CS32 Discussion
2019 Summer - Week 4
Qi Zhao
zhaoq0563@ucla.edu
Jul-19-2019
Outline

• Inheritance
• Polymorphism
Outline

• Inheritance
• Polymorphism
Inheritance

- The process of deriving a new class using another class as a base.
- Our example:

  ![Diagram](image)

  - **Dog**
    - Features of a dog
  - **Cat**
    - Features of a cat

- But there might be some common features in the two...
Inheritance

- The process of deriving a new class using another class as a base.
- Our example:
Deriving a class from another

- Dog inherits Animal.
Deriving a class from another

```java
Dog d1;
d1.setName("puppy");
d1.getAge();
d1.speak();
```

```java
Animal a1;
a1.speak();
a1.setName("abc");
```
Deriving a class from another

• What’s inherited:
  – all member functions except the overloaded assignment operator (operator=), constructors, and the destructor
  – all member variables

• However, the derived class cannot access the private members of the base class directly (e.g. Dog cannot access m_age).

• class D : public B
  – a D object is a kind of B
  – a D is a B (a Dog is an Animal)
Construction

- So, a Dog is an Animal. What happens when we construct a Dog?
- 1. The base part of the class (Animal) is constructed.
Construction

- So, a Dog is an Animal. What happens when we construct a Dog?
- The member variables of Dog are constructed.

```
Animal
  m_age 0
  m_name ""
```
Construction

- So, a Dog is an Animal. What happens when we construct a Dog?
- The body of Dog's constructor is executed.
Construction

• Suppose I want to overload Dog’s constructor to create:
  
  Dog(string initName, int initAge);

• How would I go about implementing it?
• Suppose I want to overload Dog's constructor to create:

```cpp
Dog::Dog(string initName, int initAge)
: m_age(initAge), m_name(initName)
{
}
```

**Incorrect**
Construction

- Suppose I want to overload Dog’s constructor to create:

  ```cpp
  Dog::Dog(string initName, int initAge)
  : Animal(initAge), m_name(initName)
  {}
  ```

  ```cpp
class Animal
  {
    public:
      Animal(int initAge);
      ...
  }
  ```
Destruction

• Just reverse the order of construction.
• 1. The body of destructor is executed.
• 2. The member variables are removed.
• 3. The base part of the class is destructed.
Overriding member functions

- Assume `speak()` is implemented as follows.

```cpp
void Animal::speak() const
{
    cout << "..." << endl;
}
```

- Dog inherits this function.
- But we want our Dog to really say something when ordered to speak!
Overriding member functions

```cpp
class Dog : public Animal {
public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
    void speak() const;
private:
    string m_name;
};

void Dog::speak() const {
    cout << "Woof!" << endl;
}
```

```cpp
Animal a1;
a1.speak();

• Output
...

Dog d1;
d1.speak();

• Output
  Woof!
```
Overriding member functions

```cpp
class Dog : public Animal
{
public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
    void speak() const;
private:
    string m_name;
};

void Dog::speak() const
{
    cout << "Woof!" << endl;
}
```

- Why do we call this **overriding**, not **overloading**?
- Overload – same function name, but different return type and/or different set of arguments
- Override – same function name, same return type, same everything, except defined “again” in the derived class. 
Overriding member functions

- Can I still call the base class's `speak()` on a Dog object?
- Yes, just do:

```cpp
Dog d1;
d1.Animal::speak();
```
```cpp
#include <iostream>
using namespace std;

class A {
public:
    A() { cout << "1"; }
    A(const A &obj) { cout << "2"; }
};

class B : public A {
public:
    B() { cout << "3"; }
    B(const B &obj) { cout << "4"; }
};

class C : public B {
public:
    C() { cout << "7"; }
    C(const C &obj) { cout << "8"; }
};

int main() {
    C c1;
    C c(c1);
}
```

What is the output?

