



# 第 8 章 多态性

郑 莉 清华大学

教材：C++语言程序设计（第5版） 郑莉 清华大学出版社

# 目录

- 虚函数
- 抽象类

# 虚函数

## 例8-4通过虚函数实现运行时多态

现在我们来改进一下第7章的程序

## 例8-4通过虚函数实现运行时多态

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

class Base1 {
public:
    virtual void display() const; //虚函数
};

void Base1::display() const {
    cout << "Base1::display()" << endl;
}
```

## 例8-4通过虚函数实现运行时多态

```
class Base2:public Base1 {  
public:  
    virtual void display() const;  
};  
void Base2::display() const {  
    cout << "Base2::display()" << endl;  
}  
class Derived: public Base2 {  
public:  
    virtual void display() const;  
};  
void Derived::display() const {  
    cout << "Derived::display()" << endl;  
}
```



## 例8-4通过虚函数实现运行时多态

```
void fun(Base1 *ptr) {  
    ptr->display();  
}
```

```
int main() {  
    Base1 base1;  
    Base2 base2;  
    Derived derived;  
    fun(&base1);  
    fun(&base2);  
    fun(&derived);  
    return 0;  
}
```

运行结果：  
Base1::display()  
Base2::display()  
Derived::display()



# 初识虚函数

- 用virtual关键字说明的函数
- 虚函数是实现运行时多态性基础
- C++中的虚函数是动态绑定的函数
- 虚函数必须是非静态的成员函数，虚函数经过派生之后，就可以实现运行过程中的多态。



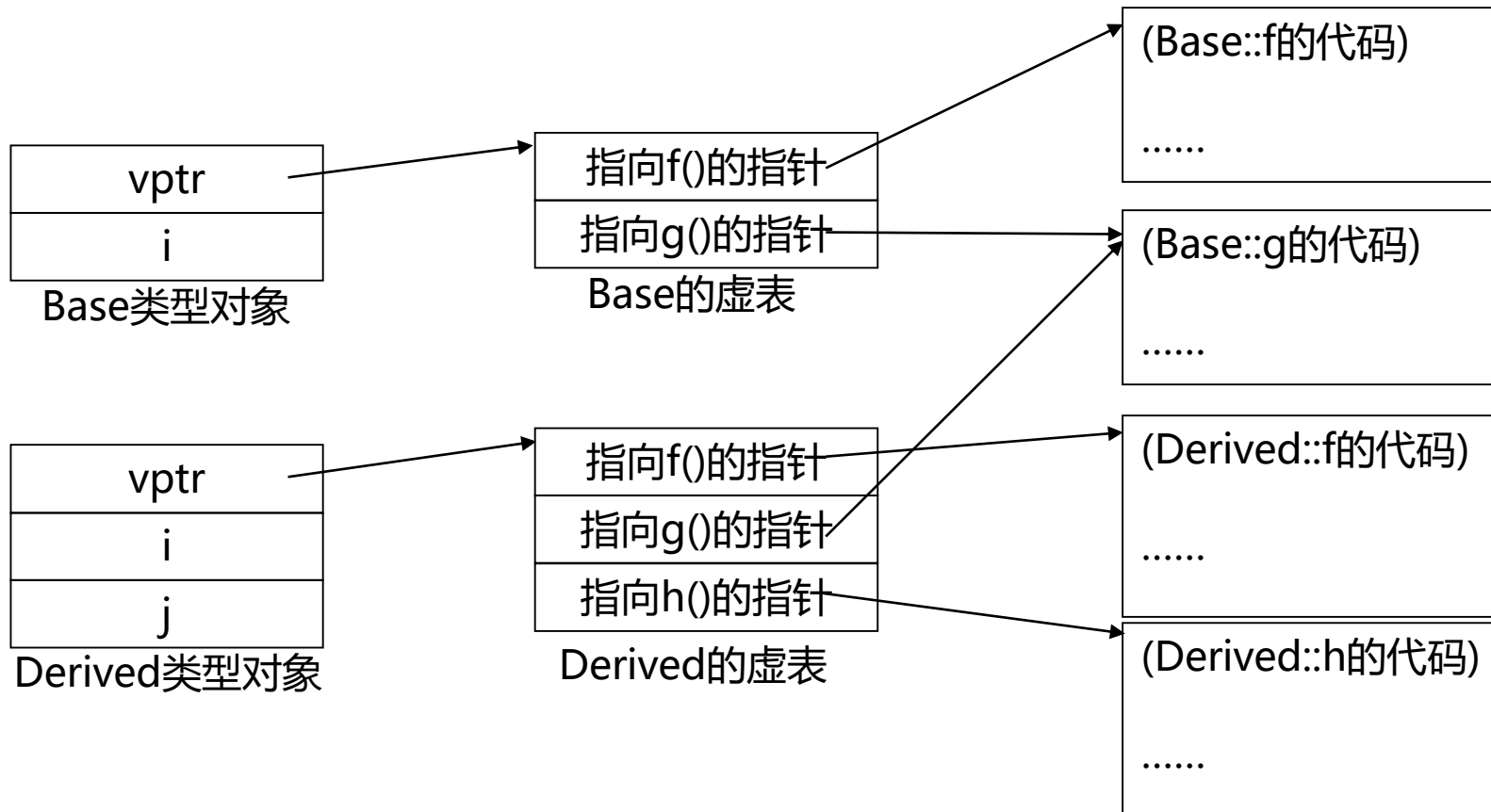
# 虚表与动态绑定

- 虚表
  - 每个多态类有一个虚表 (virtual table)
  - 虚表中有当前类的各个虚函数的入口地址
  - 每个对象有一个指向当前类的虚表的指针 (虚指针vptr)
- 动态绑定的实现
  - 构造函数中为对象的虚指针赋值
  - 通过多态类型的指针或引用调用成员函数时，通过虚指针找到虚表，进而找到所调用的虚函数的入口地址
  - 通过该入口地址调用虚函数

# 虚表示意图

