C++程序设计基础 第九章

前言

本文档由 @ItsJiale 创作，作者博客：https://jiale.domcer.com/，作者依据数学与大数据学院 2024 级大数据教学班的授课重点倾向编写而成。所有内容均为手动逐字录入，其中加上了不少自己的理解与思考，耗费近一周时间精心完成。

此文档旨在助力复习 C++ 程序设计基础，为后续学习数据结构筑牢根基。信计专业的同学，也可参考本文档规划复习内容。需注意，若个人学习过程中存在不同侧重点或对重难点的理解有差异，应以教材内容为准。倘若文档内容存在任何不妥之处，恳请各位读者批评指正。

By：ItsJiale

2025.5.4

第九章 多态与虚函数

在本章开始之前，请大家回忆一下，面向对象编程的三大特征——封装、继承、多态。前面的章节中，我们介绍了封装与继承，现在我们来介绍多态。

9.1 多态概述

C++中的多态分为静态多态与动态多态。**静态多态**就是指函数重载（***比如运算符重载***），在编译阶段就能确定调用哪个函数。**动态多态**是由继承产生的，指同一个属性或行为在基类以及其各派生类中具有不同语义（***同名函数，函数体不一样***），不同的对象根据所接收的消息做出不同的响应，这种现象被称为动态多态。面向对象程序设计中所说的多态***通常是指***动态多态。

在C++中，“消息”就是对类的成员函数的调用，不同的行为代表函数的不同实现方式，因此，多态的本质是函数的多种实现形式。

多态的实现需要满足3个条件（动态多态）：

①基类声明为虚函数

②派生类重写基类的虚函数（函数重写）

③将基类指针指向派生类对象，***通过基类指针访问虚函数***

且必须公有继承

9.2 虚函数实现多态

如果基类与派生类中有同名成员函数，根据类型兼容规则（***也就是派生类对象能够被当作基类对象使用，详细见第八章***），当使用基类指针或者基类引用操作派生类对象时，只能调用基类的同名函数。如果想要使用基类指针或基类引用调用派生类中的成员函数，就需要虚函数解决，***虚函数是实现多态的基础。***

9.2.1 虚函数

虚函数的声明方式是在成员函数的返回值类型前添加 `virtual` 关键字，格式如下：

```
class 类名{
权限控制符：
    virtual 函数返回值类型 函数名（参数列表）；
    ......//其他成员
};
```

声明虚函数时，需要注意：

①构造函数不能声明为虚函数，但析构函数可以声明为虚函数。

②虚函数不能是静态成员函数。（*静态成员函数是被* `static` *修饰、属于类本身而非单个对象实例、既能直接用类名调用又可操作类的静态成员变量和调用其他静态成员函数的类成员函数。*）

③友元函数不能声明为虚函数，但虚函数可以作为另一个类的友元函数。*（复习：友元函数不是类的成员函数）*

虚函数只能是类的成员函数，不能将类外的普通函数声明为虚函数，即 `virtual` 关键字只能修饰类中的成员函数，不能修饰类外的普通函数。***因为虚函数的作用是让派生类对虚函数重新定义***，它只能存在于类的继承层次结构中。

若类中声明了虚函数，并且派生类重新定义了虚函数，当使用基类指针或基类引用操作派生类对象调用函数时，系统会自动调用派生类中的虚函数来替代基类函数，***即此时派生类不再发布会基类对象的作用。***

```cpp
#include <iostream>
using namespace std;
class B0 {
public:
  virtual void display() {   //虚函数 让派生类对虚函数重新定义
        cout << "B0.display()" << endl;
    }
};

class B1 :public B0 {
public:
    void display() {
        cout << "B1.display()" << endl;
    }
};

class D1 :public B1 {
public:
    void display() {
        cout << "D1.display()" << endl;
    }
};

void fun(B0* p) {
    p->display();
}
int main()
{
    B0 b0;
    B1 b1;
    D1 d1;
    B0* pt;
    pt = &b0;
    fun(pt);
    pt = &b1;
    fun(pt);
    pt = &d1;
    fun(pt);
    return 0;
}
```

