C++

Programming

继承与派生

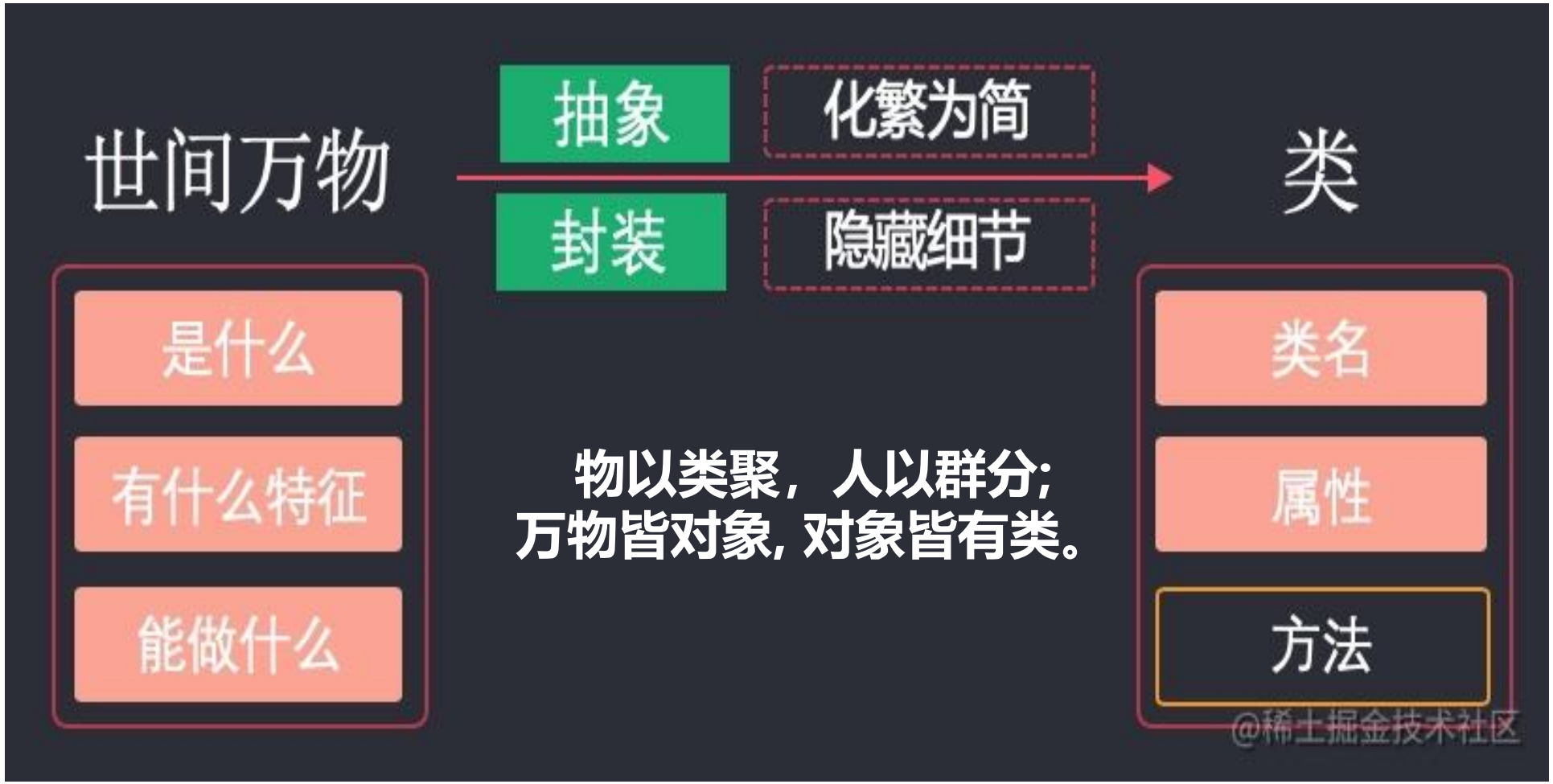
Inheritance & Derive

2025年3月31日

学而不厌 诲人不倦

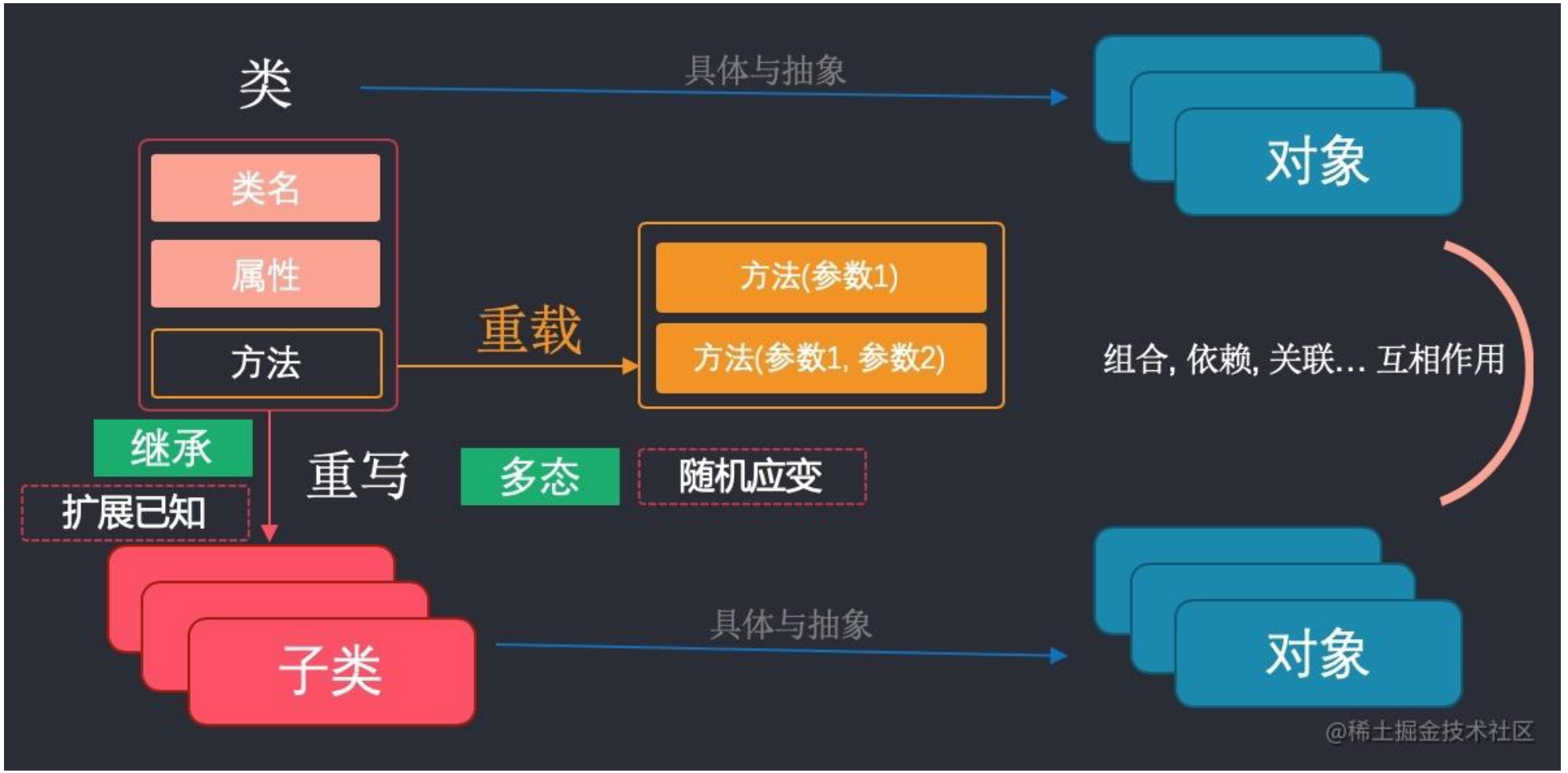
回顾

面向对象编程：抽象、封装、信息隐藏



回顾

面向对象编程：模板、继承、多态



5.3 派生类成员的访问属性

➢ 1. 公有、保护、私有继承

class 派生类名: [继承方式] 基类名

{ 派生类新增成员声明 };

