MidRC Part3

L13 Abstract Data Types

1. Concept of ADT

Definition

Contains:

1. values representing certain notions
2. operations on the values

Doesn't Contain:

1. details of *how* it is implemented

An example: a mobile phone

Advantages

Information Hiding: we don't need to know the *details* of how the obects are represented, nor do we need to know *how* the operations on those objects are implemented

Encapsulation: ADTs combine both data and operation in one entity

Benefits

Local: The other parts of the program does not depend on the implementation of the ADT. To realize other components, you only need to focus locally.

Substitutable: you can change the implementation and no users of that type can tell.

2. Classes in C++

Basic Idea: Provide a single entity that defines:

1. The **value** of an object
2. The **operation** available on that object. (Sometimes also called **member functions** or **methods**)

Class Definition

In c++, we use class:

Example

- `anInt.h`

```
class anInt{
    int v; // private by default
public:
    int getValue(); // public functions can be used to
    void setValue(int v); // access or modify the private member
    void addValue(anInt x);
};
```

`public:`, `private:`, `protected`: access specifier

- `public` - members are accessible from outside the class
- `private` - members cannot be accessed (or viewed) from outside the class
- `protected` - members cannot be accessed from outside the class, however, they can be accessed in inherited classes

When using class to implement the ADT, we should guarantee the property of information hiding and encapsulation, therefore data members of the ADT are **not allowed** to be public.

Note:

1. definitions include both data elements (like `int v`) and member functions (like `int getValue();`)
2. The default access type is private (different from structs, whose default is public)

Defining Member Functions

Example

- `anInt.cpp`

```
#include "anInt.h"
int anInt::getValue(){
    return v;
}
void anInt::setValue(int v){
    this->v = v; // "this" is a pointer to the current instance
}
void anInt::addValue(anInt x){
    (this->v) += x.v; // the private member is visible to any object of the same
type
}
```

3. Constructor

- Constructor is a special method that is automatically called when an object of a class is created.
- It is the first function called immediately after an object is created.

Note:

- The name of the constructor is the same as the name of the class.
- Constructor does not have a return type.
- The order in which elements are initialized is the order they appear in the definition, not the order in the initialization list.

Example

```
class Character{
    string name;
    int weapon;
    int role;
public:
    Character():name("A"), role(1), weapon(role<<1){}; // default constructor
    Character(string n, int w, int r): // Initialize with parameters
        name(n), weapon(w), role(r) {};
    void getInformation(){
        cout << name << " " << weapon << " " << role << "\n";
    }
};
```

Initialization Syntax:

```
IntSet::IntSet()
: numElts(0)
{}
```

Alternatively, you can write:

```
IntSet::IntSet()
{
    numElts = 0;
}
```

But it's not recommended.

4. Const Member Functions

Example:

```
int size() const;
```

The `const` keyword means: `this` pointer is now a pointer to a const instance.

That is to say: the function `size()` in the example cannot modify the object on which `size()` is called

Rules

- A const member function promise that it will not modify this object.
- A const object can only call its const member functions.
- A const member function can only call other member functions that are const too.

Example

```
const int MAXELTS = 100;
class IntSet {
    int elts[MAXELTS];
    int numElts;
    int indexOf(int v) const;
public:
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int size() const;
};
```

For example, It is NOT ok to:

```
int IntSet::size() const {
    numElts++; // A const member function promise that it will not modify this
object
    return numElts;
}
```

or

```
const IntSet is;
is.insert(2); // A const object can only call its const member functions.
```

or

```
int IntSet::size() const {
    insert(0); // A const member function can only call other member functions
that are const too.
    // omits certain steps one might take here
    remove(0); // A const member function can only call other member functions
that are const too.
    return numElts;
}
```

L14 Subtypes and Inheritance

1. Subtype

$S <: T$ indicates that S is a **subtype** of T , and T is a **supertype** of S.

- We can use S everywhere we expect T
- S is not converted to T

Example

```
void foo(T_Type t);
int main(){
    S_Type s;
    foo(s); // We can use S everywhere we expect T
}
```

How to create a subtype?

