



中國人民大學
RENMIN UNIVERSITY OF CHINA

程序设计荣誉课程

4 继承

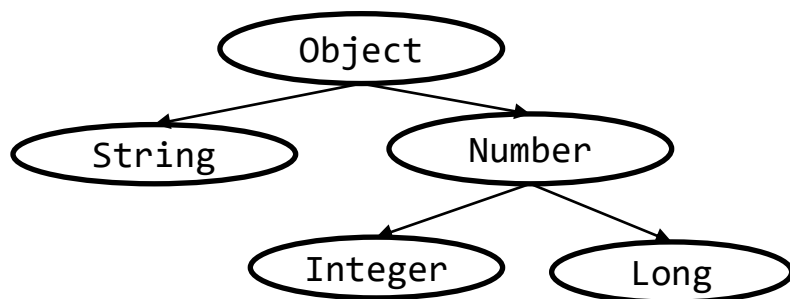
授课教师：孙亚辉

本章内容

1. 继承
2. 派生类构造函数
3. 继承与成员函数
4. 继承与成员变量

1. 继承

- **继承**（inheritance）是面向对象程序设计的一个基本特征。使用继承，可以定义相似的类型并对其相似关系建模。
- 通过继承联系在一起的类构成一种**层次关系**
 - 通常在层次关系的根部有一个**基类**（base class）
 - 其他类则直接或间接地从基类继承而来，这些继承得到的类称为**派生类**（derived class）
 - 基类负责定义在层次关系中所有类共同拥有的成员
 - 每个派生类定义各自特有的成员



继承入门

- 派生类必须使用**类派生列表**（class derivation list）明确指出它是从**哪个**（哪些）基类继承而来的。
- 继承的基本语法如下
 - **class 派生类类名 : 访问说明符 基类1类名, 访问说明符 基类2类名, ...**
 - 在派生类类名之后使用冒号“:”，然后是基类列表，每个基类可以使用一些修饰符，最基础的是访问限制符**public**、**private**、**protected**，亦可省略访问说明符，使用默认模式。
 - 基类名前亦可用关键词**virtual**，待后续介绍（虚继承）。

- 一个派生类可以从多个基类中派生

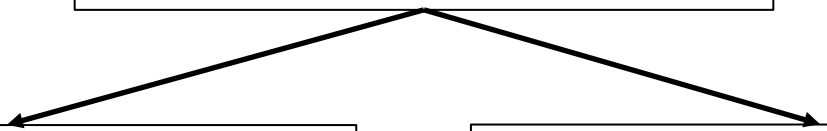
```
1  #include <iostream>
2  #include <string.h>
3  using namespace std;
4
5  class B1
6  {
7  public:
8      void output()
9      {
10         cout << "call the class B1" << endl;
11     }
12 };
13 class B2
14 {
15 public:
16     void outputB2()
17     {
18         cout << "call the class B2" << endl;
19     }
20 };
```

```
22 class A : public B1, private B2
23 {
24 public:
25     void show()
26     {
27         outputB2();
28     }
29 };
30
31 int main()
32 {
33     A a;
34
35     a.output();
36
37     //a.outputB2(); // 报错
38
39     a.show();
40
41     return 0;
42 }
```

一个基类可以有多个派生类

- 以Person为基类，Teacher与Student为派生类
 - Teacher类相较于Person增加了tid（职工号）、salary（工资）、affil（所在学院）
 - Student类相较于Person增加了sid（学号）、dorm（宿舍）

```
1. // 基类
2. class Person {
3.     int age;
4.     bool sex; // true for male
5.     char name[21];
6. };
```

A diagram with two arrows pointing downwards from the bottom of the Person class box to the top of the Teacher and Student class boxes, indicating inheritance.

```
1. // 派生类
2. class Teacher : Person {
3.     int tid;
4.     int salary;
5.     char affil[16];
6. };
```

```
1. // 派生类
2. class Student : Person {
3.     int sid;
4.     char dorm[12];
5. };
```

派生类的内存布局

- 以基类Person、派生类Teacher和main函数为例查看的内存布局

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  // 基类
6  class Person
7  {
8  public:
9      int age;          // 4 byte
10     bool sex;          // 1 byte
11     char name[21];    // 21 byte
12 };
13
14 // 派生类
15 class Teacher : public Person
16 {
17 public:
18     int tid;
19     int salary;
20     char affil[16];
21 };
```

```
23 int main()
24 {
25     Person person;
26
27     Teacher teacher;
28     int tid = teacher.tid;
29     int salary = teacher.salary;
30     char *affil = teacher.affil;
31
32     teacher.age = person.age;
33     teacher.sex = person.sex;
34     strcpy(teacher.name, person.name);
35
36     cout << "sizeof(person.age): " << sizeof(person.age) << endl;
37     cout << "sizeof(person.sex): " << sizeof(person.sex) << endl;
38     cout << "sizeof(person): " << sizeof(person) << endl;
39
40     cout << "sizeof(teacher): " << sizeof(teacher) << endl;
41 }
```

- 以基类Person、派生类Teacher和main函数为例查看的内存布局
 - Person类的大小是28字节（4字节对齐），数据成员占据内存空间26字节
 - 当派生类Teacher能访问到基类Person的所有数据成员时，Teacher的大小（sizeof(Teacher)）是52字节，而Teacher本身定义的数据成员占据空间24字节
 - 可以推断，Teacher类的大小是Teacher本身数据成员大小+基类Person的大小。

