

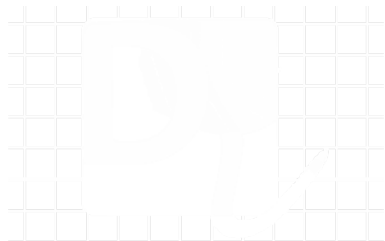
高级语言C++程序设计

Lecture 8 类和对象

南开大学 计算机学院

2022

类的定义



RECAP: 结构体

对具有相同属性的对象的一种抽象

```
struct Person {  
    double height;  
    double weight;  
    int age;  
};  
int main() {  
    Person p1 = {180, 120, 20};  
    cout<<p1.weight;  
    p1.age = 25;  
    return 0;  
}
```

对人进行抽象，包含三个属性：身高、体重、年龄

类 (Class)

更系统、更完善的抽象机制：**封装、继承、多态**等

一个简单的类

```
class Person {  
    public:  
        double height;  
        double weight;  
        int age;  
};
```

对人进行抽象，包含三个属性：身高、体重、年龄

类 (Class)

一个简单的类

```
class Person {  
    public:  
        double height;  
        double weight;  
        int age;  
};  
int main() {  
    Person p1;  
    p1.age = 25;  
    cout<<p1.age;  
    return 0;  
}
```

访问控制符为`public`，表示类的成员变量是公有的，所有人都可以访问

类 (Class)

一个简单的类

```
class Person {  
    private:  
        double height;  
        double weight;  
        int age;  
};  
int main() {  
    Person p1;  
    p1.age = 25; // 错误, age是私有成员  
    cout<<p1.age; // 错误, age是私有成员  
    return 0;  
}
```

访问控制符为`private`, 表示类的成员变量是私有的, 外人不可访问

类 (Class)

一个简单的类

```
class Person {  
    private:  
        double height;  
        double weight;  
        int age;  
    public:  
        int getAge(){ return age;}  
        void setAge(int n){ age = n;}  
};
```

增加成员函数，成员函数的访问控制符为public，成员函数可以访问类的私有成员变量

类 (Class)

一个简单的类

```
class Person {
```

```
    private:
```

```
        double height;
```

```
        double weight;
```

```
        int age;
```

```
    public:
```

```
        int getAge(){ return age;}
```

```
        void setAge(int n){ age = n;}
```

```
};
```

通过**private**，隐藏类的成员变量（静态属性）

通过**public**提供类的成员变量的访问接口

类的封装：将成员变量定义为私有，将成员函数定义为公有，用户只能通过成员函数访问成员变量！

类 (Class)

```
int main() {  
    Person p1;  
    p1.age = 25; // 错误, age是私有成员  
    cout<<p1.age; // 错误, age是私有成员  
    p1.setAge(25); // OK! setAge是公有成员函数  
    cout<<p1.getAge(); // OK! getAge是公有成员函数  
    return 0;  
}
```

调用成员函数访问成员变量的过程称为消息传递!

思考: 类的封装有何好处?

类 (Class)

```
class Person {  
    private:  
        double height;  
        double weight;  
        int age;  
    public:  
        int getAge(){ return age;}  
        void setAge(int n){  
            if(age < 0){  
                cout<<"Invalid Age!";  
                return;  
            }  
            age = n;  
        }  
};
```

- 封装是C++三大特性之一
- 可以限制用户行为，提高程序安全性
- 代码重用

年龄不能为负！

类的定义

关键字class 类名 花括号

访问
控制
符

```
class Person {
```

```
    private:
```

```
        int age;
```

```
    public:
```

```
        int getAge(){  
            return age;  
        }
```

```
        void setAge(int n){  
            age = n;  
        }
```

```
};
```

成员变量

成员函数

分号

类的定义

```
int main() {  
    Person Tom;  
  
    Tom.setAge(20);  
    cout<<Tom.getAge();  
  
    Person *pTom;  
    pTom = new Person();  
    pTom->setAge(25);  
    cout<<pTom->getAge();  
    (*pTom).setAge(24);  
}
```

定义类对象

调用成员函数：类对象
+运算符 (.)

定义类对象指针

动态生成类对象

调用成员函数：类对象
指针+箭头运算符(->)

类的定义

函数声明与函数定义分离

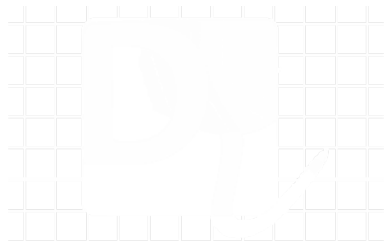
```
class Person {  
    private:  
        int age;  
    public:  
        int getAge();  
        void setAge(int n);  
};
```

类定义中只放
函数声明

```
int Person::getAge() {  
    return age;  
}  
void Person::setAge(int n) {  
    age = n;  
}
```

函数定义放在类外：
类名+域限定符(::)

构造函数与析构函数



类成员初始化

通过默认值初始化： 如果声明类对象时未对成员变量进行初始化，就使用默认值

```
class Person {  
    private:  
        int age = 0; //默认值  
    public:  
        int getAge(){ return age;}  
        void setAge(int n){ age = n;}  
};  
void main() {  
    Person p1; //age使用默认值 0  
    cout<<getAge();  
}
```

类成员初始化

通过初始化列表：如果类的数据成员是**公有的**，可以使用初始化列表进行初始化

```
class address{  
    public:  
        long telenum;  
        char addr[30];  
};
```

```
class person{  
    public:  
        char name[15];  
        int age;  
        address paddr;  
};
```

```
person p1={"Zhang Hua",23,{2475096, "NanKai  
University"}};
```

类成员初始化

通过构造函数初始化

```
class Person {  
    private:  
        int age;  
    Public:  
        Person(){age = 0;} //无参构造函数  
        Person(int n){age = n;} //有参构造函数  
};
```