```
class Base {  
public:  
    virtual void f();  
    virtual void g();  
private:  
    int i;  
};
```

```
class Derived: public Base {  
public:  
    virtual void f(); //覆盖Base::f  
    virtual void h(); //新增的虚函数  
private:  
    int j;  
};
```



# virtual 关键字

- 派生类可以不显式地用virtual声明虚函数，这时系统就会用以下规则来判断派生类的一个函数成员是不是虚函数：
  - 该函数是否与基类的被覆盖的虚函数有相同的名称、参数个数及对应参数类型、cv限定符（是否const）、引用限定符（&或&&，本课程不介绍）；
  - 该函数的返回值类型是否与基类被覆盖的虚函数返回值类型相同， 或者可以隐含转换为基类被覆盖的的虚函数的返回值类型；
- 如果派生类的函数满足上述条件，就会自动确定为虚函数。这时，派生类的虚函数便覆盖了基类的虚函数。
- 派生类中的虚函数还会隐藏基类中同名函数的所有其它重载形式。
- 一般习惯于在派生类的函数中也使用virtual关键字，以增加程序的可读性。

## 哪些成员函数可以是虚函数

- 一般非静态成员函数可以是虚函数
- 构造函数不能是虚函数
- 析构函数可以是虚函数

# 一般虚函数成员

- 虚函数的声明

`virtual` 函数类型 函数名 (形参表) ;

- 虚函数声明只能出现在类定义中的函数原型声明中，而不能在成员函数实现的时候。
- 在派生类中可以对基类中的成员函数进行覆盖。
- 虚函数一般不声明为内联函数，因为对虚函数的调用需要动态绑定，而对内联函数的处理是静态的。

# 虚析构函数

为什么需要虚析构函数？

- 可能通过基类指针删除派生类对象；

## 例8-5 虚析构函数举例

```
#include <iostream>
using namespace std;
class Base {
public:
    ~Base(); //不是虚函数
};
Base::~Base() {
    cout << "Base destructor" << endl;
}
class Derived: public Base{
public:
    Derived();
    ~Derived(); //不是虚函数
private:
    int *p;
};
```

