第5章-多态

author：岳石磊 copyright：科林明伦 内部资料禁止外泄

1.多态的概念

多态：相同的行为方式可能导致不同的行为结果，即产生了多种形态行为，即多态，同一行语句展现了多种不同的表现形态，多态本质；定义父类的指针可以指向任何继承于该类的子类的对象，且父类的指针具有子类对象的行为，多种子类表现为多种形态由父类的指针进行统一，那么这个父类额指针就具有了多种形态。

C++ 直接支持多态条件：

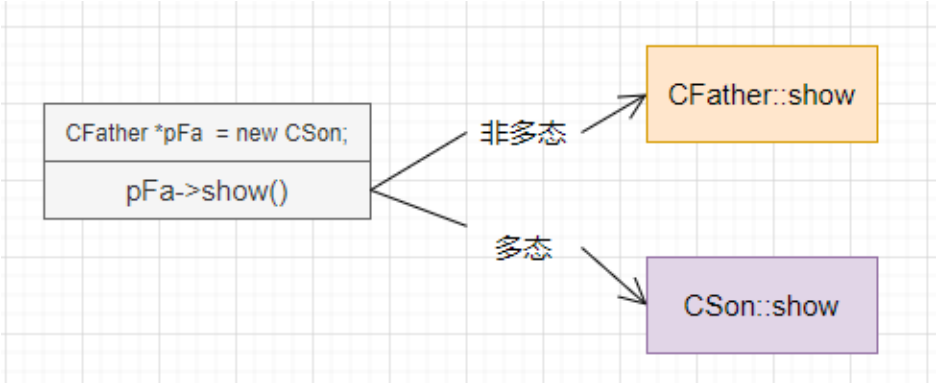
1. 必须存在继承关系，这是前提，存在父类类型的指针指向某个子类，通过该指针调用虚函数。
2. 父类中存在虚函数（virtual修饰），且子类中重写了父类的虚函数。

举例：

```
class CFather{
public:
    virtual void show(){  //虚函数
        cout<<"CFather::show"<<endl;
    }
};

class CSon:public CFather{ //继承
public:
    virtual void show(){   //重写了父类的虚函数，  即使省略 virtual 也会认为是虚函数
        cout<<"CSon::show"<<endl;
    }
};

CFather *pFa = new CSon;  //父类的指针指向子类对象
pFa->show();              //CSon::show  ，调用的是子类的函数
```



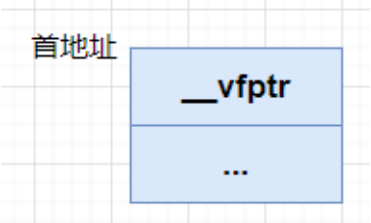
重写： 在继承条件下，子类定义了与父类中虚函数一模一样的函数（包括：函数名、参数列表、返回值）我们称之为重写。如上例中的show函数。

2. 虚函数-虚函数指针-虚函数列表

定义虚函数使用关键字 virtual 。虚函数是实现多态必不可少的条件之一。

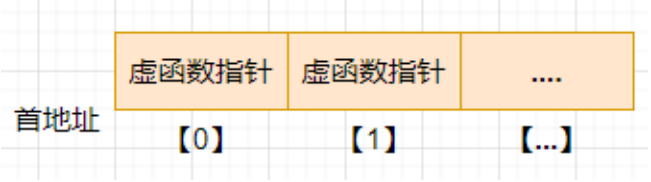
```
1  virtual void fun();
```

虚函数指针（__vfptr）：当一个类中定义了虚函数时，在定义对象的内存空间的首地址会多分配出一块内存，标识这块内存的变量称之为虚函数指针(__vfptr)。



- 属于对象的，定义对象时才存在（内存空间得以分配），多个对象多份指针。
- 是一个二级指针，类型为 void**。
- 在构造的初始化参数列表中进行初始化（编译器默认完成）。
- 指向了一个函数指针数组（虚函数列表，vftable）。
- 每个对象中的虚函数指针指向了同一个虚函数列表。

