C++编程(10)

唐晓晟
北京邮电大学电信工程学院

第12章 派生类

- 引言
- 派生类
- 抽象类
- 类层次结构的设计
- 类层次结构和抽象类
- 忠告

12.1 引言

- C++从Simula那里借用了类以及类层次结构的概念，此外，还有有关系统设计的思想
- 概念不会孤立的存在，它总与一些相关的概念共存，用类来描述概念，不可避免的需要用类描述概念之间的关系
- 派生类的概念及其相关的语言机制使得我们能够表述一种层次性的关系，即：表述一些类之间的共性

- 例如：圆和三角形概念之间有关系，因为它们都是形状，即它们之间共有形状这个概念
- 若在程序中表示一个圆和一个三角形，但是却没有涉及到形状的概念，就应该认为是丢掉了某些最基本的东西
- 这个简单的思想就是面向对象程序设计的基础

12.2 派生类

- 考虑做一个程序，处理某公司雇佣的人员

```
struct Employee
{
    string first_name;
    string family_name;
    char middle_initial;
    Date hiring_date;
    short department;
};
```

```
struct Manager
{
    Employee emp;
    // 经理的雇佣记录
    list<Employee*> group;
    // 所管理的人员
    short level;
};
```

- 程序员很容易知道Employee和Manager之间的关系，但是编译器却对此一无所知
- 好的描述方法应该能够把Manager也是Employee的事实明确地表述出来

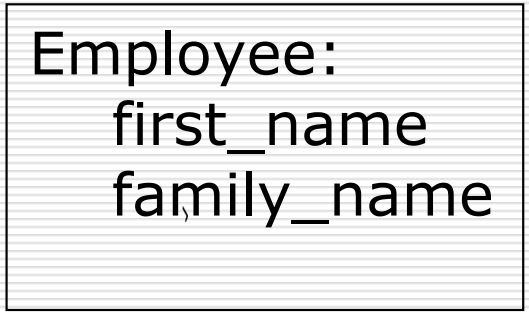
派生类示例

```
struct Manager : public Employee
{
    list<Employee*> group;
    short level;
}; // 注意，Employee必须在Manager之前定义，不能只声明
```

- 此Manager是从Employee派生(继承)出来的，也即：Employee是Manager的一个基类，而Manager则是Employee的一个子类
- 类Manager中包含了类Employee中的所有成员，再加上一些自己的成员

派生类的实现方式

一种常见的实现方式，就是将派生类的对象也表示为一个基类的对象，只是将那些特别属于派生类的信息附加在最后




```
void f(Manager ml,, Employee el)
{
    list<Employee*> elist;
    elist.push_front(&ml); // 可以，因为Manager也是Employee
    elist.push_front(&el);  // 不可，Employee不一定是Manager
}
```

派生类示例

- 可以用Derived*给Base*类型的变量赋值，不需要显式的转换，而相反的方向，则必须显式转换

```
void g(Manager mm, Employee ee)
{
    Employee* pe = &mm; // 可以
    Manager* pm = &ee; // 错误
    pm->level = 2;  // 错误：pm没有level变量
    pm=static_cast<Manager*>(pe); // 蛮力，可以
    pm->level = 2; // 没问题
}
```

12.2.1 成员函数

- 对于派生类，只能使用基类中的公用的和保护(protected)的成员，但是派生类不能使用基类的私用名字

```
class Employee{
    string first_name, family_name;
    char middle_initial;
    //...
public:
    void print() const;
    string full_name() const
    {
        return first_name + ' ' +
middle_initial + ' ' + family_name;
    }
};
```

```
class Manager : public Employee
{
    // ...
public:
    void print() const;
};
void Manager::print() const
{
    cout << "name is " <<
full_name() << "\n";
    //cout << family_name; // 错误
}
```

派生类中调用基类的同名函数

```
void Manager:print() const
{
    Employee::print();
    // 打印Employee的信息
    cout <<, level;
    // 打印Manager的特殊信息
}

void Manager:print() const
{
    print(); //错误，无穷递归调用
}
```