```
Derived::Derived() {
    p = new int(0);
}
Derived::~Derived() {
    cout << "Derived destructor" << endl;
    delete p;
}

void fun(Base* b) {
    delete b; //静态绑定, 只会调用~Base()
}

int main() {
    Base *b = new Derived();
    fun(b);
    return 0;
}
```

运行结果:  
Base destructor



## 例8-5 虚析构函数举例

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual ~Base();
};
Base::~~Base() {
    cout << "Base destructor" << endl;
}
class Derived: public Base{
public:
    Derived();
    virtual ~Derived();
private:
    int *p;
};
```

```
Derived::Derived() {
    p = new int(0);
}
Derived::~~Derived() {
    cout << "Derived destructor" << endl;
    delete p;
}

void fun(Base* b) {
    delete b;
}

int main() {
    Base *b = new Derived();
    fun(b);
    return 0;
}
```

运行结果：  
Derived destructor  
Base destructor





- 如果你打算允许其他人通过基类指针调用对象的析构函数（通过delete这样做是正常的），就需要让基类的析构函数成为虚函数，否则执行delete的结果是不确定的。

# 抽象类

## 例8-6 抽象类举例

```
//8_6.cpp
#include <iostream>
using namespace std;

class Base1 {
public:
    virtual void display() const = 0;    //纯虚函数
};

class Base2: public Base1 {
public:
    virtual void display() const; //覆盖基类的虚函数
};

void Base2::display() const {
    cout << "Base2::display()" << endl;
}
```



## 例8-6 抽象类举例

```
class Derived: public Base2 {  
public:  
    virtual void display() const; //覆盖基类的虚函数  
};  
void Derived::display() const {  
    cout << "Derived::display()" << endl;  
}  
void fun(Base1 *ptr) {  
    ptr->display();  
}  
int main() {  
    Base2 base2;  
    Derived derived;  
    fun(&base2);  
    fun(&derived);  
    return 0;  
}
```

运行结果:

Base2::display()

Derived::display()



# 纯虚函数

- 纯虚函数是一个在基类中声明的虚函数，它在该基类中没有定义具体的操作内容，要求各派生类根据实际需要定义自己的版本，纯虚函数的声明格式为：

```
virtual 函数类型 函数名(参数表) = 0;
```

# 抽象类

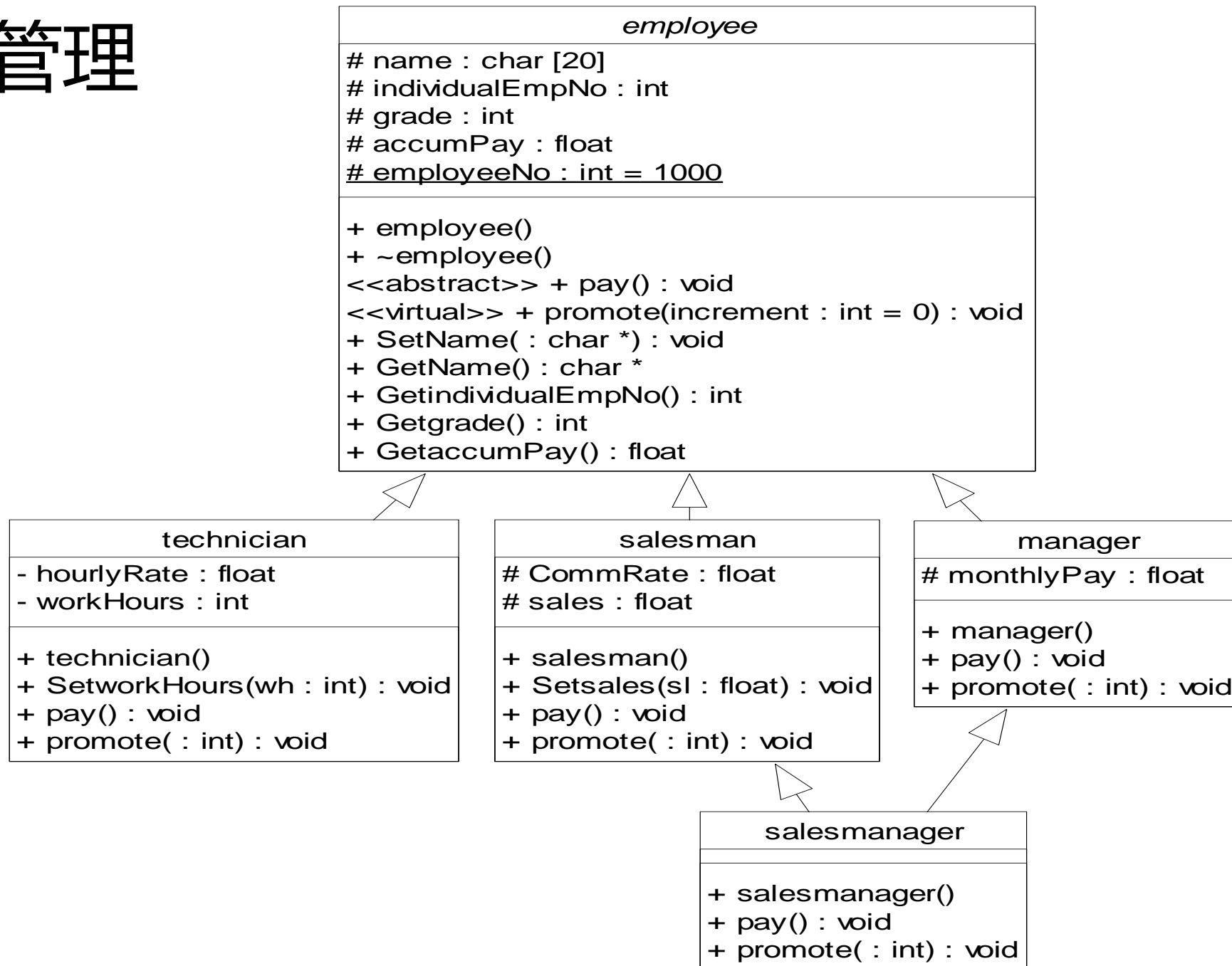
- 带有纯虚函数的类称为抽象类：

```
class 类名
{
    virtual 类型 函数名(参数表)=0;
    //其他成员.....
}
```

