



中國人民大學  
RENMIN UNIVERSITY OF CHINA

程序设计荣誉课程

# 5 多态

授课教师：孙亚辉

# 本章内容

1. 虚函数
2. 虚函数表
3. 抽象类
4. 虚析构函数
5. `override`与`final`说明符
6. 同名隐藏

# 1. 虚函数

- “继承”中的示例：Teacher继承自Person，但增加了新的数据成员，因此当一个Teacher作自我介绍时，它想要囊括它自身区别于一个普通Person的信息，所以Teacher类重新定义了introduce函数。
- 然而一旦Teacher类对象被转换为Person类对象，即使动作的发起人是Teacher，所作的自我介绍也仅会包含Person公有的信息，没有Teacher自身的信息。

```

5 // 基类
6 class Person
7 {
8 public:
9     void introduce()
10    {
11        cout << "Person introduce()" << endl;
12    }
13 };
14 // 派生类1
15 class Teacher1 : public Person
16 {
17 public:
18     void introduce()
19    {
20        cout << "Teacher1 introduce()" << endl;
21    }
22 };
23 // 派生类2
24 class Teacher2 : public Person
25 {
26 public:
27     void introduce()
28    {
29        cout << "Teacher2 introduce()" << endl;
30    }
31 };

```

```

33 int main()
34 {
35     Person pn;
36     Person* pnp = &pn;
37     pnp->introduce();
38
39     Teacher1 tc1;
40     pnp = &tc1;
41     pnp->introduce();
42
43     Teacher2 tc2;
44     pnp = &tc2;
45     pnp->introduce();
46     return 0;
47 }

```

```

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

```

在一个大厅里，大家相互之间不认识，一个介绍人（Person指针）如何介绍每一个人的信息（调用不同派生类对象的introduce函数）？

- 考虑未通过拷贝产生新的对象（内存空间）的派生类向基类的转换（转换为引用或指针类型，比如`pnp=&tc1`）。
- 一个类对象在内存空间中的结构包含了它本身及其基类的数据成员信息，不包含任何函数成员信息。
- 我们想要每个对象（不管是以本身类型出现，还是以被转换为其基类引用或指针类型出现）在任何环境下都能调用其自身重新实现了的、继承自其基类的非静态成员函数（比如`pnp = &tc1; pnp->introduce();` 想要让pnp调用tc1的introduce函数）。
- 为达此目的，目标函数的信息应该出现在类对象的内存数据中，以便该对象发起的函数调用能够在运行时定位到目标函数。C++提供了**虚函数表**（vtable）来处理这种情况，对应的目标函数需要被声明为“**虚函数**（virtual function）”。

```

5 // 基类
6 class Person
7 {
8 public:
9     virtual void introduce()
10    {
11        cout << "Person introduce()" << endl;
12    }
13 };
14 // 派生类1
15 class Teacher1 : public Person
16 {
17 public:
18     void introduce()
19     {
20         cout << "Teacher1 introduce()" << endl;
21     }
22 };
23 // 派生类2
24 class Teacher2 : public Person
25 {
26 public:
27     void introduce()
28     {
29         cout << "Teacher2 introduce()" << endl;
30     }
31 };

```

```

33 int main()
34 {
35     Person pn;
36     Person* pnp = &pn;
37     pnp->introduce();
38
39     Teacher1 tc1;
40     pnp = &tc1;
41     pnp->introduce();
42
43     Teacher2 tc2;
44     pnp = &tc2;
45     pnp->introduce();
46     return 0;
47 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
Person introduce()
Teacher1 introduce()
Teacher2 introduce()

```

通过虚函数，一个介绍人（Person指针）可以介绍每一个人的信息（调用不同派生类对象的introduce函数）

# 虚函数

- **虚函数**是可在派生类中**覆盖**其行为的成员函数，当使用基类的指针或引用来处理派生类时，对被覆盖的虚函数的调用，将会调用定义于派生类中的行为。
- 虚函数的声明如下
  - 在基类中首次声明函数时，在返回类型前使用关键字**virtual**

```
1. class Person {  
2.     virtual void introduce();  
3. };
```

# 虚函数

- 类外定义虚函数时，不能使用`virtual`限定符。如下的函数定义无法通过编译

```
5 // 基类
6 class Person
7 {
8 public:
9     virtual void introduce();
10 };
11
12 invalid specifier outside a class declaration C/C++(239)
13 View Problem Quick Fix... (Ctrl+.)
14 virtual void Person::introduce()
15 {
16     cout << "Person introduce()" << endl;
17 }
```



- 基类中的某个成员函数被声明为**virtual**，则其派生类中具有**同样名字、同样参数列表、同样返回类型**及参数列表后的**同样限定符**（如**const**）的函数，在派生类中也是虚函数。
  - 不要求派生类中的对应函数拥有**virtual**声明（但可以有）
    - 下例中Teacher类中的**introduce**也是虚函数，且是**Person::introduce**的覆盖声明

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

- 注意：cppreference.com上介绍虚函数时，不要求派生类中的函数与基类中对应的虚函数具有同样的返回类型，但《C++ Primer》一书中明确提及[返回类型要匹配](#)，且g++等编译器对于返回类型不匹配的情况会报错（不是说虚函数只要重名就可以被覆盖）。

若某个成员函数 `vf` 在类 `Base` 中被声明为 `virtual`，且某个直接或间接派生于 `Base` 的类 `Derived` 拥有一个下列几项与之相同的成员函数声明

- 名字
- 形参列表 (但非返回类型)
- `cv` 限定符
- 引用限定符

<https://zh.cppreference.com/w/cpp/language/virtual>

则类 `Derived` 中的此函数亦为虚函数（无论其声明中是否使用关键词 `virtual`）并覆盖 `Base::vf`（无论其声明中是否使用单词 `override`）。

```

5 // 基类
6 class Person
7 {
8 public:
9     virtual void introduce()
10    {
11        cout << "Person introduce()" << endl;
12    }
13 };
14 // 派生类1
15 class Teacher1 : public Person
16 {
17     public:
18         int Teacher1::introduce()
19         {
20             cout << "Teacher1 introduce()" << endl;
21             return 1;
22         }
23 };

```

return type is not identical to nor covariant with return type "void" of overridden virtual function "Person::introduce" C/C++(317)

[View Problem](#) [Quick Fix... \(Ctrl+.\)](#)

