知识点1【多态的概述】（了解）

知识点2【虚函数】（重要）

1、知识点引入

2、父类指针 保存 子类空间地址（带来的问题）

3、虚函数的定义：成员函数前加virtual修饰

4、虚函数原理

Animal的类的结构：

Dog的类存结构：

知识点3【纯虚函数】（重要）

问题1：虚函数 和纯虚函数的 区别

知识点4【虚析构函数】（重要）

知识点5【纯虚析构函数】（重要）

问题1：虚析构 和纯虚析构的区别?

知识点6【多态的常用问题】（重要）

1、多态的分类

2、谈谈你对动态捆绑机制的理解（虚函数实现原理）

3、重载、重定义、重写的区别

4、虚函数和纯虚函数的区别

5、虚析构和纯虚析构的区别

6、虚函数的作用

7、虚析构的作用

知识点7【重载、重定义、重写的区别】（重要）

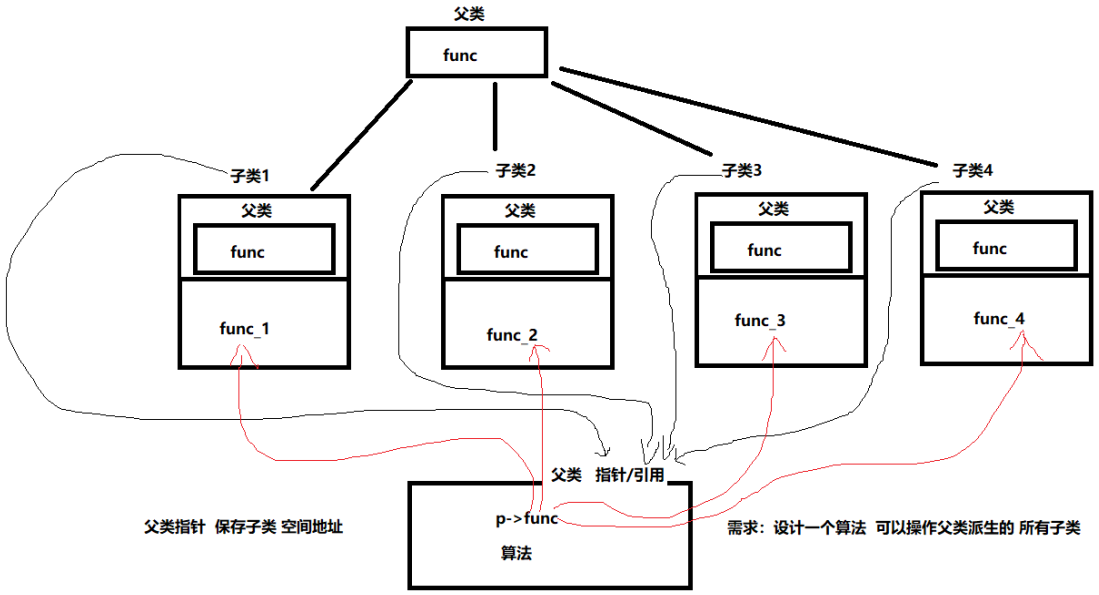
知识点1【多态的概述】（了解）

静态多态（编译时多态，早绑定）：函数重载、运算符重载

动态多态（运行时多态，晚绑定）：虚函数

知识点2【虚函数】（重要）

1、知识点引入

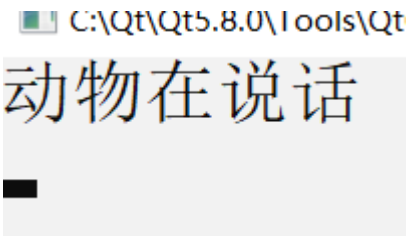


父类指针（引用）保存 子类空间地址的目的 就是让算法通用。

2、父类指针 保存 子类空间地址（带来的问题）

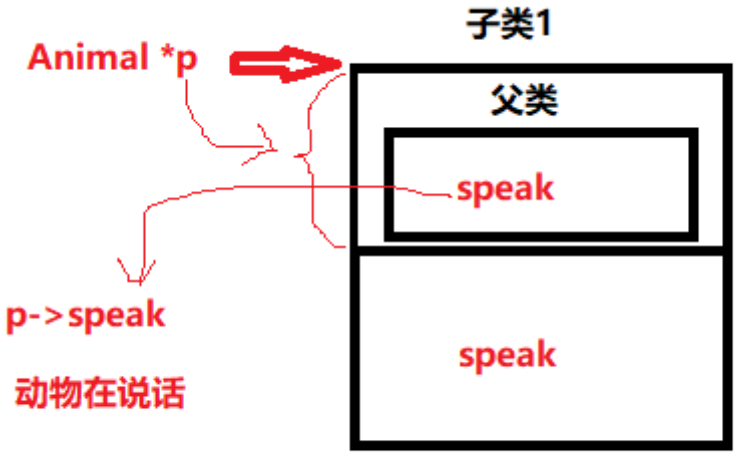
```
class Animal
{
public:
 void speak(void)
 {
 cout<<"动物在说话"<<endl;
 }
};
class Dog:public Animal
{
public:
 void speak(void)
 {
 cout<<"狗在汪汪"<<endl;
 }
};