- ❑ 构造函数必须和类同名，并且没有返回类型
- ❑ 构造函数可以有多个，即允许重载
- ❑ 类定义时没有给出构造函数，系统会自动提供一个默认的无参构造函数,如 Person(){}

构造函数

```
class Person {  
    private:  
        int age;  
    public:  
        Person() {age = 0;}  
        Person(int n) {age = n;}  
};
```

```
int main() {  
    Person Tom;  
    Person Jack(20);  
    Person *p = new Person();  
    return 0;  
}
```

构造函数在**类**
对象生成时由
系统自动调用

构造函数

```
class Person {  
    private:  
        int age;  
    public:  
        Person() { age = 0; }  
        Person(int n) { age = n; }  
};
```

```
int main() {  
    Person Tom;  
    Person Jack(20);  
    return 0;  
}
```

Tom 没有带参数，调用无参构造函数，age初始化为0

构造函数

```
class Person {  
    private:  
        int age;  
    public:  
        Person() {age = 0;}  
        Person(int n) {age = n;}  
};
```

```
int main() {  
    Person Tom;  
    Person Jack(20);  
    return 0;  
}
```

Jack 带一个参数，调用有参构造函数，age初始化为20

构造函数

```
class Person {  
    private:  
        int age;  
        char *name;  
    public:  
        Person() {  
            age = 0;  
            name = NULL;  
        }  
        Person(int n, char *p):age(n),name(p) {  
        }  
};
```

用参数初始化表的方式

也可以写成: age{n},name{p}

对象数组的初始化

```
class Person {  
    private:  
        int age;  
        char *name;  
    public:  
        Person(int n, char *p):age(n),name(p){  
        }  
};
```

```
Person poker[2] = {  
    Person(11, "spade"),  
    Person(10, "diamond"),  
};
```

利用初始化列表对每个元素初始化，共调用两次构造函数

对象指针的初始化

```
class Person {  
    private:  
        int age;  
        char *name;  
    public:  
        Person(int n, char *p):age(n),name(p){  
        }  
};
```

```
Person *poker = new Person(11, "spade");
```

使用new生成动态对象，将会调用构造函数，new的返回值为类对象指针

委托构造函数

一个构造函数在其初始化列表中调用了另一个构造函数

```
class Box {  
    double length;  
    double width;  
    double height;  
public:  
    Box(double lv, double wv, double hv):  
length{lv}, width{wv}, height{hv}{}  
  
    Box(double side):Box(side, side, side){}  
};
```

下面的构造函数在初始化列表中调用了上面的构造函数

构造函数

如果定义了带参构造函数，系统默认的对无参构造函数就不存在了，要想使用，必须显示地定义

```
class Person {  
    private:  
        int age;  
    public:  
        Person(int n){age = n;}  
};  
int main(){  
    Person a; // 错误，未定义无参构造函数  
    return 0; Person(){}  
}
```

explicit关键字

C++标准支持带有一个参数的构造函数，将参数类型隐式转换为相应的类类型

```
class Cube
{
private:
    double side;
public:
    Cube(double aSide);
    void TestFunction(Cube aCube);
};
Cube::Cube(double aSide){ //带参构造函数
    side = aSide;
}
```

explicit关键字

C++标准支持带有一个参数的构造函数，将参数类型隐式转换为相应的类类型


```
void Cube::TestFunction(Cube aCube)
{
    cout << aCube.side << endl;
}
void main()
{
    Cube c1(7.0);
    Cube c2(3.0);
    c1.TestFunction(50.0);
    c1.TestFunction(c2);
}
```

此函数接收double型实参，通过Cube的构造函数将其隐式转换为Cube类对象

explicit关键字

explicit关键字可以禁止隐式类型转换

```
class Cube
{
private:
    double side;
public:
    explicit Cube(double aSide);
    void TestFunction(Cube aCube);
};
Cube::Cube(double aSide){ //带参构造函数
    side = aSide;
}
```



此函数无法实现从double到Cube的隐式类型转换

析构函数(destructor)

类对象在生命期结束时会被释放，除回收内存空间外，还会调用**析构函数**

析构函数定义： **~ 类名(){<函数体>}**

```
class Person {  
    private:  
        int age;  
    public:  
        Person(){}  
        ~Person(){}  
};
```

- 无返回类型，无参数，不能被重载（最多只能有一个）
- 如果定义时没有给出，默认函数体为空

析构函数

类对象在生命期结束时会被释放，除回收内存空间外，还会调用**析构函数**

```
int main(){
    Person Tom;
    Person pTom = new Person();
    delete pTom;
    return 0;
}
```

调用两次构造函数，
两次析构函数

Tom通过静态分配内存创建，在main函数结束时自动被释放，调用析构函数

pTom指向动态创建的类对象，类对象在delete时被释放，调用析构函数

类对象的析构

`delete`

first calls the appropriate destructor (for class types), and then calls a *deallocation function*.

先调用析构函数（如果是类对象），再回收动态分配的空间

`delete []`

first calls the appropriate destructors for each element in the array (if these are of a class type), and then calls an array deallocation function

先为每个数组元素调用析构函数（如果是类对象），再回收动态分配的空间

类对象的析构

```
class A{  
    int x;  
};  
A *p;  
p = new A(); //动态生成一个类A的对象  
delete p; //先调用类A的析构函数，再回收A的空间  
  
p = new A[10]; //动态生成10个类A的对象  
delete p; //只为第一个类A的对象调用析构函数，然后释放动态空间（错误！）  
delete []p; //为数组中每个类A的对象调用析构函数，再回收动态分配的空间
```

析构函数

析构函数有何用处？

```
class A {  
    private:  
        int *p;  
    public:  
        A(int n){ p = new int[n];}  
        ~A(){delete []p;}  
};
```