# 抽象类的作用

- 抽象类为抽象和设计的目的而声明;
- 将有关的数据和行为组织在一个继承层次结构中, 保证派生类具有要求的行为;
- 对于暂时无法实现的函数, 可以声明为纯虚函数, 留给派生类去实现;
- 注意
  - 抽象类只能作为基类来使用;
  - 不能定义抽象类的对象。

# 例： 人员信息管理





## 例：人员信息管理

```
//employee.h
class employee{
protected:
    char name[20]; //姓名
    int individualEmpNo; //个人编号
    int grade; //级别
    float accumPay; //月薪总额
    static int employeeNo; //本公司职员编号目前最大值
public:
    employee(); //构造函数
    ~employee(); //析构函数
    virtual void pay()=0; //计算月薪函数（纯虚函数）
    virtual void promote(int increment=0); //升级函数（虚函数）
    void SetName(char *); //设置姓名函数
    char * GetName(); //提取姓名函数
    int GetindividualEmpNo(); //提取编号函数
    int Getgrade(); //提取级别函数
    float GetaccumPay(); //提取月薪函数
};
```



## 例：人员信息管理

```
class technician:public employee    //兼职技术人员类
{
private:
    float hourlyRate;    //每小时酬金
    int workHours;    //当月工作时数
public:
    technician(); //构造函数
    void SetworkHours(int wh);    //设置工作时数函数
    void pay();    //计算月薪函数
    void promote(int); //升级函数
};
```

## 例：人员信息管理

```
class salesman:virtual public employee    //兼职推销员类
{
protected:
    float CommRate; //按销售额提取酬金的百分比
    float sales;    //当月销售额
public:
    salesman(); //构造函数
    void Setsales(float sl); //设置销售额函数
    void pay(); //计算月薪函数
    void promote(int); //升级函数
};
```

## 例：人员信息管理

```
class manager:virtual public employee //经理类
{
protected:
    float monthlyPay; //固定月薪数
public:
    manager(); //构造函数
    void pay(); //计算月薪函数
    void promote(int); //升级函数
};
```

```
class salesmanager:public manager,public salesman //销售经理类
{
public:
    salesmanager(); //构造函数
    void pay(); //计算月薪函数
    void promote(int); //升级函数
};
```



