程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2025 年 3 月 5 日

大纲

继承和派生的概念

继承：在定义一个新的类B时，如果该类与某个已有的类A相似 (指的是B拥有A的全部特点)，那么就可以把A作为一个基类，而把B作为基类的一个派生类 (也称子类)。

继承和派生的概念

继承：在定义一个新的类B时，如果该类与某个已有的类A相似 (指的是B拥有A的全部特点)，那么就可以把A作为一个基类，而把B作为基类的一个派生类 (也称子类)。

- 派生类是通过对基类进行修改和扩充得到的。
  - 扩充: 在派生类中，可以添加新的成员变量和成员函数
  - 修改: 在派生类中，可以重新编写从基类继承得到的成员
- 派生类一经定义后，可以独立使用，不依赖于基类。

派生类的写法

```
class 派生类名 : 派生方式说明符 基类名 {
  //code
};
```

派生方式说明符：public，private，protected

派生类的写法

```
class 派生类名 : 派生方式说明符 基类名 {
  //code
};
```

派生方式说明符：public，private，protected

```
1  class CStudent {
2  private:
3    string sName;
4    int nAge;
5  public:
6    bool isThreeGood() {}
7    void setName(const string &name) {
8      sName = name;
9    }
10 };
11 class CUndergraduateStudent : public CStudent {
12 private:
13   int nDepartment;
14 public:
15   bool isThreeGood() {} //覆盖
16   bool canBaoYan() {}
17 }; // 派生类的写法是：类名 : 派生方式说明符 基类名
```

派生类对象的内存空间

派生类对象的大小，等于基类对象的大小，再加上派生类对象自己的成员变量的大小。在派生类对象中，包含着基类对象，而且基类对象的存储位置位于派生类对象新增的成员变量之前。

派生类对象的内存空间

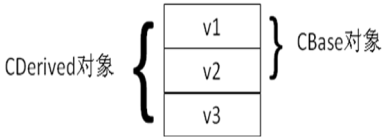
派生类对象的大小，等于基类对象的大小，再加上派生类对象自己的成员变量的大小。在派生类对象中，包含着基类对象，而且基类对象的存储位置位于派生类对象新增的成员变量之前。

```
1  class CBase {
2    int v1;
3    int v2;
4  };
5  class CDerived : public CBase {
6    int v3;
7  };
```

派生类对象的内存空间

派生类对象的大小，等于基类对象的大小，再加上派生类对象自己的成员变量的大小。在派生类对象中，包含着基类对象，而且基类对象的存储位置位于派生类对象新增的成员变量之前。

```
1  class CBase {
2    int v1;
3    int v2;
4  };
5  class CDerived : public CBase {
6    int v3;
7  };
```



覆盖

派生类可以定义一个和基类成员同名的成员，这叫覆盖。在派生类中访问这类成员时，缺省的情况是访问派生类中定义的成员。要在派生类中访问由基类定义的同名成员时，要使用作用域符号::。

覆盖

派生类可以定义一个和基类成员同名的成员，这叫覆盖。在派生类中访问这类成员时，缺省的情况是访问派生类中定义的成员。要在派生类中访问由基类定义的同名成员时，要使用作用域符号::。

```
1  class Base {
2    int j;
3  public:
4    int i;
5    void func();
6  };
7  class Derived : public Base {
8  public:
9    int i;
10   void access();
11   void func();
12 };
13 void Derived::access() {
14   j = 5;         // error
15   i = 5;         //引用的是派生类的 i
16   Base::i = 5;  //引用的是基类的 i
17   func();        //派生类的
18   Base::func(); //基类的
19 }
```

覆盖

派生类可以定义一个和基类成员同名的成员，这叫覆盖。在派生类中访问这类成员时，缺省的情况是访问派生类中定义的成员。要在派生类中访问由基类定义的同名成员时，要使用作用域符号::。

```
1  class Base {
2    int j;
3  public:
4    int i;
5    void func();
6  };
7  class Derived : public Base {
8  public:
9    int i;
10   void access();
11   void func();
12 };
13 void Derived::access() {
14   j = 5;         // error
15   i = 5;         //引用的是派生类的 i
16   Base::i = 5;   //引用的是基类的 i
17   func();        //派生类的
18   Base::func(); //基类的
19 }
```