这是第八章类型兼容的例子，当时是派生类对象发挥基类对象的作用。而现在基类成员函数添加了 `virtual` 关键字，进而***使得派生类不再发挥基类对象的作用，而是发挥派生类本身的作用。***

需要注意的是，派生类对基类虚函数重写时，必须与基类中虚函数的原型完全一致，派生类中重写的虚函数前提是否添加 `virtual` ，均被视为虚函数。

补充：

1. override

`override` 关键字的作用是检查派生类中函数是否在重写基类虚函数，如果不是重写基类虚函数，编译器就会报错。

2. final

`final` 关键字有两种用法：修饰类、修饰虚函数。当使用 `final` 关键字修饰类时，表示该类不可以被继承。

```
class Base final{    //final修饰类，Base类不能被继承
}
```

9.2.2 虚函数实现多态的机制

虚函数实现多态的机制主要依赖于**虚函数表**和**虚函数指针**

1. 虚函数表（VTable）

当一个类中包含虚函数时，编译器会为这个类创建一个虚函数表。虚函数表是一个函数指针数组，其中的每个元素都指向类中的一个虚函数。对于每个包含虚函数的类，都会有一个对应的虚函数表。例如，对于基类 `Base` ，如果它有 `virtual void func()` 这样的虚函数，那么编译器会为 `Base` 类创建一个虚函数表，表中存放指向 `func` 函数的指针。

2. 虚函数指针（VPtr）

每个包含虚函数的类的对象，编译器都会在对象中添加一个**隐藏**的指针，即虚函数指针 `VPtr` 。在对象构造的时候， `VPtr` 会被初始化，指向该对象所属类的虚函数表。也就是说，每个对象都有一个 `VPtr` ，它指向对象所属类的虚函数表。

当通过基类指针或引用调用虚函数时，程序运行时会根据基类指针或引用所指向的实际对象的类型，通过该对象的 `VPtr` 找到对应的虚函数表，然后在虚函数表中查找被调用的虚函数的指针，

并调用该函数。这样，即使基类指针或引用指向的是派生类对象，也能正确调用派生类中重写的虚函数，从而实现多态。

```cpp
#include <iostream>
using namespace std;

class Base {
public:
    virtual void func() {
        cout << "Base::func()" << endl;
    }
};

class Derived : public Base {
public:
    void func() override {
        cout << "Derived::func()" << endl;
    }
};

int main() {
    Base* ptr = new Derived();
    ptr->func();
    delete ptr;
    return 0;
}
```

1. 虚函数表（VTable）

基类 `Base`：由于 `Base` 类中有一个虚函数 `func`，编译器会为 `Base` 类创建一个虚函数表。这个虚函数表是一个特殊的数据结构，它是一个函数指针数组，其中有一个元素指向 `Base::func` 函数。可以把虚函数表想象成一个“地图”，它记录了类中所有虚函数的位置。

派生类 `Derived`：`Derived` 类继承自 `Base` 类，并且重写了 `func` 函数。编译器会为 `Derived` 类也创建一个虚函数表。在这个虚函数表中，原本指向 `Base::func` 的位置现在被替换为指向 `Derived::func` 的指针。也就是说，`Derived` 类的虚函数表“更新”了 `func` 函数的地址。

2. 虚函数指针（VPtr）

基类对象：每个 `Base` 类的对象在内存中除了包含自身的成员变量外，还会有一个隐藏的虚函数指针 `VPtr`。当创建 `Base` 类对象时，这个 `VPtr` 会被初始化为指向 `Base` 类的虚函数表。

派生类对象：`Derived` 类对象同样也有一个 `VPtr` 。当创建 `Derived` 类对象时，它的 `VPtr` 会被初始化为指向 `Derived` 类的虚函数表。

3. 多态的实现过程

在 `main` 函数中， `Base* ptr = new Derived();` 这行代码创建了一个 `Derived` 类的对象，并让一个基类 `Base` 的指针 `ptr` 指向它。虽然 `ptr` 是 `Base` 类型的指针，但它实际上指向的是一个 `Derived` 类的对象。

