

知识点1【继承中的静态成员特性】（了解）

知识点2【多继承】（了解）

多继承的格式：

多继承容易产生二义性：（解决办法1 使用作用域）

普通继承：

知识点4【虚继承】（了解）1-2 vs studio分析

虚继承：

总结：之所以产生 vbptr和vtable 目的 保证 不管多少个继承 虚基类的数据只有一份。

注意： 虚继承只能解决具备公共祖先的多继承所带来的二义性问题，不能解决没有公共祖先的多继承的。

知识点5【多态的概述】

1、概念

2、引入

总结：基类指针、引用 只能访问 子类对象中 基类部分 数据

3、使用基类指针、引用 访问 子类对象中的成员方法（虚函数）

4、拥有虚函数的类 涉及得到继承 2-2

总结：

4、虚函数的应用案例（基类指针、引用 作为函数的参数）

问题:C++的动态捆绑机制是怎么样的？

知识点1【继承中的静态成员特性】（了解）

```
1 class Base
2 {
3 public:
4 //静态成员属于类 而不属于对象
```

```

5  static int num;
6  static int data;
7
8  static void showData(void);
9
10 };
11 int Base::num = 100;
12 int Base::data = 200;
13
14 class Son:public Base
15 {
16 public:
17     static int data;//父和子类 静态成员 同名
18     static void showData(void);
19 };
20 int Son::data = 300;
21
22 void test01()
23 {
24     //从Base类中访问
25     cout<<Base::num<<endl;
26
27     // Son 也拥有了静态成员num
28     cout<<Son::num<<endl;
29
30     //父和子类 静态成员 同名 在子类中 访问子类中的成员
31     cout<<Son::data<<endl;//200
32
33     //父和子类 静态成员 同名 访问父类中的成员 必须加 Base::
34     cout<<Son::Base::data<<endl;//200
35
36
37     //父和子类 同名静态成员函数 子类默认访问子类的静态成员函数
38     Son::showData();
39     //父和子类 同名静态成员函数 子类访问父类的静态成员函数 必须加 Base::
40     Son::Base::showData();
41 }

```

运行结果：

100

100

300

200

子类中的showData

父类中showData

知识点2【多继承】（了解）

多继承的格式：

```
1 class 子类：继承方式1 父类名1,继承方式2 父类名2,继承方式3 父类名3,....
2 {
3
4 };
5 //表示子类 是由 父类名1,父类名2,父类名3...共同派生出来
```

```
1 class Base1
2 {
3 public:
4     int a;
5 };
6 class Base2
7 {
8 public:
9     int b;
```

```

10 };
11
12 class Son:public Base1,public Base2
13 {
14     //Son类 拥有了a b
15 };
16 int main(int argc, char *argv[])
17 {
18     Son ob;
19     ob.a = 100;
20     ob.b = 200;
21     return 0;
22 }

```

多继承容易产生二义性： （解决办法1 使用作用域）

```

1  class Base1
2  {
3  public:
4      int a;
5  };
6  class Base2
7  {
8  public:
9      int a;
10 };
11
12 class Son:public Base1,public Base2
13 {
14
15 };
16 int main(int argc, char *argv[])
17 {
18     Son ob;
19     //ob.a = 100;//err Base1 和 Base2中都有a成员同名
20     //解决办法:加作用域
21     ob.Base1::a = 100;
22     ob.Base2::a = 200;
23     return 0;
24 }

```

知识点3【菱形继承】 具有公共祖先 的多继承

```
1  class Animal
2  {
3  public:
4      int data;
5  };
6
7  class Sheep:public Animal
8  {
9  public:
10 };
11 class Tuo:public Animal
12 {
13 public:
14 };
15
16 class SheepTuo:public Sheep,public Tuo
17 {
18 public:
19 };
20 int main(int argc, char *argv[])
21 {
22     SheepTuo st;
23     //SheepTuo 从Sheep中继承data 从Tuo继承data 就产生二义性
24     //st.data = 200;//err
25     //第一中方式：加作用域解决
26     st.Sheep::data = 200;
27     st.Tuo::data = 300;
28
29     return 0;
30 }
```

普通继承：

```
1  class Animal
2  {
3  public:
4      int data;
5  };
```

```

class Animal      size(4):
    +---+
    |   | data
    +---+