覆盖

派生类可以定义一个和基类成员同名的成员，这叫覆盖。在派生类中访问这类成员时，缺省的情况是访问派生类中定义的成员。要在派生类中访问由基类定义的同名成员时，要使用作用域符号::。

```
1  class Base {
2    int j;
3  public:
4    int i;
5    void func();
6  };
7  class Derived : public Base {
8  public:
9    int i;
10   void access();
11   void func();
12 };
13 void Derived::access() {
14   j = 5;         // error
15   i = 5;         //引用的是派生类的 i
16   Base::i = 5;  //引用的是基类的 i
17   func();        //派生类的
18   Base::func(); //基类的
19 }
```

Obj占用的存储空间

一般来说，基类和派生类不定义同名成员变量。

单选题

派生类和基类有同名同参数表的成员函数，这种现象：

- A 叫重复定义，是不允许的
- B 叫函数的重载
- C 叫覆盖。在派生类中基类的同名函数就没用了
- D 叫覆盖。体现了派生类对从基类继承得到的特点的修改

单选题

派生类和基类有同名同参数表的成员函数，这种现象：

- A 叫重复定义，是不允许的
- B 叫函数的重载
- C 叫覆盖。在派生类中基类的同名函数就没用了
- D 叫覆盖。体现了派生类对从基类继承得到的特点的修改

答案：D

单选题

以下说法正确的是：

- A 派生类可以和基类有同名成员函数，但是不能有同名成员变量
- B 派生类的成员函数中，可以调用基类的同名同参数表的成员函数
- C 派生类和基类的同名成员函数必须参数表不同，否则就是重复定义
- D 派生类和基类的同名成员变量存放在相同的存储空间

单选题

以下说法正确的是：

- A 派生类可以和基类有同名成员函数，但是不能有同名成员变量
- B 派生类的成员函数中，可以调用基类的同名同参数表的成员函数
- C 派生类和基类的同名成员函数必须参数表不同，否则就是重复定义
- D 派生类和基类的同名成员变量存放在相同的存储空间

答案：B

继承 vs 复合

- 继承：“是”关系。
  基类 A，B是基类A的派生类。
  逻辑上要求：“一个B对象也是一个A对象”。
- 复合：“有”关系。
  类C中“有”成员变量k，k是类D的对象，则C和D是复合关系
  一般逻辑上要求：“D对象是C对象的固有属性或组成部分”。

三种继承方式

- 公有继承 (`public`)
  公有继承的特点是基类的公有成员和保护成员作为派生类的成员时，它们都保持原有的状态，而基类的私有成员仍然是私有的，不能被这个派生类的子类所访问。
- 私有继承 (`private`)
  私有继承的特点是基类的公有成员和保护成员都作为派生类的私有成员，并且不能被这个派生类的子类所访问。
- 保护继承 (`protected`)
  保护继承的特点是基类的所有公有成员和保护成员都成为派生类的保护成员，并且只能被它的派生类成员函数或友元访问，基类的私有成员仍然是私有的。

	public	protected	private
公有继承	public	protected	不可见
私有继承	private	private	不可见
保护继承	protected	protected	不可见

public 继承

1. 基类成员对其对象的可见性:
   公有成员可见，其他不可见。这里保护成员同于私有成员。
2. 基类成员对派生类的可见性:
   公有成员和保护成员可见，而私有成员不可见。这里保护成员同于公有成员。
3. 基类成员对派生类对象的可见性:
   公有成员可见，其他成员不可见。

private 继承

1. 基类成员对其对象的可见性:
   公有成员可见，其他成员不可见。
2. 基类成员对派生类的可见性:
   公有成员和保护成员是可见的，而私有成员是不可见的。
3. 基类成员对派生类对象的可见性:
   所有成员都是不可见的。

protected 继承

1. 基类成员对其对象的可见性:
   公有成员可见，其他不可见。这里保护成员同于私有成员。
2. 基类成员对派生类的可见性:
   公有成员和保护成员是可见的，而私有成员是不可见的。
3. 基类成员对派生类对象的可见性:
   所有成员都是不可见的。