## 例：人员信息管理

```
//employee.cpp
#include<iostream>
#include<cstring>
#include"employee.h"
using namespace std;
int employee::employeeNo=1000;    //员工编号基数为1000

employee::employee()
{
    individualEmpNo=employeeNo++; //新输入的员工编号
    为目前最大编号加1
    grade=1;    //级别初值为1
    accumPay=0.0;}    //月薪总额初值为0

employee::~employee()
{}
```



## 例：人员信息管理

```
void employee::promote(int increment)
{
    grade+=increment; } //升级，提升的级数由increment指定
```

```
void employee::SetName(char* names)
{
    strcpy(name,names); } //设置姓名
```

```
char* employee::GetName()
{
    return name;} //提取成员姓名
```

```
int employee::GetindividualEmpNo()
{
    return individualEmpNo;} //提取成员编号
```

```
int employee::Getgrade()
{
    return grade;} //提取成员级别
```

```
float employee::GetaccumPay()
{
    return accumPay;} //提取月薪
```



## 例：人员信息管理

```
technician::technician()
{
    hourlyRate=100;} //每小时酬金100元

void technician::SetworkHours(int wh)
{
    workHours=wh;} //设置工作时间

void technician::pay()
{
    accumPay=hourlyRate*workHours;} //计算月薪，按小时计酬

void technician::promote(int)
{
    employee::promote(2); } //调用基类升级函数，升2级
```

## 例：人员信息管理

```
salesman::salesman()
{
    CommRate=0.04;} //销售提成比例4%

void salesman::Setsales(float sl)
{
    sales=sl;} //设置销售额

void salesman::pay()
{
    accumPay=sales*CommRate;} //月薪=销售提成

void salesman::promote(int)
{
    employee::promote(0); } //调用基类升级函数，升0级
```



## 例：人员信息管理

```
manager::manager()
{
    monthlyPay=8000;}    //固定月薪8000元

void manager::pay()
{
    accumPay=monthlyPay;}    //月薪总额即固定月薪数

void manager::promote(int )
{
    employee::promote(3);}    //调用基类升级函数，升3级

salesmanager::salesmanager()
{
    monthlyPay=5000;
    CommRate=0.005;}

void salesmanager::pay()
{
    accumPay=monthlyPay+CommRate*sales; } //月薪=固定月薪+销售提成

void salesmanager::promote(int)
{
    employee::promote(2);}    //调用基类升级函数，升2级
```



## 例：人员信息管理

```
//main.cpp
#include<iostream>
#include<cstring>
#include"employee.h"
using namespace std;
int main(){
    manager m1;
    technician t1;
    salesmanager sm1;
    salesman s1;
    char namestr[20];          //输入雇员姓名时首先临时存放在namestr中
    employee *emp[4]={&m1,&t1,&sm1,&s1};
    int i;
    for(i=0;i<4;i++) {
        cout<<"请输入下一个雇员的姓名:";
        cin>>namestr;
        emp[i]->SetName(namestr);    //设置每个成员的姓名
        emp[i]->promote();           //升级，通过基类指针访问各派生类函数
    }
```



## 例：人员信息管理

```
cout<<"请输入兼职技术人员"<<t1.GetName()<<"本月的工作时数:";  
int ww;  
cin>>ww;  
t1.SetworkHours(ww);    //设置工作时间
```

```
cout<<"请输入销售经理"<<sm1.GetName()<<"所管辖部门本月的销售总额:";  
float sl;  
cin>>sl;  
sm1.Setsales(sl);    //设置销售额
```

```
cout<<"请输入推销员"<<s1.GetName()<<"本月的销售额:";  
cin>>sl;  
s1.Setsales(sl);    //设置销售额
```



## 例：人员信息管理

```
for(i=0;i<4;i++)
{
    emp[i]->pay(); //计算月薪，通过基类指针访问各派生类函数
    cout<<emp[i]->GetName()<<"编号"<<emp[i]->GetindividualEmpNo()
        <<"级别为"<<emp[i]->Getgrade()<<"级，本月工资 "
        <<emp[i]->GetaccumPay()<<endl;
}
}
```