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp:18:9: error: conflicting return type specified for 'virtual int Teacher1::introduce()'
    18 |         int introduce()
        |         ^~~~~~
test.cpp:9:18: note: overridden function is 'virtual void Person::introduce()'
     9 |         virtual void introduce()
        |         ^~~~~~

```

- 以此前的虚函数introduce声明为例，重新回顾如下几个函数调用
  1. `Person xpn = tc; xpn.introduce();` //调用`Person::introduce`
  2. `Person& rpn = tc; rpn.introduce();` //调用`Teacher::introduce`
  3. `Person* ppn = &tc; ppn->introduce();` //调用`Teacher::introduce`
  4. `void Teacher::someFunc() {introduce();} tc.someFunc();` // 调用 `Teacher::introduce`
  5. `void Person::someFunc() {introduce();} tc.someFunc();` // 调用 `Teacher::introduce`
- 可以看出，除了第一个通过拷贝创建新的类对象的类型转换之外，其余几种情况都调用到`Teacher::introduce`

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

yahui@Yahui:/media/sf\_VM\$ ./a.out

Person introduce()  
Person introduce()  
Person introduce()  
Person introduce()

```
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 };
27
28 int main()
29 {
30     Teacher tc;
31     Person xpn = tc;
32     xpn.introduce();
33     Person &rpn = tc;
34     rpn.introduce();
35     Person *ppn = &tc;
36     ppn->introduce();
37     tc.someFunc();
38 }
```

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

yahui@Yahui:/media/sf\_VM\$ ./a.out

Person introduce()  
Teacher introduce()  
Teacher introduce()  
Teacher introduce()

```
6  class Person
7  {
8  public:
9      virtual 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 };
27
28 int main()
29 {
30     Teacher tc;
31     Person xpn = tc;
32     xpn.introduce();
33     Person &rpn = tc;
34     rpn.introduce();
35     Person *ppn = &tc;
36     ppn->introduce();
37     tc.someFunc();
38 }
```

# private的虚函数也能用吗？

- 普通的成员函数调用，是在编译时根据编译器看到的对象类型直接确定调用目标，在基类中永远无法调用到派生类中的函数。使用虚函数机制，可以在基类中调用到派生类中的函数实现（上页`Person::someFunc() {introduce();}`）。
- 如果某个虚函数声明为`private`的，是否也是一样的效果？

“Hello in A”

“Hello in B”

```
1. class A {
2.     public:
3.         int x, y;
4.         void print() { say_hello(); }
5.     private:
6.         virtual void say_hello();
7. };
8. class B : public A {
9.     private:
10.         virtual void say_hello();
11. };
```

```
1. int main() {
2.     A *a = new B;
3.     a->print();
4.     return 0;
5. }
```

输出什么？

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class A
6  {
7  public:
8      int x, y;
9      void print() { say_hello(); }
10 private:
11     virtual void say_hello() { cout << "Hello in A" << endl; }
12 };
13 class B : public A
14 {
15 private:
16     virtual void say_hello(){ cout << "Hello in B" << endl; }
17 };
18
19 int main()
20 {
21     A *a = new B;
22     a->print();
23     return 0;
24 }

```

```



yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
Hello in B

```



## 派生类中覆盖基类中的虚函数时

- 基类中被覆盖的虚函数不需要可见或可访问：可声明为private
- 派生类可任意修改重新实现的虚函数的访问权限

```
1. class A {
2.     public:
3.         int x, y;
4.         void print() { say_hello(); }
5.     private:
6.         virtual void say_hello();
7. };
8. class B : public A {
9.     public:
10.        virtual void say_hello();
11. };
12. int main() {
13.     B b;
14.     b.say_hello();
15.
16.     A& a = b;
17.     a.say_hello();
18.     return 0;
19. }
```




```
1. class A {
2.     public:
3.         int x, y;
4.         void print() { say_hello(); }
5.     public:
6.         virtual void say_hello();
7. };
8. class B : public A {
9.     private:
10.        virtual void say_hello();
11. };
12. int main() {
13.     B b;
14.     b.say_hello();
15.
16.     A& a = b;
17.     a.say_hello();
18.     return 0;
19. }
```





```
1. class A {
2.     public:
3.         int x, y;
4.         void print() { say_hello(); }
5.     public:
6.         virtual void say_hello();
7. };
8. class B : public A {
9.     private:
10.         virtual void say_hello();
11. };
12. int main() {
13.     B b;
14.     A& a = b;
15.     a.say_hello();
16.     return 0;
17. }
```



- 输出

Hello in B

（即使只有基类的虚函数是public访问权限）

# 虚函数对类对象内存结构的影响

- 前文讲到，欲实现根据运行时对象正确调用目标函数，该函数的信息应该与类对象的内存数据相关联。
- 以下检查虚函数的引入对类对象的内存结构有何影响
  - 原始的Person类，大小为28字节（sizeof(Person)）
  - 增加一个虚函数，大小为40字节（64位Ubuntu系统下）
  - 增加两个虚函数，大小仍为40字节（64位Ubuntu系统下）

```
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    virtual void introduce() { cout << "introduce" << endl; }
13    virtual void introduce2() { cout << "introduce" << endl; }
14 };
```

```

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    //virtual void introduce() { cout << "introduce" << endl; }
13    //virtual void introduce2() { cout << "introduce" << endl; }
14 };
15
16 int main()
17 {
18     Person person;
19     cout << &person << endl;
20     cout << &(person.age) << endl;
21     cout << "sizeof(person): " << sizeof(person) << endl;
22     return 0;
23 }
24

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
0x7ffe034fcb30
0x7ffe034fcb30
sizeof(person): 28