虚函数列表（vftable）：是一个函数指针数组，数组每个元素为类中虚函数的地址。

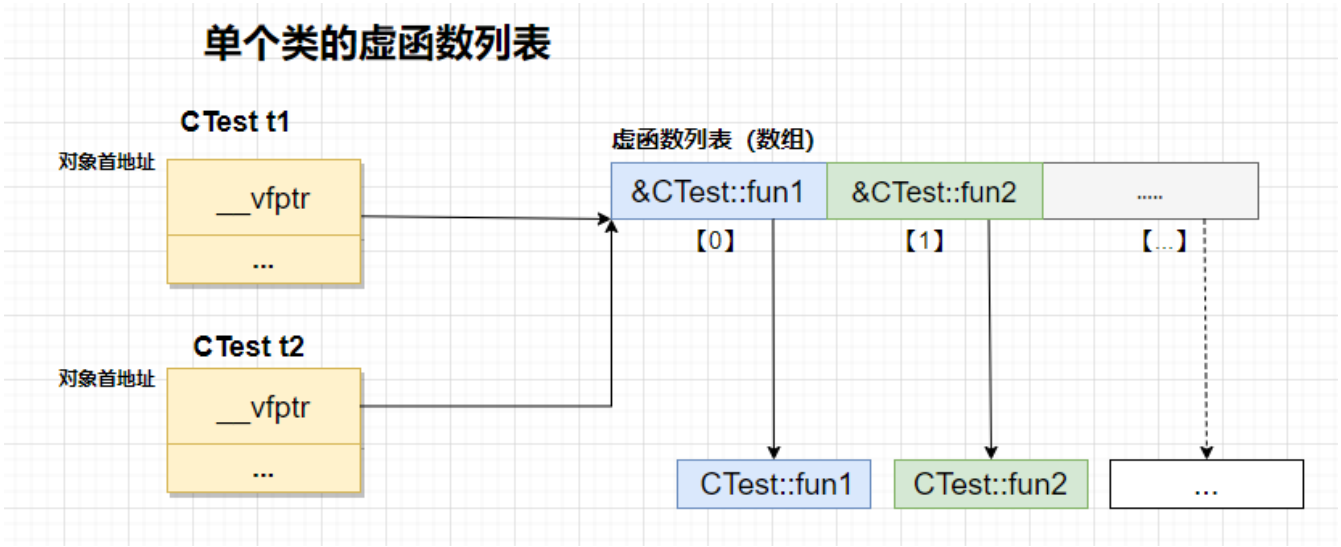


- 属于类的，在编译期存在，为所有对象共享。
- 必须通过真实存在的对象调用，无对象或空指针对象无法调用虚函数。

虚函数调用流程：

1. 定义对象获取对象内存首地址中的__vfptr。
2. 通过__vfptr指针间接引用到虚函数列表vftable。

3. 在虚函数列表中定位到其虚函数地址，通过地址调用真正的虚函数。



虚函数于普通成员函数的区别：

- 调用流程不同：虚函数的调用流程相比普通函数而言复杂的多，这是他们的本质区别。
- 调用效率不同：普通的成员函数通过函数名（即函数入口地址）直接调用执行函数，效率高速度快，虚函数的调用需要虚函数指针-虚函数列表的参与，效率低，速度慢。
- 使用场景不同：虚函数主要用于实现多态，这一点是普通的函数无法做到的。

