VE280 Final Review

August 4

Ge Tianyi

Outline

1. Subtype
2. Invariant
3. Dynamic allocation
4. Deepcopy

Subtype

Subtype

Definition

> S is a **subtype** of T, written "S<:T". T is a **supertype** of S.
> When a T object is expected, an S object can be supplied.

Example: `MyopicPlayer <: Player`

```
class MyopicPlayer: public Player {...};
// All the methods of Player type is valid for mp.
int main() {
    MyopicPlayer mp;
    Player *p_ptr = &mp;
    p_ptr->selectPiece();
    // Player is expected, MyopicPlayer is supplied
    Player &p_ref = mp;
    p_ref.selectPiece(); // Reference also works
}
```

Subtype

Three ways to create a subtype

1. Add one or more operations
   - `MyopicPlayer` has `isBadPiece`, but `Player` doe not.
2. Strengthen the postcondition of one or more operations
   - A same operation can do more things (More `EFFECTS`)
3. Weaken the precondition of one or more operations
   - A same operation has loose requirements (Less `REQUIRES`)
   - e.g. $|\cdot|$ supports real numbers |-2|=2; it also supports complex numbers |3+4i|=5

Subtype

Subclasses

	Public Members	Protected Members	Private Members
Public Inheritance	public	protected	private
Protected inheritance	protected	protected	private
Private inheritance	private	private	private

- `protected` means can be accessed by its subclass but cannot be accessed by outside callers

Subtype

Subclasses

```
class A {
public:
    int x;
protected:
    int y;
private:
    int z;
};
class B : public A {
    // x is public
    // y is protected, accessible from B
    // z is not accessible from B
};

// The other two types of inheritance
// are not our focus in this course
```

Subtype

About `Dynamic_cast<Type *>()`

```
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;
```

This only works for classes which have one or more virtual methods. Otherwise, there's compile error.

Subtype

About `Dynamic_cast<Type *>()`

```
class A {
public:
    virtual void f() {
        cout << "A::f()" << endl;
    }
};
class B : public A {
public:
    void f() {
        cout << "B::f()" << endl;
    }
};
```

```
A a, *pa;
B b, *pb;
```

```
pa = &b; pa->f(); // B::f()
```

```
pa = &b;
pb = dynamic_cast<B*>(pa);
// It succeeds because pa's actual type is B (B<:A)
// which has at least one virtual method
pb->f(); // As expected, B::f()
```

括号里面必须是B type / subtype of B type

```
pa = &a;
pb = dynamic_cast<B*>(pa);
// It fails because pa points to A type,
// which is not a B type or a subtype of B type
// It returns nullptr
pb->f(); // segmentation fault
```

```
pb = &b; pa = dynamic_cast<A*>(pb); // always valid
```

Interfaces & Invariants

Interfaces

Only provide the usage of this type. Do not expose the implementation details.

Interfaces

In order to protect hp

Singleton: when only one instance is needed

```
// player.h
extern Player *getHumanPlayer(Board *b, Pool *p);

// player.cpp
Player *getHumanPlayer(Board *b, Pool *p) {
    static HumanPlayer hp(b, p);
    // static variables will only be initialized once
    return &hp;
}
```

The user can only access the `HumanPlayer` instance by

```
// game.cpp
Player *hp = getHumanPlayer(&board, &pool);
```

Invariants

Definition

- It describes the conditions that must hold on those members for the representation to correctly implement the abstraction.
- It must hold immediately before exiting each method of that implementation Ã¢â‚¬â€œ including the constructor.

Question:

What's the invariant of `IntSet` regarding `size`?

- `size` represents the number of elements in this `IntSet`, which is always less than or equal to `MAXELTS`

Dynamic allocation

Example: Dynamic length array

```
int num = 100;
int *array = new int[num](0);
delete [] array;
```

Do not create an array with varaible length.

```
int num = 100;
int array[num]; // no! deduction warning!
```

Find the problems in p5:

```
Customer *c = new Customer();
c = gold.removeFront(); // new memory above is lost
```

```
Customer *c = new Customer(*gold.removeFront());
// the returned Customer from gold.removeFront is lost
// Violate Conservation rule
```

```
Customer *c = new Customer();
// ... take input and store into c ...

all.insertBack(c); // queue 'all' has all the customers

if (c->type == "platinum") platinum.insertBack(c);
else if(c->type == "gold") gold.insertBack(c);
else if (c->type == "silver") silver.insertBack(c);
else if (c->type == "regular") regular.insertBack(c);
// violate At-most-once Invariant
```

two containers are managing 1 c invariant

Deepcopy

Deepcopy

Why we need deepcopy?

- Sometimes a type needs to manage members in dynamic memory.