与私有继承方式的情况相同。两者的区别仅在于对派生类的派生类成员而言，对基类成员有不同的可见性。

类的保护成员

另一种存取权限说明符：`protected`

类的保护成员

另一种存取权限说明符：`protected`

- 基类的`private`成员：可以被下列函数访问
  - 基类的成员函数
  - 基类的友元函数

类的保护成员

另一种存取权限说明符：`protected`

- 基类的`private`成员：可以被下列函数访问
  - 基类的成员函数
  - 基类的友元函数
- 基类的`public`成员：可以被下列函数访问
  - 基类的成员函数
  - 基类的友元函数
  - 派生类的成员函数
  - 派生类的友元函数
  - 其他的函数

类的保护成员

另一种存取权限说明符：`protected`

- 基类的`private`成员：可以被下列函数访问
  - 基类的成员函数
  - 基类的友元函数
- 基类的`public`成员：可以被下列函数访问
  - 基类的成员函数
  - 基类的友元函数
  - 派生类的成员函数
  - 派生类的友元函数
  - 其他的函数
- 基类的`protected`成员：可以被下列函数访问
  - 基类的成员函数
  - 基类的友元函数
  - 派生类的成员函数
  - 派生类的友元函数

保护成员

```
1  class Father {
2  private:
3    int nPrivate; //私有成员
4  public:
5    int nPublic; //公有成员
6  protected:
7    int nProtected; // 保护成员
8  };
9  class Son : public Father {
10   void accessFather() {
11     nPublic = 1;    // ok;
12     nPrivate = 1;   // wrong
13     nProtected = 1; // OK, 访问从基类继承的 protected 成员
14     Son f;
15     f.nProtected = 1;
16   }
17 };
18 int main() {
19   Father f;
20   Son s;
21   f.nPublic = 1;    // Ok
22   s.nPublic = 1;    // Ok
23   f.nProtected = 1; // error
24   f.nPrivate = 1;   // error
25   s.nProtected = 1; // error
26   s.nPrivate = 1;   // error
27   return 0;
28 }
```

派生类的构造函数

```
1  class Bug {
2  private:
3    int nLegs;
4    int nColor;
5  public:
6    int nType;
7    Bug(int legs, int color);
8    void PrintBug() {}
9  };
10 class FlyBug : public Bug { // FlyBug 是 Bug 的派生类
11   int nWings;
12 public:
13   FlyBug(int legs, int color, int wings);
14 };
15 Bug::Bug(int legs, int color) {
16   nLegs = legs;
17   nColor = color;
18 }
```

派生类的构造函数

```
1  //错误的 FlyBug 构造函数
2  FlyBug::FlyBug(int legs, int color, int wings) {
3    nLegs = legs;   // 不能访问
4    nColor = color; // 不能访问
5    nType = 1;      // ok
6    nWings = wings;
7  }
```

派生类的构造函数

```
1  //错误的 FlyBug 构造函数
2  FlyBug::FlyBug(int legs, int color, int wings) {
3    nLegs = legs;   // 不能访问
4    nColor = color; // 不能访问
5    nType = 1;      // ok
6    nWings = wings;
7  }
```

```
20 //正确的 FlyBug 构造函数:
21 FlyBug::FlyBug(int legs, int color, int wings) : Bug(legs, color) {
22   nType = 1;
23   nWings = wings;
24 }
```

派生类的构造函数、析构函数

```
26  int main() {
27    FlyBug fb(2, 3, 4);
28    fb.PrintBug();
29    fb.nType = 1;
30    fb.nLegs = 2; // error.  nLegs is private
31    return 0;
32  }
```

派生类的构造函数、析构函数

```
26  int main() {
27    FlyBug fb(2, 3, 4);
28    fb.PrintBug();
29    fb.nType = 1;
30    fb.nLegs = 2; // error.  nLegs is private
31    return 0;
32  }
```