```
1. // 基类
2. class Person {
3. public:
4.     int age;
5.     bool sex;
6.     char name[21];
7. };
8. // 派生类
9. class Teacher : public Person {
10. public:
11.     int tid;
12.     int salary;
13.     char affil[16];
14. };
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
sizeof(person.age): 4
sizeof(person.sex): 1
sizeof(person): 28
sizeof(teacher): 52
```

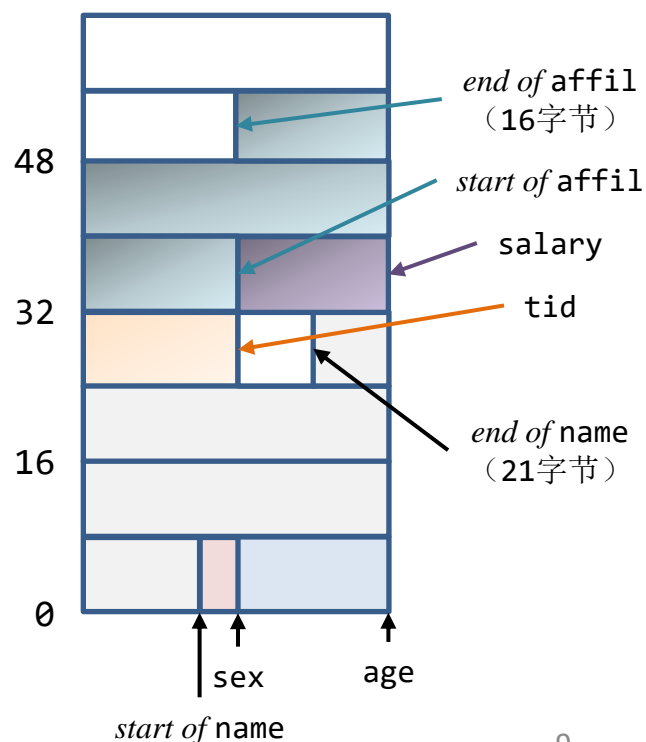

Teacher类对象的内存空间如右图所示

- 对于派生类的类对象而言，基类的数据成员占据该类对象内存空间的起始部分，而派生类本身的数据成员则占据靠后的部分。
- 这样布局的好处是，派生类对象向基类类型的转换过程中，无需额外的地址偏移操作。

– 这里的类型转换可以是如下几种：

- ① `Person pn = teacher`（如通过编译器生成的拷贝构造函数）
- ② `Person& rpn = teacher`（获得基类的引用形式）
- ③ `Person* ppn = &teacher`（获得基类的指针形式）

```
1. // 基类
2. class Person {
3. public:
4.     int age;
5.     bool sex;
6.     char name[21];
7. };
8. // 派生类
9. class Teacher : public Person {
10. public:
11.     int tid;
12.     int salary;
13.     char affil[16];
14. };
```



注意： 可以从派生类转换到基类，但是不能从基类转换到派生类。

（因为基类对象没法初始化派生类可能的新数据成员）

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  // 基类1
6  class Person
7  {
8  };
9
10 // 派生类
11 class Teacher : public Person
12 {
13 };
14
15 int main()
16 {
17     Person pn;
18     Teacher tc;
19
20     Person xpn = tc;
21     Person &rpn = tc;
22     Person *ppn = &tc;
23
24     Teacher xtc = pn;
25     Teacher &rtc = pn;
26     Teacher *ptc = &pn;
27
28     return 0;
29 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
```

```
test.cpp: In function 'int main()':
```

```
test.cpp:24:19: error: conversion from 'Person' to non-scalar type 'Teacher' requested
    24 |         Teacher xtc = pn;
        |                        ^~
```

```
test.cpp:25:20: error: invalid initialization of reference of type 'Teacher&' from expression of type 'Person'
    25 |         Teacher &rtc = pn;
        |                        ^~
```

```
test.cpp:26:20: error: invalid conversion from 'Person*' to 'Teacher*' [-fpermissive]
    26 |         Teacher *ptc = &pn;
        |                        ^~~
        |                        |
        |                        Person*
```

通过派生类的类对象调用基类函数

- 下例展示了通过派生类Teacher的类对象调用基类Person中的introduce成员函数的代码
 - Teacher并未重新实现（覆盖）introduce函数

```
5 // 基类
6 class Person
7 {
8     public:
9         void introduce(){}
10 };
11 // 派生类
12 class Teacher : public Person
13 {
14 };
15
16 int main()
17 {
18     Person person;
19     Teacher teacher;
20     teacher.introduce();
21 }
```

```
5 // 基类
6 class Person
7 {
8     public:
9         void introduce(){}
10 };
11 // 派生类
12 class Teacher : private Person
13 {
14 };
15
16 int main()
17 {
18     Person person;
19     Teacher teacher;
20     teacher.introduce();
21 }
```

```
5 // 基类
6 class Person
7 {
8     public:
9         void introduce(){}
10 };
11 // 派生类
12 class Teacher : protected Person
13 {
14 };
15
16 int main()
17 {
18     Person person;
19     Teacher teacher;
20     teacher.introduce();
21 }
```

- 派生类的对象可访问具有public继承访问权限的基类的public成员函数。

```
5 // 基类
6 class Person
7 {
8 public:
9     void introduce(){}
10 };
11 // 派生类
12 class Teacher : Person
13 {
14 };
15
16 int main()
17 {
18     Person person;
19     Teacher teacher;
20     teacher.introduce();
21 }
```

```
5 // 基类
6 class Person
7 {
8 private:
9     void introduce1() {}
10 protected:
11     void introduce2() {}
12 public:
13     void introduce3() {}
14 };
15 // 派生类
16 class Teacher : public Person
17 {
18 };
19
20 int main()
21 {
22     Person person;
23     Teacher teacher;
24     teacher.introduce1();
25     teacher.introduce2();
26     teacher.introduce3();
27 }
```

继承模式与访问模式

- 考虑这种情况：继承基类Person时没有使用任何访问限制符，检查默认继承模式时在派生类里面对基类成员的访问权限
 - 说明：类派生列表中没有明确指明访问模式时，在派生类里面可以访问从基类继承而来的非私有成员。

```
5 // 基类
6 class Person
7 {
8     public:
9         int age;
10        bool sex;
11        char name[21];
12    protected:
13        int protected_x;
14
15    private:
16        int private_x;
17 };
18
19 // 派生类
20 class Teacher : Person
21 {
22     public:
23         char affil[16] = {*name};
24         int tid = protected_x;
25         int salary = private_x;
26 };
```