类成员的访问权限由高到低依次为
public --> protected --> private，
protected 成员和 private 成员类似，也不能通过对象访问。

继承方式包括：public、private、protected。如果省略，系统默认为private。

继承方式/基类成员	public成员	protected成员	private成员
public继承	public	protected	不可见
protected继承	protected	protected	不可见
private继承	private	private	不可见

- 继承方式中的 public、protected、private 是用来指明基类成员在派生类中的最高访问权限
- 基类成员在派生类中的访问权限不得高于继承方式中指定的权限。
- 基类的 private 成员是能够被继承的，并且（成员变量）会占用派生类对象的内存，它只是在派生类中不可见，导致无法使用罢了。

5.3 派生类成员的访问属性

➢ Demo03：公有继承

```
#include<iostream>
using namespace std;
//基类People
class People{
public:
    void setname(string name);
    void setage(int age);
    void sethobby(string hobby);
    string gethobby();
protected:
    string m_name;
    int m_age;
private:
    string m_hobby;
};
```

```
//派生类Student
class Student: public People{
public:
    void setscore(float score);
protected:
    float m_score;
};
void Student::setscore(float
score){ m_score = score; }
```

```
void People::setname(string name){ m_name = name; }
void People::setage(int age){ m_age = age; }
void People::sethobby(string hobby){ m_hobby = hobby; }
string People::gethobby(){ return m_hobby; }
```

5.3 派生类成员的访问属性

➢ Demo03：公有继承

```
//派生类Pupil
class Pupil: public Student{
public:
    void setranking(int ranking);
    void display();
private:
    int m_ranking;
};
void Pupil::setranking(int
ranking){ m_ranking = ranking; }
void Pupil::display(){
cout<<m_name<<"的年龄是"<<m_age<<",
考试成绩为"<<m_score<<"分，班级排名第
"<<m_ranking<<", TA喜欢
"<<gethobby()<<"。"<<endl;
}
```

```
int main(){
   Pupil pup;
   pup.setname("小明");
   pup.setage(15);
   pup.setscore(92.5f);
   pup.setranking(4);
   pup.sethobby("乒乓球");
   pup.display();
   return 0;
}
```

http://c.biancheng.net/view/2269.html

➢ 2. 修改继承方式，改变访问权限 Demo04

```
#include<iostream>
using namespace std;
//基类People
class People {
public:
    void show();
protected:
    string m_name;
    int m_age;
};
void People::show() {
    cout << m_name << "的年龄是" << m_age << endl;
}
```

5.3 派生类成员的访问属性

➢ 2. 修改继承方式，改变访问权限 Demo04

```
//派生类Student
class Student : public People {
public:
    void learning();
public:
    using People::m_name; //将protected改为public
    using People::m_age; //将protected改为public
    float m_score;
private:
    using People::show; //将public改为private
};
void Student::learning() {
    cout << "我是" << m_name << "，今年" << m_age
<< "岁，这次考了" << m_score << "分！" << endl;
}
```

```
int main() {
    Student stu;
    stu.m_name = "小明";
    stu.m_age = 16;
    stu.m_score = 99.5f;
    stu.show(); // error
    stu.learning();
    return 0;
}
```

使用 using 改变了默认访问权限，将 show() 函数修改为 private 属性的，是降低访问权限，将 name、age 变量修改为 public 属性的，是提高访问权限。