```

```

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    virtual void introduce() { cout << "introduce" << endl; }
13    virtual void introduce2() { cout << "introduce" << endl; }
14 };
15
16 int main()
17 {
18     Person person;
19     cout << &person << endl;
20     cout << &(person.age) << endl;
21     cout << "sizeof(person): " << sizeof(person) << endl;
22     return 0;
23 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
0x7ffdef29c9e0
0x7ffdef29c9e8
sizeof(person): 40

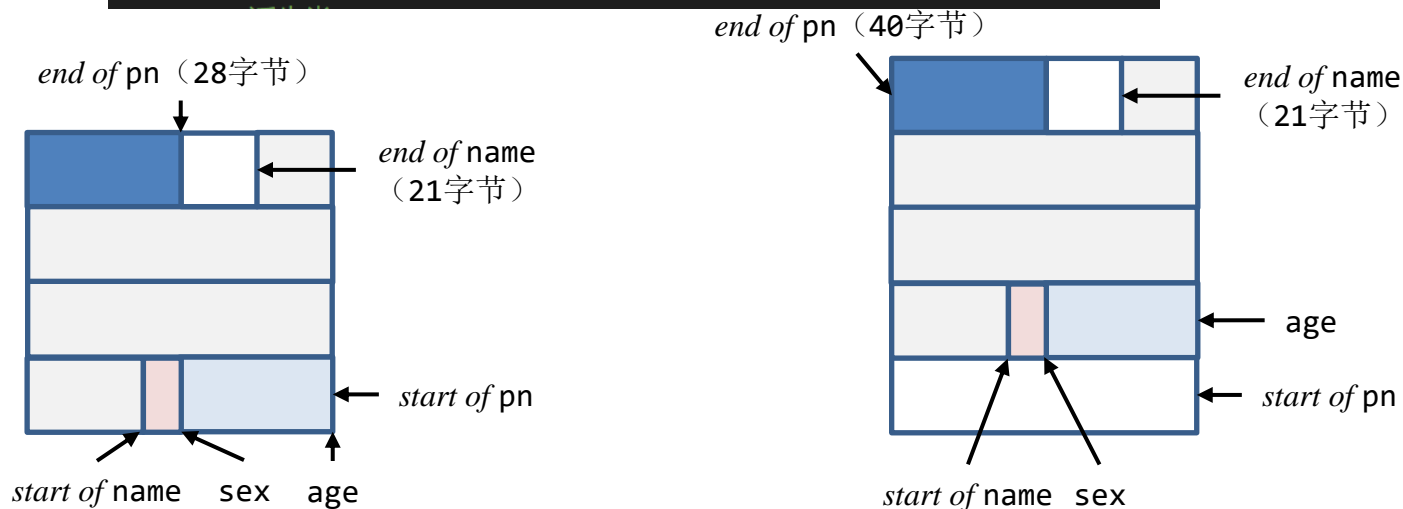
```

- 左侧展示了无virtual函数时Person类对象中数据成员内存布局；  
右侧则展示了有virtual函数时对应内存布局
  - 存在虚函数时，类对象内存结构中，在首个数据成员之前增加了一段8字节的空间（64位Ubuntu系统）——该空间将用于调用虚函数

```

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    virtual void introduce() { cout << "introduce" << endl; }
13    virtual void introduce2() { cout << "introduce" << endl; }
14 };

```



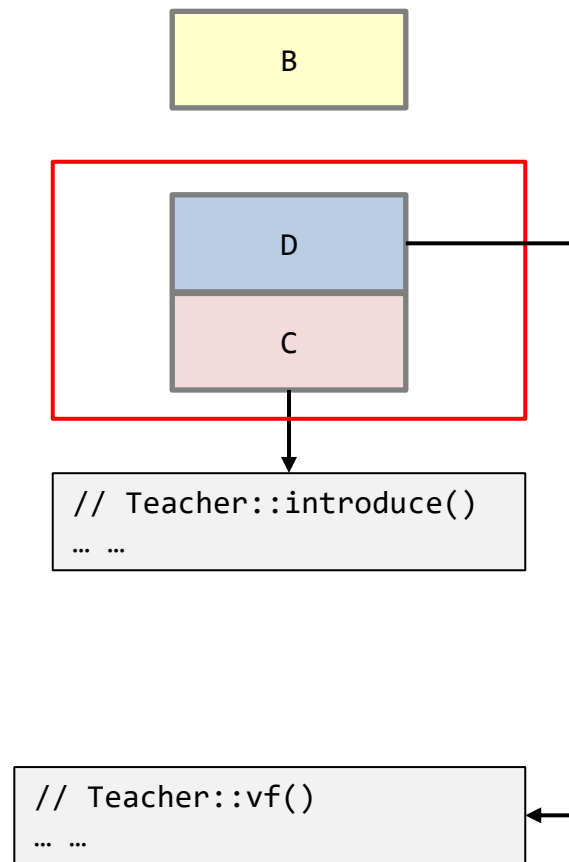
说明：上图中每一行占据8字节空间

# 本章内容

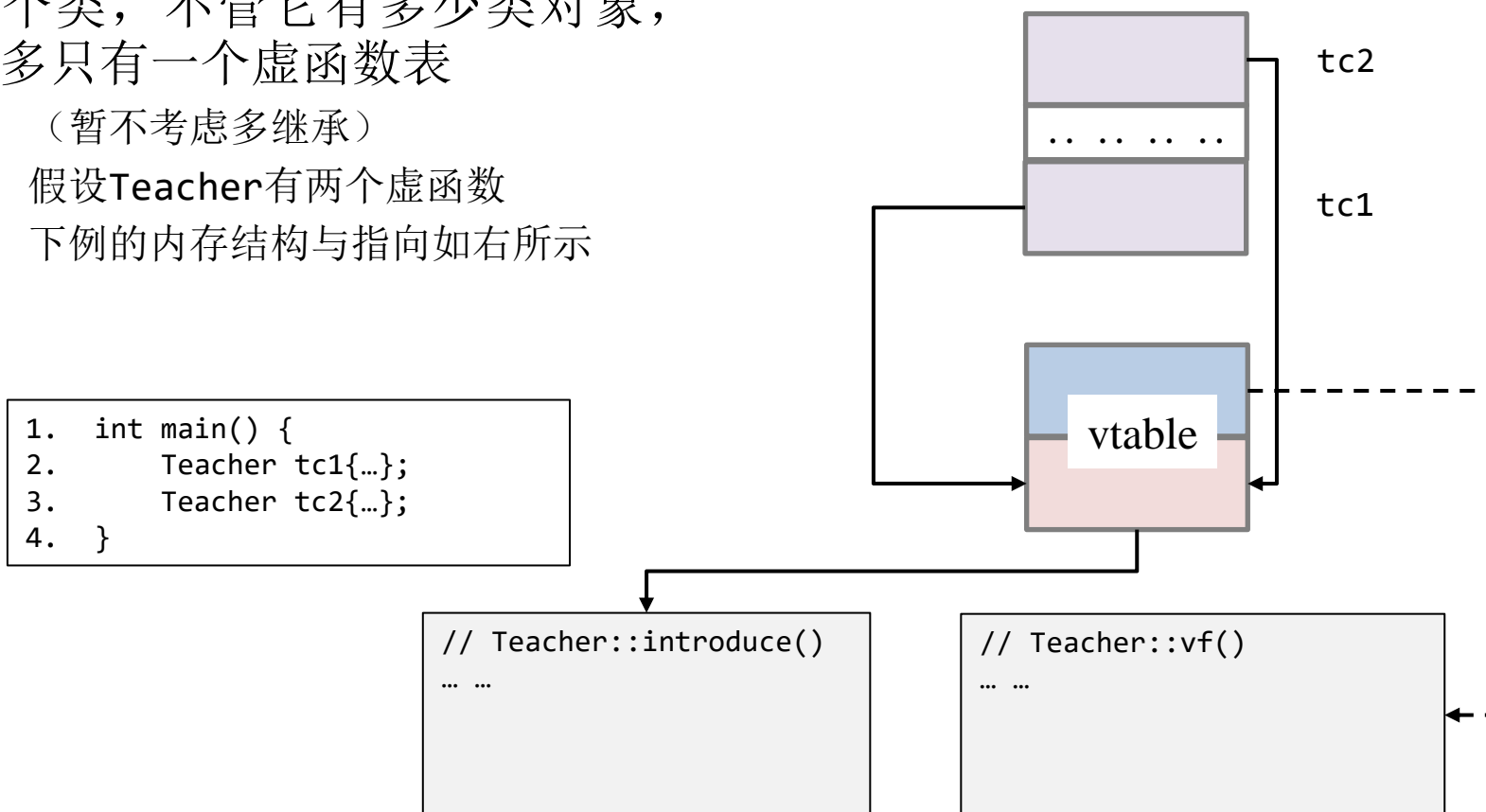
1. 虚函数
2. 虚函数表
3. 抽象类
4. 虚析构函数
5. `override`与`final`说明符
6. 同名隐藏

