Universe Scientific Publishing
Computer System Networking and Telecommunications
计算机系统网络和电信，2019,1(2)
ISSN:2661-3719 (Print); 2661-3727 (Online)

Analysis of Class and Object-oriented in C++

Tinglin LI Zhengde BAO Yawen TANG
School of Computer and Software, Jincheng College, Sichuan University, Chengdu, Sichuan, 611731

Abstract
C programming language has long occupied the top of the popular list of programming languages for a long time because of its superior performance. The class and object-oriented part of C is the most important and the core part. The use of class makes C have object-oriented programming thinking. This paper summarizes the process-oriented, object-oriented thinking and analyzes the characteristics of classes in C.

Key Words
C++, Class, Object, Properties of Class

DOI:10.18686/jsjxt.v1i2.667

浅析C++中类与面向对象

李烃林 鲍正德 唐娅雯
四川大学锦城学院计算机与软件学院，四川成都，611731

摘 要
C++程序语言以其优越的性能长期占据着当今编程语言流行榜前列，是现代计算机专业大学生的所必须掌握的一门编程语言。C++中类和面向对象的部分是它最关键也是最核心的部分，类的使用使得C++拥有了面向对象的编程思维。本文概述了面向过程、面向对象的思维并浅析了C++中类的特性。

关键字
C++；类；对象；类的特性

1.引言

C++作为C语言的扩展和升级，它在拥有数据抽象和面向对象的特性之际仍继承了C语言简便和高性能的特征，从而使得C++被广大程序开发人员所喜爱，它在计算机编程语言中正担负着越来越重要的责任。

2.面向过程与面向对象

2.1 面向过程

2.1.1 设计理念

面向过程是相对于面向对象所提出的概念，在那之前它被称为“结构化编程”。面向过程是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。[1]这类方式以步骤来拆解问题，它的重点是数据结构和算法，核心思想是结构化程序设计。

2.1.2 设计特点

1)模块化

模块化应当优先顾及全局，其次关注局部，采取从上到下、慢慢精细的设计过程把一个庞大的程序分割成一个个简单的模块，每一个模块实现一部分功能，在使用时调用相应的模块，从而实现各种复杂功能的程序。并且模块之间，应该让它单独存在，使他们各自独自地实现某个固定的功能。

2)数据与操作分离

面向过程的关注点在于如何实现功能，即模块的划分。数据是可变的，功能是固定的，因此将数据和功能函数分开，使得开发人员只需要关注于操作而不是数

据。

2.2 面向对象

2.2.1 设计理念

面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。[1]面向对象不是像解决数学问题时一个步骤一个步骤进行计算，它是把问题中的事物抽象成单独的对象，在编程时思考对象与对象之间的联系和交互。

2.2.2 设计特点

抽象：将一类事物共同的属性和行为总结出来用统一的方法去描述其共同点。

封装：一个对象的数据对外不可见，只能使用公有的函数才能去访问它。

继承：两个相似的种类可以通过继承将相同的数据和方法联系到一起，减少重复代码。

多态：同一个操作被不同对象调用时发生不一样的行为。

3.C++类

3.1 类的概念

类就是一种具有相同特征的数据和方法组合在一起的一个抽象数据类型。简单来说，类是一种开发人员自己所定义的类型，它封装了同一种类的数据和方法，使得数据之间具有关联性。

3.2 类的构成

3.2.1 普通成员

类的组成包含类名和类定义体，并且为了保证类定义体的安全性在其中设置了三道访问权限：私有权限、公有权限、保护权限（其关键字分别是 private、public 和 protected）。

3.2.2 “特殊”成员

类还有几种特殊的成员：友元类和友元函数、virtual 成员函数、内联函数、static 成员变量和 static 成员函数和 const 成员成员函数。友元类和友元函数就是在类的成员中可定义某个类或者某个函数为该类的友元，从而使得这个类和函数可以访问该类的所有数据和方法；virtual 成员函数即虚函数，是为了使得子类能够重写父类函数，从而实现类的多态性。当父类用 virtual 标记成员函数为虚函数时，它的子类的这个成员函数也会成为虚函数，可以不需要再用 virtual 标记，但是如果不标记的话，派生类的成员函数是否是虚函数需要去基类中查看，略嫌麻烦，因此不是特殊情况时，派生类的虚函数应当用 virtual 标记；内联函数是用 inline 关键字标识的，一般用于内联函数的都是一些简略的、功能简单的、常被调用的函数；static 成员变量可以让一个类的多个对象共享同一个变量，每一个对象对 static 成员变量的操作都会使得其他对象中的 static 成员变量改变；static 成员函数是专门用来操作 static 成员变量的。