➢ 1. C++基类和派生类的构造函数

类的构造函数不能被继承：即使继承了也与类的名字不同，不能成为构造函数。
在设计派生类时，对继承过来的成员变量的初始化工作也要由派生类的构造函数完成，但是大部分基类都有 private 属性的成员变量，它们在派生类中无法访问，更不能使用派生类的构造函数来初始化。
在派生类的构造函数中可以调用基类的构造函数。

```
派生类名::派生类名(基类所需的形参，本类成员所需的形参):基类名(基类参数表)
{
        本类成员初始化赋值语句;
};
```

5.4 派生类的构造函数和析构函数

➢ 1. C++基类和派生类的构造函数：Demo05

```
#include<iostream>
using namespace std;
//基类People
class People{
protected:
    string m_name;
    int m_age;
public:
People(string, int);
};

People::People(string name, int age): m_name(name),
m_age(age){}
```

5.4 派生类的构造函数和析构函数

➢ 1. C++基类和派生类的构造函数：Demo05

```
//派生类Student
class Student: public People{
private:
    float m_score;
public:
    Student(string name, int age, float score);
void display();
};
//People(name, age)就是调用基类的构造函数
Student::Student(string name, int age, float score): People(name, age),
m_score(score){ }
void Student::display(){
    cout<<m_name<<"的年龄是"<<m_age<<", 成绩是"<<m_score<<"。"<<endl;
}
```

```
int main(){
Student stu("小明", 16, 90.5);
stu.display();
return 0;
}
```

`m_score(score)`:派生类的参数初始化表

5.4 派生类的构造函数和析构函数

➢ 1. C++基类和派生类的构造函数

```
//People(name, age)就是调用基类的构造函数
Student::Student(string name, int age, float score): People(name, age),
m_score(score){ }
```

```
Student::Student(string name, int age, float score): m_score(score),
People(name, age){ }
```

```
Student::Student(string name, int age, float score){
    People(name, age);//基类构造函数不会被继承，不能当做普通成员函数调用
    m_score = score;
}
```

函数头部是对基类构造函数的调用，而不是声明，所以括号里的参数是实参，它们不但可以是派生类构造函数参数列表中的参数，还可以是局部变量、常量等，例如：

```
Student::Student(string name, int age, float score): People("小明", 16),
m_score(score){ }
```

2. 构造函数的调用顺序

基类构造函数总是被优先调用，这说明创建派生类对象时，会先调用基类构造函数，再调用派生类构造函数

继承关系： A类 --> B类--> C类

构造函数执行顺序：A类构造函数 --> B类构造函数 --> C类构造函数

- 派生类构造函数中只能调用直接基类的构造函数，不能调用间接基类的。
- 定义派生类构造函数时最好指明基类构造函数；如果不指明，就调用基类的默认构造函数（不带参数的构造函数）；如果没有默认构造函数，那么编译失败。

➢ 2. 构造函数的调用顺序 Demo06

```
#include <iostream>
using namespace std;
//基类People
class People{
public:
    People(); //基类默认构造函数
    People(string name, int age);
protected:
    string m_name;
    int m_age;
};
People::People(): m_name("xxx"), m_age(0){ }
People::People(string name, int age): m_name(name), m_age(age){}
```

5.4 派生类的构造函数和析构函数

➢ 2. 构造函数的调用顺序 Demo06

```
//派生类Student
class Student: public People{
public:
    Student();
    Student(string, int, float);
public:
    void display();
private:
    float m_score;
};
Student::Student(): m_score(0.0){ } //派生类默认构造函数
Student::Student(string name, int age, float score): People(name, age),
m_score(score){ }
void Student::display(){
    cout<<m_name<<"的年龄是"<<m_age<<", 成绩是"<<m_score<<"。"<<endl;
}
```

```
int main(){
  Student stu1;
  stu1.display();
  Student stu2("小明", 16, 90.5);
  stu2.display();
  return 0;
}
```

➢ 3. 析构函数

和构造函数类似，析构函数也不能被继承。与构造函数不同的是，在派生类的析构函数中不用显式地调用基类的析构函数，因为每个类只有一个析构函数，编译器知道如何选择，无需程序员干涉。

