    loudspeak(&Robb);

}
```

Then, we can define the corresponding function in the father type as ***virtual***.

```
class father{
    public:
    virtual void speak(){
        cout<<"I am Eddard Stark"<<endl;
    }
};
```

This time, when the *role.speak() is called, the compiler will strive to find the real type of role, not just the base type.

L15 Interfaces; Invariants

Motivation: To be consistent with the definition of ADT, we provide an "interface-only" class as a base class (Abstract Base Class). In the ABC, we only provide the declaration of some functions, and we never provide the definition of functions and member types.

```
class car{ //There we only provide an abstraction of the "car".
    virtual void run() = 0; //The car can "run"
    virtual void stop() = 0; //The car can "stop"
}
```

Remark: Function with "= 0" ended is called pure virtual function. The class with pure virtual functions is called abstract class.

Note: You cannot create any instance of an abstract class.

```
class bus{
    int people[10];
    int num;
    int capacity;
    public:
    bus();//constructor
    void run();
    void stop();
}
```

Remark: Typically, you write your interface in a public header file, and write your implementation (the derived class) in a source file. In that case, the user can get how to use the functions through the header file, but cannot get the exactly definition. This perfectly fits our requirements of ADT.

Write the following functions to help user get the instance of bus

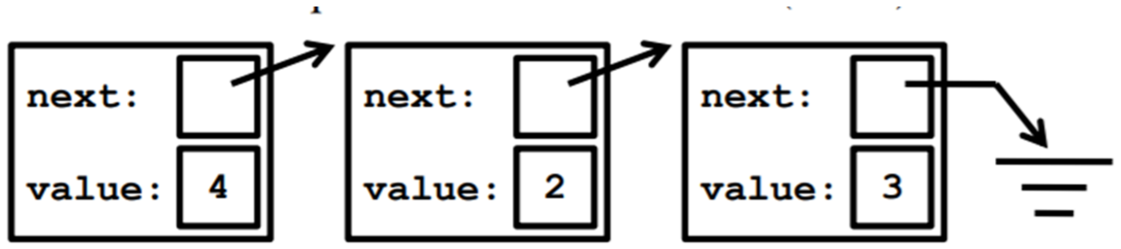
```
// header file
car * getCar();

//cpp file
static bus bus210; //This only works for only one instance is needed.
Car *getCar(){
    return &bus1;
}
```

For multiple instances are needed, use dynamic memory allocation.

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?
}

IntList::IntList(const IntList& l){
    copyList(l.first);
}

IntList &IntList::operator=(const IntList &l){
    if(this!=&l){
        removeAll();
        copyList(l.first)
    }
    return *this;
}
```

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/typename 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:

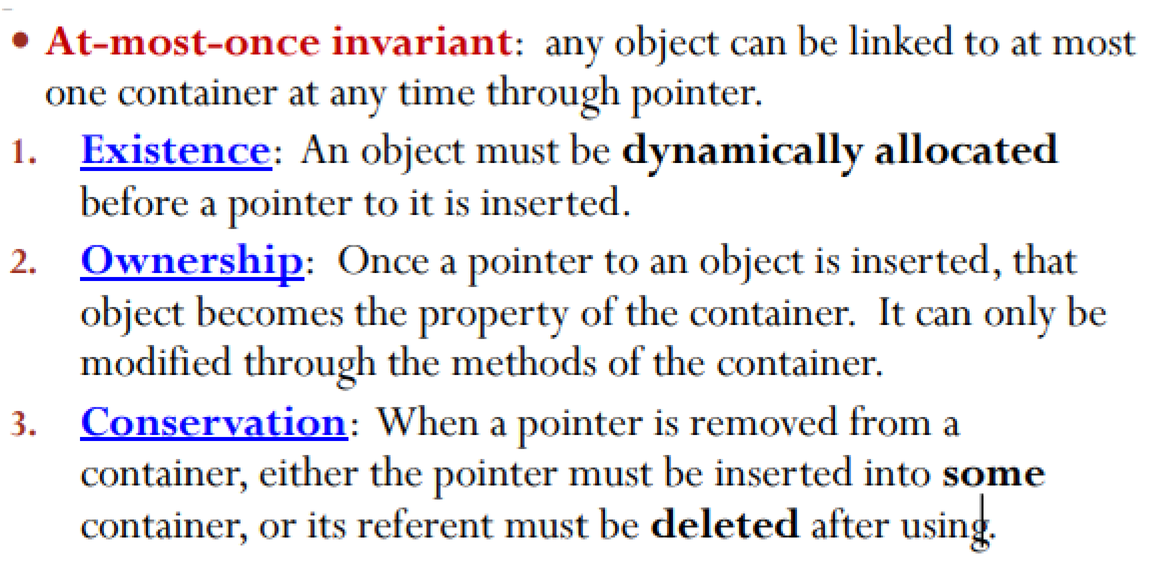
```
// Every time you generate the instance of the template class, the template is
always needed!
// Create a static list of integers
List<int> li;
// Create a dynamic list of integers
List<int> *lip = new List<int>;
// Create a dynamic list of doubles.
List<double> *ldp = new List<double>
```

