OO

jenny

February 25, 2014

Contents

修练8年C++面向对象程序设计之体会

2005-04-29 09:58 出处： 作者：林锐 责任编辑：xietaoming 六年前，我刚热恋“面向对象”（Object-Oriented）时，一口气记住了近十个定义。六年后，我从几十万行程序中滚爬出来准备写点心得体会时，却无法解释什么是“面向对象”，就象说不清楚什么是数学那样。软件工程中的时髦术语“面向对象分析”和“面向对象设计”，通常是针对“需求分析”和“系统设计”环节的。“面向对象”有几大学派，就象如来佛、上帝和真主用各自的方式定义了这个世界，并留下一堆经书来解释这个世界。

有些学者建议这样找“对象”：分析一个句子的语法，找出名词和动词，名词就是对象，动词则是对象的方法（即函数）。

当年国民党的文人为了对抗毛泽东的《沁园春·雪》，特意请清朝遗老们写了一些对仗工整的诗，请蒋介石过目。老蒋看了气得大骂：“娘希匹，全都有一股棺材里腐尸的气味。”我看了几千页的软件工程资料，终于发现自己有些“弱智”，无法理解“面向对象”的理论，同时醒悟到“编程是硬道理。”

面向对象程序设计语言很多，如Smalltalk、Ada、Eiffel、Object Pascal、Visual Basic、C++等等。C++语言最讨人喜欢，因为它兼容C 语言，并且具备C 语言的性能。近几年，一种叫Java 的纯面向对象语言红极一时，不少人叫喊着要用Java 革C++的命。我认为Java 好比是C++的外甥，虽然不是直接遗传的，但也几分象样。外甥在舅舅身上玩耍时洒了一泡尿，俩人不该为此而争吵。

关于C++程序设计的书籍非常多，本章不讲C++的语法，只讲一些小小的编程道理。如果我能早几年明白这些小道理，就可以大大改善数十万行程序的质量了。

1 C++面向对象程序设计的重要概念

早期革命影片里有这样一个角色，他说：“我是党代表，我代表党，我就是党。”后来他给同志们带来了灾难。

会用C++的程序员一定懂得面向对象程序设计吗?

不会用C++的程序员一定不懂得面向对象程序设计吗?

两者都未必。就象坏蛋入党后未必能成为好人，好人不入党未必变成坏蛋那样。

我不怕触犯众怒地说句大话：“C++没有高手，C 语言才有高手。”在用C 和C++编程8年之后，我深深地遗憾自己不是C 语言的高手，更遗憾没有人点拨我如何进行面向对象程序设计。我和很多C++程序员一样，在享用到C++语法的好处时便以为自己已经明白了面向对象程序设计。就象挤掉牙膏卖牙膏皮那样，真是暴殄天物呀。

人们不懂拼音也会讲普通话，如果懂得拼音则会把普通话讲得更好。不懂面向对象程序设计也可以用C++编程，如果懂得面向对象程序设计则会把C++程序编得更好。本节讲述三个非常基础的概念：“类与对象”、“继承与组合”、“虚函数与多态”。理解这些概念，有助于提高程序的质量，特别是提高“可复用性”与“可扩充性”。

1.1 类与对象

对象（Object）是类（Class）的一个实例（Instance）。如果将对象比作房子，那么类就是房子的设计图纸。所以面向对象程序设计的重点是类的设计，而不是对象的设计。类可以将数据和函数封装在一起，其中函数表示了类的行为（或称服务）。类提供关键字public、protected 和private 用于声明哪些数据和函数是公有的、受保护的或者是私有的。

这样可以达到信息隐藏的目的，即让类仅仅公开必须要让外界知道的内容，而隐藏其它一切内容。我们不可以滥用类的封装功能，不要把它当成火锅，什么东西都往里扔。

类的设计是以数据为中心，还是以行为为中心?

主张“以数据为中心”的那一派人关注类的内部数据结构，他们习惯上将private 类型的数据写在前面，而将public 类型的函数写在后面，如表8.1(a)所示。

主张“以行为为中心”的那一派人关注类应该提供什么样的服务和接口，他们习惯上将public 类型的函数写在前面，而将private 类型的数据写在后面，如表8.1(b)所示。

很多C++教课书主张在设计类时“以数据为中心”。我坚持并且建议读者在设计类时“以行为为中心”，即首先考虑类应该提供什么样的函数。Microsoft 公司的COM 规范的核心是接口设计，COM 的接口就相当于类的公有函数[Rogerson 1999]。在程序设计方面，咱们不要怀疑Microsoft公司的风格。

设计孤立的类是比较容易的，难的是正确设计基类及其派生类。因为有些程序员搞不清楚“继承”（Inheritance）、“组合”（Composition）、“多态”（ Polymorphism）这些概念。