构造函数分配动态内存



析构函数释放空间



```
int main() {  
    A a(10);  
    return 0;  
}
```

类对象中动态分配的内存存在
main函数结束时通过析构函数
自动释放了

析构函数

```
int main() {  
    A a(5);  
    A b(6);  
    A *p = new A(10);  
    delete p;  
    return 0;  
}
```

析构函数调用顺序与构造函数相反，先构造的后析构

构造顺序：a, b, *p
析构顺序：*p, b, a

思考：如果没有delete p; p所指向的类对象什么时候被析构？

思考

为何类对象构造/释放时要调用构造函数/析构函数，而普通数据类型（如int, double等）不用这么做？

练习

有关类的说法，错误的是

- A 类是一种用户自定义的数据类型
 - B 只有类中的成员函数才能存取类中的私有数据
 - C 在类中，如果不做特别说明，所指的数据均为私有类型
 - D 在类中，如果不做特别说明，所指的成员函数均为公有类型
-

练习

有关类的说法，错误的是

- A 对象是类的一个实例
 - B 任何一个对象只能属于一个具体的类
 - C 一个类只能有一个对象
 - D 类和对象的关系与数据类型和变量的关系相似
-

练习

假定A为一个类，int a()为该类的一个成员函数，若该成员函数在类定义体外定义，则函数头为？

A int A::a()

B int A:a()

C A::a()

D A::int a()

练习

构造函数在何时被执行？

- A 程序编译时
 - B 创建对象时
 - C 创建类时
 - D 程序装入内存时
-

练习

下面有关构造函数的描述中，正确的是

- A 构造函数可以带有返回值
 - B 构造函数的名称和类名完全相同
 - C 构造函数必须带有参数
 - D 构造函数必须定义，不能缺省
-

练习

有关构造函数的说法错误的是

- A 构造函数的名称和类名一样
 - B 构造函数在说明类变量时自动执行
 - C 构造函数无任何函数类型
 - D 构造函数有且只有一个
-

练习

下面有关析构函数特征的描述中，正确的是

- A 一个类可以有多个析构函数
 - B 析构函数与类名完全相同
 - C 析构函数不能指定返回类型
 - D 析构函数可以有一个或多个参数
-

练习

以下程序输出“#”号的个数是：

```
class Test{
public:
    Test(){}
    ~Test(){cout<<'#';}
};
int main(){
    Test temp[2], *pTemp[2];
    return 0;
}
```

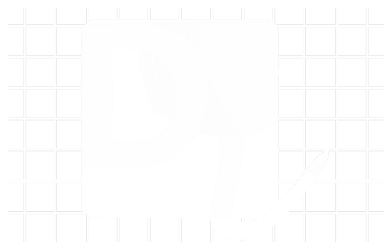
练习

下面程序的输出结果为：

```
class point{
public:
    point() {cout<<"C";}
    ~point() {cout<<"D";}
};

int main() {
    point *ptr;
    point A, B;
    point *ptr_point=new point[3];
    return 0;
}
```

拷贝构造函数



拷贝构造函数(copy constructor)

基于已有类对象构造新的类对象

```
class Person{  
    public:  
        int age;  
        Person(int n){  
            age = n;  
        }  
};  
void main(){  
    Person a(24);  
    Person b = a;  
}
```

类对象a由构造函数生成

类对象b则基于a由拷贝构造函数生成

不是先调用构造函数生成b，再由a对b进行赋值！

拷贝构造函数

定义：类名 (类名 &变量名) {<函数体>}

```
class Person {  
    private:  
        int age;  
    public:  
        Person(int n){age = n;}  
        Person(const Person & p){...}  
};
```

最好是const类型

- ❑ 拷贝构造函数名与类名相同
- ❑ 拷贝构造函数无任何返回类型
- ❑ 形参只能是类对象的引用，不能重载

拷贝构造函数

```
class Person{  
    public:  
        int age;  
        Person(int n){  
            age = n;  
        }  
};  
void main(){  
    Person a(24);  
    Person b = a;  
}
```

如果类定义没有给出拷贝构造函数，系统会给一个默认的(函数体为空)

```
Person(const Person &p){}
```

默认的拷贝构造函数将进行对位拷贝(浅拷贝)

类对象b的每个成员赋值为a的对应成员的值

拷贝构造函数

浅拷贝引发的问题?

```
class A {  
    private:  
        int *p;  
    public:  
        A(){p = new int[2];}  
        ~A(){delete []p;}  
};
```

```
void main(){  
    A a;  
    A b = a;  
}
```