当执行 `ptr->func();` 时，程序运行时不会直接根据 `ptr` 的类型（ `Base` 类型）来调用函数。而是会根据 `ptr` 所指向的实际对象（ `Derived` 类对象）的 `VPtr` 来找到对应的虚函数表。由于 `Derived` 类对象的 `VPtr` 指向 `Derived` 类的虚函数表，在这个虚函数表中 `func` 函数指针指向的是 `Derived::func` ，所以最终会调用 `Derived::func` 函数，而不是 `Base::func` 函数。

最后， `delete ptr;` 用于释放之前动态分配的内存，避免内存泄漏。

9.2.3 虚析构函数

在C++中不能声明虚构造函数，因为构造函数执行时，对象还没有创建，不能按照虚函数方式调用。*（当构造函数执行完毕，对象才算是完整地创建好了，可以被正常使用。）*但是在C++中可以声明虚析构函数，虚析构函数的声明实在“~”符号前添加 `virtual` 关键字，格式如下：

```
virtual ~ 析构函数名 () ;
```

在基类中声明虚构函数之后，基类的所有派生类的析构函数都***自动成为虚析析构函数***。

在基类声明虚构函数之后，使用基类指针或引用操作派生类对象，在析构派生类对象时，编译器会**先**调用派生类的析构函数释放派生类对象资源，**然后**再调用基类析构函数。**如果**基类没有声明虚析构函数，在析构派生类对象时，编译器只会调用基类的析构函数，不会调用派生类的析构函数，导致派生类对象申请的资源不能正常释放。

```cpp
#include <iostream>
using namespace std;
class B0 {
public:
        B0() {
            cout << "B0构造函数调用" << endl;
    }
        ~B0() { //virtual ~B0 若基类析构函数声明为虚函数 则就只会调用派生类的析构函数
            cout << "B0的析构函数调用" << endl;
        }
};
class B1 : public B0{
public:
    int* p;
    B1() {
        p = new int(10);
        cout << "B1构造函数调用" << endl;
    }
    ~B1() {
        delete p;
        cout << "B1的析构函数调用" << endl;
    }
};
int main()
{
    B0 *b = new B1();
//条件: 1. 存在继承关系
//      2. 使用基类指针指向派生类对象
//      3. 基类析构函数不是虚函数 且用的是new 和 delete这一对
    delete b;   //此时只会调用基类的析构函数 不会调用派生类的析构函数
    return 0;
}

```

注:

`p = new int(10)` 与 `delete p;` **是一对操作**

`p = new int(10)` 是申请动态内存，如果返回成功，则返回一个指针地址；反之则返回一个`null` 空指针

`delete p;` 是删除 `p` 指针的内存空间

9.3 纯虚函数与抽象类

有时候在基类中声明函数并不是基类本身的需求，而是考虑到派生类的需求，在基类中声明一个函数，函数的具体实现有派生类根据本类的需求定义。比如，动物都有叫声，但不同的动物叫声不同，因此基类（动物类）并不需要实现描述动物的叫声的函数，只需要声明即可，函数的具体实现在各个派生类中完成。在基类中，这样的函数被称为**纯虚函数**。

纯虚函数也通过`virtual`关键字声明，但是纯虚函数没有函数体（类似于Java中的abstract）。纯虚函数在声明时，需要在后面加上`“=0”`，格式如下：

```
virtual 函数返回值类型 函数名（参数列表） = 0;
```

纯虚函数并不是函数的返回值为0，它只是告诉编译器这是一个纯虚函数。

纯虚函数的作用是在基类中为派生类保留一个接口，方便派生类根据需要完成定义，实现多态。**派生类都应该实现基类的纯虚函数，**如果派生类没有实现基类的纯虚函数，则该函数在派生类中仍然是纯虚函数。

如果一个类中包含纯虚函数，那么这样的类就叫做**抽象类**。抽象类的作用主要是通过它的一个类群建立一个公共接口（纯虚函数），使它们能够更有效地发挥多态性。

抽象类只能作为基类派生新类，***不能创建抽象类的对象***，但可以定义抽象类的指针或引用，通过指针或引用操作派生类对象。抽象类可以有多个纯虚函数，如果派生类需要实例化对象，在派生类中需要全部实现基类的纯虚函数，如果派生类**没有全部实现基类的纯虚函数**，未实现的纯虚函数在派生类中仍然是纯虚函数，**则派生类也是抽象类**。