程序设计实习

信息科学技术学院 Seto1

2025 年 11 月 23 日

前言

课程评分细则:

期末机试 35%+ 期中机考 35%+ 平时成绩 30%

平时成绩包括：课程作业 13 分 + 合作大作业 10 分 + 魔兽终极版大作业 5 分 + 上机签到 2 分

* 其他加分：ACM 校内赛 & 命题等 (待定)

Seto1

2025 年 11 月 23 日

目录

第一章　类与对象

概念：通过某种语法形式，将数据结构和操作该数据结构的函数“捆绑”在一起 → 形成一个**“类”**，通过**封装**形成“边界感”（设计程序的过程，就是设计类的过程）

例：矩形类中定义“成员变量”长和宽描述属性；定义“成员函数”计算面积、周长等描述行为.

```
class CRectangle{//相当于自定义了一种类型
    public://访问权限，默认是 private 的，需要声明，实际上实现了有效的封装
        int w, h;
        int Area() {return w * h;}
        int Perimeter(){return 2 * (w + h);}
        void Init(int w_, int h_) {w = w_; h = h_;}//实例化出对象
}; //必须有分号
//sizeof(CRectangle) = 8，是所有成员变量大小之和
/* 也可写作：
class CRectangle{
    public:
        int w, h;
        int Perimeter();
        void Init( int w_, int h_ );
};
int CRectangle::Perimeter() {return 2 * (w + h);}
*/
int main( ) {
    int w, h;
    CRectangle r; //r 是 CRectangle 类的一个对象
    cin >> w >> h;
    r.Init(w, h);
    cout << r.Area() << endl << r.Perimeter();
    return 0;
```

```
}//* 对象间可以赋值，不能比较（除非重载运算符）
```

使用类的成员变量/函数方法

```
//1) 对象名. 成员名
CRectangle r1, r2; r1.w = 5; r2.Init(3, 4);
//2) 指针-> 成员名
CRectangle r1, r2;
CRectangle * p1 = & r1; CRectangle * p2 = & r2;
p1->w = 5; p2->Init(3, 4);
//3) 引用名. 成员名
CRectangle r2;
CRectangle & rr = r2;//定义引用时，一定要初始化成某个变量；只能引用变量，不能引用表达式
rr.w = 5; rr.Init(3, 4); //rr 的值变了，r2 的值也变
//某个变量的引用，和这个变量是一回事，相当于该变量的一个别名
```

引用类型的作用

```
void swap( int & a, int & b ){int tmp = a; a = b; b = tmp;}
int n1, n2; swap(n1, n2);//代替指针

int n = 4;
int & SetValue() { return n; }//返回对 n 的引用
int main() {
    SetValue() = 40; //对返回值进行赋值 → 对 n 赋值
    cout << n;//输出：40
    return 0;
}

int n = 100;
const int & r = n; //常引用类型，n = r = 100
r = 200; //编译出错，从而实现了“只读”
n = 300; //n = r = 300
//在函数传参时，常引用和普通变量区别不大，但是常引用在内存开销上更占优势
```

补充：指针 指向某个变量的地址的变量

```
int* p = &a, *q = nullptr//指针 p 存储 a 的地址,nullptr (或 NULL) 表示空指针
cout << *p << endl; //使用 * 解引用
int *array1 = new int [3];//使用 new 分配动态内存，此时指针指向第一个元素的地址
array1 [0] = 1;
*(array + 1) = 2;//二者同样都代表访问数组元素
int *array2[4] = {new int(0),new int(1)};
cout << *array2[0] << " " << *array2[2] << endl;//会在输出 0 后运行错误
delete [] array1;//释放内存
```

辨析 常量指针、指针常量和常量指针常量:

```
const int *p = &x;// p 是一个常量指针，指向 x
*p = 15; // 错误：不能通过常量指针修改指向的值
p = &y; // 可以改变指针 p 的指向

int *const p = &x; // p 是一个指针常量，指向 x
*p = 15; // 可以通过指针常量修改指向的值
p = &y; // 错误：不能改变指针的指向

const int *const p = &x; // p 既是指向常量的指针，又是常量指针
*p = 15; // 错误：不能修改指向的值
p = &x;  // 错误：不能改变指针的指向
```

成员变量的访问权限 class 默认为 private，struct 默认为 public, 使用 private（AKA. 隐藏）的好处：方便修改成员变量的类型等属性（只需更改成员函数）而不用修改所有使用该变量的语句。

函数重载 名字相同但参数个数/类型不同的一组函数（**仅函数类型不同会报错**）。

*** 避免二义性**

```
int Max(int x1,int x2){}
int Max(double x1,double x2){}
Max(2.4,3)//不合法，二义性!
```

函数的缺省参数 可使**最右边的连续**若干个参数有缺省值（成员函数也可重载/缺省）：

```
void func(int x1,int x2=2,int x3=3){}
func(10);//ok, func(10,2,3)
func(10,8);//ok, func(10,8,3)
func(10,,3)//CE!
```

构造函数（Constructor）一种用于初始化对象的成员函数，缺省有无参数的构造函数，在自定义构造函数后消失。

```
class Complex {
    double real, imag;
public://构造函数最好是 public 的，否则不能直接用来初始化对象。
    Complex(double r, double i = 0){real = r; imag = i;} //构造函数与类名相同
    Complex(Complex c1,Complex c2){
        real = c1.real+c2.real;
        imag = c1.imag+c2.imag;/* 无返回值！ */
    }//可存在多个构造函数
};
Complex c1; // error, 没有参数
Complex * pc = new Complex; // error, 没有参数
Complex c2(2); // OK, 缺省参数 Complex c2(2, 0)
Complex c3(2, 4), c4(3, 5); //OK
Complex * pc = new Complex(3, 4); //OK
```