```

```
18 void test01()
19 {
20  Animal *p = new Dog;
21  p->speak();
22 }
```



其实用户的需求：p->speak 希望等到的是“狗在汪汪” 而不是“动物在说话”。

原因在此：



3、虚函数的定义：成员函数前加virtual修饰

子类重写父类的虚函数注意：有继承、子类重写父类虚函数（函数名、返回值类型、参数类型个数顺序 必须完全一致)。

```
1  class Animal
2  {
3  public:
4   //虚函数
5   virtual void speak(void)
6   {
7   cout<<"动物在说话"<<endl;
8   }
9  };
10 class Dog:public Animal
11 {
```

```
public:
 //子类重写 父类的虚函数
 void speak(void)
 {
 cout<<"狗在汪汪"<<endl;
 }
};
class Cat:public Animal
{
public:
 //子类重写 父类的虚函数
 void speak(void)
 {
 cout<<"猫在喵喵"<<endl;
 }
};
void test01()
{
 Animal *p1 = new Dog;
 p1->speak();
 Animal *p2 = new Cat;
 p2->speak();

 delete p1;
 delete p2;
}
```

多态条件：有继承、子类重写父类的虚函数，父类指针 指向子类空间。

4、虚函数原理

Animal的类的结构：

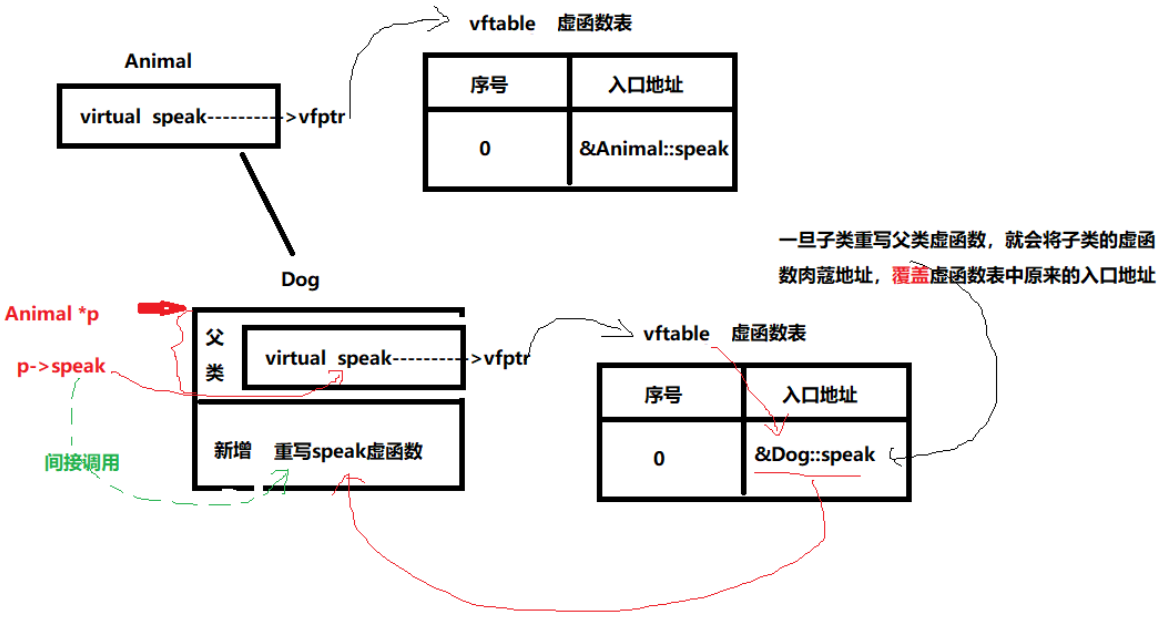
```
class Animal    size(4):
        +---
 0      | {vfptr}
        +---

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

如果一个类中的成员函数 被virtual修饰，那么这个函数就是虚函数。类就会产生一个虚函数指针（vfptr）指向了一张虚函数表（vftable）。

如果这个类 没有涉及到继承， 这时虚函数表中 纪录及时当前类的虚函数入口地址。

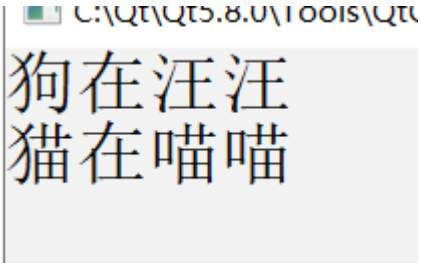
Dog的类存结构：