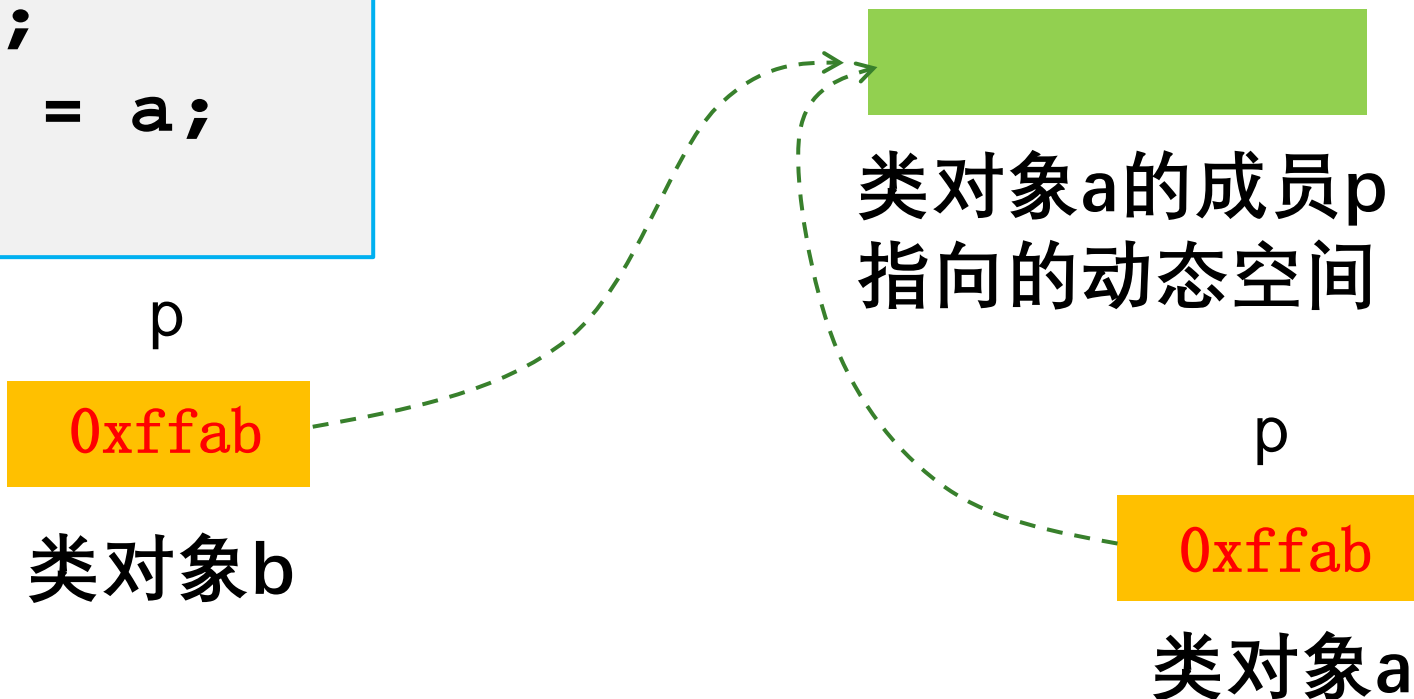


调用默认拷贝构造函数，浅拷贝

拷贝构造函数

```
void main() {  
    A a;  
    A b = a;  
}
```

浅拷贝引发的问题



类对象b的p和类对象a的p指向同一个空间，引发空间使用冲突和重复delete问题

拷贝构造函数

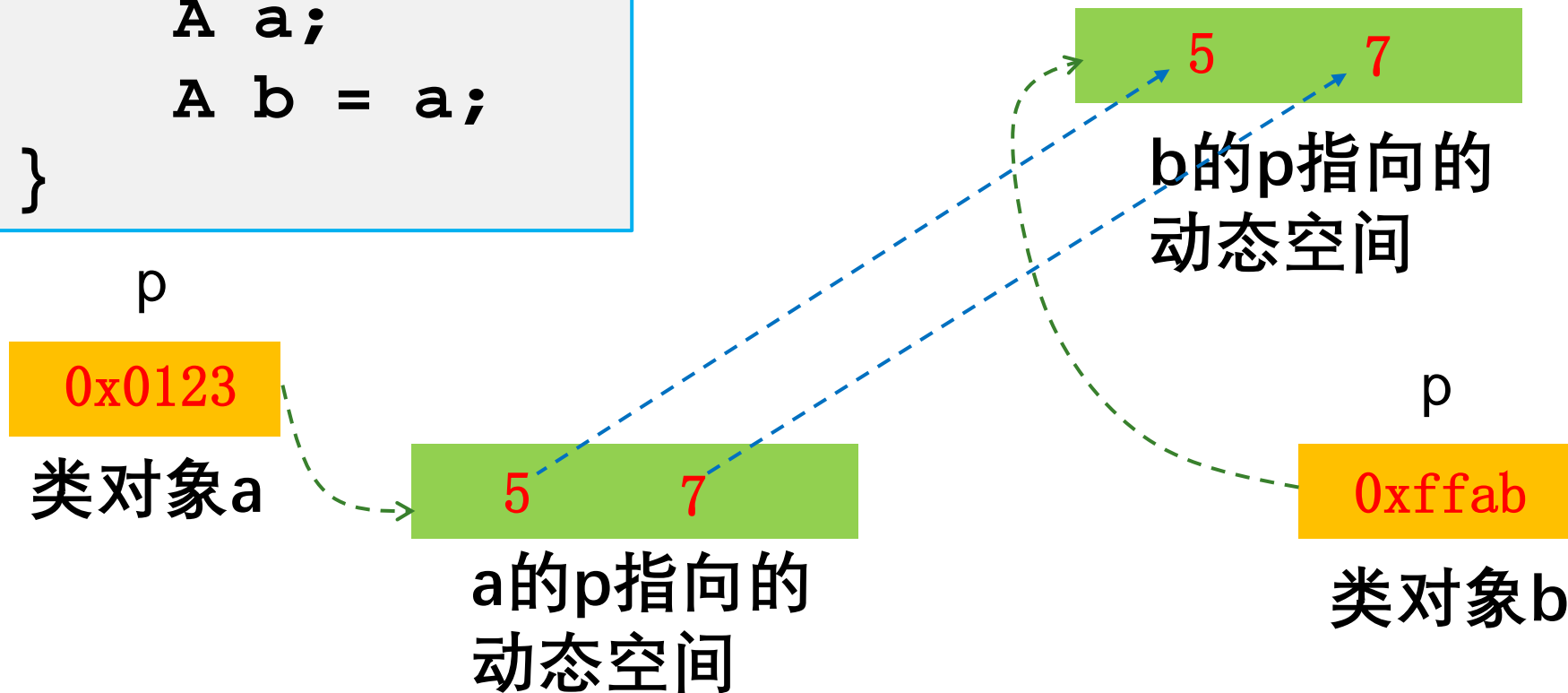
自定义拷贝构造函数，实现深拷贝

```
class A {  
    private:  
        int *p;  
    public:  
        A(){ p = new int[2];}  
        ~A(){delete []p;}  
        A(const A &a){  
            p = new int[2]; //动态生成新的空间  
            for(int i = 0; i < 2; i ++){  
                p[i] = a.p[i]; //拷贝每个数值  
            }  
        }  
};
```