4.类的三大特征

4.1 封装性

4.1.1 封装重要性

开发人员在编写程序时，常常烦恼于数据的安全性，因此作为程序语言的设计者，需要考虑一个重要的问题便是信息的隐藏性。而 C++不仅具有抽象功能，还拥有比较完善的数据封装机制，从而完美的展现了面向对象的特性。C++还将自定义数据类型的发展提高了一个档次，而类是自定义类型中最重要也是最核心的一部分。

4.1.2 封装方法

在一个类中，开发人员可以按照需求自定义相应的数据成员与函数成员，并且为其赋予相应的权限。而 C++类的封装就是通过不同的权限将数据和操作设置为是否可视化，从而实现类的信息隐藏属性。C++类使用 private 关键字将数据私有化而使得外界无法直接访问数据，只能通过用 public 关键字标识的操作接口间接访问或修改其数据，使数据和操作的具体实现被隐藏在接口后面，达到封装数据的目的。例如：

```
#include <iostream>
class Temp{
private:
    int data;
public:
    Temp(int x = 0)
```

```
    {
        data = x;
    }
    void show()
    {
        std::cout << data << std::endl;
    }
};
int main()
{
    Temp t;
    //Std::cout<<t.data<<endl;      //错误，私有成员无法被外界访问
    t.show();          //通过公有函数输出 t 中的 data 数据
    return 0;
}
```

在这个例子中，如果直接通过 t.data 的方式来访问输出 data 会报错，因为 Temp 类将 data 数据封装在了 private 里，只能通过 Temp 的 show()函数去访问它。

4.1.3 封装优点

这种方式使得类的对象只需要知道如何使用已经编写好的接口访问、修改数据或者与其他类对象通信，而不用关心其是如何具体地实现操作的，把实现细节隐藏在类中。并且当数据和方法被更改时，只要接口不变，无需去重新修改每个调用其接口的地方。

4.2 继承性

4.2.1 继承普遍性

面向对象的程序设计提供了类的继承机制，该机制自动地为一个类提供来自另一个类的操作和属性，这使得程序员只需要在新类中添加已有类中没有的成员来建立新类。[2]在现实中咱们可以发现许多相关的例子，比如将“动物”当作一个类，它能够划分很多子类，“猫”、“狗”等等。它们都包含了“动物”类的属性和操作，所以说，“动物”是“父类”、“基类”，而“猫”、“狗”则是“子类”、“派生类”。

4.2.2 继承要点

当子类在继承了父类以后，不仅能够继承父类原有的数据和操作，还可以添加新的，而且如果有必要的话子类还可以重写父类的函数，从而拥有与父类不一样的功能，从而推旧陈新。不仅仅是方法，属性也一样可以更改，比如父类的姓名长度规定为 5 位，子类可以更改成 8 位。不过值得注意的是，父类自己独有的函数不能被子类继承，由于每个类的属性和方法不一样，在构造和析构时生成的内存空间和释放的内存空间也不一样。

4.2.3 继承方式

C++中有三种继承方式，这些继承的共同特点是都不能直接访问父类的私有成员，并且它们有一些区别。如下例：

```
#include <iostream>
class A{
private:
    int x;
protected:
    int y;
public:
    int z;
    A(int x1 = 0, int y1 = 0, int z1 = 0){ x = x1; y = y1; z = z1; }
    void showA()
    {
        std::cout << x << y << z << std::endl;
    }
};
class B :public A{
public:
    void showB()
    {
        //std::cout << x << std::endl;   //错误，父类私有成员未被继承
        std::cout << y << z << std::endl;
    }
};
int main()
{
    B b;
    //b.x;//错误，父类私有成员未被继承
    //b.y;//错误，外界无法访问类的保护成员
```

```
    b.showA();      //输出 000
    b.showB();      //输出 00
    return 0;
}
```