```
class Dog       size(4):
        +---
 0      | +--- (base class Animal)
 0      | | {vfptr}
        | +---
        +---

Dog::$vftable@:
        | &Dog_meta
        |  0
 0      | &Dog::speak
```




```
#include <iostream>

using namespace std;
class Animal
{
public:
 //虚函数
 virtual void speak(void)
 {
  cout<<"动物在说话"<<endl;
  }
};
class Dog:public Animal
{
public:
#if 1
 //子类重写 父类的虚函数
 void speak(void)
 {
 cout<<"狗在汪汪"<<endl;
  }
#endif
};
class Cat:public Animal
{
```

```
26 public:
27  //子类重写 父类的虚函数
28  void speak(void)
29  {
30  cout<<"猫在喵喵"<<endl;
31  }
32 };
33 void AnimalSpeak(Animal *p)
34 {
35  p->speak();
36  return;
37 }
38
39 int main(int argc, char *argv[])
40 {
41  AnimalSpeak(new Dog);
42  AnimalSpeak(new Cat);
43
44  return 0;
45 }
```



问题:C++的动态捆绑机制是怎么样的？（视屏转文字）

知识点3【纯虚函数】（重要）

虚函数不实现函数体：

```
1 class Animal
2 {
3 public:
4  //纯虚函数
5  virtual void speak(void)=0;
6 };
```

一旦类中有纯虚函数，那么这个类 就是抽象类。

抽象类 不能实例化 对象。（Animal ob；错误）

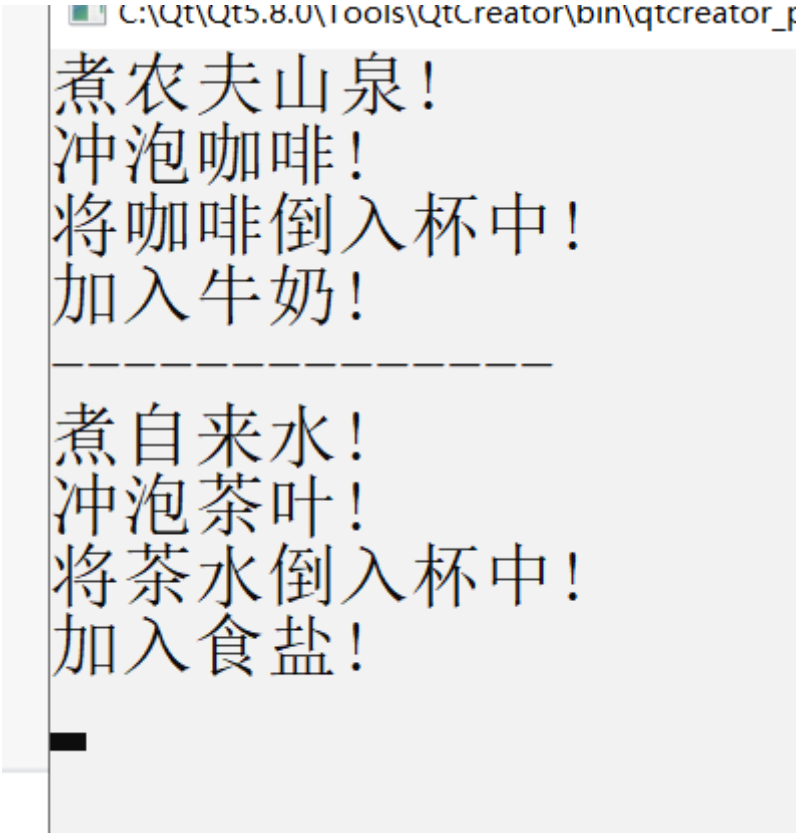
抽象类 必须被继承 同时 子类 必须重写 父类的所有纯虚函数，否则 子类也是抽象类。

抽象类主要的目的 是设计 类的接口：