构造函数可在数组中应用，每赋一个初始值就调用一个构造函数.

```
class Test {
public:
    Test(int n) { } //(1)
    Test(int n, int m) { } //(2)
    Test() { } //(3)
};
Test array1[3] = {1, Test(1,2)};// 三个元素分别用 (1), (2), (3) 初始化
Test * pArray[3] = {new Test(4), new Test(1,2)};
```

```
//两个元素分别用 (1), (2) 初始化（指针必须先构造对象）
```

不允许出现形如 Test(Test a) 类似的构造函数（因为可能涉及循环调用），需要使用**复制构造函数**进行：

```
class Complex {
public :
    double real, imag;
    Complex(double r,double i){real = r;imag = i;}
    Complex(Complex & c) {//若以常量为参数，也可用 const
        real = c.real*2;
        imag = c.imag;
    }//若无此复制构造函数，则默认有一个起拷贝功能的复制构造函数
};
void Func(Complex c){}
A Func1(){A b(1,3);return b;}//获得返回值，调用复制构造函数
Complex c1;
Complex c2(c1);//初始化语句调用复制构造函数
Complex c3;c3=c1;//赋值语句不调用复制构造函数
Complex c4=c1;//这是一个初始化语句！调用复制构造函数
Func(c1);//函数中调用对象参数，调用复制构造函数
Func1();
```

类型转换构造函数

```
class Complex {
public:
    double real, imag;
    Complex(int i) { //类型转换构造函数，只有一个参数
        real = i; imag = 0;
    }//若变为 explicit Complex，则 c1 = 9 会报错，而必须使用 c1 = Complex(9)
    Complex(double r, double i) { real = r; imag = i; }
};
Complex c1 = 9;// 9 被自动转换成一个临时 Complex 对象
```

析构函数 (Destructor) 当且仅当对象消亡时，调用析构函数，本身会生成什么也不做的缺省析构函数。

```
class CString{
private :
    char * p;
public:
    CString () {
        p = new char[10];
    }
    ~ CString () {//名字与类名相同，无参数，无返回值，前面加 ~
        delete [] p;
    }//会在 return 0; 后或 delete 运算时执行
    //delete 对象数组会析构多次
};
Cstring Func1{CString obj}{return obj;}//传参消亡时也会析构（换成常引用则不调用）
int main(){
    pTest = new CString[3]; //构造函数调用 3 次
    delete [] pTest;//析构函数调用 3 次，一定要写 []!
    Cstring Obj;
    Func1(Obj);//生成临时对象，消亡析构
    return 0;//局部/全局变量消亡析构，按生成顺序逐个析构.
    //函数中的 static 静态变量形同全局变量，在函数被调用时构造，在 return 0 处析构
}
```

this 指针　指向成员函数所作用的对象. 在 C 语言中由于成员函数无法被封装，只能使用该指针确定性地指代对象:

```
class CCar {
public:
    int price; void SetPrice(int p);
};
void CCar::SetPrice(int p) { price = p; }
/* 等价于: */struct CCar{int price;};
void SetPrice(struct CCar * this, int p) { this->price = p; }
```

而在类内，非静态成员函数中可以直接使用 this 指代该函数作用对象的指针：

```
class Complex {
public:
    double real, imag;
    void Print() { cout << real << "," << imag; }
    Complex AddOne() {this->real ++; this->Print(); return * this;}
};
```

静态成员变量/函数 加了 static 关键字的成员，**实际上是全局变量/函数**，只是被显式地写进了类里，事实上为所有对象共享，于是 sizeof 运算符不会计算静态成员变量，且其**不需要通过对象就能访问**，其目的是将与类紧密相关的全局变量和函数写到类里面，而看上去像一个整体.

```
class CRectangle{
private:
    int w, h;
    static int nTotalArea, nTotalNumber;//由于 private 无法在 main 中访问
public:
    CRectangle(int w_, int h_);
    ~CRectangle();
    static void PrintTotal();
};
CRectangle::CRectangle(int w_, int h_){
    w = w_; h = h_;
    nTotalNumber ++; nTotalArea += w * h;
}
CRectangle:: ~CRectangle(){nTotalNumber --; nTotalArea -= w * h;}
void CRectangle::PrintTotal(){cout << nTotalNumber << "," << nTotalArea << endl;}
//在静态成员函数中，不能访问非静态成员变量，不能调用非静态成员函数，不能使用 this 指针
int CRectangle::nTotalNumber = 0, CRectangle::nTotalArea = 0;
// 必须在定义类定义的外面专门对静态成员变量进行声明并初始化
int main(){
    CRectangle r1(3, 3), r2(2, 2);
    CRectangle::PrintTotal();//访问静态成员函数既可以通过类名:: 函数名
    r1.PrintTotal();}//也可以像普通成员变量一样访问
```

常量对象和常量方法 const Sample Obj; 称为常量对象，值无法改变，只能调用构造函数, 析构函数和有 const 说明的函数 (**常量方法**)

```
class Sample {
    public: int value;
            void GetValue() const;//在说明后面加 const 关键字
            void func() {}
};
void Sample::GetValue() const {value = 0; func(); /*wrong*/}//定义时也要用 const
//内部不能改非静态属性的值，也不能调用同类的非常量非静态成员函数
//若想修改某非静态变量值，则可使用 multable 关键字 (multable int)
int main(){
    const Sample o;
    o.GetValue(); //常量对象上可以执行常量成员函数
    return 0;
}
```

由于函数传参会反复调用复制构造函数造成效率降低，可使用 const T& 进行传参，若想修改，必须新建变量:

```
void PrintfObj(const Sample & o){
    o.func(); //error
    Sample & r = (Sample &) o; //必须强制类型转换
    r.func(); //ok
}
```

* 函数重载的另一种类型：名字和参数表都一样，但是一**个是 const，一个不是**.

