2020/05/21_C++_第12课_虚函数补充、默认构造和析构、抽象类、多重继承

笔记本: C++
创建时间: 2020/5/21 星期四 15:30
作者: ileemi
标签: 抽象类, 多重继承, 默认构造和析构, 虚函数补充

虚函数补充

多态在父类成员函数中的表现:

1. 父类一般成员函数中调用虚函数, 有多态效果
当父类和子类存在虚函数时，在父类中一般成员函数调用虚函数，会产生多态效果，会调用子类的虚函数
编译器会默认添加this指针，通过this指针调用，从对象的首4个字节取出虚表指针，然后通过虚表指针去虚表中拿去对应函数的地址，拿到地址调用对应的函数。

```
class A
{
public:
  void Test()
  {
    //等同于VirFunc()
    this->VirFunc();  //调用子类
    /*
    通过this指针拿取虚表指针，然后通过虚表指针从虚表中取出对应的函数地址
    1、当对象创建的时候，通过子类去调用父类的构造
    2、父类构造会把虚表指针填写成自己的
    3、到子类时，子类构造会把虚表指针填写成自己的
    4、对于子类对象而言，这个虚表对应的就是子类的
    */
  }
  virtual void VirFunc()
```

```
    {
        cout << "A::VirFunc" << endl;
    }
};

class B :public A
{
public:
    virtual void VirFunc()
    {
        cout << "B::VirFunc" << endl;
    }
};

int main()
{
    A* pA = new B;
    //pA->VirFunc();

    pA->Test();

    B b;

    return 0;
}
```

2、父类构造时，调用虚函数，不产生多态效果，此时虚表指针指向父类的虚表
3、父类析构时，调用虚函数，不产生多态效果，析构中父类析构会回填虚表，将虚表指针回填成自己的

```
class A
{
public:
    A()
    {
        /*
        当基类A 构造函数创建的时候虚表指针才来
        此时虚表指针指向父类的虚表
        */
        this->VirFunc();
    }
    ~A()
    {
        //析构中会回填虚表，将虚表指针回填成自己的
        this->VirFunc();
    }
    void Test()
```

```
    {
      this->VirFunc();
    }
    virtual void VirFunc()
    {
      cout << "A::VirFunc" << endl;
    }
};

class B :public A
{
public:
    B()
    {
      //此时虚表指针指向子类的虚表
      VirFunc();
    }
    ~B()
    {
    /*
    此时虚表指针指向子类的虚表
    接着程序会去父类运行父类的析构
    */
      VirFunc();
    }
    virtual void VirFunc()
    {
      cout << "B::VirFunc" << endl;
    }
};

int main()
{
    A* pA = new B;
    //pA->VirFunc();

    pA->Test();

    B b;

    return 0; //此时调用析构，先调用子类，然后调用父类
}
```

构造和析构可否成为虚函数?

1、构造不能成为虚函数（没有必要）构造用来初始化自己类中的数据成员

2、析构：父类的析构需要是虚析构

```
/*
父类虚表
------
A::~A
------
*/
class A
{
public:
    A()
    {
    }
    virtual ~A()
    {
        //析构中会回填虚表
        cout << "A::~A" << endl;
        /*
        调用析构时，子类通过虚表走完后，调用父类的析构，
        这个时候没有走虚表，直接拿父类的构造地址来调用，
        所以当调用完父类的析构后，就不会在去调用子类的析构
        */
    }
};

/*
子类虚表
------
B::~B
------
*/
class B :public A
{
public:
    B()
    {
    }
    ~B()
    {
    //此时虚表指针指向子类的虚表
    cout << "B::~B" << endl;
    }
};

int main()
{
```

```
    //父类的指针指向子类对象 new子类对象
    A* pA = new B

    //如果父类析构不是虚函数，delete时只能调用父类的析构，因为子类的析构已经进行的释放
    //一切的多态行为都会出发这种情况

    delete pA;    //下断点，此时输出父类析构中的数据
    /*
    同过指针调父类的析构，然后进行释放
    pA->A::~A();
    free(pA)
    */

    //此时编译器不知道以后会有哪些子类继承父类
    /*
    此时，如果想让析构按照顺序使类调用自己的析构
    在父类的析构前添加关键字  virtual
    */

    return 0;
}
```