（刚才讲的是通过派生类对象访问基类成员，这里是在派生类里面访问基类成员）

- 然后检查类派生列表使用各种访问模式时，派生类对基类成员的访问权限

```
5 // 基类
6 class Person
7 {
8 public:
9     int age;
10    bool sex;
11    char name[21];
12 protected:
13    int protected_x;
14
15 private:
16    int private_x;
17 };
18
19 // 派生类
20 class Teacher : public Person
21 {
22 public:
23     char affil[16] = {*name};
24     int tid = protected_x;
25     int salary = private_x;
26 };
```

```
5 // 基类
6 class Person
7 {
8 public:
9     int age;
10    bool sex;
11    char name[21];
12 protected:
13    int protected_x;
14
15 private:
16    int private_x;
17 };
18
19 // 派生类
20 class Teacher : private Person
21 {
22 public:
23     char affil[16] = {*name};
24     int tid = protected_x;
25     int salary = private_x;
26 };
```

```
5 // 基类
6 class Person
7 {
8 public:
9     int age;
10    bool sex;
11    char name[21];
12 protected:
13    int protected_x;
14
15 private:
16    int private_x;
17 };
18
19 // 派生类
20 class Teacher : protected Person
21 {
22 public:
23     char affil[16] = {*name};
24     int tid = protected_x;
25     int salary = private_x;
26 };
```

- 类派生列表中的访问说明符使用private、protected、public时，所得结果与上页一致，即在派生类里面可以访问基类中的protected、public成员，不能访问private成员。

- protected有什么用？
 - 将Person中的成员声明为受保护的（protected），在派生类里面可以访问，但其他外部函数中无法访问（对于外部函数而言，protected相当于private）。
 - **protected与private的区别**：在派生类里面不可以访问基类的private成员，但可以访问基类的protected成员。

```
5 // 基类
6 class Person
7 {
8     protected:
9         int protected_x;
10 };
11
12 // 派生类
13 class Teacher : protected Person
14 {
15     int tid = protected_x;
16 };
17
18 int main()
19 {
20     Person person;
21     Teacher teacher;
22
23     int tid = person.protected_x;
24     int tid = teacher.protected_x;
25 }
```

多重派生的访问控制

- 基类的访问控制符对访问基类中的数据有什么影响

```
1. // 基类
2. class Person {
3.     int age;
4.     bool sex;
5.     char name[21];
6. };
```

```
1. // 派生类
2. class Teacher : <?> Person {
3.     int tid;
4.     int salary;
5.     char affil[16];
6. };
```

```
1. // 派生类的派生类
2. class Professor : Teacher {
3.     void func() {
4.         // 是否能够访问age
5.     }
6. };
```

思考：在Professor类中与main函数中，分别在什么情况下能够访问Person类中的成员age？

class Teacher : public Person:

- Teacher中可访问Person中的public和protected成员
- Professor中可访问Person中的public和protected成员
- main中可通过Teacher访问Person中的public成员

```
5 // 基类
6 class Person
7 {
8     int private_age;
9 public:
10     int public_age;
11 protected:
12     int protected_age;
13 };
14
15 // 派生类
16 class Teacher : public Person
17 {
18     int x = private_age;
19     int y = public_age;
20     int z = protected_age;
21 };
```

```
23 // 派生类
24 class Professor : Teacher
25 {
26     int x2 = private_age;
27     int y2 = public_age;
28     int z2 = protected_age;
29 };
30
31 int main()
32 {
33     Teacher Teacher;
34     Professor professor;
35
36     Teacher.private_age;
37     Teacher.public_age;
38     Teacher.protected_age;
39
40     professor.private_age;
41     professor.public_age;
42     professor.protected_age;
43 }
```

class Teacher : protected Person

- Teacher中可访问Person中的public和protected成员；但是，Person中的public成员在Teacher类对象中相当于变成了protected访问权限
- Professor中可访问Person中的public和protected成员
- main中不可通过Teacher访问Person中的public成员

访问说明符将基类中比当前访问说明符可见性更强的访问控制权限降低到访问说明符的可见性。

```
5 // 基类
6 class Person
7 {
8     int private_age;
9 public:
10    int public_age;
11 protected:
12    int protected_age;
13 };
14
15 // 派生类
16 class Teacher : protected Person
17 {
18     int x = private_age;
19     int y = public_age;
20     int z = protected_age;
21 };
```

```
23 // 派生类
24 class Professor : Teacher
25 {
26     int x2 = private_age;
27     int y2 = public_age;
28     int z2 = protected_age;
29 };
30
31 int main()
32 {
33     Teacher Teacher;
34     Professor professor;
35
36     Teacher.private_age;
37     Teacher.public_age;
38     Teacher.protected_age;
39
40     professor.private_age;
41     professor.public_age;
42     professor.protected_age;
43 }
```

class Teacher : <无>/private Person (默认的继承访问权限是private)

- Teacher中可访问Person中的public和protected成员；但是，Person中的public和protected成员在Teacher类对象中相当于变成了private访问权限
- Professor中不可访问Person中的任何成员
- main中不可通过Teacher访问Person中的public成员

```
5 // 基类
6 class Person
7 {
8     int private_age;
9 public:
10    int public_age;
11 protected:
12    int protected_age;
13 };
14
15 // 派生类
16 class Teacher : Person
17 {
18     int x = private_age;
19     int y = public_age;
20     int z = protected_age;
21 };
```

```
23 // 派生类
24 class Professor : Teacher
25 {
26     int x2 = private_age;
27     int y2 = public_age;
28     int z2 = protected_age;
29 };
30
31 int main()
32 {
33     Teacher Teacher;
34     Professor professor;
35
36     Teacher.private_age;
37     Teacher.public_age;
38     Teacher.protected_age;
39
40     professor.private_age;
41     professor.public_age;
42     professor.protected_age;
43 }
```