1.2 继承与组合

如果A 是基类，B 是A 的派生类，那么B 将继承A 的数据和函数。示例程序如下：

```
class A {
public:
    void Func1(void);
    void Func2(void);
};

class B : public A {
public:
    void Func3(void);
    void Func4(void);
};

// Example
main() {
    B b; // 的一个对象B
    b.Func1(); // B 从A 继承了函数Func1
    b.Func2(); // B 从A 继承了函数Func2
    b.Func3();
    b.Func4();
}
```

这个简单的示例程序说明了一个事实：C++的“继承”特性可以提高程序的可复用性。正因为“继承”太有用、太容易用，才要防止乱用“继承”。我们要给“继承”立一些使用规则：

1.2.1 如果类A 和类B 毫不相关，不可以为了使B 的功能更多些而让B 继承A 的功能。

不要觉得“不吃白不吃”，让一个好端端的健壮青年无缘无故地吃人参补身体。

1.2.2 如果类B 有必要使用A 的功能，则要分两种情况考虑：

1. 若在逻辑上B 是A 的“一种”（a kind of ），则允许B 继承A 的功能。如男人（Man）是人（Human）的一种，男孩（Boy）是男人的一种。那么类Man 可以从类Human 派生，类Boy 可以从类Man 派生。示例程序如下：

```
class Human {...

};

class Man : public Human {...

};

class Boy : public Man {...

};
```

2. 若在逻辑上A 是B 的“一部分”（a part of），则不允许B 继承A 的功能，而是要用A和其它东西组合出B。例如眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类Head 应该由类Eye、Nose、Mouth、Ear 组合而成，不是派生而成。示例程序如下：

```
class Eye {
public:
    void Look(void);
};

class Nose {
public:
    void Smell(void);
};

class Mouth {
public:
    void Eat(void);
};

class Ear {
public:
```

```
    void Listen(void);
};

// 正确的设计，冗长的程序
class Head {
public:
    void Look(void) { m_eye.Look(); }
    void Smell(void) { m_nose.Smell(); }
    void Eat(void) { m_mouth.Eat(); }
    void Listen(void) { m_ear.Listen(); }
private:
    Eye m_eye;
    Nose m_nose;
    Mouth m_mouth;
    Ear m_ear;
};
```

如果允许Head 从Eye、Nose、Mouth、Ear 派生而成，那么Head 将自动具有Look、Smell、Eat、Listen 这些功能:

```
// 错误的设计
class Head : public Eye, public Nose, public Mouth, public Ear {
};
```

上述程序十分简短并且运行正确，但是这种设计却是错误的。很多程序员经不起“继承”的诱惑而犯下设计错误。

一只公鸡使劲地追打一只刚下了蛋的母鸡，你知道为什么吗?

因为母鸡下了鸭蛋。

本书3.3 节讲过“运行正确”的程序不见得就是高质量的程序，此处就是一个例证。

1.3 虚函数与多态

除了继承外，C++的另一个优良特性是支持多态，即允许将派生类的对象当作基类的对象使用。如果A 是基类，B 和C 是A 的派生类，多态函数Test 的参数是A 的 指针。那么Test 函数可以引用A、B、C 的对象。示例程序如下:

```
class A {
public:
    void Func1(void);
};

void Test(A *a) {
    a->Func1();
}

class B : public A {...

};

class C : public A {...

};

// Example
main() {
    A a;
    B b;
    C c;
    Test(&a);
    Test(&b);
    Test(&c);
};
```

以上程序看不出“多态”有什么价值，加上虚函数和抽象基类后，“多态”的威力就显示出来了。

C++用关键字virtual 来声明一个函数为虚函数，派生类的虚函数将（override）基类对应的虚函数的功能。示例程序如下:

```
class A {
public:
    virtual void Func1(void){ cout<< “This is A::Func1 \”n}
};

void Test(A *a) {
    a->Func1();
}

class B : public A {
public:
```

```
    virtual void Func1(void){ cout<< ‘‘This is B::Func1 \’’n}
};

class C : public A {
public:
    virtual void Func1(void){ cout<< ‘‘This is C::Func1 \’’n}
};

// Example
main() {
    A a;
    B b;
    C c;
    Test(&a); // 输出This is A::Func1
    Test(&b); // 输出This is B::Func1
    Test(&c); // 输出This is C::Func1
};
```

如果基类A 定义如下:

```
class A {
public:
    virtual void Func1(void)=0;
};
```

那么函数Func1 叫作纯虚函数，含有纯虚函数的类叫作抽象基类。抽象基类只管定义纯虚函数的形式，具体的功能由派生类实现。

结合“抽象基类”和“多态”有如下突出优点:

（1）应用程序不必为每一个派生类编写功能调用，只需要对抽象基类进行处理即可。这一 招叫“以不变应万变”，可以大大提高程序的可复用性（这是接口设计的复用，而不是代码实现的复用）。