成员对象 一个类的成员变量是另一个类的对象，有成员对象的类称为**封闭类**.

```
class CTyre { //轮胎类
    int radius,width;
public:
    CTyre(int r, int w):radius(r), width(w) { }
};
class CEngine {};
```

```
class CCar { //汽车类（封闭类）
    int price; CTyre tyre; CEngine engine;
public:
    CCar(int p, int tr, int tw);//一定要声明构造函数，否则不知道成员对象如何初始化
};
CCar::CCar( int p, int tr, int tw ): price(p), tyre(tr, tw)//初始化列表
//* 常量型/引用型成员变量一定要用初始化列表初始化，普通的不一定
{};//先执行成员对象的构造函数，再执行封闭类的构造函数，析构相反。
//成员对象调用顺序与其在类中的声明顺序一致，与初始化列表中的无关!
int main(){
    CCar car(20000, 17, 225);
    CCar car1(car);//此时会调用 CTyre, CEngine 的复制构造函数
    return 0;
}
```

友元 一个类的**友元函数**可以访问该类的私有成员:

```
class CCar ; //提前声明 CCar 类，以便后面的 CDriver 类使用
class CDriver{public: void ModifyCar( CCar * pCar) ;};
class CCar{
private:
    int price;
    friend int MostExpensiveCar( CCar cars[], int total); //将全局函数声明为友元
    friend void CDriver::ModifyCar(CCar * pCar); //将另一个类的成员函数声明为友元
};
//友元类：如在 CCar 中声明 friend class CDriver 则 CDriver 类可访问 CCar 私有变量
void CDriver::ModifyCar( CCar * pCar) {pCar->price += 1000;}
int MostExpensiveCar( CCar cars[], int total) {
    int tmpMax = -1;
    for( int i = 0; i < total; ++i ) tmpMax=max(tmpMax,cars[i].price);
    return tmpMax;
}
```

* 友元类之间的关系不能传递，不能继承.

第二章　运算符重载

背景：C++ 预定义了一组运算符, 但只能用于基本的数据类型，我们希望对自定义类, 也能够直接使用 C++ 提供的运算符

运算符重载的**实质是函数重载**: “返回值类型 operator 运算符 (形参表)”:

```
class Complex {
public:
    double real, imag;
    Complex( double r = 0.0, double i= 0.0 ):real(r), imag(i) { }
    Complex operator-(const Complex & c);//运算符可被重载为成员函数
};
Complex operator+(const Complex & a, const Complex & b){//参数个数即为目数
    return Complex( a.real+b.real, a.imag+b.imag);//运算符也可被重载为普通函数
}
Complex Complex::operator-(const Complex & c){//参数个数为目数减一
    return Complex(real - c.real, imag - c.imag);
}
//a + b 等价于 operator+(a, b); a - b 等价于 a.operator-(b)
```

赋值运算符 = 重载 使得等号两边的类型可以不匹配，**只能重载为成员函数**（$=, \rightarrow, (), []$ 运算符必须为成员函数）. 例如，定义长度可变的字符串类 String:

```
class String {
private:
    char * str;
public:
    String () : str(new char[1]) { str[0] = 0; }
    const char * c_str() { return str; }
    String(const String & s) {
```

```
        str = new char[strlen(s.str)+1];
        strcpy(str, s.str);
    }//同样避免了浅拷贝
    String & operator = (const char * s);
    ~String() { delete [] str; }
};
//重载 "=" 使得 obj = "hello" 能够成立
String & String::operator = (const char * s){
//这里用 String& 是为了保留特性 (a=b)=c, 即 (a.operator=(b)).operator=(c);
    if( this == &s ) return * this;//为自赋值创造条件
    delete [] str;//注意先清空
    str = new char[strlen(s)+1];
    strcpy(str, s);
    return *this;
}//重载运算符避免了直接赋值带来的浅拷贝: S1 析构或改变时 S2 指向的地址被 delete
int main(){
    String s;
    s = "Good Luck," ; //等价于 s.operator = ("Good Luck,");
    // String s2 = “hello!” ; //error, 这是初始化语句, 没定义传参的构造函数
    return 0;
}
```

运算符重载为友元 重载为成员函数无法满足类似 5+a 的表达式，重载为普通函数又无法访问类的私有函数，于是采取两全其美的方式——重载为友元:

```
class Complex {
public:
    Complex(double r= 0.0, double i= 0.0): real(r), imaginary(i){};
    Complex operator+(int r){return Complex(real + r, imaginary);}//a+5
    friend Complex operator+(int r, const Complex & C);//5+a
private:
    double real, imaginary;
};
```

«/» **运算符的重载** 本身在 iostream 中是通过成员函数重载的：

```
class ostream {
    ostream & operator<< (int n) {Output(n);return * this;}
};
```

对于类中重载，给出一个例子：

```
class Point {
    private:
        int x,y;
    public:
        Point() {};
        friend istream& operator>>(istream& os, Point& p) {
            os >> p.x >> p.y;
            return os;
        }
        friend ostream& operator<<(ostream& os, const Point& p) {
            os << p.x << "," << p.y;
            return os;
        }
};
```