```

```

1 class Sheep:public Animal
2 {
3     public:
4 };

```

```

class Sheep      size(4):
    +---+
    |   | +---+ (base class Animal)
    |   | | data
    |   | +---+
    +---+

```

```

1 class Tuo:public Animal
2 {
3     public:
4 };

```

```

class Tuo      size(4):
    +---+
    |   | +---+ (base class Animal)
    |   | | data
    |   | +---+
    +---+

```

```

1 class SheepTuo:public Sheep,public Tuo
2 {
3 public:
4 };

```

```

class SheepTuo  size(8):
    +----
0    |   +---- (base class Sheep)
0    |   |   +---- (base class Animal)
0    |   |   |   data
    |   |   +----
    |   +----
4    |   +---- (base class Tuo)
4    |   |   +---- (base class Animal)
4    |   |   |   data
    |   |   +----
    |   +----
    +----

```

知识点4【虚继承】（了解）1-2 vs studio分析

virtual修饰继承方式

```

1 //继承的动作 虚继承
2 //父类：虚基类
3 class 子类:virtual public 父类
4 {
5 };

```

虚继承：

```

1 class Animal
2 {
3 public:
4     int data;
5 };

```

```
class Animal      size(4):
    +---+
    |   data
    +---+
```

```
1 class Sheep:virtual public Animal
2 {
3     public:
4 };
```

```
class Sheep      size(8):
    +---+
    | {vbptr}
    +---+
    +--- (virtual base Animal)
    | data
    +---+

Sheep::$vbtable@:
0      | 0
1      | 4 (Sheepd(Sheep+0)Animal)
vbi:   class offset o.vbptr o.vbte fVtorDisp
      Animal      4      0      4 0
```

vbptr (虚基类指针) 其中v是virtual 虚 b是base 基类 prt指针

(vbptr指向虚基类表)

vbtable(虚基类表) 保存了当前的虚指针相对于虚基类的首地址的偏移量

```
1 class Tuo:virtual public Animal
2 {
3     public:
4 };
```



```

class Tuo      size(8):
    +---
    0      | {vbptr}
    +---
    +--- (virtual base Animal)
    4      | data
    +---

Tuo::$vtable@:
    0      | 0
    1      | 4 (Tuod(Tuo+0)Animal)
vbi:      class offset o.vbptr  o.vbte fVtorDisp
          Animal 4 0 4 0

```

总结：之所以产生 vbptr和vtable 目的 保证 不管多少个继承 虚基类 的数据只有一份。

```

1  class SheepTuo:public Sheep,public Tuo
2  {
3  public:
4  };

```

```

class SheepTuo  size(12):
    +---
    0      | +--- (base class Sheep)
    0      | | {vbptr}
    +---
    4      | +--- (base class Tuo)
    4      | | {vbptr}
    +---
    +---
    +--- (virtual base Animal)
    8      | data
    +---

```

```

SheepTuo::$vtable@Sheep@:
 0          | 0
 1          | 8 (SheepTuod(Sheep+0)Animal)

SheepTuo::$vtable@Tuo@:
 0          | 0
 1          | 4 (SheepTuod(Tuo+0)Animal)
vbi:         class  offset o.vbptr  o.vbte fVtorDisp
              Animal      8         0      4 0

```

案例1:

```

1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include<string.h>
4 using namespace std;
5
6 class Animal
7 {
8 public:
9     int data;
10 };
11
12 class Sheep :virtual public Animal
13 {
14 public:
15 };
16 class Tuo :virtual public Animal
17 {
18 public:
19 };
20
21 class SheepTuo :public Sheep, public Tuo
22 {
23 public:
24 };
25 int main(int argc, char* argv[])
26 {
27     SheepTuo st;
28     st.data = 200;
29

```

```

30 //通过Sheep的vbptr 寻找vbptr距离虚基类首地址的偏移量
31 //&st == vbptr
32 /*(int *)&st sheep 的虚基类表的起始位置
33 int off_set = (int)*((int*)(*(int*)&st) + 1);
34 cout << off_set << endl;
35
36 //通过sheep的vbptr 和 off_set定位虚基类的首地址
37 cout << ((Animal*)((char*)&st + off_set))->data << endl;
38
39 return 0;
40 }