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!
```

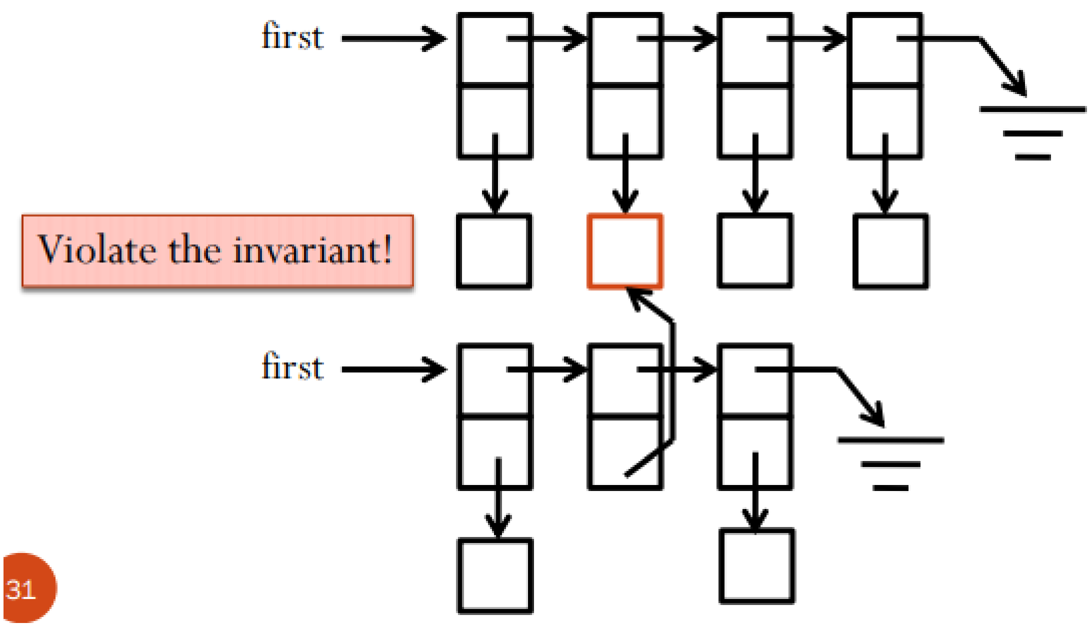
Three rules and One invariant (almost to appear in the exam)



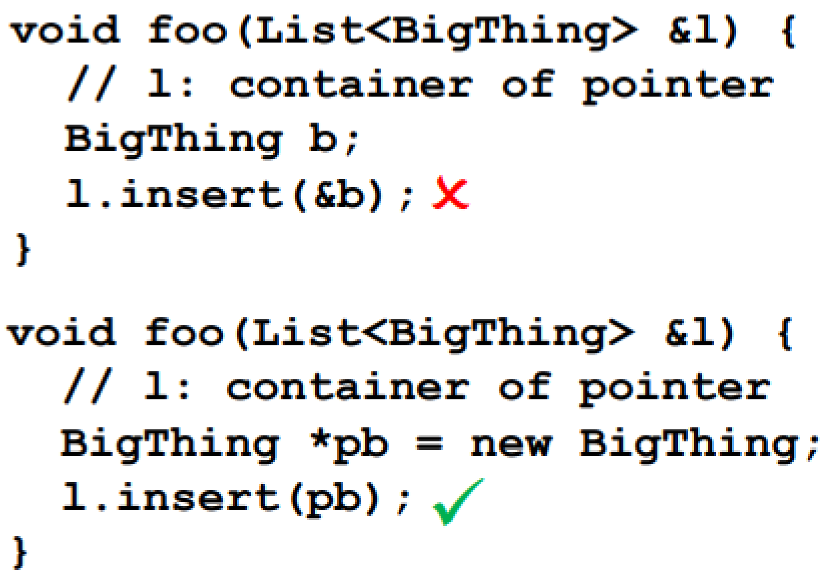
- **At-most-once invariant**: any object can be linked to at most one container at any time through pointer.

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()) {
        T* temp = remove(); //?
        delete temp;
    }
}

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);
    T *temp = new T(*(list->value));
    insert(temp); //?
}
```

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 Bigthing(*list->value);//?
    insert(*list->value.clone());
}
```

```
//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
}
```

Questions

1. What is the output?

```
#include <iostream>
using namespace std;
class Foo {
    public:
    void f() { cout << "a"; };
    virtual void g() = 0;
    virtual void c() = 0;
};
class Bar : public Foo
{
    public:
    void f() { cout << "b"; };
    virtual void g() { cout << "c"; };
    void c() { cout << "d"; };
    virtual void h() { cout << "e"; };
};
class Baz : public Bar
{
    public:
    void f() { cout << "f"; };
    virtual void g() { cout << "g"; };
    void c() { cout << "h"; };
    void h() { cout << "i"; };
};
class Qux : public Baz
{
    public:
    void f() { cout << "j"; };
    void h() { cout << "k"; };
};
int main() {
    Bar bar; bar.g();
    Baz baz; baz.h();
    Qux qux; qux.g();
    Foo &f1 = qux;
    f1.f(); f1.g(); f1.c();
    Bar &b1 = qux;
    b1.f(); b1.c(); b1.h();
    Baz &b2 = qux;
    b2.f(); b2.c(); b2.h();
    return 0;
}
```

Answer:

```
int main() {
Bar bar; bar.g(); // c
Baz baz; baz.h(); // i
Qux qux; qux.g(); // g
Foo &f1 = qux;
f1.f(); f1.g(); f1.c(); // a g h
Bar &b1 = qux;
b1.f(); b1.c(); b1.h(); // b h k
Baz &b2 = qux;
b2.f(); b2.c(); b2.h(); // f h k
return 0;
}
```