拷贝构造函数

```
void main() {  
    A a;  
    A b = a;  
}
```

深拷贝



a和b的p分别指向不同的内存空间，不会产生使用冲突，对象析构时也不会造重复delete

拷贝构造函数

调用场合(一)

```
void main(){  
    A a;  
    A b = a;  
    A c(a);  
}
```

定义类对象时直接用
已有类对象初始化



拷贝构造函数

调用场合(二)

```
void f(A var) {  
}
```

```
int main() {  
    A a;  
    f(a);  
}
```

类对象为函数实参

形参var由实参a拷贝构造

拷贝构造函数

调用场合(三)

```
A f() {  
    A a;  
    return a;  
}
```

```
void main () {  
    f();  
}
```

函数返回类对象时，系统会产生一个临时变量，临时变量由拷贝构造函数基于return后面的变量生成



临时变量

拷贝构造



a

编译器优化可能会省略临时变量

拷贝构造函数

为何拷贝构造函数的参数为引用类型？

```
class Person {  
    private:  
        int age;  
    public:  
        Person(int n){age = n;}  
        Person(const Person & p){...}  
};
```

如果不是引用，拷贝构造函数的参数也需要通过拷贝构造函数生成（场合二），形成无限循环！

类对象生成方式总结

A a;	直接用类名定义类对象，调用 构造函数
A *a = new A();	动态生成类对象，调用 构造函数
A a = b; A c(b);	用已有类对象初始化生成类对象，调用 拷贝构造函数
非引用类型的函数形参	调用 拷贝构造函数
return时的临时类对象	调用 拷贝构造函数

类对象赋值

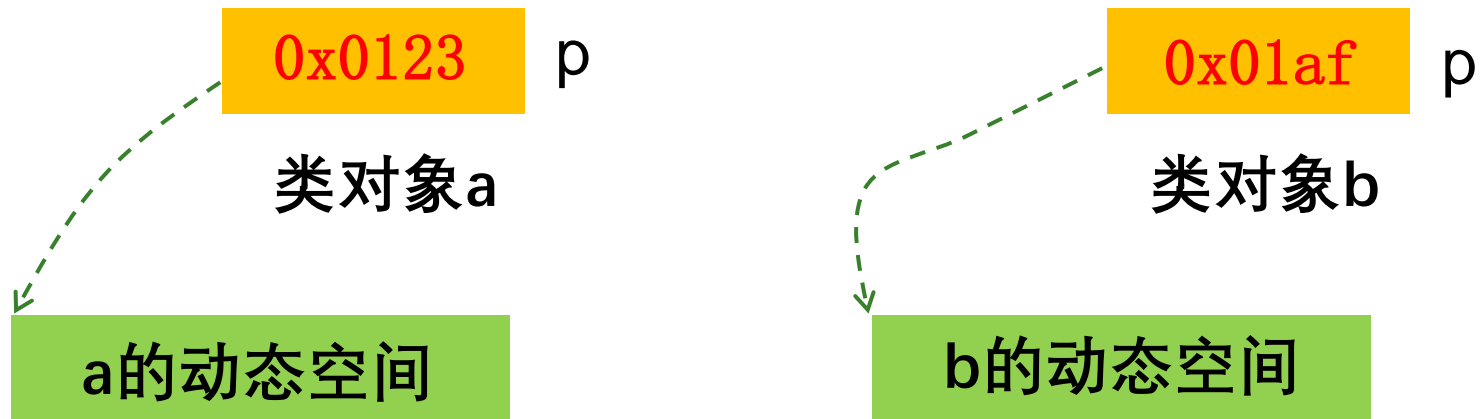
```
class A {  
    private:  
        int *p;  
    public:  
        A(){p = new int[2];}  
        ~A(){delete []p;}  
};  
int main() {  
    A a;  
    A b;  
    a = b;  
    return 0;  
}
```