自定义数组：重载 []

```
int & operator[](int subscript){
    return ptr[subscript];
}//实际上还是在使用 C++ 中原本定义的数组
const Array & operator=( const Array & a){//重载自定义数组赋值符号
    if( ptr == a.ptr ) return * this;//防止自赋值
    delete [] ptr;
    ptr = new int[ a.size ];
    memcpy( ptr, a.ptr, sizeof(int ) * a.size);//内存拷贝函数
    size = a.size;
    return * this;}
```

重载类型转换运算符

```
operator int() const{ //必须为成员函数，不指定返回类型，形参为空
    return n;
}//不需显式调用 (int)n，在需要类型转换时自动调用
```

自增/自减运算符的重载 其特殊之处在于自增减运算符有前置/后置之分（先/后调用值）为了区分，我们规定：**前置运算符作为一元运算符重载**.

```
//以大整数类中的自增重载举例：
CHugeInt operator++(){//不传参数
    *this = *this + 1;
    return *this;
}//前置自增
CHugeInt operator++(int k){//传一个没有用的参数，即 a.operator++(0)
    CHugeInt temp = *this;//先存储原本的值，再进行自增
    *this = *this + 1;
    return temp;
}//后置自增
```

第三章　继承与派生

背景：类与类之间发生作用，之间有某些成员变量重合，希望减少重复工作量，于是提出基类和派生类（继承）的概念：

定义 若类 B 拥有类 A 的全部特点，则可定义两者间的**继承关系**：A 称为**基类**,B 称为**派生类**, 其可以添加新的成员/**重新编写**继承得到的成员，定义后不依赖于基类.

```
class CStudent {
private:
    string sName;
    int nAge;
public:
    bool IsThreeGood() { }
    void SetName( const string & name ) { sName = name; }
};//基类
class Undergraduate : public CStudent {//类名：派生方式（也有公有/私有/保护）基类名
private:
    int nDepartment;
public:
    bool IsThreeGood() { ... } //覆盖（重新编写函数）
    bool CanBaoYan() { ... }//扩充
};//继承类
```

* 虽然派生类拥有基类的全部成员（大小等于基类成员变量 + 派生类成员变量大小），但派生类的成员函数中, **不能访问基类中的 private 成员**.

解决方案 第三种访问权限：**protected**，其可被派生类的成员函数/友元函数访问

```
class Father {
    protected: int nProtected; // 保护成员
};
```

```
class Son : public Father{
void AccessFather () {
    nProtected = 1; // OK, 访问从基类继承的保护成员
    Son s; s.nProtected = 1; //OK, 访问其它同类对象的基类保护成员
    Father f; f.nProtected = 1; //wrong, f 不是函数所作用的当前对象
}
};
```

说明 继承是一个“是”的关系（一个 B 对象一定是一个 A 对象），需要区别于**复合**关系，即一个类的对象拥有作为其成员的**其它类的对象**.

```
//复合关系举例 1 圆中有点（圆心）:
class CPoint{
    double x, y;
    friend class CCircle; //友元，便于 CCirle 类操作其圆心
};
class CCircle{
    double r;
    CPoint center;//作为封闭类拥有 CPoint 类对象作为其成员
};//复合关系
//复合关系举例 2 狗中有人，人中有狗？(循环定义)
class CDog;//提前声明
class CMaster{
    CDog *dogs[10];//不使用指针会报错
};
class CDog{
    CMaster *m;
};//必须使用对象指针避免循环定义
```

覆盖 派生类可以定义一个**和基类成员同名**的成员，默认访问派生类中定义的成员，若要访问由基类定义的同名成员时，要使用作用域符号“::”

```
class base {
    int j;
public:
```

```
    int i;
    void func();
};//基类
class derived : public base{
public:
    int i;//覆盖变量
    void access();
    void func();
};//派生类
void derived::access(){
    j = 5; //error
    i = 5; base::i = 5;//前面为访问派生类，后面为基类
    func();  base::func(); //同理
}
```

派生类的构造/析构函数 在创建派生类的对象时，需要调用基类的构造函数以初始化继承的成员（该构造函数执行时**先执行基类的构造函数，再调用成员对象类的构造函数，最后执行派生类成员变量构造函数**——析构的时候完全相反）可以显式调用（像成员对象一样提供基类构造函数参数）或隐式调用（省略基类构造函数时，自动调用基类的缺省构造函数）

公有继承的赋值

```
class base {};
class derived : public base {};
base b;
derived d;
b = d;//派生类的对象可以赋值给基类对象
base & br = d;//派生类对象可以初始化基类引用
base * pb = & d;//派生类对象的地址可以赋值给基类指针
//这里 pb 虽然指向一个派生类的对象，但不能访问派生类中扩展的成员
Derived *pd = pb;//但可以强制指针类型转换以进行访问
```

直接基类和间接基类 类 A 派生类 B，类 B 派生类 C，则类 A 是类 C 的**间接基类**，在声明派生类时，派生类的首部只需要列出它的直接基类，沿着类的层次自动向上继承它的间接基类（先构造间接基类，再构造直接基类，析构相反）

第四章　多态

虚函数 前面有 virtual 关键字的函数:

```
class base {
    virtual int get() ;//virtual 关键字只用在类定义里的函数声明中
    //静态成员函数不能是虚函数,virtual 也可写在返回值后
};
int base::get() {}//写函数体时不用写 virtual
```