```
#include <iostream>

using namespace std;
//抽象制作饮品
class AbstractDrinking{
public:
 //烧水
 virtual void Boil() = 0;
 //冲泡
  virtual void Brew() = 0;
  //倒入杯中
  virtual void PourInCup() = 0;
  //加入辅料
  virtual void PutSomething() = 0;
  //规定流程
  void MakeDrink(){
  this->Boil();
  Brew();
  PourInCup();
  PutSomething();
  }
 };

//制作咖啡
class Coffee : public AbstractDrinking{
public:
  //烧水
  virtual void Boil(){
  cout << "煮农夫山泉!" << endl;
  }
  //冲泡
  virtual void Brew(){
  cout << "冲泡咖啡!" << endl;
  }
  //倒入杯中
  virtual void PourInCup(){
  cout << "将咖啡倒入杯中!" << endl;
  }
```

```
 //加入辅料
 virtual void PutSomething(){
 cout << "加入牛奶!" << endl;
 }
};

//制作茶水
class Tea : public AbstractDrinking{
public:
 //烧水
 virtual void Boil(){
 cout << "煮自来水!" << endl;
 }
 //冲泡
 virtual void Brew(){
 cout << "冲泡茶叶!" << endl;
 }
 //倒入杯中
 virtual void PourInCup(){
 cout << "将茶水倒入杯中!" << endl;
 }
 //加入辅料
 virtual void PutSomething(){
 cout << "加入食盐!" << endl;
 }
};

//业务函数
void DoBussiness(AbstractDrinking* drink){
 drink->MakeDrink();
 delete drink;
}

int main(int argc, char *argv[])
{
 DoBussiness(new Coffee);
 cout << "--------------" << endl;
 DoBussiness(new Tea);

 return 0;
```

```
79 }
80
```



问题1：虚函数 和纯虚函数的 区别

虚函数：virtual修饰 有函数体 不会导致父类为抽象类。

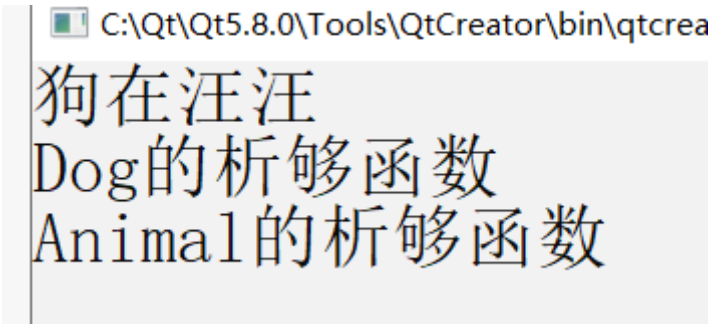
纯虚函数：virtual修饰，=0，没有函数体 导致父类为抽象类。子类必须重写父类的所有纯虚函数。

知识点4【虚析构函数】（重要）

虚析构：通过父类指针 释放整个子类空间。

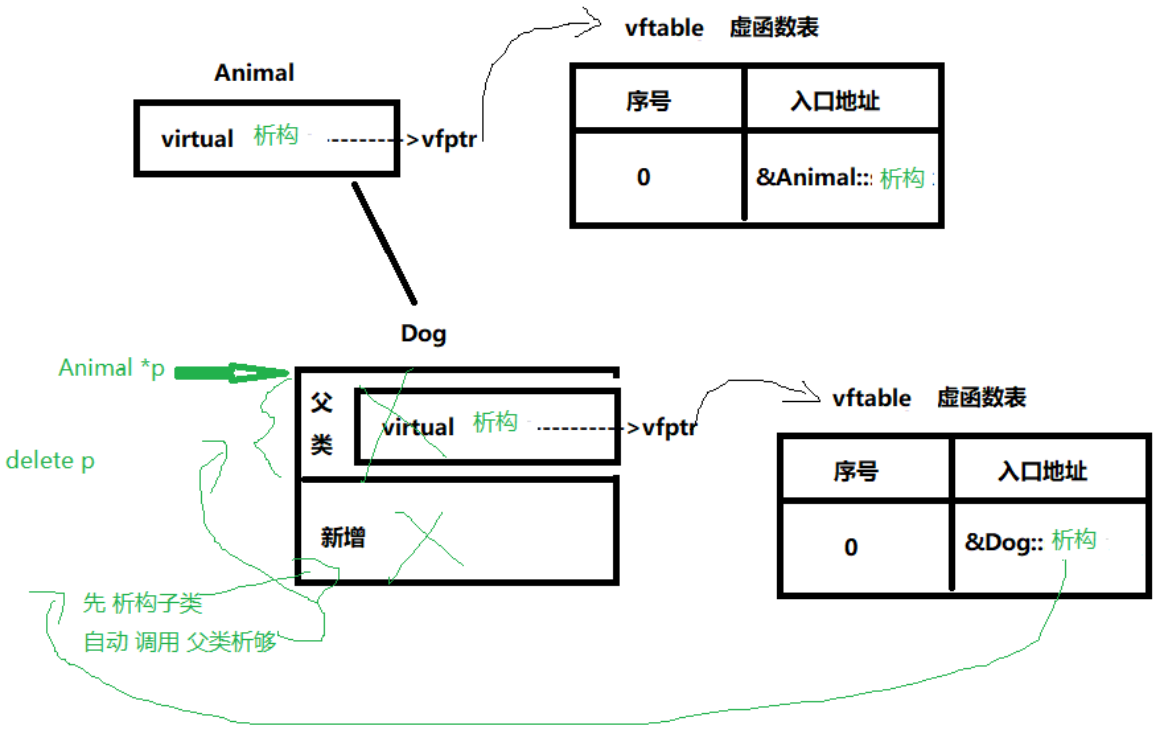
```
1  class Animal
2  {
3  public:
4   //虚函数
5   virtual void speak(void)
6   {
7   cout << "动物在说话" << endl;
8   }
9   //虚析构
10   virtual ~Animal()
11   {
12   cout<<"Animal的析构函数"<<endl;
```