访问说明符对类型转换的影响

- 对于如下类型转换（包含通过派生类进行基类的拷贝初始化）
 - `Person pn = teacher;`
 - `Person& rpn = teacher;`
 - `Person* ppn = &teacher;`
- 对于不同的访问说明符（`class Teacher : 访问说明符 Person`）：
 - `public`: 上述三种均可正常编译执行
 - 其他: 上述三种均不能通过编译

```

5 // 基类
6 class Person
7 {
8     int private_age;
9 public:
10    int public_age;
11 protected:
12    int protected_age;
13 };
14 // 派生类
15 class Teacher : public Person
16 {
17 };
18
19 int main()
20 {
21     Teacher teacher;
22     Person pn = teacher;
23     Person &rpn = teacher;
24     Person *ppn = &teacher;
25 }

```

```

5 // 基类
6 class Person
7 {
8     int private_age;
9 public:
10    int public_age;
11 protected:
12    int protected_age;
13 };
14 // 派生类
15 class Teacher : private Person
16 {
17 };
18
19 int main()
20 {
21     Teacher teacher;
22     Person pn = teacher;
23     Person &rpn = teacher;
24     Person *ppn = &teacher;
25 }

```

```

5 // 基类
6 class Person
7 {
8     int private_age;
9 public:
10    int public_age;
11 protected:
12    int protected_age;
13 };
14 // 派生类
15 class Teacher : protected Person
16 {
17 };
18
19 int main()
20 {
21     Teacher teacher;
22     Person pn = teacher;
23     Person &rpn = teacher;
24     Person *ppn = &teacher;
25 }

```

对于代码中的某个给定节点来说，如果基类的公有成员是可访问的，则派生类向基类的类型转换是可行的；反之则不行。

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  // 基类
6  class Person
7  {
8      int private_age;
9  public:
10     int public_age;
11  protected:
12     int protected_age;
13 };
14 // 派生类
15 class Teacher : protected Person
16 {
17 };
18
19 int main()
20 {
21     Teacher teacher;
22     Person pn = teacher;
23     Person &rpn = teacher;
24     Person *ppn = &teacher;
25 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:22:17: error: 'Person' is an inaccessible base of 'Teacher'
    22 |         Person pn = teacher;
        |                        ^~~~~~
test.cpp:23:19: error: 'Person' is an inaccessible base of 'Teacher'
    23 |         Person &rpn = teacher;
        |                        ^~~~~~
test.cpp:24:20: error: 'Person' is an inaccessible base of 'Teacher'
    24 |         Person *ppn = &teacher;
        |                        ^~~~~~

```

改变个别成员的可访问性

- 有时我们需要改变派生类继承的某个名字访问级别，通过使用 `using` 声明，可将基类中 `protected/public` 成员在派生类中转为 `protected/public/private` 成员。如： `protected` to `public`:

```
5 // 基类
6 class Person
7 {
8     int private_age;
9     public:
10    int public_age;
11    protected:
12    int protected_age;
13 };
14 // 派生类
15 class Teacher : public Person
16 {
17     public:
18     //using Person::protected_age;
19 };
20
21 int main()
22 {
23     Teacher teacher;
24     teacher.public_age;
25     teacher.protected_age;
26 }
```

```
5 // 基类
6 class Person
7 {
8     int private_age;
9     public:
10    int public_age;
11    protected:
12    int protected_age;
13 };
14 // 派生类
15 class Teacher : public Person
16 {
17     public:
18     using Person::protected_age;
19 };
20
21 int main()
22 {
23     Teacher teacher;
24     teacher.public_age;
25     teacher.protected_age;
26 }
```

- public to protected:

```
5 // 基类
6 class Person
7 {
8     int private_age;
9 public:
10    int public_age;
11 protected:
12    int protected_age;
13 };
14 // 派生类
15 class Teacher : public Person
16 {
17     public:
18     using Person::protected_age;
19 };
20
21 int main()
22 {
23     Teacher teacher;
24     teacher.public_age;
25     teacher.protected_age;
26 }
```

```
5 // 基类
6 class Person
7 {
8     int private_age;
9 public:
10    int public_age;
11 protected:
12    int protected_age;
13 };
14 // 派生类
15 class Teacher : public Person
16 {
17     public:
18     using Person::protected_age;
19     protected:
20     using Person::public_age;
21 };
22
23 int main()
24 {
25     Teacher teacher;
26     teacher.public_age;
27     teacher.protected_age;
28 }
```


派生类只能为那些它可以访问的名字提供using声明

```
5 // 基类
6 class Person
7 {
8     int private_age;
9 public:
10    int public_age;
11 protected:
12    int protected_age;
13 };
14 // 派生类
15 class Teacher : public Person
16 {
17     public:
18     using Person::protected_age;
19     using Person::private_age;
20 };
21
22 int main()
23 {
24     Teacher teacher;
25     teacher.public_age;
26     teacher.protected_age;
27 }
```

回顾：在类的作用域中使用using声明只能指向类的基类成员

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4  using std::cout;
5
6  // 基类
7  class Person
8  {
9  public:
10     int age;        // 4 byte
11     bool sex;       // true for
12     char name[21];  // 21 byte
13 };
14
15 // 派生类
16 class Teacher : private Person
17 {
18 public:
19     int tid;
20     int salary;
21     char affil[16];
22     using Person::age; // 改变
23     using std::cout;
24 };
25
26
27 int main()
28 {
29
30     Teacher teacher;
31 }
```

yahui@Yahui:/media/sf_VM\$ g++ test.cpp

test.cpp:23:16: error: using-declaration for non-member at class scope

```
23 |     using std::cout;
    |                   ^~~~
```

本章内容

1. 继承
2. 派生类构造函数
3. 继承与成员函数
4. 继承与成员变量

2. 派生类构造函数

上述代码中基类和派生类都没有明确地定义构造函数，都是使用默认的构造函数。使用`=delete`禁止编译器生成默认构造函数，如下代码编译器报错

- 由于Teacher中没有明确地初始化其基类的数据成员，编译器试图使用基类的默认构造函数进行初始化（基类的数据成员），由于默认构造函数不存在，因此编译出错

```
5 // 基类
6 class Person
7 {
8     Person() = delete;
9 };
10 // 派生类
11 class Teacher : private Person
12 {
13 };
14
15
16 int main()
17 {
18     Teacher teacher;
19 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:18:13: error: use of deleted function 'Teacher::Teacher()'
   18 |         Teacher teacher;
      |         ^~~~~~
test.cpp:11:7: note: 'Teacher::Teacher()' is implicitly deleted because the default definition would be ill-formed:
   11 | class Teacher : private Person
      |         ^~~~~~
test.cpp:11:7: error: use of deleted function 'Person::Person()'
test.cpp:8:5: note: declared here
    8 |     Person() = delete;
      |     ^~~~~~
```