在创建派生类的对象时，需要调用基类的构造函数：初始化派生类对象中从基类继承的成员。在执行一个派生类的构造函数之前，总是先执行基类的构造函数。

派生类的构造函数、析构函数

```
26  int main() {
27    FlyBug fb(2, 3, 4);
28    fb.PrintBug();
29    fb.nType = 1;
30    fb.nLegs = 2; // error.  nLegs is private
31    return 0;
32  }
```

在创建派生类的对象时，需要调用基类的构造函数：初始化派生类对象中从基类继承的成员。在执行一个派生类的构造函数之前，总是先执行基类的构造函数。

调用基类构造函数的两种方式

- 显式方式：在派生类的构造函数中，为基类的构造函数提供参数.
  `derived::derived(arg_derived-list) : base(arg_base-list)`
- 隐式方式：在派生类的构造函数中，省略基类构造函数时，派生类的构造函数则自动调用基类的默认构造函数.

派生类的构造函数、析构函数

```
26  int main() {
27    FlyBug fb(2, 3, 4);
28    fb.PrintBug();
29    fb.nType = 1;
30    fb.nLegs = 2; // error.  nLegs is private
31    return 0;
32  }
```

在创建派生类的对象时，需要调用基类的构造函数：初始化派生类对象中从基类继承的成员。在执行一个派生类的构造函数之前，总是先执行基类的构造函数。

调用基类构造函数的两种方式

- 显式方式：在派生类的构造函数中，为基类的构造函数提供参数.
  `derived::derived(arg_derived-list) : base(arg_base-list)`
- 隐式方式：在派生类的构造函数中，省略基类构造函数时，派生类的构造函数则自动调用基类的默认构造函数.

派生类的析构函数被执行时，执行完派生类的析构函数后，自动调用基类的析构函数。

派生类的构造函数、析构函数

```
4  class Base {
5  public:
6    int n;
7    Base(int i) : n(i) {
8      cout << "Base " << n << " constructed" << endl;
9    }
10   ~Base() {
11     cout << "Base " << n << " destructed" << endl;
12   }
13 };
14 class Derived : public Base {
15 public:
16   Derived(int i) : Base(i) {
17     cout << "Derived constructed" << endl;
18   }
19   ~Derived() {
20     cout << "Derived destructed" << endl;
21   }
22 };
23 int main() {
24   Derived Obj(3);
25 }
```

派生类的构造函数、析构函数

```
4  class Base {
5  public:
6    int n;
7    Base(int i) : n(i) {
8      cout << "Base " << n << " constructed" << endl;
9    }
10   ~Base() {
11     cout << "Base " << n << " destructed" << endl;
12   }
13 };
14 class Derived : public Base {
15 public:
16   Derived(int i) : Base(i) {
17     cout << "Derived constructed" << endl;
18   }
19   ~Derived() {
20     cout << "Derived destructed" << endl;
21   }
22 };
23 int main() {
24   Derived Obj(3);
25 }
```

输出：

```
Base 3 constructed
Derived constructed
Derived destructed
Base 3 destructed
```

包含成员对象的派生类的构造函数写法

```
1  class Bug {
2  private:
3    int nLegs;
4    int nColor;
5  public:
6    int nType;
7    Bug(int legs, int color) {}
8    void PrintBug(){}
9  };
10 class Skill {
11 public:
12   Skill(int n) {}
13 };
14 class FlyBug : public Bug {
15   int nWings;
16   Skill sk1, sk2;
17 public:
18   FlyBug(int legs, int color, int wings);
19 };
20 FlyBug::FlyBug(int legs, int color, int wings)
21   : Bug(legs, color), sk1(5), sk2(color), nWings(wings) {}
```

封闭派生类对象的构造函数执行顺序

在创建派生类的对象时:

1. 先执行基类的构造函数，用以初始化派生类对象中从基类继承的成员；
2. 再执行成员对象类的构造函数，用以初始化派生类对象中成员对象。
3. 最后执行派生类自己的构造函数

封闭派生类对象消亡时析构函数的执行顺序

在派生类对象消亡时：

1. 先执行派生类自己的析构函数
2. 再依次执行各成员对象类的析构函数
3. 最后执行基类的析构函数

析构函数的调用顺序与构造函数的调用顺序相反。

public 继承的赋值兼容规则

```
1 class Base {};
2 class Derived : public Base {};
3 Base b;
4 Derived d;
```