3. 多态实现的原理

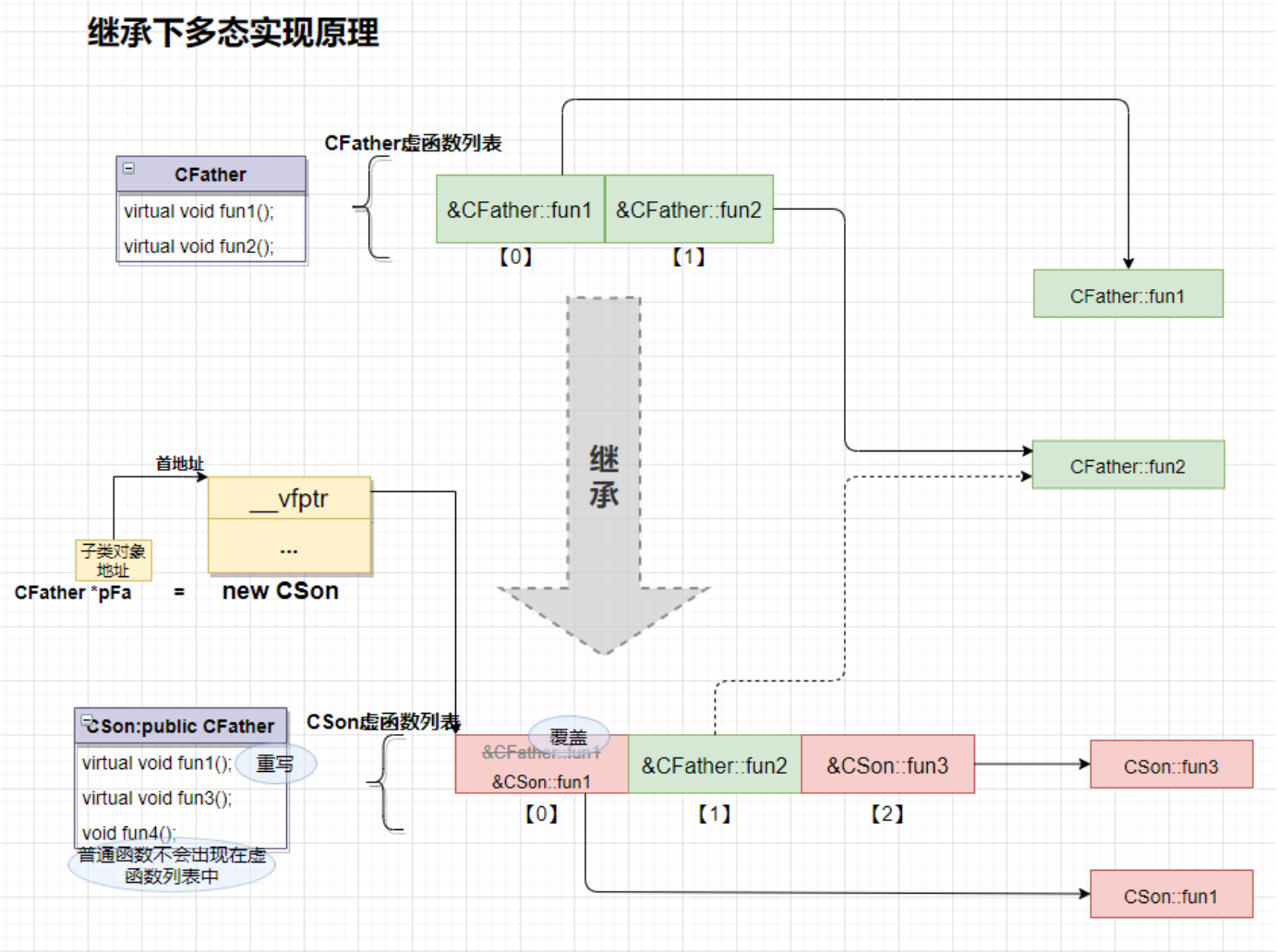
前提：虚函数列表是属于类的，父类和子类都会有各自的虚函数列表，`__vfptr`属于对象的，每个对象都有各自`__vfptr`。

原理：

1. 由于子类继承父类，不但继承了父类的成员，也会继承父类的虚函数列表。
2. 编译器会检查子类是否有重写父类的虚函数，如果有，在子类的虚函数列表中会替换掉父类的虚函数，一般称之为**覆盖**，覆盖后便指向了子类的虚函数。
3. 如果子类没有重写的父类虚函数，父类虚函数会保留在子类的虚函数列表中。
4. 如果子类定义了独有的虚函数，按顺序依次添加到虚函数列表结尾。
   以上这些过程在编译阶段就完成了。

流程：父类指针指向子类对象，`__vfptr`在子类的初始化参数列表中被初始化，指向子类的虚函数列表，**申请哪个子类对象`__vfptr`就指向了哪个子类的虚函数列表。**调用虚函数时执行虚函数的调用流