•析构函数的执行顺序和构造函数的执行顺序也刚好相反：

•创建派生类对象时，构造函数的执行顺序和继承顺序相同，即先执行基类构造函数，再执行派生类构造函数。

•而销毁派生类对象时，析构函数的执行顺序和继承顺序相反，即先执行派生类析构函数，再执行基类析构函数。

5.4 派生类的构造函数和析构函数

➢ 3. 析构函数 Demo07

```
#include <iostream>
using namespace std;
class A{
public:
    A(){cout<<"A constructor"<<endl;}
   ~A(){cout<<"A destructor"<<endl;}
};
class B: public A{
public:
    B(){cout<<"B constructor"<<endl;}
   ~B(){cout<<"B destructor"<<endl;}
};
```

```
class C: public B{
public:
    C(){cout<<"C constructor"<<endl;}
   ~C(){cout<<"C destructor"<<endl;}
};
int main(){
   C test;
   return 0;
}
```

➢ 1. 多重继承的声明与构造函数

派生类只有一个基类，称为单继承（Single Inheritance）。
C++也支持多继承（Multiple Inheritance），一个派生类可以有两个或多个基类。

```
class D: public A, private B, protected C{
    //类D新增加的成员
}
```

```
D(形参列表): A(实参列表), B(实参列表), C(实参列表){
    //其他操作
}
```

基类构造函数的调用顺序和它们在派生类构造函数中出现的顺序无关，而是和声明派生类时基类出现的顺序相同。先调用 A 类的构造函数，再调用 B 类构造函数，最后调用 C 类构造函数。

5.5 多重继承

➢ 1. 多重继承的声明与构造函数 Demo08

```
#include <iostream>
using namespace std;
//基类
class BaseA{
public:
    BaseA(int a, int b);
   ~BaseA();
protected:
    int m_a;
    int m_b;
};
BaseA::BaseA(int a, int b): m_a(a), m_b(b){
    cout<<"BaseA constructor"<<endl;
}
BaseA::~BaseA(){
    cout<<"BaseA destructor"<<endl;
}
```

5.5 多重继承

➢ 1. 多重继承的声明与构造函数 Demo08

```
//基类
class BaseB{
public:
    BaseB(int c, int d);
   ~BaseB();
protected:
    int m_c;
    int m_d;
};
BaseB::BaseB(int c, int d): m_c(c), m_d(d){
    cout<<"BaseB constructor"<<endl;
}
BaseB::~BaseB(){
    cout<<"BaseB destructor"<<endl;
}
```

➢ 1. 多重继承的声明与构造函数 Demo08

```
//派生类
class Derived: public BaseA, public BaseB{
public:
    Derived(int a, int b, int c, int d, int e);
   ~Derived();
public:
    void show(){ cout<<m_a<<","<<m_b<<","<<m_c<<","<<m_d<<","<<m_e<<endl;};
private:
    int m_e;
};
Derived::Derived(int a, int b, int c, int d, int e): BaseA(a, b), BaseB(c,
d), m_e(e){
    cout<<"Derived constructor"<<endl;
}
Derived::~Derived(){
    cout<<"Derived destructor"<<endl;
}
```

```
int main(){
Derived obj(1, 2, 3, 4, 5);
obj.show();
return 0;
}
```

2. 多重继承时命名冲突的解决 Demo09

```
#include <iostream>
using namespace std;
//基类
class BaseA{
public:
    BaseA(int a, int b);
   ~BaseA();
public:
    void show();
protected:
    int m_a;
    int m_b;
};
BaseA::BaseA(int a, int b): m_a(a), m_b(b){
    cout<<"BaseA constructor"<<endl;}
BaseA::~BaseA(){
    cout<<"BaseA destructor"<<endl;
}
```

```
void BaseA::show(){
    cout<<"m_a = "<<m_a<<endl;
    cout<<"m_b = "<<m_b<<endl;
}
```