12.2.2 构造函数和析构函数

```
class Employee{
    string first_name, family_name;
    short department;
    //...
public:
    Employee(const string& n, int d);
};
class Manager : public Employee
{
    list<Employee*> group;
    short level;
public:
    Manager(const string& n, int d, int
level);
};
```

```
Employee::Employee(const
string& n, int d) :
family_name(n),
department(d) {}

Manager::Manager(const
string& n, int d, int lvl) :
Employee(n,d), level(lvl) {}

Manager::Manager(const
string& n, int d, int lvl) :
family_name(n),
department(d), level(lvl) {}
// 错误
```

构造是自下而上进行：基类、成员、派生类，销毁时则顺序相反

12.2.3 复制

- 类对象的复制由复制构造函数和赋值操作定义
- 注意，如果没有为Manager定义复制运算符，编译器会为你生成一个，也即：该运算符是不继承的，构造函数也是如此

```
class Employee
{
    Employee& operator=(const Employee&);
    Employee(const Employee&);
};
void f(const Manager& m) {
    Employee e = m; // 从m的Employee部分创建e
    // 这种情况叫做切割
    e = m; // 用m的Employee部分给e赋值
}
```

12.2.4 类层次结构

- 派生类也可以做为基类

```
class Employee{/*   */};
class Manager:public Employee{/*   */};
class Director:public Manager{/*   */};
```

- 一组相关的类按照习惯被称为一个类层次结构

```
class Temporary{/*   */};
class Secretary:public Employee{/*   */};
class Tsec:public Temporary, public Secretary{/*   */};
class Consultant:public Temporary, public Manager{/*   */};
```

- C++可以表示类的有向无环图

12.2.5 类型域

- 为了使派生类不仅仅是一种完成声明的方便简写形式，必须解决如下问题：对于给定的类型为Base*的指针，被指的对象到底属于哪个派生类？
- 四种基本的解决方案
  - (1)保证被指的只能是唯一类型的对象
  - (2)在基类里安排一个类型域，供函数检查
  - (3)使用dynamic_cast
  - (4)使用虚函数

方案2的示例

```
struct Employee
{
    enum Empl_type{M,E};
    Empl_type type;
    Employee():type(E){}
    string ...
};

struct Manager:public
Employee
{
    Manager(){type=M;}
    list<Employee*> group;
    ...
};
```

```
void print_employee(const Employee* e)
{
    switch(e->type){
        case Employee::E:
            cout << e->family_name << e-
>department;
            break;
        case Employee::M:
            cout << e->family_name << e-
>department;
            const Manager* p =
static_cast<const Manager*>(e);
            cout << p->level;
            break;
    }}
```

方案2的示例

- 这种方式可以工作，特别是在由一个人维护的小程序里
- 致命弱点：程序员按照特定的方式来操纵这些类型，这种方式是编译器无法检查的
- 如果引入Employee的新的派生类，需要改动太多的原有代码，维护麻烦
- 对于模块化以及信息隐藏原则是一个很大的破坏，基类是某种“公用信息”的展台

12.2.6 虚函数

- 虚函数能克服类型域解决方案的缺陷，它使得程序员可以在基类里声明一些能够在各个派生类里重新定义的函数，编译器和转载程序能保证对象和应用于它们的函数之间正确的对应关系
- 派生类里，相关函数的参数类型必须和基类完全相同
- 虚函数第一次声明的那个类里，必须对其进行定义，除非为纯虚函数

```
class Employee{
    string first_name, family_name;
    short department;
public:
    Employee(const string& n, int d);
    virtual void print() const; // 指明print()的作用就像是一个界面
};
```

虚函数示例

```
void Employee::print() const{
    cout << family_name <<
department;
}
class Manager:public Employee {
    list<Employee*> group;
    short level;
public:
    Manager(const string& n, int d,
int lvl);
    void print() const;
};
void Manager::print() const {
    Employee::print();
    cout << level;
}
```

```
void print_list(const
list<Employee*>& s)
{
for(list<Employee*>::
const_iterator p=s.begin();
p!=s.end(); ++p)
    (*p)->print();
}
int main(){
    Employee e("Brown",1234);
    Manager m("Smith",1234,2);
    List<Employee*> empl;
    empl.push_front(&e);
    empl.push_front(&m);
    print_list(empl);
}
```