```

注意：vsstudio中运行



注意： 虚继承只能解决**具备公共祖先的多继承**所带来的二义性问题，不能解决没有公共祖先的多继承的。

虚继承：不管继承多少次 虚基类 只有一份数据。

知识点5 【多态的概述】

1、概念

多态是c++的特征之一

多态的分类：静态多态（静态联编）、动态多态（动态联编）

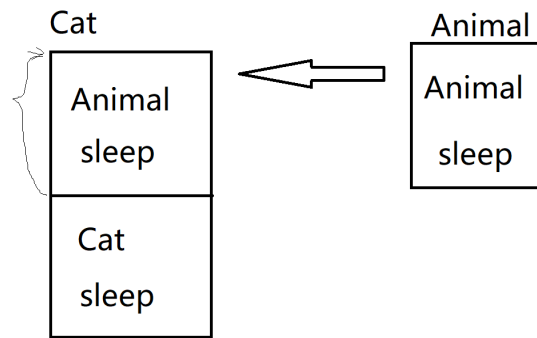
静态多态（静态联编）：函数入口地址 是在 编译阶段 确定（运算符重载、函数重载）

动态多态（动态联编）：函数入口地址 是在 运行阶段 确定（虚函数）

2、引入

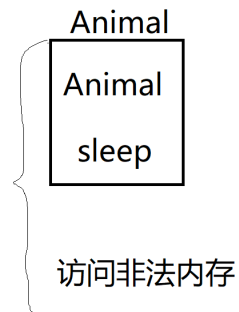
基类指针、引用 指向子类对象 (安全)

`Animal *p = new Cat`



子类指针、引用 指向基类对象 不安全

`Cat *p = new Animal;`

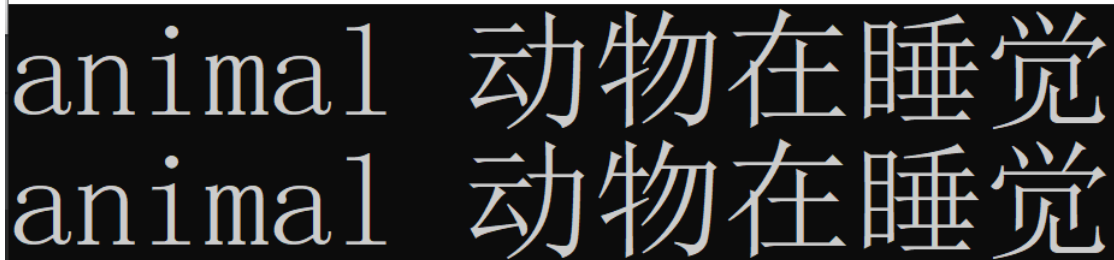


```
1  class Animal
2  {
3  public:
4  void sleep(void)
5  {
6  cout<<"animal 动物在睡觉"<<endl;
7  }
8  };
9
10 class Cat:public Animal
11 {
12 public:
13 void sleep(void)
14 {
15 cout<<"Cat 猫在睡觉!! 喵喵"<<endl;
16 }
17 };
18 void test01()
19 {
20 //用基类（指针或引用） 保存 子类对象（向上转换）
21 Animal *p = new Cat;
22 p->sleep();//调用的是基类的sleep
23
24 Cat cat;
```

```
25  Animal &ob = cat;
26  ob.sleep();//调用的是基类的sleep
27 }
```

运行结果：

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe



总结：基类指针、引用 只能访问 子类对象中 基类部分 数据

3、使用基类指针、引用 访问 子类对象中的成员方法（虚函数）

使用**virtual**修饰成员函数 该成员函数就是**虚函数**。

```
1  class Animal
2  {
3  public:
4      //虚函数
5      virtual void sleep(void)
6      {
7          cout<<"animal 动物在睡觉"<<endl;
8      }
9  };
```

```
class Animal    size(4):
```

```
    +---+
    | {vfptr}|
    +---+
```

```
Animal::$vftable@:
    |
    | &Animal_meta
    | 0
    |
    | 0
    | &Animal::sleep
```

vfptr虚函数指针 指向的是虚函数表 (vftable)

vftable表存放的是 vfptr做保存的函数入口地址

注意: //如果 Animal没有涉及到继承 函数指针变量 就指向自身sleep

```
1  class Animal
2  {
3  public:
4      //虚函数 本质 是一个函数指针变量
5      virtual void sleep(void)
6      {
7          cout<<"animal 动物在睡觉"<<endl;
8      }
9  };
10 void test01()
11 {
12     //如果 Animal没有涉及到继承 函数指针变量 就指向自身sleep
13     Animal ob;
14     ob.sleep();
15 }
```