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  // 基类1
6  class Person
7  {
8  public:
9      int age;
10     bool sex;
11     char name[21];
12     //Person() = delete; // 使用=delete可禁止编译器生成默认构造函数 编译
13     Person(){cout << "初始化派生类的对象时要调用基类的构造函数" << endl;}
14 };
15
16 // 派生类
17 class Teacher : private Person
18 {
19 };
20
21 int main()
22 {
23     Teacher teacher;
24 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
初始化派生类的对象时要调用基类的构造函数

```

初始化派生类的对象时要调用基类的构造函数。

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  // 基类1
6  class Person
7  {
8  public:
9      int age;
10     bool sex;
11     char name[21];
12     Person() { cout << "1" << endl; }
13 };
14
15 // 派生类
16 class Teacher : private Person
17 {
18 public:
19     Teacher() { cout << "2" << endl; }
20 };
21
22 int main()
23 {
24     Teacher teacher;
25 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
1
2

```

初始化派生类的对象时，先调用基类的构造函数，再调用派生类的构造函数。

```

5 // 基类
6 class Person
7 {
8 public:
9     int age;
10    bool sex;
11    char name[21];
12
13    Person(int x, bool y, const char *z)
14    {
15        age = x;
16        sex = y;
17        strcpy(name, z);
18    }
19 };
20 // 派生类
21 class Teacher : public Person
22 {
23 public:
24     int salary;
25     //using Person::Person;
26 };
27
28 int main()
29 {
30     Teacher teacher;
31 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:30:13: error: use of deleted function 'Teacher::Teacher()'
   30 |     Teacher teacher;
      |           ^~~~~~
test.cpp:21:7: note: 'Teacher::Teacher()' is implicitly deleted because the default definition would be ill-formed:
   21 | class Teacher : public Person
      |       ^~~~~~
test.cpp:21:7: error: no matching function for call to 'Person::Person()'
test.cpp:13:5: note: candidate: 'Person::Person(int, bool, const char*)'
   13 |     Person(int x, bool y, const char *z)
      |     ^~~~~~
test.cpp:13:5: note: candidate expects 3 arguments, 0 provided
test.cpp:6:7: note: candidate: 'constexpr Person::Person(const Person&)'
    6 | class Person
      |       ^~~~~~
test.cpp:6:7: note: candidate expects 1 argument, 0 provided
test.cpp:6:7: note: candidate: 'constexpr Person::Person(Person&&)'
test.cpp:6:7: note: candidate expects 1 argument, 0 provided

```

如何使用基类的自定义的构造函数呢？

默认情况，基类的构造函数不被继承。

```
5 // 基类
6 class Person
7 {
8 public:
9     int age;
10    bool sex;
11    char name[21];
12
13    Person(int x, bool y, const char *z)
14    {
15        age = x;
16        sex = y;
17        strcpy(name, z);
18    }
19 };
20 // 派生类
21 class Teacher : public Person
22 {
23 public:
24     int salary;
25     //using Person::Person;
26 };
27
28 int main()
29 {
30     Teacher teacher(1, true, "a");
31 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:30:33: error: no matching function for call to 'Teacher::Teacher(int, bool, const char [2])'
   30 |     Teacher teacher(1, true, "a");
      |                               ^
test.cpp:21:7: note: candidate: 'constexpr Teacher::Teacher(const Teacher&)'
   21 | class Teacher : public Person
      |      ^~~~~~
test.cpp:21:7: note: candidate expects 1 argument, 3 provided
test.cpp:21:7: note: candidate: 'constexpr Teacher::Teacher(Teacher&&)'
test.cpp:21:7: note: candidate expects 1 argument, 3 provided
```


解决方案1——定义派生类的构造函数:

- 派生类需要直接调用基类的构造函数来初始化它的基类部分。
- 每个类控制它自己的成员初始化过程。

```
5 // 基类
6 class Person
7 {
8 public:
9     int age;
10    bool sex;
11    char name[21];
12
13    Person(int x, bool y, const char *z)
14    {
15        age = x;
16        sex = y;
17        strcpy(name, z);
18    }
19 };
20 // 派生类
21 class Teacher : public Person
22 {
23 public:
24     int salary;
25     Teacher(int m, int x2, bool y2, const char *z2) :
26         salary(m), Person(x2, y2, z2) {}
27 };
28
29 int main()
30 {
31     Teacher teacher(1, 1, true, "s");
32 }
```

解决方案2——派生类能够通过using声明语句重用基类的构造函数
(注意: 通过using声明语句重用基类的构造函数时不需要指明基类的构造函数的参数列表)

```
5 // 基类
6 class Person
7 {
8 public:
9     int age;
10    bool sex;
11    char name[21];
12
13    Person(int x, bool y, const char *z)
14    {
15        age = x;
16        sex = y;
17        strcpy(name, z);
18    }
19 };
20 // 派生类
21 class Teacher : public Person
22 {
23 public:
24     int salary;
25     using Person::Person;
26 };
27
28 int main()
29 {
30     Teacher teacher(1, true, "a");
31 }
```

派生类能通过using声明语句重用基类的带默认参数的构造函数。

```
5  // 基类
6  class Person
7  {
8  public:
9      int age;
10     bool sex;
11     char name[21];
12     Person(int x, bool y, const char *z = "s")
13     {
14         age = x;
15         sex = y;
16         strcpy(name, z);
17     }
18 };
19 // 派生类
20 class Teacher : public Person
21 {
22 public:
23     int salary;
24     using Person::Person;
25 };
26
27 int main()
28 {
29     Teacher teacher2(1, true);
30     cout << teacher2.name << endl;
31 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
s
```