2. 多重继承时命名冲突的解决 Demo09

```
//基类
class BaseB{
public:
     BaseB(int c, int d);
    ~BaseB();
     void show();
protected:
     int m_c;
     int m_d;
};
BaseB::BaseB(int c, int d): m_c(c), m_d(d){
     cout<<"BaseB constructor"<<endl;
}
BaseB::~BaseB(){
     cout<<"BaseB destructor"<<endl;
}
```

```
void BaseB::show(){
     cout<<"m_c = "<<m_c<<endl;
     cout<<"m_d = "<<m_d<<endl;
}
```

➢ 2. 多重继承时命名冲突的解决 Demo09

```
//派生类
class Derived: public BaseA, public BaseB{
public:
    Derived(int a, int b, int c, int d, int e);
   ~Derived();
public:
    void display();
private:
    int m_e;
};
Derived::Derived(int a, int b, int c, int d, int e): BaseA(a, b), BaseB(c, d), m_e(e){
    cout<<"Derived constructor"<<endl;
}
Derived::~Derived(){
    cout<<"Derived destructor"<<endl;
}
```

```
void Derived::display(){
    BaseA::show(); //调用BaseA类的show()函数
    BaseB::show(); //调用BaseB类的show()函数
    cout<<"m_e = "<<m_e<<endl;
}
```

➢ 类的组合：

在一个类中以另一个类的对象作为数据成员

继承是纵向的，组合是横向的

```
class Teacher
{
    private:
        int num;
        string name;
        char sex;
    public:
        ...
};
```

```
class BirthDate
{
    private:
        int year;
        int month;
        int day;
    public:
        ...
};
```

```
class Professor:public Teacher
{
    private:
        BirthDate:birthday;
    public:
        ...
};
```

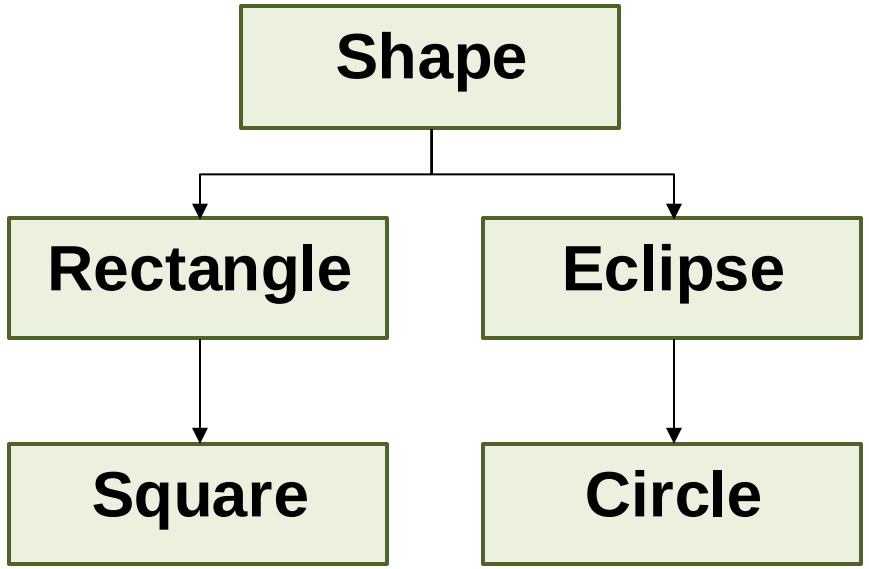
小结

Chapter 5 继承与派生

- 5.1 继承与派生的概念
- 5.2 派生类的声明方式与构成
- 5.3 派生类成员的访问属性
- 5.4 派生类的构造函数和析构函数
- 5.5 多重继承

类的继承与派生

1. 设计基类shape
2. 设计基类的两个派生类Rectangle和Eclipse
3. 从Rectangle类派生Square类
4. 从Eclipse类派生Circle类



Thank You !

Q & A