虚函数的作用 通过**被赋值为派生类指针**的基类指针或**被赋值为派生类对象**的基类引用，调用基类和派生类的同名虚函数时，**根据指针/引用指向的对象类型**决定调用的函数，这种机制叫做**多态**.

```
class CBase{
public:
    virtual void Func(){}
    void Func1(){Func();}//等价于 this->Func();(与指向类型一致)
};
class CDerived:public CBase{
public:
    virtual void Func(){}//实际上只需要在基类中确认其为 virtual, 派生类中可以不写
};
int main(){
    CDerived ODerived;
    CBase *p = &ODerived;
    p-> Func();//CDerived.Func();
    CBase &r = ODerived;
    r.Func1();//CBase.Func1(); 进而调用 CDerived.Func()!
    return 0;}
```

多态的实质 基类定义共同接口，派生类进行不同实现——通过基类以相同的方式操作不同派生类的行为，从而增强程序可扩充性。

例 对于一款游戏，有多种互相攻击、反击的生物，若使用非多态机制，则需要 n^2 个攻击函数（即使加了继承机制），代码量太大且新加生物代码升级成本太高，为此引入多态机制:

```
class CCreature { // " 怪物 " 类
protected :
    int m_nLifeValue, m_nPower;
public:
    virtual void Attack(CCreature * pCreature);
    virtual void Hurted(int nPower);
    virtual void FightBack(CCreature * pCreature);
};
class CDragon : public CCreature {
public:
    virtual void Attack(CCreature *p){
        p->Hurted(m_nPower);p->FightBack(this);
    }
    virtual void Hurted(int nPower);
    virtual void FightBack(CCreature * p){p->Hurted(m_nPower/2);}
};
```

就此将攻击、反击函数的代码量从 n^2 级别降至 n 级别，从而提高可扩充性.

在另一例中，用**基类指针数组**存放指向**各种派生类对象**的指针，通过遍历该数组，实现多种几何形体的各种处理, 此处展示其中将数组关于面积排序的比较函数:

```
CShape *pShapes[100];//存放各种几何形体
int MyCompare(const void* s1,const void* s2){//qsort 固定类型
    CShape **p1, **p2;//注意此处的 CShape** 类型
    p1 = (CShape**)s1, p2 = (CShape**)s2;
    double a1=(*p1)->Area(),a2=(*p2)->Area();//(*p1) 类型为 CShape*, 为基类指针
    if (a1<a2) return -1;else if (a2<a1) return 1;return 0;
}
```

于是我们得到了面向对象编程的三大基本概念: **数据抽象**（类与封装）、**继承**（派生类继承基类成员）、**动态绑定**（多态，编译器在运行时决定在哪里调用函数，即**动态联编机制**）.

多态的实现机制 有虚函数的类都有一个**虚函数表**，列出了该类的虚函数地址，由此会在内存中**多出 4 个字节**.

虚函数的访问权限 由于动态联编机制，在编译访问权限检查时检查的是**指针的类型** (即若其对应 private 则编译错误), 而真正运行访问时才是**指向对象的类型**.

虚析构函数 通过基类的指针删除派生类对象时, 通过静态联编机制**只调用基类的析构函数而不调用派生类析构函数!** （但删除一个**派生类对象**时, 先调用派生类析构函数, 然后调用基类析构函数）

⇒ **解决方案** 把**基类的析构函数**声明为 virtual（一般来说，若该类定义了虚函数，则须将其析构函数也定义成虚函数，但**不允许以虚函数作为构造函数**）

纯虚函数 没有函数体的虚函数，包含纯虚函数的类叫**抽象类**, 只能作为基类派生新类或定义指针/引用类型指向派生类对象（实现多态），**不能创建抽象类的对象**.

```
class A {
public:
    virtual void f() = 0; //纯虚函数
    void g( ) { this->f() ; } //在抽象类的成员函数内可以调用纯虚函数
    A(){} //构造函数或析构函数内部不能调用纯虚函数
};//抽象类
class B : public A{
public:
    void f(){ cout<<"B:f()"<<endl; }
    //抽象类派生类只有实现了基类中的所有纯虚函数才能成为非抽象类
};
int main(){
    B b;b.g();//输出 B:f()
    return 0;
}
```

第五章　python“速成”

变量与数据类型　python 中有五个标准的数据类型：数字（包括 int、float(双精度)、**complex (1+2j)**)、str（字符串)、list（列表)、tuple（元组)、dict（字典），相当于实现类名，可用于创建对象（a=str() 作为**构造函数**）可用 isinstance(a,Type) 函数查询某个数据是否是某个类型.

python 的指针特性 所有在赋值号左侧的变量**都是指针**，对变量赋值意味着将变量指向某处.

```
a,b = [1,2,3,4],[1,2,3,4]
print (a is b) #False
c = a # 指向同一个地址
print (a is c) #True
a[3] = 5
print(c) #[1,2,3,5]
c[2] = 6
print(a) #[1,2,6,5]
# 对 list,dict 等可修改类型，关注 a is b（指向同一地址）和 a == b（值相同）的区别
# 事实上列表的元素 a[0] 等也是指针，但实现方式有所不同（同一内容的指向同一地址）
```

布尔类型 True 就是 1，False 就是 0, 同时认为空 str/tuple/list/dict 都相当于 False（可以写 if not []，但不等于 False)。

浮点类型 若需判断两个浮点数相等需要使用：(import sys) abs(a-b) <= sys.float_info.epsilon round(x) 遵循四舍六入五成双, 可以通过 str(Float) 函数转成字符串。

类型转换函数

```
print(int("f0",16)) #240
print(int("1010",2)) #10
print(chr(97)) #a
print(ord('a')) #97
print(repr(str)) # 可执行字符串，包含转义字符等
```