此例中 B 公有继承了 A，因此 A 的公有方法能够直接被 B 调用并访问到 A 的私有成员，B 无法调用 A 中的私有成员 x，但是 B 能够在公有函数中调用从 A 继承下来的保护成员 y。

当 B 保护继承 A 时，A 的公有成员和保护成员会变成B的保护成员，因此B将无法直接调用A的showA()方法，此时可以在 B 的公有成员中写一个调用 showA()方法的函数即可。

当 B 私有继承 A 时，A 的公有成员和保护成员会变成 B 的私有成员，因此 B 不仅无法调用 A 的 showA()方法，A 的成员均无法向下再继承。

4.2.4 继承优缺点

C++的继承可以使得子类无须重写父类雷同的代码，减少代码的反复编程，从而提高代码的可复用性，而且可以扩展其功能。不过这种继承方式也会导致一个重要的问题——二义性，一种常见的二义性就是当一个子类继承了多个父类时，它们的数据或函数中有相同的名字从而形成矛盾，这种情况在 C++中能够使用作用域解析运算符在表达式中书写具体的父类名来解决此类二义性。而当子类在继承时可以通过多种路径与同一个父类相关联时，便会造成进一步的二义性，比如说一个父类 A 中含有一个数据成员 X，有两个子类 B 和 C 都公有继承了 A，同时一个子类 D 多重继承了 B 和 C，这个时候 D 可以通过两种途径 B 和 C 来与 A 关联，因此就产生了一个潜在的二义性，在 D 对象中是包含了两个 A 还是一个 A？所以在 C++中使用了虚基类使得每个无论有多少条路径可以到达父类的子类对象中只有唯一的父类部分。

虽然 C++都给出了相应的解决方案，不过在实际开发中，就越发地考验开发人员的技术和经验，需要开发人员在使用一些用法时要小心地避免产生不当的影响，因此使得 C++的上手难度也相应地提高了。

4.3 多态性

4.3.1 多态定义

多态是指“一种操作，不同表现”，即同一个函数在不同的情况下被调用时发生不一样的行为。多态是伴随着继承而出现的特性，可以说多态是为了解决继承所出现的各种问题而出现的解决方案。

4.3.2 多态方式

在 C++中，编译时多态和运行时多态是实现多态的两种方式。编译时多态性是指在程序编译阶段即可确定下来的多态性，包括强制多态和重载多态两种形式，运行时多态性是指必须等到程序动态运行时才可确定的多态性，主要通过继承结合动态绑定获得，分别体现在包含多态与类型参数化多态两方面。[3]

1) 强制多态

强制也称为类型转换，比如当表达式中有不同类型的数据时，可以将部分类型强制转换成另一种类型，从而保证运算符两边参数一致。然而在 C++中，这种方式并不是能够解决所有类似问题，因此 C++提供了另一种方法——过载多态。

2) 过载多态

过载多态也叫重载多态，指的是一样名字的函数或操作符在不一样的情况下表现出不一样的行为，它包含函数重载和运算符重载。

函数重载允许我们在定义一个函数时，可以通过传入参数的不同实现不同的操作，比较经典的是类的构造函数，可以对类的对象进行不同的初始化。运算符重载可以让我们重新定义相关运算符的操作，从而可以实现复杂的类之间的四则运算等。如下例：

```
#include <iostream>
class C{
private:
    int x;
    int y;
public:
    C(){ x = 1; y = 1; }
    C(int a, int b){ x = a; y = b; }
    C operator +(C c1)
    {
        C c2(c1.x + x, c1.y + y);
        return c2;
    }
    void show()
```

```
    {
        std::cout << x << "\t" << y << std::endl;
    }
};
int main()
{
    C c1;
    C c2(1, 2);
    C c3 = c1 + c2;        //运算符重载后可直接进行自定义类型的加法
    c1.show();      //输出 1   1
    c2.show();      //输出 1   2
    c3.show();      //输出 2   3
    return 0;
}
```