父类的指针指向子类的对象，再进行释放指针时：

每个类再进行析构的时候会将虚表指针填写成自己的虚表地址
一般情况下，析构都写成虚析构

父类的指针指向子类对象 new子类对象

析构时先析构子类，就是因为虚表在开始析构的时候是子类虚表，析构完子类以后，进入父类析构，不走虚表，直接拿父类的虚表指针

实例：

```
class CHero
{
public:
    CHero()
    {
        cout << "CHero::CHero" << endl;
    }
    //这里再没有添加virtual关键字的时候，释放空间的时候，类对象只会释放父类的析构
    virtual ~CHero()
    {
        cout << "CHero::~CHero" << endl;
    }
};
```

```
class CWarrior:public CHero
{
public:
    CWarrior()
    {
        cout << "CWarrior::CWarrior" << endl;
    }
    ~CWarrior()
    {
        cout << "CWarrior::~CWarrior" << endl;
    }
};

class CMage :public CHero
{
public:
    CMage()
    {
        cout << "CMage::CMage" << endl;
    }
    ~CMage()
    {
        cout << "CMage::~CMage" << endl;
    }

private :
    //再父类的析构没有添加virtual 关键字的时候，此时这块内存不会得到释
放
    char* m_p;
};

void Clear(CHero* paryHeros[], int nCount )
{
    for (size_t i = 0; i < nCount; i++)
    {
        delete paryHeros[i];
     }
}

int main()
{
      CHero* aryHeros[] = {
        new CWarrior,
        new CMage,
        new CWarrior,
        new CWarrior,
        new CMage,
```

```
            new CWarrior,
            new CWarrior,
            new CMage
        };

        //各种使用

        //清理
        Clear(aryHeros, sizeof(aryHeros)/sizeof(aryHeros[0]));

        return 0; //析构时应该调用各自对用的析构，再父类的析构前添加关键字 virtual
}
```

默认构造和析构

默认构造

1、编译器产生构造的时机以及相关情况

- 当类没有任何构造的时候, 编译器会自动提供一个默认构造（没有参数，再VS2019下不产生代码）
- 当类有构造的时候, 编译器不会再提供默认构造
- 当父类存在构造，子类没有构造的时候，编译器同样会对子类生成一个构造（此时的构造是产生代码的）

2、编译会生成默认构造的情况

- 父类有默认构造, 子类没有构造
- 类中有虚函数，没有默认构造的时候
- 类对象作为成员, 成员有默认构造

默认析构

1、父类有析构,子类没有析构, 编译器会提供一个析构
2、成员有析构, 自己没有析构, 编译器会提供一个析构
3、有虚函数

抽象类（纯虚函数）

纯虚函数：（解决基类不实例化对象）
纯虚函数不能实例化基类对象

纯虚函数 - 函数声明后加 =0, 告诉编译器，函数没有主体
1、含有纯虚函数的类叫做抽象类, 抽象类不能实例化对象
2、如果子类没有重写父类的纯虚函数, 那么子类也是抽象类

规范化接口:
强制子类实现某些接口(成员函数)

虚函数和纯虚函数：

虚函数：

虚函数 是在基类中使用关键字 virtual 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数。我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为动态链接，或后期绑定。

纯虚函数：
您可能想要在基类中定义虚函数，以便在派生类中重新定义该函数更好地适用于对象，但是您在基类中又不能对虚函数给出有意义的实现，这个时候就会用到纯虚函数。

```
class CAnimal
{
public:
    virtual void Speak()=0;
    virtual void Eat()=0;
};

class CDog :public CAnimal
{
public:
    virtual void Speak() { cout << "汪汪" << endl; }
    virtual void Eat() { cout << "啃骨头" << endl; }
};

class CCat :public CAnimal
{
public:
    virtual void Speak() { cout << "喵喵" << endl; }
    virtual void Eat() { cout << "吃鱼" << endl; }
};

class A
{
public:
    virtual void Test() = 0;
};

int main()
{
```

```
    //纯虚函数不能实例化基类对象
    //A* pATest = new A;
    //A a;

    CAnimal* pA = new CDog;
    pA->Speak();
    pA->Eat();

    CAnimal* pA0 = new CCat;
    pA0->Speak();
    pA0->Eat();


    //在没有进行抽象类（虚函数）前，可以new一个父类
    //为了限制这种情况，可以在父类中对虚函数进行初始化值 在函数 () 后的添加 " = 0;"

    //CAnimal* pA1 = new CAnimal;
    //pA1->Speak();
    //pA1->Eat();

    return 0;
}
```