程，则实现了多态。



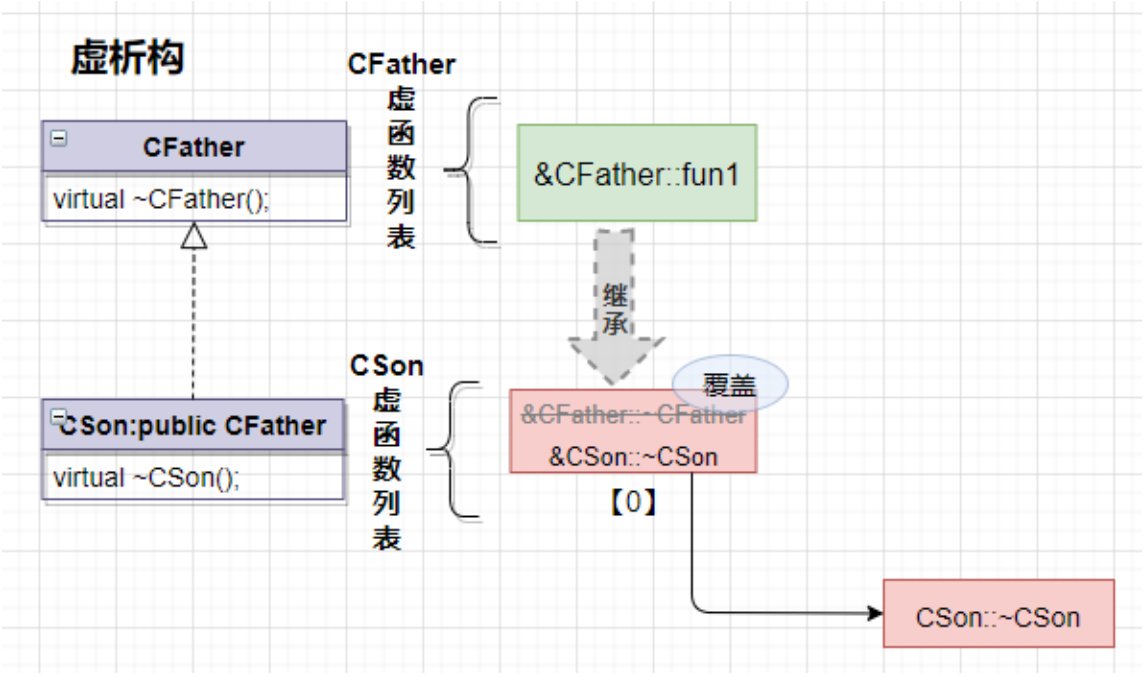
4. 虚析构

在多态下，父类的指针指向子类的对象，最后在回收空间的时候，是按照父类的指针类型delete的，所以只调用了父类的析构，子类的析构并没有执行，这样的话就有可能导致内存泄漏。

```
CFather* pFa = new CSon;
delete pFa;  //父类的指针，调用父类的析构，相当于  pFa->~CFather();
```

这个问题用**虚析构**来解决，即把父类的析构函数变为虚析构函数，`delete pFa;`时，调用析构会发生**多态行为**，从而真正调用的是子类的析构，最后回收对象内存空间时，再调用父类的析构。

```
class CFather{
    virtual ~CFather();  //虚析构
};
class CSon:public CFather{
    ~CSon();     //即使不加关键字virtual，子类的析构函数默认也为虚析构了
};
```



在用多态时，父类的析构一定为虚析构。

5. 纯虚函数

在多态下，有时抽象出来的父类的虚函数作为接口函数，并不知道如何实现或不需要实现就是为了多态而生的，只有继承的子类才明确如何实现，可以把父类的虚函数变为**纯虚函数**。
纯虚函数的特点是：当前类不必实现，而子类必须要重写实现纯虚函数。

```
//抽象类
class CFather{
    virtual void show()=0;  //不必实现
};
//具体类
class CSon:public CFather{
    virtual void show(){    //子类一定重写并实现
        ...
    };
};
```

包含纯虚函数的类叫**抽象类**，抽象类不能实例化对象，继承这个抽象类的派生类叫**具体类**，具体类必须重写定义抽象类的里面的所有的纯虚函数。

6. 多态的缺点

多态缺点：

1. 效率：调用虚函数效率低，速度慢。
2. 空间：虚函数指针占用空间，多个对象会有多个虚函数指针，虚函数列表会随着继承的层级递增 虚表大小只增不减。

3. 安全：类中的私有的函数，不能为虚函数，否则会有安全隐患。

```
class CFather {
private:
    virtual void fun() = 0;
};
class CSon :public CFather {
private:
    virtual void fun() {
        cout << "CSon::fun" << endl;
    }
};
//用自己的方法，可以在类外调用虚函数，存在安全隐患。
CFather* pFa = new CSon;
void (*p_fun)() = (void (*)())(((int*)(*(int*)pFa))[0]);
(*p_fun)();   //CSon::fun
```