输入与输出

```
lst = input().split() # 将输入多项按分隔符（缺省为空格）拆分成一个列表
print(1,2,3,end="") # 以空格分隔，缺省 end 为换行
print(" 我叫 %s 今年 %d 岁！" %('小明',10)) # 格式控制符
# 输入输出重定向
import sys
f, g = open("a.in", "r"), open("a.out", "w")
sys.stdin, sys.stdout = f, g
#The Main Program...
f.close()
g.close()
```

字符串

```
x = "abcdefg"
print(x[-1]) #g（从右往左看）
print(x[0:3]) #abc（左闭右开）
print(x[:]) #abcdefg（缺省为起点和终点）
x[2] = 3 #error，字符串不可修改！
print(r'ab\ncd') #ab\ncd，消除转义
# 可以作加法、数乘；可用 in/not in 判断是否为子串，用 s.count() 计算子串个数
a = 3
eval("1+a") #4，特殊的类型转换，看作表达式求值
s = "abab"
print(s.find("ab"),s.rfind("ab")) #0,2（向左/右）寻找子串，找不到返回-1
#startswith/endswith 判断是否以某子串开头、结尾
#strip,lstrip,rstrip 去除两端/左侧/右侧空白字符;strip(s) 等去除 s 中所有字符:
print("cd\t12 34 5 c".lstrip("d\tc")) #12 34 5 c
```

字符串的分割 split(x)（x 缺省为空格回车制表符）函数将字符串分割成列表（两个相邻分隔符之间会被分隔出一个**空串**）

```
import re # 可引入正则表达式形成多个分割串的效果
a = 'Beautiful, is; better*than\nugly'
```

```
print(re.split(';| |,|\*|\n',a)) # 分隔串用 / 隔开，注意 * 用转义符号
# ['Beautiful', '', 'is', '', 'better', 'than', 'ugly']
```

字符串的格式化（两种） 赋值/输出均可用:

```
s = "My name is %s,I am %.2fm tall." % ("tom",h)
x = "Hello {0} {1:>10},you get ${2:0.4f}".format("Mr.","Jack",3.2)
# 第一个数字表示序号，后面的表示宽度/精度/类型,> 表示右对齐（< 左对齐，^ 中对齐）
#>>Hello Mr.     Jack,you get $3.2000
```

字节流类型 bytes 数据类型，可以与字符串相互转换:

```
bs = 'ABC 好的'.encode('utf-8')
# b'ABC\xe5\xa5\xbd\xe7\x9a\x84',b 表示数据类型,\x 表示十六进制字节
s = str(bs,encoding = "utf-8") # 转换回字符串
```

元组 由数个逗号分隔的值组成，相当于 vector，一旦创建，不可修改/增删元素/排序，可多元嵌套. 但是，元组的**元素本身有可能被修改**。例如，如果是列表，就可以修改该列表；同时，虽然不能增删元素，但元组间可以连接（相当于创建新元组）

```
v = ("hello",[1, 2, 3], [3, 2, 1])
v[1][0] = 'world' #ok
v[1] = 32 #error
v += (10,20) #ok
# 更直观的例子:
x = (1,2,3)
b = x # b is x == True
x += (100,) #b is x == False,b 不变，仅创建了一个新的 x
```

单元素元组 赋值时需要在最后加逗号，否则被识别为普通变量（singleton = ’hello’,）

列表 (可以修改, 列表和元组可以互转):

```
list1 = ['Google', 'Runoob', 1997, 2000]
del list1[2] # 删除
list1 += [100] # 没有创建新列表!
list2 = list1[1:3] # 这里创建了新的列表, 即修改 list2 不影响 list1
[a,b,c,d] = list1 # 分别赋值, 元组模式也很常用
```

列表生成式

```
[x * x for x in range(1, 11) if x % 2 == 0] #[4,16,36,64,100]
[m + n for m in 'ABC' for n in 'XYZ'] # 双重循环, 9 个元素
L = ['Hello', 'World', 18, 'Apple', None]
[s.lower() for s in L if isinstance(s,str)]
#['hello', 'world', 'apple'] # 重要的处理输入的方法
```

列表相关函数

```
a, b = [1,2,3], [5, 6]
a.append(b) # 将 b 作为一整个列表插入
a.extend(b) # 将 b 中每一个元素分别插入
a.insert(1,'K'); a.insert(3,'K')
a.remove('K') # 删除第一个该元素
a.index('K') # 索引位置, 未找到则报错
# 高级函数
c = [1,2,3,4,5]
ls1 = list(map(f1,c)) #map 实现列表/元组的映射, 返回一个延时求值对象, 可转换成列表
ls1 = list(filter(f2,c)) # 筛选出满足 f2 的元素
ls1 = reduce(f3,c,10) #10 可缺省为 0, 实现列表的累积
#reduce(sum,[1,2,3,4,5],10)=25
```

用列表定义二维数组

```
# 常见误区 (例: 定义全零 3*3 数组)
array = [0,0,0]
matrix = [array*3] #wrong,[[0,0,0,0,0,0,0,0,0]]
```

```
matrix = [array] * 3 #wrong, 所有列表指向同一地址，对某一列表的修改会影响其他列表
matrix = [[0 for j in range(3)] for i in range(3)] #correct
```

列表/元组比大小 逐个元素比大小，如果某个是另一个前缀，则短的小，若无法比较则出错.