运行结果：

请输入下一个雇员的姓名:Zhang

请输入下一个雇员的姓名:Wang

请输入下一个雇员的姓名:Li

请输入下一个雇员的姓名:Zhao

请输入兼职技术人员Wang本月的工作时数:40

请输入销售经理Li所管辖部门本月的销售总额:400000

请输入推销员Zhao本月的销售额:40000

Zhang编号1000级别为4级，本月工资8000

Wang编号1001级别为3级，本月工资4000

Li编号1002级别为3级，本月工资7000

Zhao编号1003级别为1级，本月工资1600



# override 与 final

override与final都不是语言关键字 (keyword) , 只有在特定的位置才有特别含意, 其他地方仍旧可以作为一般标识符 (identifier) 使用。

# override

- 显式函数覆盖
- 声明该函数必须覆盖基类的虚函数，编译器可发现“未覆盖”错误
- 复习：覆盖要求
  - 函数签名 (signature) 完全一致
  - 函数签名包括：函数名 参数列表 const

## 例：因为漏写了const导致未成功覆盖

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

class Base {
public:
    virtual void f1(int) const;
    virtual ~Base() {
    };
};

void Base::f1(int) const {
    cout << "Base f1" << endl;
    return;
}
```

```
class Derived: public Base {
public:
    void f1(int);
    ~Derived() {
    };
};

void Derived::f1(int) {
    cout << "derived f1" << endl;
}
```

```
int main() {
    Base *b;
    b = new Base;
    b->f1(1);
    b = new Derived;
    b->f1(1);
    return 0;
}
```

运行结果  
Base f1  
Base f1

# 显式覆盖的作用

- 声明显式函数覆盖，在编译期间发现未覆盖的错误。
- 运用显式覆盖，编译器会检查派生类中声明override的函数，在基类中是否存在可被覆盖的虚函数，若不存在，则会报错。

- 例：

```
struct Base {  
    virtual void some_func(float);  
};  
struct Derived : Base {  
    virtual void some_func(int) override; // 错误：Derive::some_func並沒有override  
    Base::some_func  
    virtual void some_func(float) override; // 正确  
};
```



# final

用来避免类被继承，或是基类的函数被覆盖

## 例：final类和final函数

```
struct Base1 final { };
```

```
struct Derived1 : Base1 { }; // 编译错误：Base1为final，不允许被继承
```

```
struct Base2 {  
    virtual void f() final;  
};
```

```
struct Derived2 : Base2 {  
    void f(); // 编译错误：Base2::f 为final，不允许被覆盖  
};
```

# 运算符重载概述

# 思考

- 有如下复数类定义：

```
class Complex {  
public:  
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) { }  
    void display() const;           //输出复数  
private:  
    double real; //复数实部  
    double imag; //复数虚部  
};
```

- 能对复数类对象使用+、-运算符吗？

# 运算符重载的意义

- 运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时导致不同的行为。
- 针对自定义的类，可以对原有运算符进行重载。
- 例如：
  - 使复数类的对象可以用 “+” 运算符实现加法；
  - 是时钟类对象可以用 “++” 运算符实现时间增加1秒。

# 运算符重载的规定

- C++ 几乎可以重载全部的运算符，而且只能够重载C++中已经有的。
  - 不能重载的运算符： `"."` 、 `"*"` 、 `"::"` 、 `"?:"`
- 重载之后运算符的优先级和结合性都不会改变。
- 可以重载为类的非静态成员函数。
- 可以重载为非成员函数（必要时可以声明为友元）。

# 运算符重载为成员函数

## 例8-1 复数类加减法运算重载为成员函数

- 要求:
  - 将+、-运算重载为复数类的成员函数。
- 规则:
  - 实部和虚部分别相加减。
- 操作数:
  - 两个操作数都是复数类的对象。



## 例8-1 复数类加减法运算重载为成员函数

```
#include <iostream>
using namespace std;
class Complex {
public:
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) { }
    //运算符+重载成员函数
    Complex operator + (const Complex &c2) const;
    //运算符-重载成员函数
    Complex operator - (const Complex &c2) const;
    void display() const;    //输出复数
private:
    double real; //复数实部
    double imag;    //复数虚部
};
```



## 例8-1 复数类加减法运算重载为成员函数