## 2. 虚函数表

- 对于前例，**Teacher**类对象起始位置处存放的数据**B**是另一块内存（红色框中）的地址，而该内存低地址处存放的数据是**Teacher**类中虚函数**introduce**的代码地址。
- 假设**Person**定义了第二个虚函数**vf**，且**Teacher**同样覆盖实现了**vf**，右图展示了栈上类对象起始8个字节（64位系统）；该8字节为指针，指向的内存区域（红色框中）存储了分别指向**Teacher**中两个虚函数代码的地址。红框中以连续内存形式存储类的虚函数代码段地址的结构称为**虚函数表**（**vtable**）
  - 类对象中指向虚函数表的数据（x64中前8字节、x32中前4字节）被称为**虚指针**



- 一个类，不管它有多少类对象，最多只有一个虚函数表
  - （暂不考虑多继承）
  - 假设**Teacher**有两个虚函数
  - 下例的内存结构与指向如右所示

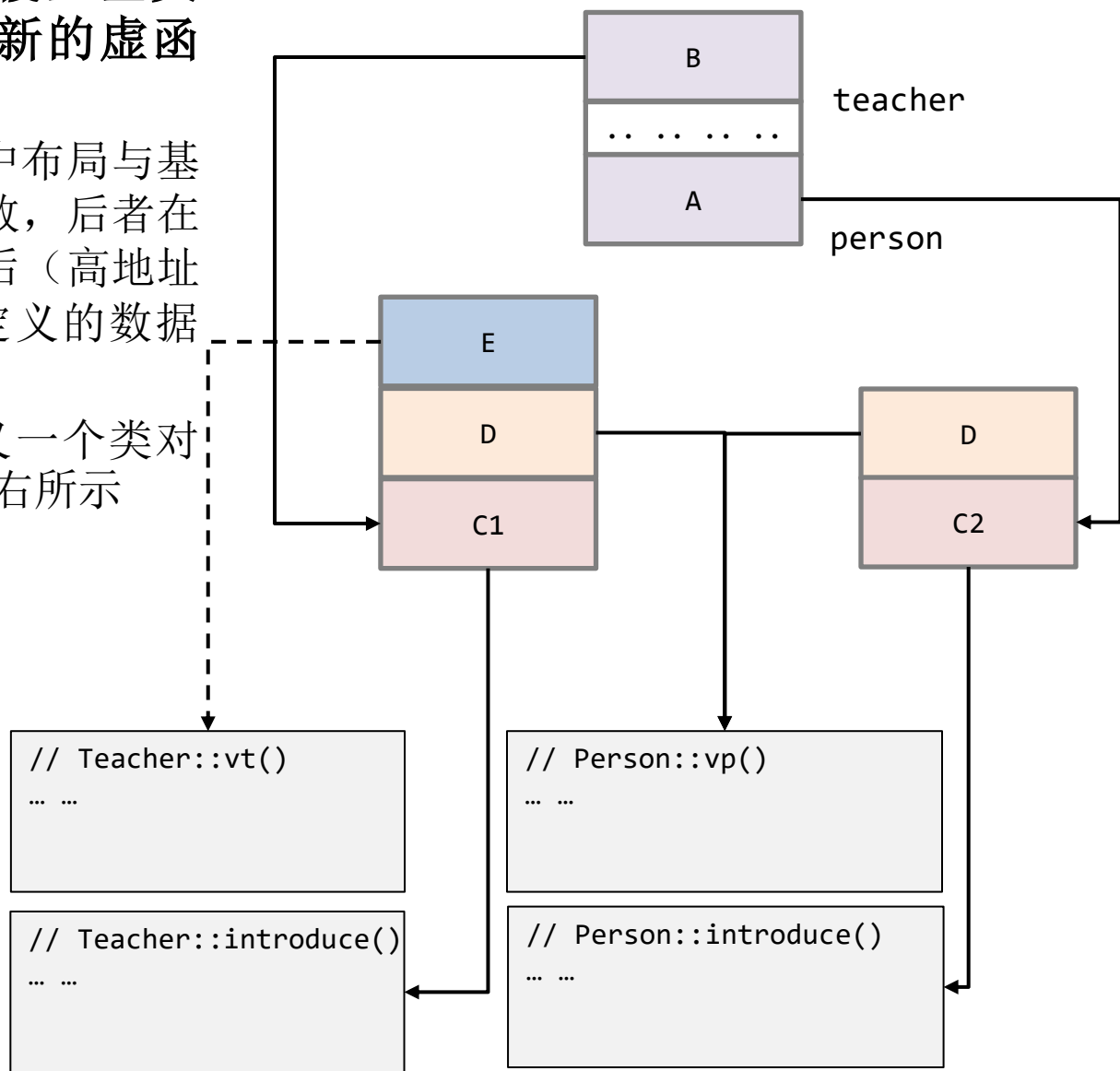




- 派生类的虚函数表涵盖了它从基类直接继承的虚函数、覆盖基类的虚函数以及自定义的新的虚函数

- 前两者在派生类vtable中布局与基类vtable中对应函数一致，后者在vtable中位于前两者之后（高地址处，类比派生类中新定义的数据成员）
- 如下继承关系，各定义一个类对象，内存结构与指向如右所示