派生类能够通过using声明语句重用基类的默认构造函数。

```
5 // 基类
6 class Person
7 {
8 public:
9     int age;
10    bool sex;
11    char name[21];
12 };
13 // 派生类
14 class Teacher : public Person
15 {
16 public:
17     int salary;
18     //using Person::Person;
19     Teacher(int x){x = salary;}
20 };
21
22 int main()
23 {
24     Teacher teacher;
25 }
```



```
5 // 基类
6 class Person
7 {
8 public:
9     int age;
10    bool sex;
11    char name[21];
12 };
13 // 派生类
14 class Teacher : public Person
15 {
16 public:
17     int salary;
18     using Person::Person;
19     Teacher(int x){x = salary;}
20 };
21
22 int main()
23 {
24     Teacher teacher;
25 }
```

本章内容

1. 继承
2. 派生类构造函数
3. 继承与成员函数
4. 继承与成员变量

3. 继承与成员函数

- 当派生类未覆盖实现基类中的方法时，通过派生类对象调用基类中可访问的普通成员函数时，是直接调用对应函数。

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
Person introduce()
```

```
5 // 基类1
6 class Person
7 {
8 public:
9     void introduce()
10    {
11        cout << "Person introduce()" << endl;
12    }
13    void Person_someFunc()
14    {
15        introduce();
16    }
17 };
18
19 // 派生类
20 class Teacher : public Person
21 {
22 };
23
24 int main()
25 {
26     Teacher tc;
27     tc.introduce();
28 }
```

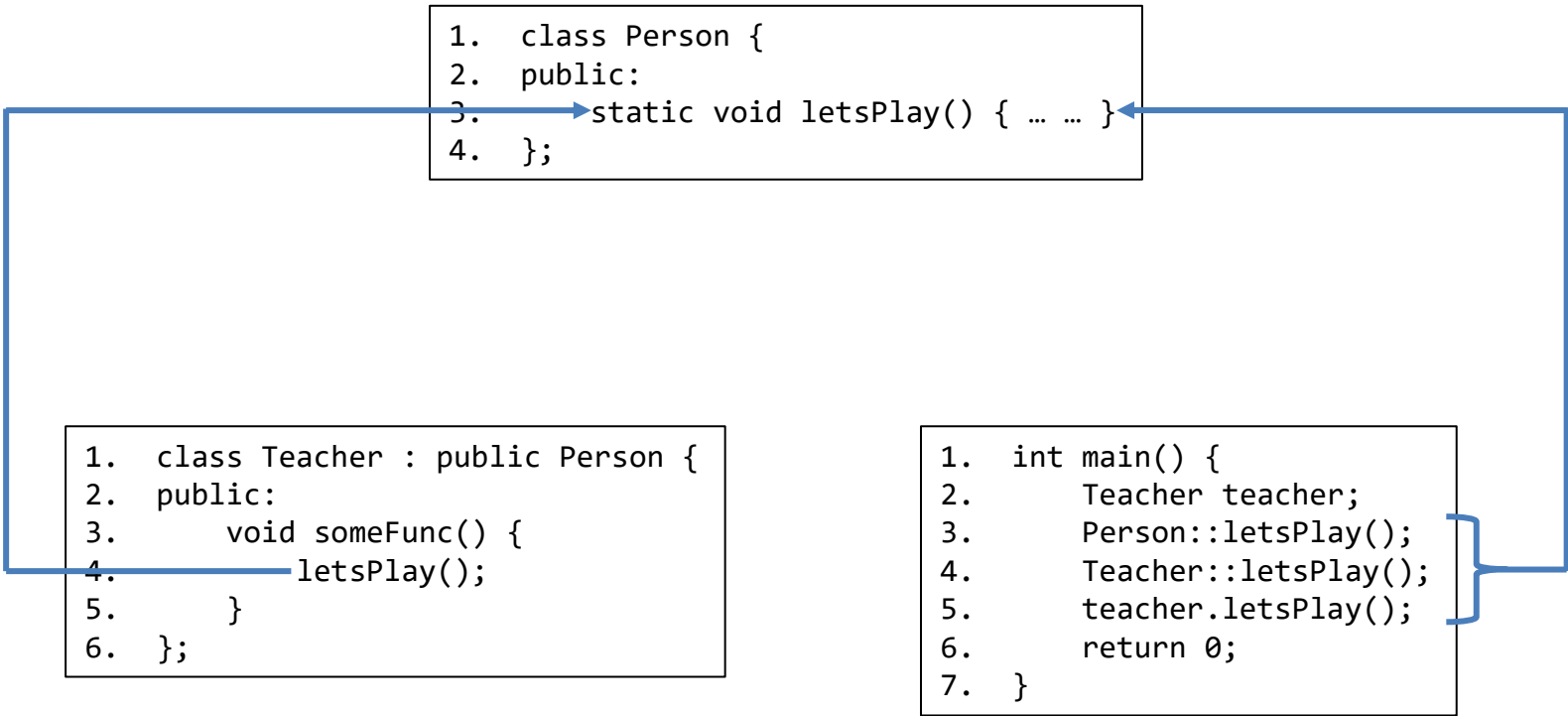
继承与成员函数

- 本小节将介绍另外几种情况
 - 调用基类中静态函数（派生类未重新实现）
 - 调用基类中静态函数（派生类重新实现——覆盖）
 - 派生类覆盖了基类中的普通成员函数

继承与静态成员函数

- 假设基类 `Person` 定义了一个 `public` 的静态成员函数 `letsPlay()`，且 **派生类 `Teacher` 没有重新实现该函数**时，如下的访问方式都是正确的
 - `Person::letsPlay();` // `Person` 定义了 `letsPlay`
 - `Teacher::letsPlay();` // `Teacher` 继承了 `letsPlay`
 - `teacher.letsPlay();` // 通过派生类的类对象访问
 - `void Teacher::someFunc() { letsPlay(); }`
- 此时，整个继承体系中只存在该静态成员 `letsPlay` 的唯一定义


```
1. class Person {  
2. public:  
3.     static void letsPlay() { ... ... }  
4. };
```