（2）派生类的功能可以被基类指针引用，这叫向后兼容，可以提高程序的可扩充性和可维护性。以前写的程序可以被将来写的程序调用不足为奇，但是将来写的程序可以被以前写的程序调用那可了不起。

2 良好的编程风格

内功深厚的武林高手出招往往平淡无奇。同理，编程高手也不会用奇门怪招写程序。良好的编程风格是产生高质量程序的前提。

2.1 命名约定

有不少人编程时用拼音给函数或变量命名，这样做并不能说明你很爱国，却会让用此程序的人迷糊（很多南方人不懂拼音，我就不懂）。程序中的英文一般不会太复杂，用词要力求准确。匈牙利命名法是Microsoft 公司倡导的[Maguire 1993]，虽然很烦琐，但用习惯了也就成了自然。没有人强迫你采用何种命名法，但有一点应该做到：自己的程序命名必须一致。

以下是我编程时采用的命名约定:

（1）宏定义用大写字母加下划线表示，如$\text{MAX}_{\text{LENGTH}}$;

（2）函数用大写字母开头的单词组合而成，如SetName, GetName ;

（3）指针变量加前缀p，如*pNode ;

（4）BOOL 变量加前缀b，如bFlag ;

（5）int 变量加前缀i，如iWidth ;

（6）float 变量加前缀f，如fWidth ;

（7）double 变量加前缀d，如dWidth ;

（8）字符串变量加前缀str，如strName ;

（9）枚举变量加前缀e，如eDrawMode ;

（10）类的成员变量加前缀m，如$\text{m}_{\text{strName}}$, m_{iWidth} ;

对于int, float, double 型的变量，如果变量名的含义十分明显，则不加前缀，避免烦琐。如用于循环的int 型变量i,j,k ; float 型的三维坐标（x,y,z）等。

2.2 使用断言

程序一般分为Debug 版本和Release 版本，Debug 版本用于内部调试，Release 版本发行给用户使用。断言assert 是仅在Debug 版本起作用的宏，它用于检查“不应该”发生的情况。以下是一个内存复制程序，在运行过程中，如果assert 的参数为假，那么程序就会中止（一般地还会出现提示对话，说明在什么地方引发了assert）。

```
//复制不重叠的内存块
void memcpy(void *pvTo, void *pvFrom, size_t size) {
    void *pbTo = (byte *) pvTo;
    void *pbFrom = (byte *) pvFrom;
    assert( pvTo != NULL && pvFrom != NULL );
    while(size - - > 0 )
        *pbTo + + = *pbFrom + + ;
    return (pvTo);
}
```

assert 不是一个仓促拼凑起来的宏，为了不在程序的Debug 版本和Release 版本引起差别，assert 不应该产生任何副作用。所以assert 不是函数，而是宏。程序员可以把assert 看成一个在任何系统状态下都可以安全使用的无害测试手段。

很少有比跟踪到程序的断言，却不知道该断言的作用更让人沮丧的事了。你化了很多时间，不是为了排除错误，而只是为了弄清楚这个错误到底是什么。有的时候，程序员偶尔还会设计出有错误的断言。所以如果搞不清楚断言检查的是什么，就很难判断错误是出现在程序中，还是出现在断言中。幸运的是这个问题很好解决，只要加上清晰的注释即可。这本是显而易见的事情，可是很少有程序员这样做。这好比一个人在森林里，看到树上钉着一块“危险”的大牌子。但危险到底是什么？树要倒？有废井？有野兽？除非告诉人们“危险”是什么，否则这个警告牌难以起到积极有效的作用。难以理解的断言常常被程序员忽略，甚至被删除。[Maguire 1993]

以下是使用断言的几个原则：

（1）使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的。

（2）使用断言对函数的参数进行确认。

（3）在编写函数时，要进行反复的考查，并且自问：“我打算做哪些假定？”一旦确定了的 假定，就要使用断言对假定进行检查。

（4）一般教科书都鼓励程序员们进行防错性的程序设计，但要记住这种编程风格会隐瞒错误。当进行防错性编程时，如果“不可能发生”的事情的确发生了，则要使用断言进行报警。

2.3 new、delete 与指针

在C++中，操作符new 用于申请内存，操作符delete 用于释放内存。在C 语言中，函数malloc 用于申请内存，函数free 用于释放内 存。由于C++兼容C 语言，所以new、delete、malloc、free 都有可能一起使用。new 能比malloc 干更多的事，它可以申请对象的内存，而malloc 不能。C++和C 语言中的指针威猛无比，用错了会带来灾难。对于一个指针p，如果是用new申请的内存，则必须用delete 而不能用free 来释放。如果是用malloc 申请的内存，则必须用free 而不能用delete 来释放。在用delete 或用free 释放p 所指的内存后，应该马上显式地将p 置为NULL，以防下次使用p 时发生错误。示例程序如下：

```
void Test(void) {
    float *p;

    p = new float[100];
    if(p==NULL) return;...
    // do something
        delete p;
    p=NULL; // 良好的编程风格

    // 可以继续使用p
    p = new float[500];
    if(p==NULL) return;...
    // do something else
        delete p;
    p=NULL;
}
```