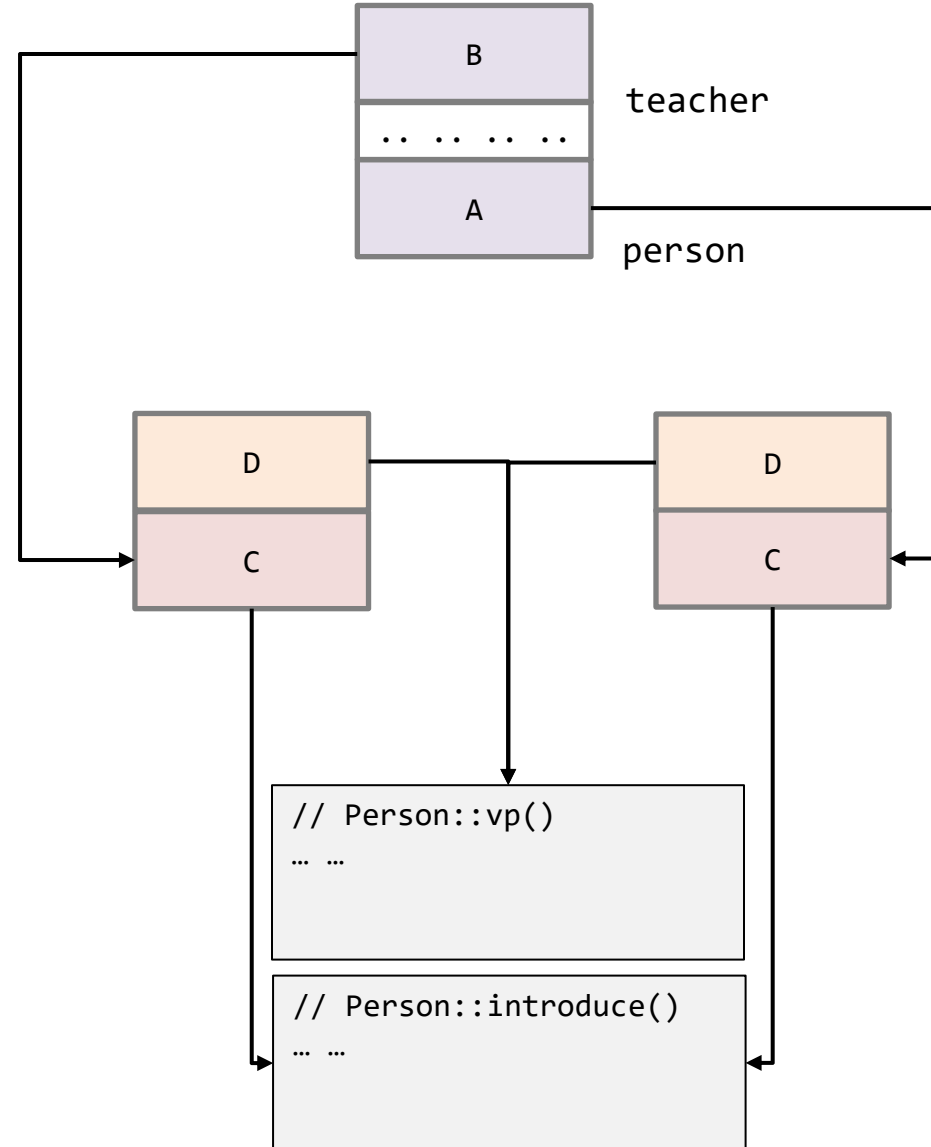
```
1. class Person {  
2.     virtual void introduce();  
3.     virtual void vp();  
4. };  
5. class Teacher : public Person{  
6.     void introduce();  
7.     virtual void vt();  
8. };
```



- 如果派生类没有重新声明继承自基类的任何虚函数，则派生类的虚函数表是基类虚函数表的直接拷贝

- 派生类并不直接使用基类的虚函数表
- 派生类的虚函数表包含了基类中 **private** 的虚函数（派生类未覆盖）

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



# 虚函数表

- 虚函数表由编译器在编译时生成，在程序运行时载入内存并初始化，在类对象初始化时设置虚指针。
- 以如下示例代码来查看编译产生的虚函数表信息。

```
test.cpp
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Person
6  {
7  public:
8      virtual void introduce() { cout << "" << endl; };
9      virtual void vp() { cout << "" << endl; };
10 };
11 class Teacher : public Person
12 {
13 public:
14     virtual void introduce() { cout << "" << endl; };
15     virtual void vt() { cout << "" << endl; };
16 };
17
18 int main()
19 {
20     Teacher tc;
21     Person ps;
22 }
23
```

g++ -o test.out -save-temps -masm=intel  
-m32 test.cpp

test.s - Notepad

	File	Edit	Format	View	Help
	.size	_ZTV7Teacher,	20		
_ZTV7Teacher:					
	.long	0			
	.long	_ZTI7Teacher			
	.long	_ZN7Teacher9introduceEv			
	.long	_ZN6Person2vpEv			

- 引用了4个虚函数的汇编码如下图所示
  - 两个虚函数表 `_ZTV7Teacher` 与 `_ZTV6Person`（demangle之后的名字分别是 `vtable for Teacher` 与 `vtable for Person`）
  - 前者包含 `Teacher` 类中3个虚函数的代码段地址和 `_ZTI7Teacher`（`typeid for Teacher`）的地址
  - 后者包含 `Person` 类中2个虚函数代码段地址与该类的 `typeid` 对象的地址

<code>_ZTV7Teacher:</code>		<code>_ZTV6Person:</code>	
<code>.long</code>	<code>0</code>	<code>.long</code>	<code>0</code>
<code>.long</code>	<code>_ZTI7Teacher</code>	<code>.long</code>	<code>_ZTI6Person</code>
<code>.long</code>	<code>_ZN7Teacher9introduceEv</code>	<code>.long</code>	<code>_ZN6Person9introduceEv</code>
<code>.long</code>	<code>_ZN6Person2vpEv</code>	<code>.long</code>	<code>_ZN6Person2vpEv</code>
<code>.long</code>	<code>_ZN7Teacher2vtEv</code>		