137138
Outline

• Inheritance

• Polymorphism
Virtual functions: Motivation

- Back to this diagram:

```
Animal
  Implement common features of all animals here

Dog
  Implement features specific to dogs here

Cat
  Implement features specific to cats here
```

- Suppose we have `speak()` overridden in `Cat`, where it goes “Meow!”;
Virtual functions: Motivation

- C++ allows a pointer to the base class to point to a derived class.

- What do you think
  pAni->speak();
  will do?
  should do?

```cpp
Animal *pAni;
int x;
cin >> x;
switch (x)
{
    case 1:
        pAni = new Dog;
        break;
    case 2:
        pAni = new Cat;
        break;
    default:
        pAni = new Animal;
        break;
}
```
Virtual functions: Motivation

- What it will do:
  “...”, no matter what x is.
- What it should do:
  “Woof!” if x == 1,
  “Meow!” if x == 2,
  “...” otherwise
- We want the overridden function to be called!

```cpp
Animal *pAni;
int x;
cin >> x;

switch (x)
{
  case 1:
    pAni = new Dog;
    break;
  case 2:
    pAni = new Cat;
    break;
  default:
    pAni = new Animal;
    break;
}
```
Virtual functions

class Animal
{
    public:
        Animal();
        Animal();
        virtual ~Animal();
        int getAge() const;
        virtual void speak() const;
    private:
        int m_age;
};

class Dog : public Animal
{
    public:
        Dog();
        ~Dog();
        string getName() const;
        void setName(string name);
        void speak() const;
    private:
        string m_name;
};

• pAni->speak();
Polymorphism

- Late binding / dynamic binding
  - The appropriate version is selected during runtime!
- **Polymorphism**
  - pAni can take multiple forms.

```cpp
Animal *pAni;
int x;
cin >> x;
switch (x)
{
    case 1:
        pAni = new Dog;
        break;
    case 2:
        pAni = new Cat;
        break;
    default:
        pAni = new Animal;
        break;
}
```
```cpp
#include<iostream>

using namespace std;

class Base {
public:
    Base() { cout<<"Constructor: Base"<<endl; }
    ~Base() { cout<<"Destructor :Base"<<endl; }
};

class Derived: public Base {
public:
    Derived() { cout<<"Constructor: Derived"<<endl; }
    ~Derived() { cout<<"Destructor : Derived"<<endl; }
};

int main() {
    Base *Var = new Derived();
    delete Var;
    return 0;
}
```

What is the output?
Virtual functions

```cpp
class Animal {
public:
    Animal();
    virtual ~Animal();
    int getAge() const;
    virtual void speak() const;
private:
    int m_age;
};
```

```cpp
class Dog : public Animal {
public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
    virtual void speak() const;
private:
    string m_name;
};
```

• Wait, what’s that virtual doing before the destructor of Animal?
Animal speaks?

- speak is a common feature among all (or many) animals.
- But it really means something only if we know what this animal is.
- Option 1:
  - Get rid of speak() function in Animal, and implement it in all the derived classes.
    - Then we can’t do pAni->speak()...
- Option 2:
  - Make it a **pure virtual function**.
Pure virtual functions

- You declare it in the base class, but don’t define it, and add “= 0” in the declaration.
- It is a dummy function.
- The derived class **must** implement all the pure virtual functions of its base class.

```cpp
class Animal
{
public:
    Animal();
    virtual ~Animal();
    int getAge() const;
    virtual void speak() const = 0;
private:
    int m_age;
};
```
Abstract base class

- If a class has at least one pure virtual function, it is called an **abstract base class**.

```cpp
Animal a1; // won’t compile
Animal *pAni = new Animal; // won’t compile
Animal *pAni = new Dog; // still works
```

- Animal is like a “common” interface without complete implementation. Or, one can think of it as a “framework.”
#include <iostream>
using namespace std;

class A
{
public:
    virtual void fun() { cout << "A::fun() " ; }
};
class B: public A
{
public:
    void fun() { cout << "B::fun() " ; }
};
class C: public B
{
public:
    void fun() { cout << "C::fun() " ; }
};
int main()
{
    B *bp = new C;
    bp->fun();
    return 0;
}
See you next week!