各种花里胡哨的列表的排序

```
a = [5,7,6,3,4,1,2]
b = sorted(a) # b=[1,2,3,4,5,6,7],a 不变
a.sort(reverse = True) # 参数可选，缺省为正序，此处为倒序
# 自定义比较函数 key
sorted("This is a test string from Andrew".split(), key=str.lower))
# ['a', 'Andrew', 'from', 'is', 'string', 'test', 'This'], 不区分大小写排序
#lambda 表达式实现特定关键字排序 (例: students 中元素为大小为 3 的元组):
from operator import *
students.sort(key = lambda x: x[2] ) # 关于最后一个关键字排序，关键字相同保序
#lambda x: x[2] 表示一个简单的无函数名的函数，x 为输入值，x[2] 为输出值
students.sort(key = itemgetter(0,1)) # 先关于 0 位置关键字排序，再关于 1 位置
def f(x):
    return (-x[2],x[1],x[0])
students.sort(key = f) # 先按 2 位置倒序排，再按 1 位置，再按 0 位置
# 元组本身不能排序，可用 sorted 方法得到新元组
```

列表的拷贝

```
a = [1,[2],3,4]
b = a # 标准浅拷贝
# 浅拷贝机制使得 a += [10] 和 a = a + [10] 不同，后者新建了一个列表，与元组相似
b = a[:] # 似乎是深拷贝，简单地改变 a 不影响 b
a[1].append(3) # 再列表中嵌有可修改的列表时深拷贝失效，b=[1,[2,3],3,4]
import copy
b = copy.deepcopy(a) # 真正的深拷贝
```

python 函数可返回多参数，实际是返回一个元组. 参数传递都是**按值传递**，即形参改变不影响实参. 若要改变实参（如 swap）则需传可变对象（如列表），可以通过 from import 跨文件引用函数、变量。

```
def func(a, b = 1, c = 2):
    print ('a=', a , 'b=', b, 'c=', c)
func(c = 50, a = 60) # 默认参数和实参带名字,a=60 b=1(缺省) c=50

def func1(*b): # 表示 b 是元组，用来传递参数个数不定的情况.
    #...
func1(*[1,2,3]) # 此时 * 表示解包，传进去 1 2 3 三个参数
func1((1,2,3)) # 未解包，视为一个元组元素

def func2(**b): #**b 表示字典，自动将传入参数转换为字典
    #...
func(p1 = 2, p2 = 3, p3 = 4) # 传入参数 b 为 {'p1':2,'p2':3, 'p3':4}

x,y = 100,100
def func3():
    global y # 声明使用全局变量
    x = 10 # 缺省使用局部变量
    y = 10
func3() # global x = 100,y = 10
```

字典 每个元素由“键：值”两部分组成，可以根据“键”进行查找。“键”必须可哈希（列表、集合、字典等可变结构不可作为键），因而可支持**快速查找**，可以通过赋值添加元素，使用 del 删除，可以通过赋值语句**修改**值。

```
# 字典的构造（若有重复键，后面的键值覆盖前面的）
d = dict([('name','Gumby'),('age',42)]) # 只要是不可变对象即可
d = dict(name = 'Gumby', age = 42) # 必须是特定对象（键不能是具体数值）
# 字典之间相互的赋值是浅拷贝，可以用 d.copy() 实现伪·深拷贝（可变对象修改会同步变）
# 可以用 copy.deepcopy(x) 实现真正的深拷贝

# 字典的输出
phonebook = {'Alice':'2341', 'Beth':'9102', 'Cecil':'3258'}
print("Cecil's phone number is %(Cecil)s." %phonebook) #* 输出格式化
```

```
# 字典函数
d.keys(),d.values(),d.items() # 取键序列，值序列和元素序列
d.pop(x) # 删除键为 x 的元素，若没有则异常
d.get(k,v) # 若有元素键为 k，则返回值，否则返回 v
```

集合 可变对象，**无重复元素（自动去重）**，元素无顺序，也可快速查找，也同样只能以不可变对象为元素，可以使用 set(x) 函数实现字符串/列表转集合.

```
# 集合相关函数
s.add(x),s.update(s1) # 添加元素，合并集合
s.discard(x),s.remove(x) # 同为删除元素，后者找不到会报异常
# 集合特殊操作
a&b,a|b,a-b # 求交，求并，求差
a>b,a>=b,a<b # 求子集/真子集关系
```

面向对象程序设计 所有类自动派生自 object 类，object 类有 ___init___（构造函数）___del___（析构函数 * **构造函数和析构函数都只能有一个！**），___lt/gt/ge/le/eq/ne___（比较大小，很多可能在初始化时置为 None，需要自己写比较大小），可以使用 dir(A) 返回各个函数.

* 所有的成员函数都要先有参数 self，相当于 this 指针.

```
# 类的成员变量可以针对某一对象随时添加：
class A:
    n = 100 # 静态成员变量（类属性），本质是与类耦合的全局变量，可以通过 A.n 调用
    __p = 200 # （伪）私有类属性，需要通过 A._A__p 访问
    def __init__(self,x):
        self.x = x
    def func(self):
        self.xx = 9 # 不会报错
    def __eq__(self, other): # 缺省是比较地址，即 a is b，ne 缺省是 eq 反向
        return self.x == other.x
    def __str__(self): # 相当于重载类型转换构造函数 str
        return "({0.x},{0.y})".format(self) # 其中的 0 表示 self 的 id
    @staticmethod # 显式指定静态方法
    def func1(): # 不需要 self 参数
        pass
```

```
    @classmethod # 显式指定类方法
    def func2(cls,x):
        return x + cls.n # 可以通过 cls.n 访问类属性
a,b = A(1),A(2)
a.n2 = 28 #ok, 创建 a 中成员变量 n2
a.func() #ok, 在成员函数中创建成员变量 xx
print(b.n2,b.xx) # 均 error, 都没有在对象 b 中创建
a.n = 20 # 这里将 a 对象的静态成员变量变为非静态!
```

```
#property 方法: 将取值和赋值绑定在同一属性上。
class foo:
    def __init__(self):
        self.name = 'yoda'
        self.work = 'master'
    def get_person(self):
        return self.name,self.work
    def set_person(self,value):
        self.name,self.work = value
    person = property(get_person,set_person) #property 方法一

    @property #property 方法二
    def person(self):
        return self.name,self.work
    @person.setter # 若不写这行及下面的内容, 则默认只读属性, 不可修改
    def person(self,value):
        self.name,self.work = value
# 在主程序中可修改 A.person, 或是输出 A.person
```