如前所述，派生类的虚函数表涵盖了它从基类直接继承的虚函数、覆盖基类的虚函数以及自定义的新的虚函数

前两者在派生类 `vtable` 中布局与基类 `vtable` 中对应函数一致，后者在 `vtable` 中位于前两者之后

# 多个虚函数对类对象内存结构的影响

- 下面Person中有两个虚函数，为什么还是只多了8字节？（64位系统下）

```
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     virtual void introduce() { cout << "Person introduce" << endl; }
13     virtual void introduce2() { cout << "Person introduce" << endl; }
14 };
15
16 int main()
17 {
18     Person person;
19     cout << &person << endl;
20     cout << &(person.age) << endl;
21     cout << "sizeof(person): " << sizeof(person) << endl;
22     return 0;
23 }
```

```
yahui@Yahoo:/media/sf_VM$ g++ test.cpp
yahui@Yahoo:/media/sf_VM$ ./a.out
0x7ffffda0c2bf0
0x7ffffda0c2bf8
sizeof(person): 40
```

# 本章内容

1. 虚函数
2. 虚函数表
3. 抽象类
4. 虚析构函数
5. `override`与`final`说明符
6. 同名隐藏

# 3. 抽象类

- 前文介绍的虚函数在基类中声明时，基类能够给出其有意义的实现，派生类也可以通过覆盖基类中的实现从而给出更具体的实现。
- **纯虚函数**是在所声明的类中不能给出有意义的实现的虚函数，其实现**一般**留给派生类去做。
- 纯虚函数的声明，是在虚函数声明之后，“;”之前，加入“= 0”，如
  - `virtual void pureVF(... ..) = 0;`
- 虽然没必要，但是声明纯虚函数的类可以给出该函数的定义/实现

```
1. class Person {  
2. public:  
3.     virtual void somef() = 0;  
4. };  
5. void Person::somef() { . . . . . }
```

# 抽象类

- 含有纯虚函数的类是**抽象类**（abstract class）
  - 抽象类不能直接创建类对象

```
5  class AbsClass {
6  public:
7      virtual void somePVF() = 0;
8  };
9  void AbsClass::somePVF(){}
10
11  int main()
12  {
13      AbsClass ak;
14  }
```

```
yahui@Yahui:/media/sf_VM$ g++ -save-temps -g -m32 -O0 -o test test.cpp
test.cpp: In function 'int main()':
test.cpp:13:14: error: cannot declare variable 'ak' to be of abstract type 'AbsK
lass'
   13 |         AbsClass ak;
       |         ^~
test.cpp:5:7: note: because the following virtual functions are pure within 'A
bsKlass':
   5 |     class AbsKlass {
       |     ^~~~~~
test.cpp:7:18: note: 'virtual void AbsKlass::somePVF()'
   7 |         virtual void somePVF() = 0;
       |         ^~~~~~
```



# 抽象类

抽象类的派生类也可以是抽象类：（1）不声明基类中的纯虚函数，（2）重复声明基类中的纯虚函数，（3）声明自己的纯虚函数

- 如果抽象类的派生类没有完全实现所继承的全部纯虚函数，则该派生类也是抽象类（下面的代码同样报错）
  - 抽象类也可以有自己的构造函数与其他成员函数

```
5  class AbsKlass {
6  public:
7      virtual void somePVF() = 0;
8      virtual void somePVF2() = 0;
9  };
10
11  class AbsKlass2: AbsKlass {
12  public:
13      void somePVF(){};
14      virtual void somePVF2() = 0;
15  };
16
17
18  int main()
19  {
20      AbsKlass2 ak;
21  }
```

- 为什么要有抽象类？以抽象类OnBeatingListener为例
  - 该类的主要功能是作为事件“挨打”的监听器，其内声明了一个纯虚函数onBeating作为接口，当实现了该接口的类对象处于“挨打”中时，该接口被调用，即被打者要做出响应
  - 假设Person实现了该接口，在实现中根据情况做出不同响应，如：用手打回去、拿起武器打回去等
  - 而OnBeatingListener本身不适宜直接创建类对象，因为它不表示某个能够挨打的目标，只有继承了该类并实现了obBeating接口的类才可作为挨打的目标，如Person、Cat、Dog等

```
1. class OnBeatingListener {
2. public:
3.     virtual void onBeating() = 0;
4. };
5. class Person : public OnBeatingListener {
6. public:
7.     virtual void onBeating();
8. };
9. void Person::onBeating() {
10.     if (...) { ... }
11.     else if (...) { ... }
12.     ...
13.     else { ... }
14. }
```

- 非抽象基类可以实例化，有虚函数表；抽象基类不能实例化，可以没有虚函数表（取决于编译器）

```

5  class OnBeatingListener {
6  public:
7      virtual void onBeating();
8      virtual void introduce(){};
9      virtual void vp(){};
10 };
11 void OnBeatingListener::onBeating(){}
12
13 class Person: public OnBeatingListener {
14 public:
15     void onBeating(){}
16     void introduce(){};
17     void vp(){};
18 };
19
20 int main()
21 {
22     Person ak;
23 }

```

```

_ZTV6Person:
    .long    0
    .long    _ZTI6Person
    .long    _ZN6Person9onBeatingEv
    .long    _ZN6Person9introduceEv
    .long    _ZN6Person2vpEv
_ZTV17OnBeatingListener:
    .long    0
    .long    _ZTI17OnBeatingListener
    .long    _ZN17OnBeatingListener9onBeatingEv
    .long    _ZN17OnBeatingListener9introduceEv
    .long    _ZN17OnBeatingListener2vpEv

```

```

5  class OnBeatingListener {
6  public:
7      virtual void onBeating() = 0;
8      virtual void introduce(){};
9      virtual void vp(){};
10 };
11 void OnBeatingListener::onBeating(){}
12
13 class Person: public OnBeatingListener {
14 public:
15     void onBeating(){}
16     void introduce(){};
17     void vp(){};
18 };
19
20 int main()
21 {
22     Person ak;
23 }

```

g++ -o test.out -save-temps -masm=intel -m32 test.cpp  
查看s文件：发现仅有\_ZTV6Person  
(gcc version 9.3.0)