类对象相互赋值时
默认进行对象数据
成员的对位拷贝

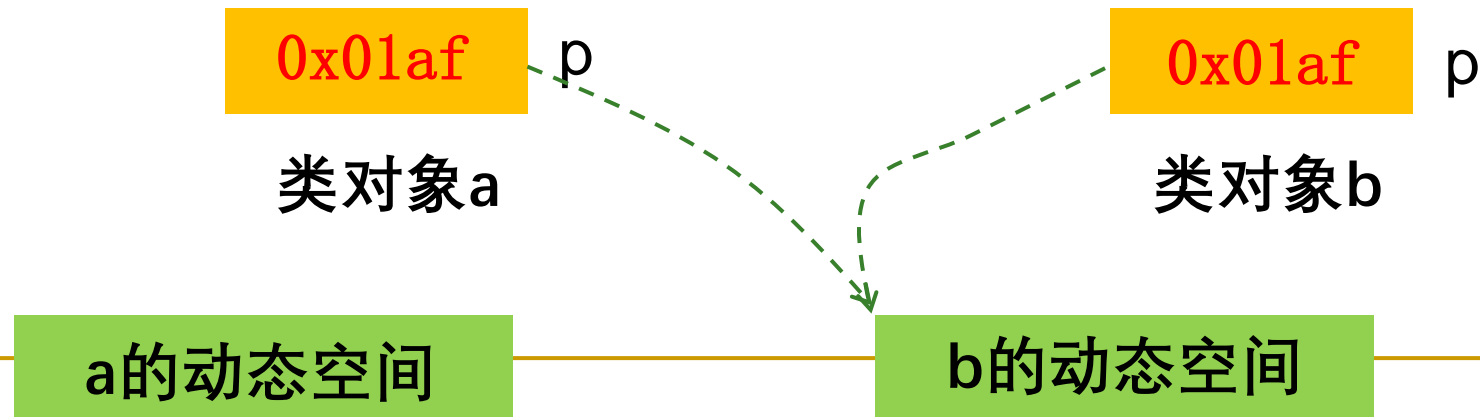
a的成员变量p将被
赋值为b的成员变量
p的值，两个p指向
同一个空间！

类对象赋值

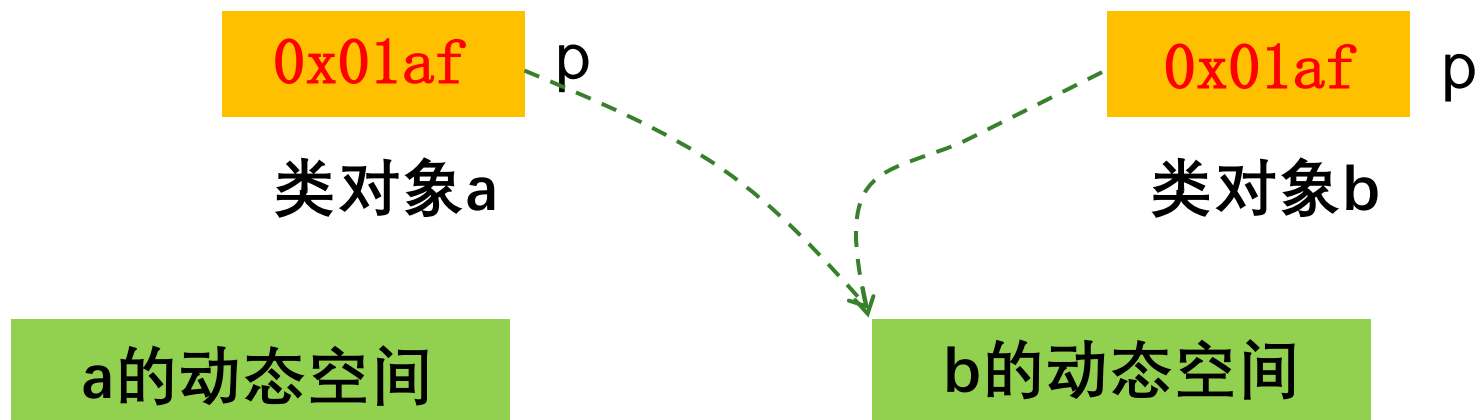
$a = b$; 执行之前, a 和 b 有自己独立的动态空间



$a = b$; 执行之后, a 的指针 p 指向 b 的动态空间



类对象赋值



a和b被析构时：

浅拷贝

- (1) a原有空间丢失，发生内存泄漏
- (2) b的动态空间被析构两次

通过重载赋值运算符，可以避免此类浅拷贝

练习

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

class A {
public:
    A(){cout<<1;}
    A(const A &){cout<<2;}
    ~A(){cout<<3;}
};

A test() {
    A a;
    return a;
}
```

```
int main(){
    A b;
    A c = b;
    test();
    b = test();
    return 0;
}
```

输出:

121233123333

Codeblocks+GCC编译:

- (1)关闭compiler flags里面的所有优化选项
- (2)添加“-fno-elide-constructors”编译选项来关闭关于临时变量拷贝构造的优化

练习

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

class A {
public:
    A(){cout<<1;}
    A(const A &){cout<<2;}
    ~A() {cout<<3;}
};

A test(A a) {
    return a;
}
```

```
int main(){
    A b;
    A c = b;
    test(b);
    test(c);
    return 0;
}
```

输出:
122233223333

练习

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

class A {
public:
    A(){cout<<1;}
    A(const A &){cout<<2;}
    ~A() {cout<<3;}
};

A test(A &a) {
    return a;
}
```

```
int main(){
    A b;
    A c = b;
    test(b);
    test(c);
    return 0;
}
```

输出:
12232333

参数类型为引用，表示形参是实参的别名，不会调用拷贝构造函数生成新的类对象

练习

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

class A {
public:
    A(){cout<<1;}
    A(const A &){cout<<2;}
    ~A() {cout<<3;}
};

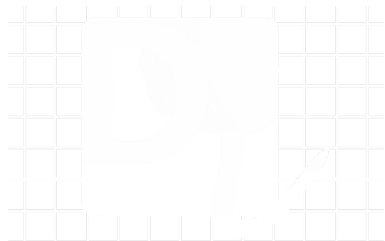
A &test(A &a) {
    return a;
}
```

```
int main(){
    A b;
    A c = b;
    test(b);
    test(c);
    return 0;
}
```

输出:
1233

函数返回类型为引用，表示返回的是return后面变量本身，不会调用拷贝构造函数生成临时变量

类的静态成员



类的普通成员

```
class A {  
    public:  
        int m;  
        int n;  
};
```

```
void main() {  
    A a, b;  
    cout<<sizeof(a);  
    cout<<sizeof(b);  
}
```



类对象a



类对象b

每生成一个类对象，都会分配新的存储空间

类的静态成员

```
class A {  
    public:  
        static int m; //声明  
        int n;  
};  
int A::m = 1; //定义和初始化
```

```
void f() {  
    A a, b;  
    cout<<sizeof(a);  
    cout<<sizeof(b);  
}
```



m

m属于整个类A



n

类对象a



n

类对象b

static 成员属于整个类，由所有类对象共享，在类定义的时候生成并分配空间

类的静态成员变量

```
class A {  
    public:  
        static int m; //声明  
        int n;  
};  
int A::m = 1; //定义和初始化
```

定义和初始化必须在类外进行，定义时不再加static


```
void main() {  
    A::m = 2; //直接用类名+(::)访问  
    A a;  
    a.m = 3; //通过类对象访问  
    A *p = new A();  
    cout<<p->m; //通过类对象的指针访问  
}
```