Example: Dlist

```
template <class T>
void Dlist<T>::copyAll(const Dlist<T> &l) {
    node *tmp = l.first;
    while (tmp) {
        T *op = new T(*tmp->op);
        // tmp->op is managed by "l"
        // need to create new dynamic object
        insertBack(op);
        tmp = tmp->next;
    }
}
```

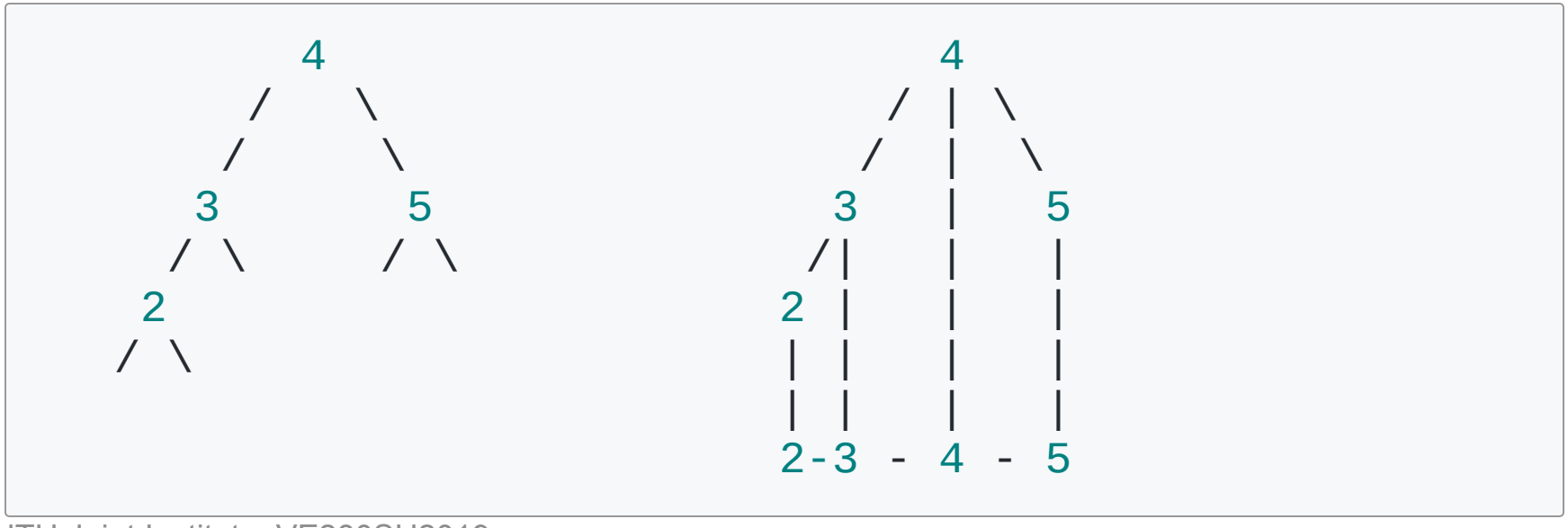
Deepcopy

Question:

Recall `binary tree` and **in-order traversal** that we met in p2.
We define that

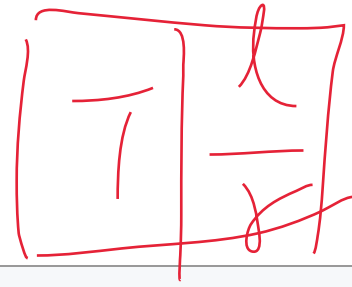
> a **good tree** is a binary tree withascending in-order traversal.

For example, the binary tree below is good (2-3-4-5).



How to deep copy a template good tree?

Provided interface:



```
template <class T>
class GoodTree {
    T *op;
    GoodTree *left;
    GoodTree *right;
public:
    void removeAll();
    // EFFECTS: remove all things of "this"
    void insert(T *op);
    // REQUIRES: T type has a linear order "<"
    // EFFECTS: insert op into "this". Assume no
    //          duplicate op.
};
```

For each `op`, there's only one valid insertion position.

You may use `removeAll` and `insert` in your `copyAll` method.

Deepcopy

Sample Answer:

```
template <class T>
void GoodTree<T>::copy_helper(const GoodTree<T> *t) {
    if (t == nullptr) return;
    T *tmp = new(t->op); insert(tmp);
    copy_helper(t->left);
    copy_helper(t->right);
}

template <class T>
void GoodTree<T>::copyAll(const GoodTree<T> &t) {
    removeAll(); // you can also call removeAll outside
    copy_helper(&t);
}
```

- If use `friend` helper function?
- How to implement `removeAll`? How about `insert`?

The rule of the Big Three: `Dlist`

A destructor

```
template <class T>
Dlist<T>::~Dlist() { removeAll(); }
```

A copy constructor

```
template <class T>
Dlist<T>::Dlist(const Dlist &l): first(0), last(0) {
    if (this != &l) copyAll(l);
}
```

An assignment operator

both variables and const things are accepted

```
template <class T>
Dlist<T> &Dlist<T>::operator=(const Dlist<T> &l) {
    if (this != &l) { removeAll(); copyAll(l); }
    return *this;
}
```

Thanks. Good luck! 👍

August 4

Copyright © 2019 Ge Tianyi