```
1. class Teacher : public Person {  
2. public:  
3.     void someFunc() {  
4.         letsPlay();  
5.     }  
6. };
```

```
1. int main() {  
2.     Teacher teacher;  
3.     Person::letsPlay();  
4.     Teacher::letsPlay();  
5.     teacher.letsPlay();  
6.     return 0;  
7. }
```

```

5 // 基类1
6 class Person
7 {
8 public:
9     static void letsPlay()
10    {
11        cout << "static Person letsPlay()" << endl;
12    }
13 };
14 // 派生类
15 class Teacher : public Person
16 {
17     void someFunc() { letsPlay(); }
18 };
19
20 int main()
21 {
22     Person::letsPlay();
23     Teacher::letsPlay();
24     Teacher tc;
25     tc.letsPlay();
26 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
static Person letsPlay()
static Person letsPlay()
static Person letsPlay()

```

- 假设基类Person定义了一个public的静态成员函数letsPlay(), 但派生类Teacher重新实现该函数时, 如下的访问方式都是正确的, 但结果稍有不同
 - Person::letsPlay(); // 调用Person::letsPlay
 - Teacher::letsPlay(); // 调用Teacher::letsPlay
 - teacher.letsPlay(); //调用Teacher::letsPlay
 - void Teacher::someFunc() { letsPlay(); }
teacher.someFunc(); //调用Teacher::letsPlay
 - void Person::someFunc() { letsPlay(); }
teacher.someFunc(); //Teacher 未实现 someFunc 。调用
Person::letsPlay

```
1. class Person {
2. public:
3.     static void letsPlay() { ... .. }
4.     void someFunc() {
5.         letsPlay();
6.     }
7. };
```

```
1. class Teacher : public Person {
2. public:
3.     void someFunc() {
4.         letsPlay();
5.     }
6.     static void letsPlay() {
7.         ... ..
8.     }
9. };
```

Teacher实现了someFunc

```
1. int main() {
2.     Teacher teacher;
3.     Person::letsPlay();
4.     Teacher::letsPlay();
5.     teacher.letsPlay();
6.     teacher.someFunc();
7.     return 0;
8. }
```

Teacher未实现someFunc

```

5 // 基类1
6 class Person
7 {
8 public:
9     static void letsPlay()
10    {
11        cout << "static Person letsPlay()" << endl;
12    }
13    void someFunc()
14    {
15        letsPlay();
16    }
17 };
18 // 派生类
19 class Teacher : public Person
20 {
21 public:
22     //void someFunc() { letsPlay(); }
23     static void letsPlay()
24     {
25         cout << "static Teacher letsPlay()" << endl;
26     }
27 };
28
29 int main()
30 {
31     Person::letsPlay();
32     Teacher::letsPlay();
33     Teacher tc;
34     tc.letsPlay();
35     tc.someFunc();
36 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
static Person letsPlay()
static Teacher letsPlay()
static Teacher letsPlay()
static Person letsPlay()

```

```

5 // 基类1
6 class Person
7 {
8 public:
9     static void letsPlay()
10    {
11        cout << "static Person letsPlay()" << endl;
12    }
13    void someFunc()
14    {
15        letsPlay();
16    }
17 };
18 // 派生类
19 class Teacher : public Person
20 {
21 public:
22     void someFunc() { letsPlay(); }
23     static void letsPlay()
24     {
25         cout << "static Teacher letsPlay()" << endl;
26     }
27 };
28
29 int main()
30 {
31     Person::letsPlay();
32     Teacher::letsPlay();
33     Teacher tc;
34     tc.letsPlay();
35     tc.someFunc();
36 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
static Person letsPlay()
static Teacher letsPlay()
static Teacher letsPlay()
static Teacher letsPlay()

```

继承与普通成员函数

- 以 `introduce` 函数为例，当仅有基类 `Person` 定义了该函数时，`Teacher` 的类对象调用该函数时直接调用 `Person` 中的 `introduce`。
- 如果 `Teacher` 重新实现了该函数，考虑如下代码
 1. `Person pn{...}; pn.introduce();` // 调用 `Person::introduce`
 2. `Teacher tc{...}; tc.introduce();` // 调用 `Teacher::introduce`
 3. `Person xpn = tc; xpn.introduce();` // 调用哪个?
 4. `Person& rpn = tc; rpn.introduce();` // 调用哪个?
 5. `Person* ppn = &tc; ppn->introduce();` // 调用哪个?
 6. `void Teacher::someFunc() {introduce();} tc.someFunc();` // 调用哪个?
 7. `void Person::someFunc() {introduce();} tc.someFunc();` // 调用哪个?

```

5 // 基类1
6 class Person
7 {
8 public:
9     void introduce()
10    {
11        cout << "Person introduce()" << endl;
12    }
13    void someFunc()
14    {
15        introduce();
16    }
17 };
18 // 派生类
19 class Teacher : public Person
20 {
21 public:
22     void introduce()
23     {
24         cout << "Teacher introduce()" << endl;
25     }
26     void someFunc()
27     {
28         introduce();
29     }
30 };

```

```

32 int main()
33 {
34     Person pn;
35     pn.introduce();
36     Teacher tc;
37     tc.introduce();
38     Person xpn = tc;
39     xpn.introduce();
40     Person &rpn = tc;
41     rpn.introduce();
42     Person *ppn = &tc;
43     ppn->introduce();
44     pn.someFunc();
45     tc.someFunc();
46 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
Person introduce()
Teacher introduce()
Person introduce()
Person introduce()
Person introduce()
Person introduce()
Teacher introduce()

```


- 对于普通的成员函数，调用成员函数时的类实例对应哪个类，如果该类中实现了被调函数，则实际调用该函数，否则调用其基类中对应的函数。
 - 普通：非静态、非虚函数
 - 类实例：如 `Person pn = tc`, `Person &rpn = tc`, `Person *ppn=&tc` 这几处的 `Person`、`Person&` 和 `Person*`，对应的类实例的实际类是 `Person`，而不是 `Teacher`，即使后两者实际指向的是一个 `Teacher` 类对象
- 既然后两者（引用和指针）实际指向的是 `Teacher` 类对象，如何才能通过 `rpn.introduce()` 和 `ppn->introduce()` 调用到 `Teacher` 中的 `introduce()` 实现？

- 基类的指针和引用强制转换为派生类的指针和引用：