可哈希 不可变的内置对象, 如元组, 字符串, 是可哈希的, 而字典, 列表, 集合 set 是不可哈希的, 自定义对象默认可哈希, 哈希值为其对象 id(地址非内容), 重载了 ___eq___ 后不可哈希, 再重载 ___hash___ 则又为可哈希. 有定义 hash(x) 则为可哈希, 整型变量的 hash(x)=x, 否则为 x.___hash___(), 哈希值可以看作是槽编号, 一个槽里面可以放多个哈希值相同的元素.

运算符重载

```
class A:
    def __iadd__(self,other):
        return A(self.x+other) # 实现 += 操作，若不加 A()，则 A 类型对象转换为整型
    def __add__(self,other): # 实现对象在前的加法
    def __radd__(self,other): # 实现对象在后的加法
    def __getitem__(self,index): #[] 索引取值重载
        return index ** 2
    def __setitem__(self,index,val): #[] 索引赋值重载
        print(index,val)
    def __str__(self): # 输出重载，对于类对象可以直接使用 print(a)
        return str(self.x)
    def __call__(self,other): #p=A(2) 后可以将 p 作为函数对象调用 p(2) 返回 4
        return self.value * other
```

对象列表的排序

```
def __lt__(self,other): #1. 重载 lt 运算符
a.sort(key = attrgetter('x','y')) #2.(先按 x 后按 y)
a.sort(key = lambda p: p.y) #3.(按 y 排序)
```

泛型 python 自带泛型设计，因为变量类型本来就是可变的，因此任何函数都是相当于模板.

继承 使用 class B(A): 实现派生类,在 B 的构造函数中用 A.___init___(self,x,y) 显式调用基类构造函数 (本来对象类型就是运行时确定，没有明显的多态), 特别地, 若 b=B(), 则 isinstance(b,A)=True.

函数式程序设计

```
class A:
    def __init__(self,n):
        self.n = n
    def __call__(self,x):
        return self.n + x
def add(x,y,f): # 函数可以作为函数的参数
    return f(x)+f(y)
g = abs # 函数可以给变量赋值
print(add(1,-10,g),add(1,10,A(5))) #11,21
def combine(f,g):
```

```
    return lambda x:f(g(x))
print(combine(Inc,Square)(4)) # 实现函数的复合，17
b = lambda x:lambda y:x+y # 返回值是 lambda 表达式的 lambda 表达式
a = b(3) #a 是一个闭包，即带自由变量的函数
print(a(2))
#lambda 表达式的更复杂应用：
low,high = 3,7
x = list(filter(lambda x,l = low,h = high: h>x>l,lst))

from functools import partial
add = lambda a,b:a+b
add1024 = partial(add,1024) # 偏应用函数 partial 固定函数的 a 参数
```

迭代器 可以用 for i in x: 形式遍历的对象 x，称为可迭代对象，必须实现 ___iter___（返回自身）和 ___next___（返回后继）方法（在容器末尾使用 next 时会抛出异常），相当于如下 while 循环：

```
it = iter([1,2,3,4])
while True:
    try:
        print(next(it))
    except StopIteration: # 异常
        break
# 自定义实现迭代器：
clss MyRange: # 用一次就消亡了，不支持多次迭代
    def __init(self,n):
        self.idx,self.n = 0,n
    def __iter__(self):
        return self
        # 解决多次迭代方案：分为容器类和迭代器类，每次重新构建 iter：
        #return MyRangeIterator(self.n)
    def __next__(self):
        if self.idx < self.n:
            val = self.idx # 记录当前值
            self.idx += 1
```

```
            return val
        else:
            raise StopIteration() # 抛出异常
```

生成器 延时求值对象，内部包含计算过程，真正需要时才完成计算。

```
a = (i*i for i in range(5))
print(a) # 不输出值
for x in a:
    print(x) # 在此处生成值
def test_yield():
    print("test")
    yield 1 #yield 关键字用于定义生成器，作为 return 使用
    t = yield (1,2) #t 的值要从外部获取
    print(t)
    yield 3
a = test_yield() # 无输出
print(next(a)) #test\n 1
print(next(a)) #(1,2)，且相当于 a.send(None)
z = a.send("check") # 向函数中传入 t=“check”，同时 z=3，在 send(None) 之前不能 send(x)
print(z) #3
print(next(a)) # 异常
def myMap(func, iterable):
    for arg in iterable:
        yield func(arg)
names = ["ana", "bob", "dogge"]
x = myMap(lambda x: x.capitalize(), names) # 实现 map
#['Ana', 'Bob', 'Dogge']
```

装饰器 可以在不改变原有函数代码的情况下扩展函数的功能：

```
def good(func):
    def wrapper(*args):
        print("%s called" % func.__name__ )
        return func(*args)+5
```

```
    return wrapper
@good #good 是一个装饰器
# 事实上，外面还可以套一层传参的函数 def good2(var)，使用 @good2(100)
def mysum(a,b,c):
    return a + b + c
print(mysum(3,4,5)) # 等价于 good(mysum)(3,4,5)，输出：mysum called\n 17
```