运行结果:

animal 动物在睡觉

4、拥有虚函数的类 涉及得到继承 2-2

```
1  class Animal
2  {
3  public:
4      //虚函数 本质 是一个函数指针变量
5      virtual void sleep(void)
6      {
7          cout<<"animal 动物在睡觉"<<endl;
8      }
9  };
10 class Cat:public Animal
11 {
12 public:
13     virtual void sleep(void)
14     {
15         cout<<"猫在睡觉!!喵喵"<<endl;
16     }
17 };
```

```

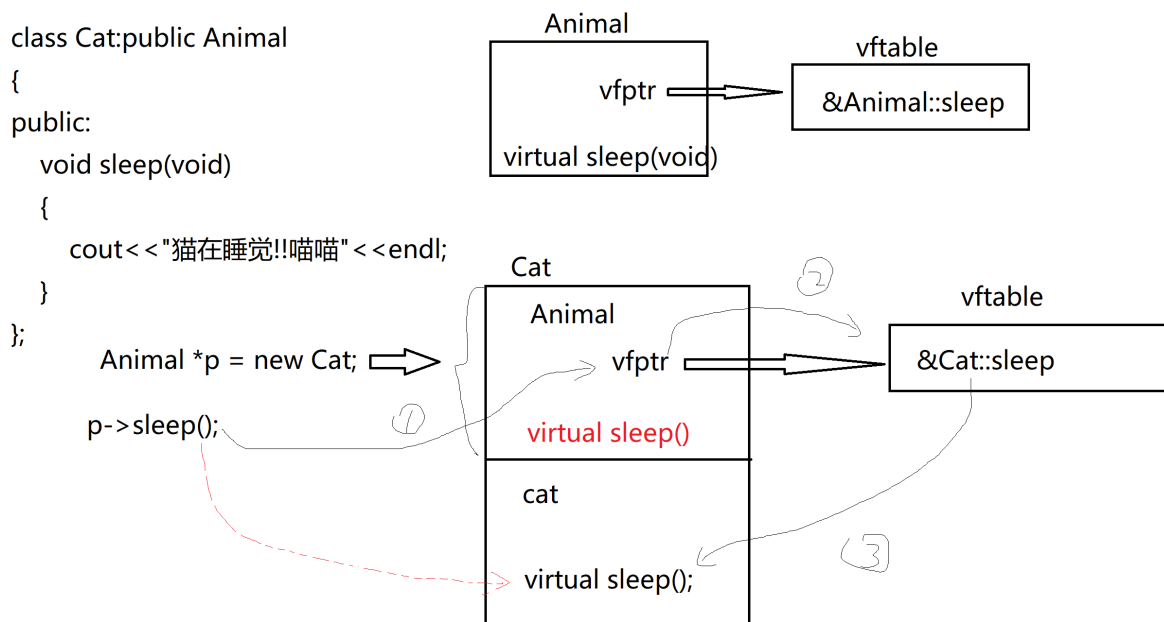
class Cat      size(4):
    +---+
    0      +---+ (base class Animal)
    0      | {vfptr}
           +---+
           +---+
Cat::$vtable@:
    &Cat_meta
    0
    0      &Cat::sleep

```

```

class Cat:public Animal
{
public:
    void sleep(void)
    {
        cout<<"猫在睡觉!!喵喵"<<endl;
    }
};

```



```

1 void test01()
2 {
3     Animal *p = new Cat;
4     p->sleep();//调用的是cat中sleep
5 }