- 派生类的对象可以赋值给基类对象

```
1 b = d;
```

- 派生类对象可以初始化基类引用

```
1 Base &br = d;
```

- 派生类对象的地址可以赋值给基类指针

```
1 Base *pb = &d;
```

如果派生方式是 `private`或`protected`，则上述三条不可行。

基类与派生类的指针强制转换

- 公有派生的情况下, 派生类对象的指针可以直接赋值给基类指针

基类与派生类的指针强制转换

- 公有派生的情况下，派生类对象的指针可以直接赋值给基类指针

```
1  Base *ptrBase = &objDerived;
```

ptrBase指向的是一个Derived类的对象；
*ptrBase可以看作一个Base类的对象，访问它的public成员直接通过ptrBase即可

基类与派生类的指针强制转换

- 公有派生的情况下，派生类对象的指针可以直接赋值给基类指针

```
1  Base *ptrBase = &objDerived;
```

ptrBase指向的是一个Derived类的对象；
*ptrBase可以看作一个Base类的对象，访问它的public成员直接通过ptrBase即可

- 即便基类指针指向的是一个派生类的对象，也不能通过基类指针访问基类没有，而派生类中有的成员。
不能通过ptrBase访问objDerived对象中属于Derived类而不属于Base类的成员

基类与派生类的指针强制转换

- 公有派生的情况下，派生类对象的指针可以直接赋值给基类指针

```
1  Base *ptrBase = &objDerived;
```

ptrBase指向的是一个Derived类的对象；
*ptrBase可以看作一个Base类的对象，访问它的public成员直接通过ptrBase即可

- 即便基类指针指向的是一个派生类的对象，也不能通过基类指针访问基类没有，而派生类中有的成员。
不能通过ptrBase访问objDerived对象中属于Derived类而不属于Base类的成员
- 通过强制指针类型转换，可以把ptrBase转换成Derived类的指针

```
1  Base *ptrBase = &objDerived;
2  Derived *ptrDerived = (Derived *)ptrBase;
```

基类与派生类的指针强制转换

- 公有派生的情况下，派生类对象的指针可以直接赋值给基类指针

```
1  Base *ptrBase = &objDerived;
```

ptrBase指向的是一个Derived类的对象；
*ptrBase可以看作一个Base类的对象，访问它的public成员直接通过ptrBase即可

- 即便基类指针指向的是一个派生类的对象，也不能通过基类指针访问基类没有，而派生类中有的成员。
不能通过ptrBase访问objDerived对象中属于Derived类而不属于Base类的成员
- 通过强制指针类型转换，可以把ptrBase转换成Derived类的指针

```
1  Base *ptrBase = &objDerived;
2  Derived *ptrDerived = (Derived *)ptrBase;
```

程序员要保证ptrBase指向的是一个Derived类的对象，否则很容易会出错。

基类与派生类的指针强制转换

```
4  class Base {
5  protected:
6    int n;
7  public:
8    Base(int i) : n(i) {
9      cout << "Base " << n << " constructed" << endl;
10   }
11   ~Base() {
12     cout << "Base " << n << " destructed" << endl;
13   }
14   void print() {
15     cout << "Base:n=" << n << endl;
16   }
17 };
18 class Derived : public Base {
19 public:
20   int v;
21   Derived(int i) : Base(i), v(2 * i) {
22     cout << "Derived constructed" << endl;
23   }
24   ~Derived() {
25     cout << "Derived destructed" << endl;
26   }
27   void func() {}
28   void print() {
29     cout << "Derived:v=" << v << endl;
30     cout << "Derived:n=" << n << endl;
31   }
32 };
```

基类与派生类的指针强制转换

```
33
34  int main() {
35    Base objBase(5);
36    Derived objDerived(3);
37    Base *pBase = &objDerived;
38    // pBase->Func(); //err;Base 类没有 Func() 成员函数
39    // pBase->v = 5; //err; Base 类没有 v 成员变量
40    pBase->print();
41    // Derived * pDerived = & objBase; //error
42    Derived *pDerived = (Derived *)(&objBase);
43    pDerived->print(); //慎用，可能出现不可预期的错误
44    pDerived->v = 128; //往别人的空间里写入数据，会有问题
45    objDerived.print();
46    return 0;
47  }
```