虚函数进一步说明

- 上述例子完全可行，即使print_list()是在派生类Manager之前写好并编译好的，这是类机制中最关键的一个方面
- 从Employee的函数中取得“正确的”行为，而又不依赖于实际使用的到底是哪个Employee，就是所谓的多态性
- 一个带有虚函数的类型被称为是一个多态类型，要在C++中取得多态性的行为，被调用的函数就必须是虚函数

12.3 抽象类

- 有些可以作为其他类的基类的类，但是不需要从为这种类生成一个实际的对象，或者说，该类仅仅提供了一种概念上的抽象
- 可以使用纯虚函数来表达这种抽象

```
class Shape
{
public:
    virtual void rotate(int){error("Shape::rotate");}
    virtual void draw(){error("Shape::draw");}
};
Shape s; // 愚蠢的声明，对s的任何操作都导致错误
```

抽象类

- 将类Shape声明为纯虚函数
- 如果一个类中存在一个或者几个纯虚函数，这个类就是一个抽象类
- 不能创建抽象类的对象，只能使用该类作为其他类的基类

```
class Shape
{
public:
    virtual void rotate(int) = 0;
    virtual void draw() = 0;
    virtual bool is_closed() = 0;
};
Shape s; // 错误
```

抽象类示例

```
class Point{/**/};
class Circle : public Shape
{
public:
    void rotate(int){}
    void draw();
    bool is_closed(){return true;}
    Circle(Point p, int r);
private:
    Point center;
    int radius;
};
```

一个未在派生类里定义的纯虚函数仍旧还是一个纯虚函数，也使得该类仍为一个抽象类，这就使得我们可以分步骤构筑起一个实现

抽象类的最重要用途就是提供一个界面，而又不保露任何实现的细节

12.4 类层次结构的设计

- 一个简单的设计问题：为某个程序提供一种方式，通过它可以从某种用户界面取得一个整数
- 想法：采用一个类Ival_box，它知道能够接受的输入值的取值范围，程序可以向Ival_box要求一个值，必要时要求它去提示用户输入，此外，程序可以询问Ival_box，看看程序最后一次查看之后用户是否修改过有关的值

12.4.1 一个传统的层次结构

```
class Ival_box
{
protected:
    int val;
    int low, high;
    bool changed;
public:
    Ival_box(int ll, int hh){
        changed = false;
        val = low = ll;
        high = hh;
    }
    virtual int get_value(){
        changed = false;
        return val;
    }
    virtual void set_value(int i){
        changed = true;
        val = i;
    } // 为用户
    virtual void reset_value(int i){
        changed = false;
        val = i;
    } // 为应用
    virtual void prompt() {}
    virtual bool was_changed() const{
        return changed;
    }
};
```

Ival_box的使用

```
void interact(Ival_box* pb)
{
    pb->prompt();
    int i = pb->get_value();
    if(pb->was_changed()){
        //...
    }
    else {
        //...
    }
}
```

```
class Ival_slider:public Ival_box{};
void some_fct()
{
    Ival_box* p1 = new Ival_slider(0,5);
    interact(p1);
    Ival_box* p2 = new Ival_dial(1,12);
    interact(p2);
}
```

- 大部分图形界面系统都提供了一个类，其中定义了屏幕实体的基本性质，如果采用“Big Bucks Inc”的系统，各个类定义如下

```
class Ival_box : public BBwindow{/*…*/};
class Ival_slider : public Ival_box{/*…*/};
class Ival_dial : public Ival_box{/*…*/};
class Flashing_ival_slider : public Ival_slider{/*…*/};
class Popup_ival_slider : public Ival_slider{/*…*/};
```