```

运行结果：

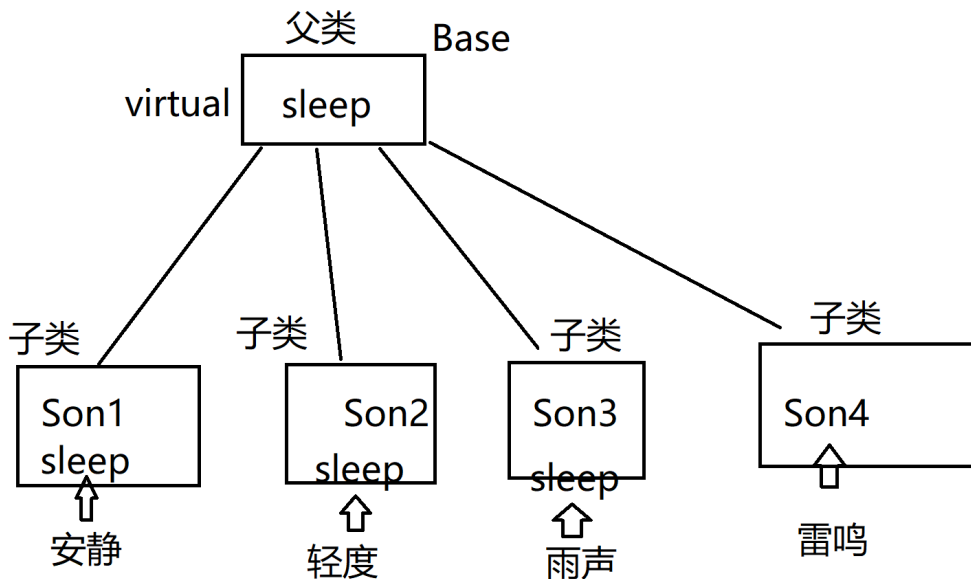
猫在睡觉!!喵喵

总结:

当虚函数涉及到继承的时候 子类 会继承 父类的 (虚函数指针vfptr 虚函数表vftable) ,编译器会将虚函数表中的函数入口地址 更新 成子类的 同名 (返回值、参数都相同) 的函数入口地址。

如果基类指针、引用 访问虚函数的时候 就会 间接的调用 子类的虚函数。

4、虚函数的应用案例 (基类指针、引用 作为函数的参数)



sleep_fun(Base &ob)

函数体:ob.sleep();

```
1 #include <iostream>
2
3 using namespace std;
4 class Base
```

```
5 {
6 public:
7     virtual void sleep(void)
8     {
9         cout<<"父亲在睡觉"<<endl;
10    }
11 };
12
13 class Son1:public Base
14 {
15 public:
16     void sleep(void)
17     {
18         cout<<"Son1在安静的睡觉"<<endl;
19     }
20 };
21 class Son2:public Base
22 {
23 public:
24     virtual void sleep(void)
25     {
26         cout<<"Son2在轻度的睡觉"<<endl;
27     }
28 };
29 class Son3:public Base
30 {
31 public:
32     virtual void sleep(void)
33     {
34         cout<<"Son3在雨声般的睡觉"<<endl;
35     }
36 };
37 class Son4:public Base
38 {
39 public:
40     virtual void sleep(void)
41     {
42         cout<<"Son4在鼾声如雷"<<endl;
43     }
44 };
```

```

45
46 //以基类指针作为函数的参数 函数可以操作该基类派生出的任意子类对象
47 void sleepFun(Base &ob)
48 {
49     ob.sleep();
50 }
51
52 int main(int argc, char *argv[])
53 {
54     Son1 ob1;
55     Son2 ob2;
56     Son3 ob3;
57     Son4 ob4;
58     sleepFun(ob1);
59     sleepFun(ob2);
60     sleepFun(ob3);
61     sleepFun(ob4);
62     return 0;
63 }
64

```

运行结果：

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe

Son1在安静的睡觉
 Son2在轻度的睡觉
 Son3在雨声般的睡觉
 Son4在鼾声如雷

问题:C++的动态捆绑机制是怎么样的?

首先,我们看看编译器如何处理虚函数。当编译器发现我们的类中有虚函数的时候,编译器会创建一张虚函数表,把虚函数的函数入口地址放到虚函数表中,并且在类中秘密增加一个指针,这个指针就是vpointer(缩写vptr),这个指针是指向对象的虚函数表。在多态调用的时候,根据vptr指针,找到虚函数表来实现动态绑定。