三者访问的是同一个m

类的静态成员函数

```
class A {  
    public:  
        static int m;  
        int n;  
        static void f1(){}  
};  
int A::m = 1;
```

静态成员函数属于整个类，所有类对象都可调用



```
void main () {  
    A::f1(); //通过类名+ (::) 直接调用  
    A a;  
    a.f1(); //通过类对象调用  
    A *p = new A();  
    p->f1(); //通过类对象的指针调用  
}
```

类的静态成员函数

```
class A {  
    public:  
        static int m;  
        int n;  
        static void f1(){ m++; } //ok?  
        static void f2(){ n++; } //ok?  
        void f3(){ n++; } //ok?  
        void f4() { m++; } //ok?  
        static void f5() { f3(); } //ok?  
};  
int A::m = 1;
```

以上类成员函数定义是否合法？

类的静态成员函数

```
class A {  
    public:  
        static int m;  
        int n;  
        static void f1() { m++; } //ok  
        static void f2() { n++; } //ok?  
        void f3() { n++; } //ok?  
        void f4() { m++; } //ok?  
        static void f5() { f3(); } //ok?  
};  
int A::m = 1;
```

静态成员函数可以
访问静态成员变量

静态成员变量和静态成员函数都属于整个类，在类定义的时候已经存在

类的静态成员函数

```
class A {  
    public:  
        static int m;  
        int n;  
        static void f1() { m++; } //ok  
        static void f2() { n++; } //ERROR!  
        void f3() { n++; } //ok?  
        void f4() { m++; } //ok?  
        static void f5() { f3(); } //ok?  
};  
int A::m = 1;
```

静态成员函数不能访问非静态成员变量

非静态成员变量属于某个类对象，而静态成员函数属于整个类，无法确定访问的是哪个类对象的变量

类的静态成员函数

```
class A {  
    public:  
        static int m;  
        int n;  
        static void f1() { m++; } //ok  
        static void f2() { n++; } //ERROR!  
        void f3() { n++; } //ok  
        void f4() { m++; } //ok?  
        static void f5() { f3(); } //ok?  
};  
int A::m = 1;
```

非静态成员函数可以
访问非静态成员变量

非静态成员变量和非静态成员函数同属一个类对象

类的静态成员函数

```
class A {  
    public:  
        static int m;  
        int n;  
        static void f1() { m++; } //ok  
        static void f2() { n++; } //ERROR!  
        void f3() { n++; } //ok?  
        void f4() { m++; } //ok  
        static void f5() { f3(); } //ok?  
};  
int A::m = 1;
```

非静态成员函数可以
访问静态成员变量

静态成员变量被所有类对象共享

类的静态成员函数

```
class A {  
    public:  
        static int m;  
        int n;  
        static void f1() { m++; } //ok  
        static void f2() { n++; } //ERROR!  
        void f3() { n++; } //ok?  
        void f4() { m++; } //ok  
        static void f5() { f3(); } //ERROR!  
};  
int A::m = 1;
```

静态成员函数不能调用非静态成员函数

非静态成员函数可能访问了非静态成员变量

类静态成员的应用

```
class Singleton
{
    public:
        static Singleton * Instance();
    private:
        Singleton() { //构造函数
            m_Instance = NULL;
        }

        ~Singleton() {} //析构函数
        static Singleton * m_Instance;
};
Singleton * Singleton::m_Instance = NULL;
```

类静态成员的应用

```
Singleton * Singleton::Instance() {  
    if (NULL == m_Instance){  
        m_Instance = new Singleton();  
    }  
    return m_Instance;  
}
```

```
void main(){  
    Singleton s; ERROR! 构造函数私有，无法调用  
    Singleton *p1, *p2;  
    p1=Singleton::Instance(); OK! 生成一个类对象  
    p2=Singleton::Instance(); OK! 不再生成新对象  
}
```

Singleton最多仅有一个类对象，称为单体！

类的常量数据成员

常量数据成员初始化之后不能更改，初始化只能由构造函数完成

```
class A {  
    public:  
        const int m; //常量数据成员定义  
        int n;  
        A(int x) { m = x; }  
};  
void main () {  
    A a(5);  
    cout<<a.m;  
    a.m = 10; //ERROR,不能更改!  
}
```

类的常量成员函数

常量成员函数不能修改类的非static成员变量

```
class A {  
    public:  
        int m;  
        int n;  
        void f() const ;  
};  
void A::f() const {  
    m = 10; // ERROR, 不能对m进行更改!  
    cout<<a.m;  
}
```

const放在函数尾部

返回值为常量的成员函数

函数的返回值不能更改

```
class A {  
    public: const放在函数头部  
        int m;  
        int n;  
        const int & f(){return m;}  
};  
void main() {  
    A a;  
    a.f() ++; // ERROR! 函数返回值不能修改  
    cout<<a.m;  
}
```


this指针

```
class A {  
    public:  
        int m;  
        int n;  
        void f() { m ++; }  
};
```

```
void main () {  
    A a;  
    A b;  
    a.f();  
    b.f();  
}
```

为什么函数f()可以找到正确的变量位置并进行操作？

非静态成员函数含有一个指向被调用对象的指针，即**this指针**

this指针

通常this指针是隐藏的，以下情况需要显示使用：

□ 成员函数的形参和类的数据成员同名

```
class A {  
    public:  
        int m;  
        void f(int m) { this.m = m; }  
};
```

带有this指针的表示类的数据成员

this指针

通常this指针是隐藏的，以下情况需要显示使用：

□ 非静态成员函数返回的是对象本身或对象地址

```
class A {  
    public:  
        int m;  
        A* f(){return this;} // 返回对象地址  
        A f1(){return *this;} // 返回对象本身  
};
```