12.4.1.1 评论

- 缺点1：使用BBWindow作为Ival_box的基类(BBWindow只是一个实现细节)，如果想让Ival_box能够在其他的系统中使用，必须定义多个不同版本

```
class Ival_box : public BBwindow{/*…*/};
class Ival_box : public CWwindow{/*…*/};
class Ival_box : public IBwindow{/*…*/};
class Ival_box : public LSwindow{/*…*/};
```

- 缺点2：每个派生类都共享着在Ival_box里声明的基本数据，也即“在Ival_box中不应该包含具体数据
- 缺点3：修改BBWindow将使得重新编译甚至重写相关的代码
- 缺点3：无法在一个混合式的环境(多种图形界面系统)里使用

12.4.2 抽象类

- 解决上述问题
  - 用户界面系统应该是一个实现细节，对于不希望了解它的用户应是隐蔽的
  - 类Ival_box应该不包括数据
  - 在用户界面修改后，不应该要求重新编译那些使用了Ival_box这一族类的代码
  - 针对不同界面系统的Ival_box应能在我们的程序里共存
- 本小节介绍的是一种能够最清晰地映射到C++语言中的途径

将Ival_box设计为一个抽象界面

```
class Ival_box {
public:
    virtual int get_value() = 0;
    virtual void set_value(int i)
= 0;
    virtual void reset_value(int
i) = 0;
    virtual void prompt() = 0;
    virtual bool was_changed()
const = 0;
    virtual ~Ival_box() {}
};
```

```
class Ival_slider : public Ival_box,
protected BBwindow
{
public:
    Ival_slider(int, int);
    ~Ival_slider();
    int get_value();
    void set_value(int i);
    //..
protected:
    //覆盖BBwindow虚函数的函数
    //例如：draw(), mousehit()...
private:
    // slider所需要的数据
};// 多重继承的方式
```

其他

- 虚析构函数假定派生类需要某种清理工作，因此，用户需要在派生类里面适当地覆盖该虚析构函数，虚析构函数保证会调用有关的类析构函数，完成正确的清理工作

```
void f(Ival_box* p){
    /*…*/ delete p;
}
```

- 现在的类层次结构

```
class Ival_box {/*…*/};
class Ival_slider : public Ival_box, protected BBwindow{/*…*/};
class Ival_dial : public Ival_box, protected BBwindow{/*…*/};
class Flashing_ival_slider : public Ival_slider{/*…*/};
class Popup_ival_slider : public Ival_slider{/*…*/};
```

12.4.3 其他实现方式

- 与传统方式相比，这种设计更加清晰也更容易维护，而且并不低效
- 但是没有解决版本控制问题

```
class Ival_box {/*…*/};
class Ival_slider : public Ival_box, protected BBwindow{/*…*/};
class Ival_slider : public Ival_box, protected CWwindow{/*…*/};
```

- 一个明显的解决方案是采用不同的名字，定义几个不同的Ival_slider类

```
class Ival_box {/*…*/};
class BB_ival_slider : public Ival_box, protected BBwindow{/*…*/};
class CW_ival_slider : public Ival_box, protected CWwindow{/*…*/};
```

其他实现方式(2)

- 进一步设计，可以将Ival_box和实现细节隔离开，从Ival_box派生出来一个抽象的Ival_slider类，然后从它派生出来针对各个系统的Ival_slider类

```
class Ival_box {/*…*/};
class Ival_slider:public Ival_box{/*…*/};
class BB_ival_slider:public Ival_slider, protected BBwindow{/*…*/};
class CW_ival_slider:public Ival_slider, protected CWwindow{/*…*/};
```

- 通常，还可以利用实现层次方面的一些更特殊的类，把事情做的更好一些

```
class BB_ival_slider:public Ival_slider, protected BBslider{/*…*/};
class CW_ival_slider:public Ival_slider, protected CWslider{/*…*/};
```

其他实现方式(3)