此例中，类 C 的构造函数通过传入不同参数对类 C 进行不同的初始化，并且将加法运算符进行运算符重载使得两个类 C 能够互相进行加法运算。

3) 参数多态

参数多态是指与类模板相关联，类模板所包含的操作的类型只能用类型参数进行实例化，使得一个结构能有多种类型。模板是 C++实现参数化多态性的工具，分为函数模板和类模板二种，类模板中的成员函数均为函数模板，因此函数模板是为类模板服务的。[4]如下例：

```
#include <iostream>
template <class T> class D{
public:
    D(){}
    T add(T a, T b){ return a + b; }
};
int main()
{
    int a = 1, b = 2;
    float c = 1.1, d = 2.2;
    D<int> d1;
    D<float> d2;
    std::cout << d1.add(a, b) << std::endl ;
//输出 3
    std::cout << d2.add(c, d) << std::endl;
    //输出 3.3
    return 0;
}
```

此例中，用类模板的方式定义了类 D，使得类 D 中的数据类型可以动态改变，并且在类中可以使用函数模板的方式定义 add 函数，从而让 add 函数无需固定某个具体数据类型的数据进行加法运算，从而实现多态。

4) 包含多态

包含多态是指同样的函数名在父类和子类中有不同的操作或行为，而这种函数必须是虚函数并且需要子类继承父类。C++虚函数具有在程序运行时能够依据类型的不同使用运行时类型识别 RTTI 的方法而确定正确的匹配函数的能力，C++在编译时采用“滞后编译”的技术来实现这种能力，以使函数能呈“多态”。[5]若一个虚函数没法详细给出合理的定义时，可以设置为纯虚函数。

```
class A{
public:
    virtual void show()
    {
        std::cout << "I'm class A function" << std::endl;
    }
};
class B :public A{
public:
    virtual void show()
    {
        std::cout << "I'm class B function" << std::endl;
    }
};
int main()
{
    A a,*pa;
    B b;
    pa = &b;        //父类指针指向子类
    a.show();       //输出 I'm class A function
    b.show();       //输出 I'm class B function
    pa->show();     //输出 I'm class B function
    return 0;
}
```

此例中，将类 A 中的 show 函数标记位虚函数，在

B 类中重写这个函数，当 A 类的指针指向它的子类 B 时，可调用 B 类重写的函数，若未标记为虚函数，则仍然调用 A 类的函数。

4.3.3 多态优点

子类不需要为每一个子类写函数调用，只用在父类里处理就可以了，加强了程序的可复用性，并且由于子类的函数能够被父类所调用，因此程序能够向后兼容。

5.C++类与面向对象优点

C++中的类以抽象数据类型为其理论基础，完美地呈现了面向对象思想的信息隐蔽和数据抽象等特点。类将同一种对象抽象为一种数据类型，将其基本属性和操作进行封装，并进行权限访问限制，从而减少数据泄露的风险。类的使用，提高了代码的可复用性、灵活性和易扩展性，降低了代码的冗余度，大幅度地提高了编写代码的效率。

6.总结

C++作为 C 语言的超集，在加入了面向对象的特性后，成为了一种兼具结构化和面向对象编程的混合式语言，从而使得它在编程的海洋中畅行无阻。而这与 C++中类的优越性密不可分，C++中的类可以使得我们更好地理解和掌握面向对象的思维方式，进而反过来灵活运用 C++。

参考文献

[1]杜瑞庆,李小慧.高等院校《C++程序设计》教学探讨[J].职业时空,2006(21):19.
[2]张宇.C++中类继承方式的讨论[J].电脑知识与技术,2012,8(12):2781-2785.
[3]江勇驰.C++语言中多态性的分析[J].广州航海高等专科学校学报,2005(01):55-57.
[4]姚云霞.浅析 C++中类的多态性[J].陇东学院学报,2012,23(01):9-11.
[5]王彤.C++类的多态性的再探讨[J].科技信息(学术研究),2008(14):6-8.

作者简介

第一作者：李烃林（1998-），男，汉，四川省成都市，本科，四川大学锦城学院，研究方向：软件工程。
第二作者（通讯作者）：鲍正德（1989-），男，汉，黑龙江哈尔滨，研究生，四川大学锦城学院，研究方向：电子商务。
第三作者：唐娅雯（1999-），女，汉，四川省资阳市，本科，四川大学锦城学院，研究方向：信息管理、J2EE