```
Complex Complex::operator+(const Complex &c2) const{  
    //创建一个临时无名对象作为返回值  
    return Complex(real+c2.real, imag+c2.imag);  
}
```

```
Complex Complex::operator-(const Complex &c2) const{  
    //创建一个临时无名对象作为返回值  
    return Complex(real-c2.real, imag-c2.imag);  
}
```

```
void Complex::display() const {  
    cout<<"("<<real<<","<<imag<<")"<<endl;  
}
```



## 例8-1 复数类加减法运算重载为成员函数

```
int main() {  
    Complex c1(5, 4), c2(2, 10), c3;  
    cout << "c1 = "; c1.display();  
    cout << "c2 = "; c2.display();  
    c3 = c1 - c2; //使用重载运算符完成复数减法  
    cout << "c3 = c1 - c2 = "; c3.display();  
    c3 = c1 + c2; //使用重载运算符完成复数加法  
    cout << "c3 = c1 + c2 = "; c3.display();  
    return 0;  
}
```

输出的结果为：

c1 = (5, 4)

c2 = (2, 10)

c3 = c1 - c2 = (3, -6)

c3 = c1 + c2 = (7, 14)



# 重载为类成员的运算符函数定义形式

函数类型 operator 运算符 (形参)

{

.....

}

参数个数=原操作数个数-1 (后置++、--除外)

## 双目运算符重载规则

- 如果要重载 B 为类成员函数，使之能够实现表达式 `oprd1 B oprd2`，其中 `oprd1` 为 A 类对象，则 B 应被重载为 A 类的成员函数，形参类型应该是 `oprd2` 所属的类型。
- 经重载后，表达式 `oprd1 B oprd2` 相当于 `oprd1.operator B(oprd2)`

## 思考：

- 单目运算符前置语后置++、--如何区分呢？

## 例8-2重载前置++和后置++为时钟类成员函数

- 前置单目运算符，重载函数没有形参
- 后置++运算符，重载函数需要有一个int形参
- 操作数是时钟类的对象。
- 实现时间增加1秒钟。

## 例8-2重载前置++和后置++为时钟类成员函数

```
#include <iostream>
using namespace std;
class Clock { //时钟类定义
public:
    Clock(int hour = 0, int minute = 0, int second = 0);
    void showTime() const;
    //前置单目运算符重载
    Clock& operator ++ ();
    //后置单目运算符重载
    Clock operator ++ (int);
private:
    int hour, minute, second;
};
```





## 例8-2重载前置++和后置++为时钟类成员函数

```
Clock::Clock(int hour, int minute, int second) {  
    if (0 <= hour && hour < 24 && 0 <= minute && minute < 60  
        && 0 <= second && second < 60) {  
        this->hour = hour;  
        this->minute = minute;  
        this->second = second;  
    } else  
        cout << "Time error!" << endl;  
}  
void Clock::showTime() const { //显示时间  
    cout << hour << ":" << minute << ":" << second << endl;  
}
```



## 例8-2重载前置++和后置++为时钟类成员函数

```
Clock & Clock::operator ++ () {  
    second++;  
    if (second >= 60) {  
        second -= 60; minute++;  
        if (minute >= 60) {  
            minute -= 60; hour = (hour + 1) % 24;  
        }  
    }  
    return *this;  
}
```

```
Clock Clock::operator ++ (int) {  
    //注意形参表中的整型参数  
    Clock old (*this);  
    ++(*this);    //调用前置 “++” 运算符  
    return old;  
}
```



## 例8-2重载前置++和后置++为时钟类成员函数