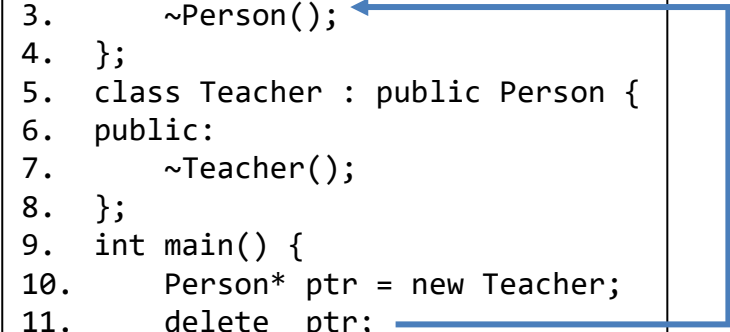
# 本章内容

1. 虚函数
2. 虚函数表
3. 抽象类
4. 虚析构函数
5. `override`与`final`说明符
6. 同名隐藏

## 4. 虚析构函数

- 当类对象生命周期结束时，析构函数被调用，用于销毁该对象内的资源。
- 当使用`delete`销毁动态分配的类对象、且类对象为基类指针类型时，**非虚的**析构函数**不能**正确调用到实际的派生类中定义的析构函数
- 以如下代码为例：直接调用了`Person::~~Person()`

```
1. class Person {  
2. public:  
3.     ~Person();  
4. };  
5. class Teacher : public Person {  
6. public:  
7.     ~Teacher();  
8. };  
9. int main() {  
10.     Person* ptr = new Teacher;  
11.     delete ptr;  
12. }
```



```

5  class Person
6  {
7  public:
8      ~Person()
9      {
10         cout << "~Person" << endl;
11     }
12 };
13 class Teacher : public Person
14 {
15 public:
16 int x;
17     ~Teacher()
18     {
19         cout << "~Teacher" << endl;
20     }
21 };
22 int main()
23 {
24     Person *ptr = new Teacher;
25     delete ptr;
26 }

```

```

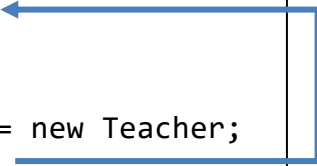
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
~Person

```

# 虚析构函数

当基类的析构函数被定义为虚函数时，派生类中的析构函数也继承了其虚属性，能确保销毁动态类对象时执行到正确的析构函数。

```
1. class Person {  
2. public:  
3.     virtual ~Person();  
4. };  
5. class Teacher : public Person {  
6. public:  
7.     ~Teacher();  
8. };  
9. int main() {  
10.     Person* ptr = new Teacher;  
11.     delete ptr;  
12. }
```

A blue line with an arrow originates from the 'delete ptr;' statement on line 11 and points to the '~Teacher()' destructor on line 7, illustrating that the virtual destructor of the derived class is called.

```

5  class Person
6  {
7  public:
8      virtual ~Person()
9      {
10         cout << "~Person" << endl;
11     }
12 };
13 class Teacher : public Person
14 {
15 public:
16 int x;
17     ~Teacher()
18     {
19         cout << "~Teacher" << endl;
20     }
21 };
22 int main()
23 {
24     Person *ptr = new Teacher;
25     delete ptr;
26 }

```

```

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

```

- 派生类的析构函数会自动调用基类的析构函数。
- 一个类如果定义了虚函数，则最好将析构函数也定义成虚函数。
- 析构函数可以是虚函数，但是**构造函数不能是虚函数**。



**构造函数不能是虚函数**。原因：虚函数的执行依赖于虚函数表，在构造函数中进行虚指针的初始化工作，在构造对象期间，虚指针还没有被初始化，无法通过虚指针调用虚构造函数初始化对象。

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Person
6  {
7  public:
8      virtual Person()
9      {
10         cout << "Person" << endl;
11     }
12 };
13 class Teacher : public Person
14 {
15 };
16 int main()
17 {
18     Person *ptr = new Teacher;
19     delete ptr;
20 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp:8:5: error: constructors cannot be declared 'virtual' [-fpermissive]
   8 |     virtual Person()
     |     ^~~~~~
```

# 本章内容

1. 虚函数
2. 虚函数表
3. 抽象类
4. 虚析构函数
5. **override**与**final**说明符
6. 同名隐藏

# 5. override与final说明符

- **override**说明符能够指定一个虚函数覆盖另一个虚函数
  - 使用时，关键字**override**紧随成员函数声明（但在纯说明符=0之前），如
    - `virtual void introduce() override = 0;`
  - **override**能够确保该函数为虚函数并覆盖某个基类中的虚函数
    - 如果在基类中不存在对应的虚函数（参数列表也要对应），编译器报错

```
1. class Person : public OnBeatingListener {  
2.     public:  
3.         virtual void onBeating() override; // 正确  
4.         virtual void onBeating(int) override; // 错误  
5.     };
```

- 对于第4行，若没有**override**说明符，编译通过，当存在**override**说明符时，编译器报错

```
error: 'virtual void Person::onBeating(int)' marked 'override',  
but does not override
```

```

5  class OnBeatingListener
6  {
7  public:
8      virtual void onBeating() = 0;
9  };
10 void OnBeatingListener::onBeating() {}
11
12 class Person : public OnBeatingListener
13 {
14 public:
15     void onBeating() override {}
16     void onBeating(int) override {}
17 };
18
19 int main()
20 {
21     Person ak;
22 }

```

```

PS C:\Users\Yahui\Drive\VM> cd "c:\Users\Yahui\Drive\VM\" ; if ($?) { g++ test.cpp -o test } ; if ($?) { .\test }
test.cpp:16:10: error: 'void Person::onBeating(int)' marked 'override', but does not override

```

```

16 |     void onBeating(int) override {}
    |           ^~~~~~

```

重载（overload）和重写（override）的区别：  
 重载Overloading是一个类中多态性的一种表现，  
 重写Overriding是基类与派生类之间多态性的一种表现。

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class OnBeatingListener
6  {
7  public:
8      virtual void onBeating() {}
9  };
10
11 class Person : public OnBeatingListener
12 {
13 public:
14     void onBeating() override {}
15 };
16
17 int main()
18 {
19     Person ak;
20 }

```

**override**能够确保该函数为虚函数并覆盖某个基类中的虚函数，否则报错

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class OnBeatingListener
6  {
7  public:
8      void onBeating() {}
9  };
10
11 class Person : public OnBeatingListener
12 {
13 public:
14     void onBeating() override {}
15 };
16
17 int main()
18 {
19     Person ak;
20 }

```

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp:14:10: error: 'void Person::onBeating()' marked 'override', but does not override
   14 |     void onBeating() override {}
      |           ^~~~~~

```

- 用于修饰虚函数的**final**说明符，指定某个虚函数不能在派生类中被覆盖
- 使用方式为
  - `virtual void introduce() final;`
- 非虚函数不能使用**final**

```
error: 'void Person::nonFinal()' marked 'final', but is not virtual
void nonFinal() final;
    ^~~~~~
```

- 派生类若试图覆盖基类中使用**final**声明的虚函数，编译器报错

```
error: virtual function 'virtual void Teacher::introduce()'
void introduce();
    ^~~~~~
error: overriding final function 'virtual void Person::introduce()'
virtual void introduce() final ;
            ^~~~~~
```

```

5  class OnBeatingListener
6  {
7  public:
8      virtual void onBeating() final {};
9  };
10 void OnBeatingListener::onBeating() {}
11
12 class Person : public OnBeatingListener
13 {
14 public:
15     void onBeating() {}
16 };
17
18 int main()
19 {
20     Person ak;
21 }