我们还要预防“野指针”，“野指针”是指向“垃圾”内存的指针，主要成因有两种：

（1）指针没有初始化。

（2）指针指向已经释放的内存，这种情况最让人防不胜防，示例程序如下：

```
class A {
public:
    void Func(void...){}
};

void Test(void) {
    A *p; {
        A a;
        p = &a; // 注意a 的生命期
    }
    p->Func(); // p 是“野指针”，程序出错
}
```

2.4 使用const

在定义一个常量时，const 比#define 更加灵活。用const 定义的常量含有数据类型，该常量可以参与逻辑运算。例如：

```
const int LENGTH = 100; // LENGTH 是int 类型
const float MAX=100; // MAX 是float 类型
#define LENGTH 100 // LENGTH 无类型
#define MAX 100 // MAX 无类型
```

除了能定义常量外，const 还有两个“保护”功能：

2.4.1 一、强制保护函数的参数值不发生变化

以下程序中，函数f 不会改变输入参数name 的值，但是函数g 和h 都有可能改变name的值。

```
void f(String s); // pass by value
void g(String &s); // pass by referance
void h(String *s); // pass by pointer

main() {
```

```
    String name‘‘=’’Dog;
    f(name); // name 的值不会改变
    g(name); // name 的值可能改变
    h(name); // name 的值可能改变
}
```

对于一个函数而言，如果其‘&’或‘*’类型的参数只作输入用，不作输出用，那么应当在该参数前加上const，以确保函数的代码不会改变该参数的值（如果改变了该参数的值，编译器会出现错误警告）。因此上述程序中的函数g 和h 应该定义成:

```
void g(const String &s);
void h(const String *s);
```

2.4.2 二、强制保护类的成员函数不改变任何数据成员的值

以下程序中，类stack 的成员函数Count 仅用于计数，为了确保Count 不改变类中的任何数据成员的值，应将函数Count 定义成const 类型。

```
class Stack {
public:
    void push(int elem);
    void pop(void);
    int Count(void) const; // const 类型的函数
private:
    int num;
    int data[100];
};

int Stack::Count(void) const {
    ++ num; // 编译错误，num 值发生变化
    pop(); // 编译错误，pop 将改变成员变量的值
    return num;
}
```

2.5 其它建议

（1）不要编写一条过分复杂的语句，紧凑的C++/C 代码并不见到能得到高效率的机器代码，却会降低程序的可理解性，程序出错误的几率也会提高。

（2）不要编写集多种功能于一身的函数，在函数的返回值中，不要将正常值和错误标志混在一起。

（3）不要将BOOL 值TRUE 和FALSE 对应于1 和0 进行编程。大多数编程语言将FALSE定义为0，任何非0 值都是TRUE。Visual C++将TRUE定义为1，而Visual Basic 则将TRUE定义为-1。示例程序如下:

```
BOOL flag;

if(flag)       { // do something } // 正确的用法
if(flag==TRUE) { // do something } // 危险的用法
if(flag==1)    { // do something } // 危险的用法
if(!flag)      { // do something } // 正确的用法
if(flag==FALSE){ // do something } // 不合理的用法
if(flag==0)    { // do something } // 不合理的用法
```

（4）小心不要将“= =”写成“=”，编译器不会自动发现这种错误。

（5）不要将123 写成0123，后者是八进制的数值。

（6）将自己经常犯的编程错误记录下来，制成表格贴在计算机旁边。

3 小结

C++/C 程序设计如同少林寺的武功一样博大精深，我练了8 年，大概只学到二三成。所以无论什么时候，都不要觉得自己的编程水平天下第一，看到别人好的技术和风格，要虚心学习。本章的内容少得可怜，就象口渴时只给你一颗杨梅吃，你一定不过瘾。我借花献佛，推荐一本好书：Marshall P. Cline 著的《C++ FAQs》[Cline 1995]。你看了后一定会赞不绝口。会编写C++/C 程序，不要因此得意洋洋，这只是程序员基本的技能要求而已。如果把系统分析和系统设计比作“战略决策”，那么编程充其量只是“战术”。如果指挥官是个大笨蛋，士兵再勇敢也会吃败仗。所以我们程序员不要只把眼光盯在程序上，要让自己博学多才。我们应该向北京胡同里的小孩们学习，他们小小年纪就能指点江山，评论世界大事。