3、强制子类实现父类提供的接口（虚函数）

多重继承

语法：

一个派生类可以继承多个基类，基类之间需要用逗号隔开

多重继承类的构造和析构的顺序

1、构造：按照继承顺序，先构造父类,再构造子类
2、析构：与构造的顺序相反
3、内存的分布：先父类，再子类，父类按照继承顺序依次排布

命名冲突：

当两个或多个基类中有同名的成员时，如果直接访问该成员，就会产生命名冲突，编译器不知道使用哪个基类的成员。这个时候需要在成员名字前面加上类名和域解析符 `::` ，以显式地指明到底使用哪个类的成员，消除二义性。

```
class CSofa
{
public:
    CSofa() { cout << "CSofa::CSofa()" << endl; }
```

```
    ~CSofa() { cout << "CSofa::~CSofa()" << endl; }
    void Sit() { cout << "坐沙发" << endl; }
    int m_nSVal = 0xAAAAAAAA;
    int m_nColor = 5;   //颜色  此时和CBed的m_nColor数据名相同，应该在进行提取抽象，进行继承
};
class CBed
{
public:
    CBed() { cout << "CBed::CBed()" << endl; }
    ~CBed() { cout << "CBed::~CBed()" << endl; }
    void Sleep() { cout << "床上睡觉" << endl; }
    int m_nBVal = 0xBBBBBBBB;
    int m_nColor = 6;   //颜色
};
class CSofaBed: public CSofa, public CBed
{
public:
    CSofaBed() { cout << "CSofaBed::CSofaBed()" << endl; }
    ~CSofaBed() { cout << "CSofaBed::~CSofaBed()" << endl; }
    int m_nFBVal = 0xDDDDDDDD;
};

int main()
{
    CSofaBed fb;
    fb.Sit();
    fb.Sleep();

    CSofa* pS = &fb;//安全
    pS->m_nSVal = 0x87654093;

    CBed* pB = &fb;//安全，指针会自动转换到CBed数据的地址
    pB->m_nBVal = 0x66666666;

    //fb.m_nColor = 8; //访问不明确，编译器不知道要访问的是哪个父类的成员
    fb.CSofa::m_nColor = 8; //指明成员所属

  return 0;
}
```

对上面的 CSofa 类和 CBed 类存在相同名字的数据成员，进行提取抽象，使 CSofa 和 CBed 继承抽象出来的类。

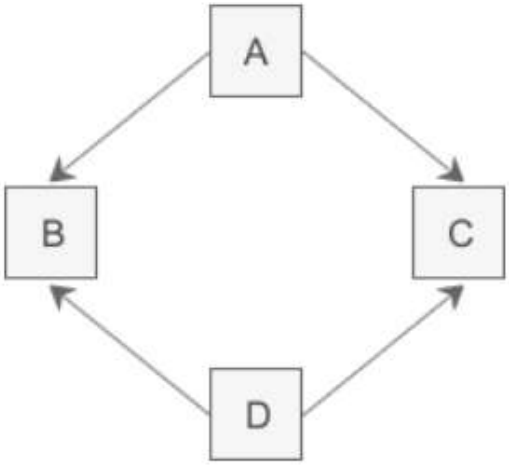
```
class CFurniture  //家具
{
```

```
public:
    CFurniture() { cout << "CFurniture::CFurniture()" << endl; }
    ~CFurniture() { cout << "CFurniture::~CFurniture()" << endl; }
    int m_nColor = 5;
};
//当没有添加 virtual 关键字的时候，访问 祖类 CFurniture 中的数据成员仍会包访问不明确的数据
class CSofa : virtual public CFurniture
{
public:
    CSofa() { cout << "CSofa::CSofa()" << endl; }
    ~CSofa() { cout << "CSofa::~CSofa()" << endl; }
    void Sit() { cout << "坐沙发" << endl; }
    int m_nSVal = 0xAAAAAAAA;
    int m_nColor = 5;   //颜色  此时和CBed的m_nColor数据名相同，应该在进行提取抽象，进行继承
};
class CBed : virtual public CFurniture
{
public:
    CBed() { cout << "CBed::CBed()" << endl; }
    ~CBed() { cout << "CBed::~CBed()" << endl; }
    void Sleep() { cout << "床上睡觉" << endl; }
    int m_nBVal = 0xBBBBBBBB;
    int m_nColor = 6;   //颜色
};
class CSofaBed: public CSofa, public CBed
{
public:
    CSofaBed() { cout << "CSofaBed::CSofaBed()" << endl; }
    ~CSofaBed() { cout << "CSofaBed::~CSofaBed()" << endl; }
    int m_nFBVal = 0xDDDDDDDD;
};

int main()
{
    CSofaBed fb;
    fb.m_nColor = 8; //指明成员所属

  return 0;
}
```

菱形继承（钻石继承）



菱形继承

多继承的派生类继承了所有父类的成员。尽管概念上非常简单，但是多个基类的相互交织可能会带来错综复杂的设计问题，命名冲突就是不可回避的一个。

多继承时很容易产生命名冲突，即使我们很小心地将所有类中的成员变量和成员函数都命名为不同的名字，命名冲突依然有可能发生，比如典型的是菱形继承

存在问题：两个父类分别从祖宗类继承一份数据, 子类从两个父类各自继承一份数据, 子类就会有祖宗类的两份数据

为了解决多继承时的命名冲突和冗余数据问题，C++ 提出了虚继承，使得在派生类中只保留一份间接基类的成员

使用：

在继承方式前面加上 virtual 关键字就是虚继承

虚继承 - 解决菱形继承的问题
构造: 先祖宗, 再父类, 再子类
析构: 与构造的顺序相反

成员内存分布：
父类1
父类2
子类
祖类

不建议经常使用
使用场景：当父类后继承一个抽象类和非抽象的时候