In ADT, there are three ways to create a subtype from the supertype:

1. Add one or more operations.
2. Strengthen the postcondition of one or more operations.

- EFFECTS clause
- Return type

3. Weaken the precondition of one or more operations.

- REQUIRES clause
- Argument type

```
class Animal{
protected:
    int food;
public:
    int eat(int x){
        // EFFECTS: the value of food decreases by x
        food -= x;
    }
    void sleep(int t){
        // REQUIRES: t is a positive even integer
    }
};
class Bird: public Creature{
public:
    void fly(); // Add one operation
    int eat(int x){ // Strengthen the postcondition
        // EFFECTS: the value of food decreases by x and a message is printed
        food -= x;
        cout << "Animal eats.\n";
    }
    void sleep(int t){ // Weaken the precondition
        // REQUIRES: t is a positive integer
    }
};
```

Substitution Principle

1. The new method must do everything the old method did, but it is allowed to do more as well.
2. It must require no more of the caller than the old method did, but it can require less.

2. Inheritance

C++ has a mechanism to enable subtyping, called **subclassing**, or sometimes **inheritance**. Subclass of T is not a subtype of T if it violates the substitution principle. (**subclass** $\neq$ **subtype**)

Example

`Sheep` is not a subtype:

```
class Animal{
protected:
    int food;
public:
    void eat(int x){
        food -= x;
    }
};
class Sheep: public Animal{
    int grass;
public:
    void eat(int x){ // different from Animal::eat()
        if(grass>=x) food -= x;
        else food -= grass;
    }
};
```

Rules of Inheritance

Type of Inheritance \ Visibility of Member in T	Public	Protected	Private
Public	Public	Protected	---
Protected	Protected	Protected	---
Private	Private	Private	---

3. Virtual Function

- **Apparent type:** the declared type of the pointer of reference.
- **Actual type:** the real type of the pointer of reference.

```
class Foo{...}
class Bar: public Foo{...}
...
    Bar bar; // static instance
    Foo foo; // static instance
    Foo *fp = bar; // non-static instance
    Foo &fr = bar; // non-static instance
    Bar *bp = foo; // not allowed
...
```

For static instance, the apparent type is equal to the actual type.

For non-static instance, the member function can be marked as `virtual`, so that the function call is based on the **actual type**, rather than the **apparent type** of the pointer of reference.

The property of virtual is inheritable.

Example

```
class A{
public:
    void f(){std::cout<<"A::f\n";}
    virtual void g()=0;
    virtual void h(){std::cout<<"A::h\n";}
};
class B: public A{
public:
    virtual void f(){std::cout<<"B::f\n";}
    void g(){std::cout<<"B::g\n";}
    void h(){std::cout<<"B::h\n";}
};
class C: public B{
public:
    void f(){std::cout<<"C::f\n";}
    void h(){A::h();}
    };
class D: public C{
public:
    virtual void g(){std::cout<<"D::g\n";}
    void h(){std::cout<<"D::h\n";}
};
```

Variable	D d;	A *pa = &d;	B *pb = &d;	C c;	A &ra = c;
_.f()/->f()	C::f	A::f	C::f	C::f	A::f
_.g()/->g()	D::g	D::g	D::g	B::g	B::g
_.h()/->h()	D::h	D::h	D::h	A::h	A::h

Interface (L15)

Pure virtual function: A member function without implementation, which is assigned to zero.

```
virtual void insert(int v) = 0; // no implementation
```

Abstract class: A class with one or more pure virtual functions is an abstract class (pure virtual class). It is only used to define the interface. You **cannot** create any object of the abstract class. **However**, However, you can always define references and pointers to an abstract class.

```
// In .h file
class Player {
// A virtual base class, providing the player interface
public:
    virtual bool draw(Deck d) = 0;
    virtual void play(Hand h) = 0;
    virtual void shuffled() = 0;
    virtual ~Player() { }
};
```

In `.h` and `.cpp` :
`.h` file contains the interface and function declarations.
`.cpp` file contains information of the derived classes, including declaration and implementation.

Creating Instances From the Interface

Since only the content in `.h` file is exposed to the user (or the main function), we are not allowed to declare the instance of the derived class as normal. There are two alternatives provided to solve this issue:

- **static instance**
  If your ADT has only one instance, you can use `static` to declare it.

  ```
  // .h
  IntSet *getIntSet();
  ```

  ```
  // .cpp
  static IntSetImpl impl;
  IntSet *getIntSet(){
      return &impl;
  }
  ```

- **dynamic instance**

  If your ADT has multiple instances, you can use `new` to declare it.

  ```
  // .h
  IntSet *getIntSet();
  ```

```cpp
// .cpp
IntSet *getIntSet(){
    return new IntSetImpl();
}
```

Reference

[1] Zhang Wenjing. VE280 RC6-7. 2023FA.

[2] Zhongqiang Ren. VE280 Lecture 13AB. 2024SP.

[3] Weikang Qian. VE280 Lecture 14-15. 2023FA.