```
int main() {  
    Clock myClock(23, 59, 59);  
    cout << "First time output: ";  
    myClock.showTime();  
    cout << "Show myClock++:   ";  
    (myClock++).showTime();  
    cout << "Show ++myClock:   ";  
    (++myClock).showTime();  
    return 0;  
}
```

运行结果:

First time output: 23:59:59

Show myClock++: 23:59:59

Show ++myClock: 0:0:1



## 前置单目运算符重载规则

- 如果要重载 U 为类成员函数，使之能够实现表达式 U oprd，其中 oprd 为 A 类对象，则 U 应被重载为 A 类的成员函数，无形参。
- 经重载后，  
表达式 U oprd 相当于 oprd.operator U()

## 后置单目运算符 ++和--重载规则

- 如果要重载 ++或--为类成员函数，使之能够实现表达式 `oprd++` 或 `oprd--`，其中 `oprd` 为A类对象，则 ++或-- 应被重载为 A 类的成员函数，且具有一个 `int` 类型形参。
- 经重载后，表达式 `oprd++` 相当于 `oprd.operator ++(0)`

# 运算符重载为非成员函数

有些运算符不能重载为成员函数，例如二元运算符的左操作数不是对象，或者是不能由我们重载运算符的对象

## 例8-3

### 重载Complex的加减法和 “<<” 运算符为非成员函数

- 将+、-（双目）重载为非成员函数，并将其声明为复数类的友元，两个操作数都是复数类的常引用。
- 将<<（双目）重载为非成员函数，并将其声明为复数类的友元，它的左操作数是std::ostream引用，右操作数为复数类的常引用，返回std::ostream引用，用以支持下面形式的输出：

```
cout << a << b;
```

该输出调用的是：

```
operator << (operator << (cout, a), b);
```

## 例8-3 重载复数的加减法和 “&lt;&lt;” 运算符为非成员函数

```
//8_3.cpp
#include <iostream>
using namespace std;

class Complex {
public:
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) { }
    friend Complex operator+(const Complex &c1, const Complex &c2);
    friend Complex operator-(const Complex &c1, const Complex &c2);
    friend ostream & operator<<(ostream &out, const Complex &c);
private:
    double real; //复数实部
    double imag; //复数虚部
};
```





## 例8-3 重载复数的加减法和 "&lt;&lt;" 运算符为非成员函数

```
Complex operator+(const Complex &c1, const Complex &c2){  
    return Complex(c1.real+c2.real, c1.imag+c2.imag);  
}  
Complex operator-(const Complex &c1, const Complex &c2){  
    return Complex(c1.real-c2.real, c1.imag-c2.imag);  
}  
  
ostream & operator<<(ostream &out, const Complex &c){  
    out << "(" << c.real << ", " << c.imag << ")";  
    return out;  
}
```

## 例8-3 重载复数的加减法和 “&lt;&lt;” 运算符为非成员函数

```
int main() {  
    Complex c1(5, 4), c2(2, 10), c3;  
    cout << "c1 = " << c1 << endl;  
    cout << "c2 = " << c2 << endl;  
    c3 = c1 - c2; //使用重载运算符完成复数减法  
    cout << "c3 = c1 - c2 = " << c3 << endl;  
    c3 = c1 + c2;      //使用重载运算符完成复数加法  
    cout << "c3 = c1 + c2 = " << c3 << endl;  
    return 0;  
}
```

# 运算符重载为非成员函数的规则

- 函数的形参代表依自左至右次序排列的各操作数。
- 重载为非成员函数时
  - 参数个数=原操作数个数（后置++、--除外）
  - 至少应该有一个自定义类型的参数。
- 后置单目运算符 ++和--的重载函数，形参列表中要增加一个int，但不必写形参名。
- 如果在运算符的重载函数中需要操作某类对象的私有成员，可以将此函数声明为该类的友元。

# 运算符重载为非成员函数的规则

- 双目运算符 B重载后,  
表达式 `opr1 B opr2`  
等同于 `operator B(opr1,opr2 )`
- 前置单目运算符 B重载后,  
表达式 `B oprd`  
等同于 `operator B(oprd )`
- 后置单目运算符 ++和--重载后,  
表达式 `opr B`  
等同于 `operator B(oprd,0 )`

# 小结

- 主要内容
  - 多态性的概念、运算符重载、虚函数、纯虚函数、抽象类、override 和 final
- 达到的目标
  - 掌握运算符重载原理和方法
  - 理解动态多态性的原理，掌握通过虚函数实现的多态性的方法
  - 掌握纯虚函数和抽象类的概念和设计方法