```
 }
};
class Dog :public Animal
{
public:
 //子类重写 父类的虚函数
 void speak(void)
 {
 cout << "狗在汪汪" << endl;
 }
 ~Dog()
 {
 cout<<"Dog的析构函数"<<endl;
 }
};

void test01()
{
 Animal* p1 = new Dog;
 p1->speak();

 delete p1;
}
```



构造的顺序：父类--->成员---->子类

析构的顺序：子类--->成员---->父类



知识点5【纯虚析构函数】（重要）

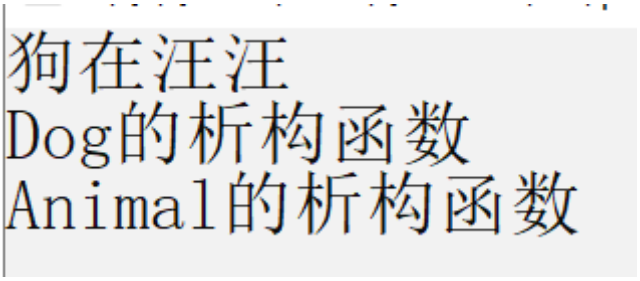
纯虚析构的本质：是析构函数，各个类的回收工作。而且析构函数不能被继承。

必须为纯虚析构函数提供一个函数体。

纯虚析构函数 必须在类外实现

```
#include <iostream>

using namespace std;
class Animal
{
public:
 //纯虚函数
 virtual void speak(void)=0;

 //纯虚析构函数 必须在类外实现
 virtual ~Animal()=0;
};
class Dog :public Animal
{
public:
 //子类重写 父类的虚函数
 void speak(void)
 {
 cout << "狗在汪汪" << endl;
```

```
 }
 ~Dog()
 {
 cout<<"Dog的析构函数"<<endl;
 }
};

void test01()
{
 Animal* p1 = new Dog;
 p1->speak();

 delete p1;

}

int main(int argc, char* argv[])
{
 test01();
 return 0;
}

Animal::~Animal()
{
 cout<<"Animal的析构函数"<<endl;
}
```



问题1：虚析构 和纯虚析构的区别?

虚析构：virtual修饰，有函数体，不会导致父类为抽象类。

纯虚析构：virtual修饰，=0，函数体必须类外实现，导致父类为抽象类。

知识点6【多态的常用问题】（重要）

1、多态的分类

2、谈谈你对动态捆绑机制的理解（虚函数实现原理）

3、重载、重定义、重写的区别

4、虚函数和纯虚函数的区别

5、虚析构和纯虚析构的区别

6、虚函数的作用

7、虚析构的作用

知识点7【重载、重定义、重写的区别】（重要）

重载：同一作用域，同名函数，参数的顺序、个数、类型不同 都可以重载。函数的返回值类型不能作为重载条件（函数重载、运算符重载）

重定义：有继承，子类 重定义 父类的同名函数（非虚函数）， 参数顺序、个数、类型可以不同。子类的同名函数会屏蔽父类的所有同名函数（可以通过作用域解决）

重写（覆盖）：有继承，子类 重写 父类的虚函数。返回值类型、函数名、参数顺序、个数、类型都必须一致。