VE280 Final Review

August 4

Zian Ke

Outline

1. Container of Pointers
2. Operator Overloading

Container of Pointers

Container of Pointers

1 invariant + 3 rules

- **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.

Container of Pointers

Coding Task #1

Given the implementation of `Set`, please modify it into a container of pointers.

```
template<class T>
class Set {
    T *elts; int numElts; int sizeElts;
public:
    Set(int size = MAXELTS);
    Set(const Set &s);
    Set &operator=(const Set &s);
    ~Set();
    void insert(T v);
    void remove(T v);
    bool query(T v) const;
    void display() const;
};
```

Container of Pointers

Coding Task #1

> Given the implementation of `Set`, please modify it into a container of pointers.

1. Change the stored element type from `T` to `T*`.
    - Member declarations to be updated:
        - `T **elts;`
        - `void insert(T *vp);`
    - User of `Set` must use `new` before `insert`.
    - **Existence**: An object must be dynamically allocated before a pointer to it is inserted.

Container of Pointers

Coding Task #1

> Given the implementation of `Set`, please modify it into a container of pointers.

2. Dynamic allocation in deep copy.
    - Use `new` in `copyFrom`:
        - `elts[i] = new T(*s.elts[i]);`
    - **At-most-once invariant**: Any object can be linked to at most one container at any time through pointer.

Container of Pointers

Coding Task #1

> Given the implementation of `Set`, please modify it into a container of pointers.

3. Avoid copy-by-value in other member functions.
    - Use const reference:
        - `void remove(const T &v);`
        - `bool query(const T &v) const;`

Container of Pointers

Coding Task #1

> Given the implementation of `Set`, please modify it into a container of pointers.

4. De-allocate memory of removed elements.
    - Use `delete`:
        - `void remove(const T &v);`
        - `~Set();`
        - `void copyFrom(const Set &s);`
    - **Conservation**: When a pointer is removed from a container, either the pointer must be inserted into some container, or its referent must be deleted.

Container of Pointers

Coding Task #1

> Given the implementation of `Set`, please modify it into a container of pointers.

5. One more thing to do with `insert`.
    - Suppose the user of `Set` attempts to insert duplicates.
    - `delete` an inserted element if it already exists:
        - `if (indexOf(*vp) != MAXELTS) delete vp;`
    - **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.

Container of Pointers

Coding Task #1

> Given the implementation of `Set`, please modify it into a container of pointers.

6. What if `remove` and `query` return pointers.
    - Do not use `delete` in `remove`, simply return the element.
        - `return elts[victim];`
    - `delete` should be done by the user of `Set`.
        - `delete set.remove(value);`
    - User of `Set` must not `delete` a `query` result.

Operator Overloading

Operator Overloading

Unary operators vs. Binary operators

- 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.

Operator Overloading

Unary operators

- Defined as class member function

```
Complex Complex::operator-() const {
  return Complex(-(this->real), -(this->imaginary));
}
```

- Defined as ordinary nonmember function

```
Complex operator-(const Complex &obj) {
  return Complex(-(obj.real), -(obj.imaginary));
}
```

Operator Overloading

Binary operators

- Defined as class member function

```
bool Complex::operator==(const Complex &rhs) const {
  return this->real == rhs.real &&
         this->imaginary == rhs.imaginary;
}
```

- Defined as ordinary nonmember function

```
bool operator==(const Complex &lhs, const Complex &rhs) {
  return lhs.real == rhs.real &&
         lhs.imaginary == rhs.imaginary;
}
```

Operator Overloading

Common operator overloadings

- Unary operators
  - `A A::operator-() const;`
  - `A& A::operator++(); // ++a`
  - `A A::operator++(int); // a++`
  - `A& A::operator--(); // --a`
  - `A A::operator--(int); // a--`
  - ...

Operator Overloading

Common operator overloadings

- Binary operators
  - `A& A::operator= (const A& rhs);`
  - `const T& A::operator[] (size_t pos) const;`
  - `T& A::operator[] (size_t pos);`
  - `A& A::operator+= (const A& rhs);`
  - `A& A::operator-= (const A& rhs);`
  - `A operator+ (const A& lhs, const A& rhs);`
  - `istream& operator>> (istream& is, A& rhs);`
  - `ostream& operator<< (ostream& os, const A& rhs);`

Operator Overloading

Common operator overloadings

- Binary operators (continued)
  - `bool operator== (const A& lhs, const A& rhs);`
  - `bool operator!= (const A& lhs, const A& rhs);`
  - `bool operator< (const A& lhs, const A& rhs);`
  - `bool operator<= (const A& lhs, const A& rhs);`
  - `bool operator> (const A& lhs, const A& rhs);`
  - `bool operator>= (const A& lhs, const A& rhs);`
  - ...

Operator Overloading

Coding Task #2

Given the implementation of `Set` (container of pointers), please use it to store self-defined `Rectangle` type.

```
template<class T>
class Set {
    const T **elts; int numElts; int sizeElts;
public:
    Set(int size = MAXELTS);
    Set(const Set &s);
    Set &operator=(const Set &s);
    ~Set();
    void insert(const T *vp);
    const T *remove(const T &v);
    const T *query(const T &v) const;
    void display() const;
};
```

Operator Overloading

Coding Task #2

> Given the implementation of `Set` (container of pointers), please use it to store self-defined `Rectangle` type.

1. Define a `Rectangle` struct.

```
struct Rectangle {
    int height;
    int width;
    int getArea() const;
    string toString() const;
};
```

Operator Overloading

Coding Task #2

> Given the implementation of `Set` (container of pointers), please use it to store self-defined `Rectangle` type.

2. Replace `double` in `main.cpp` with `Rectangle`. Also update the part of initializing random rectangles.

```
for (Rectangle &rectangle : test) {
  rectangle.height = rand();
  rectangle.width = rand();
}
```

Operator Overloading

Coding Task #2

> Given the implementation of `Set` (container of pointers), please use it to store self-defined `Rectangle` type.

3. Overload operators of `Rectangle`. We use nonmember functions here to overload binary operators.
    - `bool operator>(const Rectangle &lhs, const Rectangle &rhs);`
    - `bool operator==(const Rectangle &lhs, const Rectangle &rhs);`
    - `std::ostream &operator<<(std::ostream &os, const Rectangle &rect);`

Operator Overloading

Coding Task #2

> Given the implementation of `Set` (container of pointers), please use it to store self-defined `Rectangle` type.

4. Change `Rectangle` from `struct` to `class`.
    - `height` and `width` become private member variables.
    - Add a constructor to initialize random rectangle.
    - Add `friend` function declarations for all overloaded operators.

Thanks. Good luck! 👍

August 4