```
5 // 基类1
6 class Person
7 {
8 public:
9     int x;
10    Person(int xx) : x(xx) {}
11    void introduce()
12    {
13        cout << "Person introduce()" << endl;
14    }
15    void Person_someFunc()
16    {
17        introduce();
18    }
19 };
20 // 派生类
21 class Teacher : public Person
22 {
23 public:
24     int y;
25     Teacher(int xx, int yy) : Person(xx), y(yy) {}
26     void introduce()
27     {
28         cout << "Teacher introduce()" << endl;
29     }
30     void Teacher_someFunc()
31     {
32         introduce();
33     }
34 };
```

```
36 int main()
37 {
38     Teacher tc(1, 2);
39     Person &rpn = tc;
40     rpn.introduce();
41
42     Teacher &ss = (Teacher &)rpn;
43     ss.introduce();
44     cout << ss.y << endl;
45
46     Person *ppn = &tc;
47     ppn->introduce();
48
49     Teacher *s = (Teacher *)ppn;
50     s->introduce();
51     cout << s->y << endl;
52 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
Person introduce()
Teacher introduce()
2
Person introduce()
Teacher introduce()
2
```

如何调用基类中的同名函数

- 在Teacher的introduce中，想要调用Person中的introduce实现？
- 在其他地方（如main）想要通过Teacher的类对象调用Person中的introduce（Teacher中覆盖实现了该函数）？

```
1. class Person {  
2.     public:  
3.         void introduce() {  
4.             ... ..  
5.         }  
6. };
```

```
1. class Teacher : public Person {  
2.     public:  
3.         void introduce() {  
4.             Person::introduce();  
5.             ... ..  
6.         }  
7. };
```

```
1. int main() {  
2.     Teacher teacher;  
3.     teacher.Person::introduce();  
4. }
```

```

5 // 基类1
6 class Person
7 {
8 public:
9     void introduce()
10    {
11        cout << "Person introduce()" << endl;
12    }
13 };
14 // 派生类
15 class Teacher : public Person
16 {
17 public:
18     void introduce()
19    {
20        cout << "Teacher introduce()" << endl;
21    }
22 };
23
24 int main()
25 {
26     Teacher tc;
27     tc.Person::introduce();
28 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
Person introduce()

```

本章内容

1. 继承
2. 派生类构造函数
3. 继承与成员函数
4. 继承与成员变量

4. 继承和成员变量

- 派生类类对象的内存空间中，靠下（低地址）的是基类的数据成员，靠上（高地址）的是派生类的数据成员
- 思考：如下的代码，派生类的类对象内存布局是怎样的？
（注意：基类的成员变量和派生类的成员变量重名，基类的成员变量会被覆盖吗？）

```
1. class A {  
2.   private:  
3.     int x;  
4. };  
5. class B : public A {  
6.   private:  
7.     int x;  
8. };
```

B::X

A::x

```
1. class A {  
2.   protected:  
3.     int x;  
4. };  
5. class B : public A {  
6.   protected:  
7.     int x;  
8. };
```

A::x

```
1. class A {  
2.   public:  
3.     int x;  
4. };  
5. class B : public A {  
6.   public:  
7.     int x;  
8. };
```

B::x



- 最简单的判别方法：检查A和B的大小
 - `cout << sizeof(A) << “ ” << sizeof(B) << endl;`
 - 三种情况下，输出都是48，说明派生类中的变量并没有覆盖基类中的同名变量在内存空间中的位置



- 如何通过派生类对象或在派生类中访问基类中的同名变量？
 - 使用类名限定符

```
1. void B::print() {  
2.     cout << x << endl;  
3.     cout << A::x << endl;  
4. }
```

```
1. int main() {  
2.     B b;  
3.     cout << b.x << endl;  
4.     cout << b.A::x << endl;  
5.     return 0;  
6. }
```

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  // 基类1
6  class Person
7  {
8      int a = 2;
9  public:
10     int x = 2;
11     void introduce()
12     {
13         cout << "Person introduce()" << endl;
14     }
15     friend void print(Person a);
16 };
17
18 // 派生类
19 class Teacher : public Person
20 {
21 public:
22     void introduce()
23     {
24         cout << "Teacher introduce()" << endl;
25     }
26     friend void print(Teacher a);
27 };
28
29 void print(Teacher a) { cout << a.x << endl; cout << a.a << endl; }
30 void print(Person a) { cout << a.a << endl; }
31
32 int main()
33 {
34

```

yahui@Yahui:/media/sf_VM\$ g++ test.cpp
test.cpp: In function 'void print(Teacher)':
test.cpp:29:56: error: 'int Person::a' is private within this context
29 | void print(Teacher a) { cout << a.x << endl; cout << a.a << endl; }
| ^
test.cpp:8:9: note: declared private here
8 | int a = 2;
| ^

回顾：友元关系是单向、非传递、非继承的


```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  // 基类1
6  class Person
7  {
8      int a = 3;
9
10 public:
11     int x = 2;
12     void introduce()
13     {
14         cout << "Person introduce()" << endl;
15     }
16     friend void print(Person a);
17 };
18 // 派生类
19 class Teacher : public Person
20 {
21 public:
22     void introduce()
23     {
24         cout << "Teacher introduce()" << endl;
25     }
26     friend void print(Teacher a);
27 };
```

```
30 void print(Teacher a)
31 {
32     cout << a.x << endl;
33     //cout << a.a << endl;
34     print((Person)a);
35 }
36 void print(Person a) { cout << a.a << endl; }
37
38 int main()
39 {
40     Teacher a;
41     print(a);
42 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
2
3
```

回顾：友元关系是单向、非传递、非继承的

作业

- YOJ-426 矩形相加
- YOJ-607 命名空间-变量与函数
- YOJ-411 完善Professor类