- 最终，完整的类层次可以是将我们原来的面向应用的概念层次结构当作界面而组成的

```
class Ival_box {/*…*/};
class Ival_slider : public Ival_box{/*…*/};
class Ival_dial : public Ival_box {/*…*/};
class Flashing_ival_slider : public Ival_slider{/*…*/};
class Popup_ival_slider : public Ival_slider{/*…*/};

class BB_ival_slider:public Ival_slider,protected BBslider{/*…*/};
class BB_flashing_ival_slider:public Flashing_ival_slider, protected
BBwindow_with_bells_and_whistles{/*…*/};
class BB_popup_ival_slider:public Popup_ival_slider,protected
BBslider{/*…*/};
class CW_ival_slider:public Ival_slider,protected CWslider{/*…*/};
```

12.4.3.1 评论

- 这种抽象类的设计非常灵活，而且，与那种依赖于一个定义用户界面系统的公共基类的等价设计方式比较，处理起来几乎同样容易
- 从应用系统的角度看，这两种设计的意义是等价的，几乎所有的代码都能不加修改，而且能以同样的方式使用
- 使用抽象类设计，几乎所有的用户代码都得到了保护，能够抵御实现层次上的结构变化，而且在这种变化后不需要重新编译

12.4.4 对象创建的局部化

- 对象的创建必须通过特定于实现的名字去做，例如CW_ival_dial和BB_flashing_ival_slider等，我们希望尽可能减少出现这些特殊名字的地方
- 解决的方法是引入一个间接(可以通过多种方式做到)，一种简单的方式就是引入一个抽象类，用来表示一组创建操作

```
class ival_maker
{
public:
    virtual Ival_dial* dial(int,int)=0; // 做拨盘
    virtual Popup_ival_slider* popup_slider(int,int) = 0;
    //做弹出式滑块
}; // 这样的类被称为是一个工厂(factory)，它的函数有时被称为
    // 虚构造函数(有点易于令人误解)
```

对象创建的局部化

```
class BB_maker : public ival_maker // 做BB版本
{
public:
    Ival_dial* dial(int,int);
    Popup_ival_slider*
        popup_slider(int,int);
};
class LS_maker : public ival_maker // 做LS版本
{
public:
    Ival_dial* dial(int,int);
    Popup_ival_slider*
        popup_slider(int,int);
};
```

对象创建的局部化

```
Ival_dial* BB_maker::dial(int a, int b)
{
    return new BB_ival_dial(a,b);
}

Ival_dial* LS_maker::dial(int a, int b)
{
    return new LS_ival_dial(a,b);
}

void user(Ival_maker* pim)
{
    Ival_box* pb=pim->dial(0,99);
    //...
}
```

```
BB_maker BB_impl;
// 为BB用户
LS_maker LS_impl;
// 为LS用户
void driver()
{
    user(&BB_impl);
    // 使用BB
    user(&LS_impl);
    // 使用LS
}
```

12.5 类层次结构和抽象类

- 一个抽象类就是一个界面
- 类层次结构是一种逐步递增地建立类的方式，位于其中的类一方面为用户提供了有用的功能，同时也作为实现更高级或者更特殊的类的构造基础块，这种层次结构对于支持以逐步求精方式进行的程序设计是非常理想的
- 抽象类的层次结构是一种表述概念的清晰而强有力的方法，又不会用实现细节或者运行时的额外开销去妨碍这种表述

类层次结构和抽象类

- 虚函数的调用是廉价的，与它跨过的抽象边界的种类无关，调用一个抽象类的成员函数的开销与调用任何其他virtual函数完全一样
- 结论：一个系统展现给用户的应该是一个抽象类的层次结构，其实现所用的是一个传统的层次结构

12.6 忠告

- [1]避免类型域
- [2]用指针和引用避免切割问题
- [3]用抽象类设计的中心集中到提供清晰的界面方面
- [4]用抽象类使界面最小化
- [5]用抽象类从界面中排除实现细节
- [6]用虚函数使新的实现能够添加进来，又不会影响用户代码
- [7]用抽象类去尽可能减少用户代码的重新编译
- [8]用抽象类使不同的实现能够共存
- [9]一个有虚函数的类应该有一个虚析构函数
- [10]抽象类通常不需要构造函数
- [11]让不同概念的表示也不相同