12. Inheritance

June 22, 2023

1 Inheritance

…is an important concepts in object-oriented programming. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

1.1 1. Types of inheritance

There are three types of inheritance – public, protected, and private.

Base Class

Public Inheritance

Protected Inheritance

Private Inheritance

Public Members

public

protected

private

Protected Memebers

protected

protected

private

Private Members

private

private

private

[ ]:
```
class Base {
protected:
int base;
private:
```

```
int base_priv;
public:
Base(int b) : base(b) {};
void baseMemberFunct();
};

class Derived: public Base {
private:
int derived;
public:
Derived(int d, int b) : derived(d), Base(b) {};
void derivedMemberFunct();
};
```

[ ]:
```
Derived d(1,2);
d.base_priv = 0;
```

: yes : no

1.2 2. Polymorphism in C++

Polymorphism is an important feature of object oriented programming and means literally having many forms. There are two types of polymorphism in C++. The first type compile time polymorphism includes function and operator overloading. The second type runtime polymorphism includes function overring, explained below.

1.2.1 2.1 Overriding

A virtual function is defined within a base class and redefined (overriden) in a derived class.

[ ]:
```
class Base {
public:
virtual void foo(double a, int b);
};
class Derived: public Base {
public:
void foo(double i, int j);
};

void Base::foo(double a, int b){
std::cout << "Base foo!"
<< std::endl;
}
void Derived::foo(double i, int j ){
std::cout << "Derived foo!"
<< std::endl;
}
```

```
Base b;
b.foo(4.5, 2);
Derived d;
d.foo(4, 6);
```

The output here will be: Base foo! Derived foo!

Limitation: calls to virtual functions take longer and are more difficult for the compiler to optimize, so use them wisely. However, since C++20 you can use constexpr virtual functions, which will be evaluated during compilation time.

A pure virtual function must be included in each derived class. The base class in this case is abstract and can serve only as a base class.

[ ]:
```
class Base {
public:
virtual void foo(double a, int b)=0;
};
class Derived: public Base {
public:
void foo(double a, int b) override;
};
```

[ ]:
```
Base b; // error, can not instantiate an abstract class
Derived d; // OK!
```

The specifier override can be used to ensure that the function is virtual in the base class. C++11

1.2.2 2.2 Prohibiting overriding

Use the final specifier to make sure a function can not be overriden.C++11

[ ]:
```
class Base {
public:
virtual void foo();
virtual void mee();
int bar() final;
};
class Middle: public Base {
public:
void foo() override final;
virtual void mee();
};
class Derived: public Middel {
public:
void void mee();
};
```

1.2.3 2.3 Example of Polymorphism – let's put all of this to use

[ ]:
```
class Animal {
public:
virtual void eat()=0; // abstract base class
};

class Dog: public Animal { public: void eat() override;};

void Dog::eat() { std::cout << "eating meat" << std::endl; }

class Cat: public Animal { public: void eat() override;};

void Cat::eat() { std::cout << "eating fish" << std::endl; }

class Cow: public Animal { public: void eat() override; };

void Cow::eat() { std::cout << "eating grass" << std::endl; }


std::vector<Animal *> animals;
Animal *dog = new Dog();
Animal *cat = new Cat();
Animal *cow = new Cow();
animals.push_back(dog);
animals.push_back(cat);
animals.push_back(cow);
for(auto animalPtr: animals) {
animalPtr->eat();
}
```

Output: eating meat eating fish eating grass

1.2.4 2.4 Special member functions

[ ]:
```
class Base {
protected:
int base;
public:
Base(int b) : base(b) {};
Base(const Base& rhs) { ... }
Base& operator=(const Base &rhs){ ... }
virtual ~Base() { ... };
};
```

[ ]:
```
class Derived: public Base {
private:
int derived;
```

```
public:
Derived(int d, int b) : derived(d), Base(b) {}
Derived(const Derived & rhs):Base(rhs) {
derived = rhs.derived;
}
Derived & operator=( const Derived & rhs) {
Base::operator=(rhs);
derived = rhs.derived;
return *this;
}
virtual ~Derived(){};
};
```

[ ]:
```
Base * b = new Derived(1,2);
delete b; // calls Derived::~Derived()
```

Virtual destructors are not really "overriden", but called in a reverse order from the highest derived class to the base class.

[ ]:
```
Base::~Base() { std::cout << "deleting Base" << std::endl };
Derived::~Derived() { std::cout << "deleting Derived" << std::endl };

Base *b = new Derived;
delete b;
```

What do you think would be the output?