```

```

PS C:\Users\Yahui\Drive\VM> cd "c:\Users\Yahui\Drive\VM\" ; if ($?) { g++ test.cpp -o test } ; if ($?) { .\test }
test.cpp:10:6: error: redefinition of 'void OnBeatingListener::onBeating()'
10 | void OnBeatingListener::onBeating() {}
    | ~~~~~^~~~~~
test.cpp:8:18: note: 'virtual void OnBeatingListener::onBeating()' previously defined here
8 |     virtual void onBeating() final {};
    | ~~~~~^~~~~~
test.cpp:15:10: error: virtual function 'virtual void Person::onBeating()' overriding final function
15 |     void onBeating() {}
    | ~~~~~^~~~~~
test.cpp:8:18: note: overridden function is 'virtual void OnBeatingListener::onBeating()'
8 |     virtual void onBeating() final {};
    | ~~~~~^~~~~~

```

## 非虚函数不能使用final

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class OnBeatingListener
6  {
7  public:
8      void onBeating() final {}
9  };
10
11  class Person : public OnBeatingListener
12  {
13  public:
14      void onBeating() {}
15  };
16
17  int main()
18  {
19      Person ak;
20  }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp:8:10: error: 'void OnBeatingListener::onBeating()' marked 'final', but
is not virtual
```

```
8 | void onBeating() final {}
  | ^~~~~~
```



# 本章内容

1. 虚函数
2. 虚函数表
3. 抽象类
4. 虚析构函数
5. `override`与`final`说明符
6. 同名隐藏

## 6. 同名隐藏

- 思考如下代码

```
1. class A {  
2.     public:  
3.         virtual void print() {  
4.             cout << "A::print()" << endl;  
5.         }  
6. };  
7. class B : public A {  
8.     public:  
9.         virtual void print(int x) {  
10.             cout << "B::print(int) " << x << endl;  
11.         }  
12. };
```

- 类B的虚函数表为  
(64位系统)

```
1. _ZTV1B:  
2.     .quad    0  
3.     .quad    _ZTI1B  
4.     .quad    _ZN1A5printEv  
5.     .quad    _ZN1B5printEi
```

- 如下代码的运行结果是什么

```
6. class A {  
7.     public:  
8.         virtual void print() {  
9.             cout << "A::print()" << endl;  
10.        }  
11. };  
12. class B : public A {  
13.     public:  
14.         virtual void print(int x) {  
15.             cout << "B::print(int) " << x << endl;  
16.        }  
17. };
```

```
20. int main() {  
21.     B *b = new B;  
22.     b->print();  
23.     b->print(10);  
24.     delete b;  
25.     return 0;  
26. }
```

- 编译错误

```
A.cpp: In function 'int main()':
A.cpp:22:11: error: no matching function for call to 'B::print()'
    b->print();
           ^
A.cpp:14:16: note: candidate: virtual void B::print(int)
    virtual void print(int x) {
                  ^~~~~~
A.cpp:14:16: note: candidate expects 1 argument, 0 provided
```

# 重新定义将隐藏方法


无论参数列表是否相同，**重新定义同名函数将隐藏所有的同名基类方法**

- 如果存在两个或多个具有包含关系的作用域，外层声明了一个标识符，而内层没有再次声明同名标识符，那么外层标识符在内层依然可见，如果在内层声明了同名标识符，则外层标识符在内层不可见，这时称内层标识符隐藏了外层同名标识符，这种现象称为隐藏规则。
- 在类的派生层次结构中，基类的成员和派生类新增的成员都具有类作用域。二者的作用范围不同，是相互包含的两个层，**派生类在内层**。这时，如果派生类声明了一个和某个基类成员同名的新成员，派生的新成员就隐藏了外层同名成员，直接使用成员名只能访问到派生类的成员。**如果派生类中声明了与基类同名的新函数，即使函数的参数表不同，从基类继承的同名函数的所有重载形式也都被隐藏。****如果要访问被隐藏的成员，就需要使用类作用域分辨符和基类名来限定。**

print是不是虚函数都会被同名隐藏

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class A
6  {
7  public:
8      void print()
9      {
10         cout << "A::print()" << endl;
11     }
12 };
13 class B : public A
14 {
15 public:
16     void print(int x)
17     {
18         cout << "B::print(int) " << x << endl;
19     }
20 };
21
22 int main()
23 {
24     B b;
25     b.print();
26 }
```

解决方案



```
22 int main()
23 {
24     B b;
25     b.A::print();
26 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:25:13: error: no matching function for call to 'B::print()'
   25 |         b.print();
      |         ^
test.cpp:16:10: note: candidate: 'void B::print(int)'
   16 |         void print(int x)
      |         ^~~~~~
test.cpp:16:10: note: candidate expects 1 argument, 0 provided
```

# 经验规则

- 如果基类声明被重载了，则应在派生类中重新定义所有的基类版本
  - 如果不需要修改，则新定义可只调用基类版本（下面蓝色代码）
  - **思考**：如果基类中同名方法太多，难道一定得每个都在派生类中按照如下方式声明吗？

```
1. class A {  
2.     public:  
3.         virtual void print() { . . . . . }  
4. };  
5. class B : public A {  
6.     public:  
7.         virtual void print() {  
8.             A::print();  
9.         }  
10.        virtual void print(int x) { . . . . . }  
11. };
```

```

1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class A
6  {
7  public:
8      void print()
9      {
10         cout << "A::print()" << endl;
11     }
12     void print(int a)
13     {
14         cout << "A::print(int)" << endl;
15     }
16     void print(string a)
17     {
18         cout << "A::print(string)" << endl;
19     }
20 };
21 class B : public A
22 {
23 public:
24     using A::print;
25     void print(double x)
26     {
27         cout << "B::print(double) " << endl;
28     }
29 };
30
31 int main()
32 {
33     B b;
34     b.print();
35     b.print(1);
36     b.print("a");
37     b.print(1.0);
38 }

```

使用**using**声明解决继承时的同名隐藏问题：  
把基类中的重载函数一次性地放入派生类里面（**print**均为虚函数也是一样的结果）

```

yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
A::print()
A::print(int)
A::print(string)
B::print(double)

```



# 作业

- YOJ-382 计算工资系统（继承和派生）
- YOJ-397 商品的销售总款和平均售价
- YOJ-401 Shape