基类与派生类的指针强制转换

```
33
34  int main() {
35    Base objBase(5);
36    Derived objDerived(3);
37    Base *pBase = &objDerived;
38    // pBase->Func(); //err;Base 类没有 Func() 成员函数
39    // pBase->v = 5; //err; Base 类没有 v 成员变量
40    pBase->print();
41    // Derived * pDerived = & objBase; //error
42    Derived *pDerived = (Derived *)(&objBase);
43    pDerived->print(); //慎用，可能出现不可预期的错误
44    pDerived->v = 128; //往别人的空间里写入数据，会有问题
45    objDerived.print();
46    return 0;
47  }
```

输出结果:

```
Base 5 constructed
Base 3 constructed
Derived constructed
Base:n=3
Derived:v=-1276156084
Derived:n=5
Derived:v=6
Derived:n=3
Derived destructed
Base 3 destructed
Base 5 destructed
```

//pDerived->v 位于别人的空间里

直接基类与间接基类

类 A 派生类 B，类 B 派生类 C，类 C 派生类 D，……

- 类 A 是类 B 的直接基类
- 类 B 是类 C 的直接基类，类 A 是类 C 的间接基类
- 类 C 是类 D 的直接基类，类 A、B 是类 D 的间接基类

在声明派生类时，只需要列出它的直接基类

- 派生类沿着类的层次自动向上继承它的间接基类
- 派生类的成员包括
  - 派生类自己定义的成员
  - 直接基类中的所有成员
  - 所有间接基类的全部成员

直接基类与间接基类

```
4  class Base {
5  public:
6    int n;
7    Base(int i) : n(i) {
8      cout << "Base " << n << " constructed" << endl;
9    }
10   ~Base() {
11     cout << "Base " << n << " destructed" << endl;
12   }
13 };
14 class Derived : public Base {
15 public:
16   Derived(int i) : Base(i) {
17     cout << "Derived constructed" << endl;
18   }
19   ~Derived() {
20     cout << "Derived destructed" << endl;
21   }
22 };
23 class MoreDerived : public Derived {
24 public:
25   MoreDerived() : Derived(4) {
26     cout << "More Derived constructed" << endl;
27   }
28   ~MoreDerived() {
29     cout << "More Derived destructed" << endl;
30   }
31 };
32 int main() { MoreDerived Obj; }
```

直接基类与间接基类

```
4  class Base {
5  public:
6    int n;
7    Base(int i) : n(i) {
8      cout << "Base " << n << " constructed" << endl;
9    }
10   ~Base() {
11     cout << "Base " << n << " destructed" << endl;
12   }
13 };
14 class Derived : public Base {
15 public:
16   Derived(int i) : Base(i) {
17     cout << "Derived constructed" << endl;
18   }
19   ~Derived() {
20     cout << "Derived destructed" << endl;
21   }
22 };
23 class MoreDerived : public Derived {
24 public:
25   MoreDerived() : Derived(4) {
26     cout << "More Derived constructed" << endl;
27   }
28   ~MoreDerived() {
29     cout << "More Derived destructed" << endl;
30   }
31 };
32 int main() { MoreDerived Obj; }
```

输出结果：

```
Base 4 constructed
Derived constructed
More Derived constructed
More Derived destructed
Derived destructed
Base 4 destructed
```

单选题

以下说法正确的是：

- A 派生类对象生成时，派生类的构造函数先于基类的构造函数执行
- B 派生类对象消亡时，基类的析构函数先于派生类的析构函数执行
- C 如果基类有无参构造函数，则派生类的构造函数就可以不带初始化列表
- D 在派生类的构造函数中部可以访问基类的成员变量

单选题

以下说法正确的是：

A 派生类对象生成时，派生类的构造函数先于基类的构造函数执行

B 派生类对象消亡时，基类的析构函数先于派生类的析构函数执行

C 如果基类有无参构造函数，则派生类的构造函数就可以不带初始化列表

D 在派生类的构造函数中部可以访问基类的成员变量

答案：C