指向成员的指针

说明格式：<类型名> <类名>::*<指针变量名>;
<类型名> (<类名>::*<函数指针名>)(形参表);

```
class X {  
public:  
    void f(int);  
    int a;  
};  
void X::f(int x)  
{  
    a = x;  
}
```

```
void main(){  
    int X::* pmi = &X::a;  
    void (X::* pmf)(int) =  
    &X::f;  
    X objx;  
    objx.*pmi = 10;  
    cout << objx.a << endl;  
    (objx.*pmf)(5);  
    cout << objx.a << endl;  
}
```

类的友元

类的友元

```
class A {  
    private:  
        int m;  
};  
void f() {  
    A a;  
    a.m = 2; // ERROR  
}
```

类的私有成员不允许被外界访问

C++允许一些特殊的函数或者类访问其他类的私有成员，被称为友元！

友元函数

将普通函数声明为类的友元函数，此函数就可以访问类的私有成员

```
class A {  
    private:  
        int m;  
    public:  
  
};
```

声明函数为类的友元函数：
friend + 函数声明

```
friend void f();
```

```
void f() { //f是普通函数，不属于类A
```

```
    A a;
```

```
    a.m = 2; //ok, 因为f是类A的友元函数
```

```
}
```


友元成员函数

将类B的成员函数声明为类A的友元函数，此函数就可以访问类A的私有成员

```
class A;  
class B {  
    public:  
        void f(A &a);  
};
```

```
class A {  
    private:  
        int m;  
    public:
```

声明类B的函数为类A的友元函数



```
    friend void B::f(A &a);
```

```
};
```

```
void B::f(A &a){a.m = 2;} //ok, B::f()为类A的友元函数
```


友元成员函数

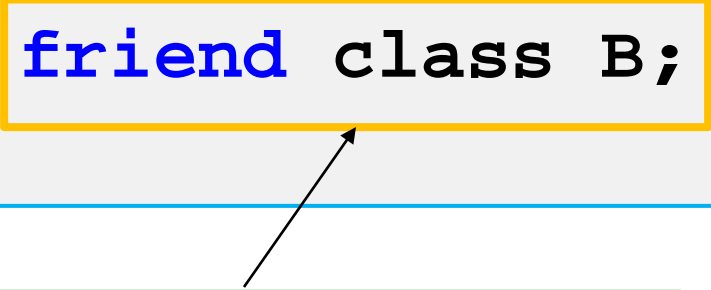
将类B的成员函数声明为类A的友元函数，此函数就可以访问类A的私有成员

```
class A; 必须先对A进行声明，否则类B里面的f(A &a)报编译
class B { 错误
    public:
        void f(A &a);
};
class A {
    private:
        int m;
    public:
        friend void B::f(A &a); 必须将先定义的类(类B)的成员函数作为后定义
};                                类(类A)的友元函数，调换顺序会出现语法错误
void B::f(A &a){a.m = 2;} ← 必须将类成员函数f的函
                                数体放在类A定义的后面
                                ，因为函数体中a.m=2访
                                问了类A的成员变量
```

友元类


将类B声明为类A的友元类，类B所有的成员函数都可以访问类A的任何成员

```
class B;  
class A {  
    private:  
        int m;  
    public:  
        friend class B;  
};
```



声明类B为类A的友元类

```
class B {  
    public:  
        void f() {  
            A a;  
            a.m = 2;  
        }  
};
```



ok, 因为类B是类A的友元类

友元类

将类B声明为类A的友元类，类B所有的成员函数都可以访问类A的任何成员

```
class B;  
class A {  
    private:  
        int m;  
    public:  
        friend class B;  
};
```

```
class B {  
    public:  
        void f(){  
            A a;  
            a.m = 2;  
        }  
};
```

注意

- (1) 顺序为：类B的声明，类A的定义，类B的定义
- (2) 成员函数f()的函数体必须在类A的定义之后

类的友元

单向性：如果类A是类B的友元类，并不意味着类B也是类A的友元类

非传递性：如果类A是类B的友元类，类B是类C的友元类，并不意味着类A也是类C的友元类

类的对象成员

一个类对象是另一个类的成员：

```
class B {  
    public:  
        int n;  
        B(int x){n = x;}  
};
```

```
class A {  
    public:
```

类A的构造函数中，对b进行初始化（会调用B的构造函数）

B b; //类B的对象是类A的数据成员

```
    int m;
```

```
    A(int x, int y):b(x){m = y;}  
};
```

类的对象成员

类成员构造顺序

```
class B {  
    public:  
        int n;  
        B(int x){n = x;}  
};
```

构造顺序: b1, b2, A自己
析构顺序: A自己, b2, b1

```
class A {  
    public:  
        int m;  
        B b1;  
        B b2;  
        A (int x1, int x2, int y):b2(x2),b1(x1){  
            m = y;}  
};
```

先按照对象成员的**声明顺序**执行
相应的构造函数，再调用自己的
构造函数，析构函数顺序相反

类的对象成员

两个类相互包含

```
class A;  
class B {  
    public:  
        A *a;  
};  
class A {  
    public:  
        B b;  
};
```

- 最前面需要放后定义的类的声明
- 先定义类只能包含后定义类的指针或者引
- 后定义类可以包含先定义类的对象

如果两个类相互包含彼此类对象，会形成无限循环

类的对象成员

类包含自己

```
class A {  
    private:  
        int m;  
        A a; // ERROR  
};
```

```
class A {  
    private:  
        int m;  
        A *p; // OK  
};
```

- ❑ 类不能包含自己的类对象，否则会陷入无限循环
- ❑ 类可以包含自己类的指针或者引用

类的运算符重载

类的运算符重载

C++提供的运算符只能用于基本数据类型，而不能用于自定义数据类型

```
class A {  
    public:  
        int n;  
};
```

```
void main() {  
    A a, b;  
    a + b; // 错误，加法不能用于类对象  
}
```

运算符重载可以赋予运算符新的功能，使它能够用于自定义的数据类型

类的运算符重载

运算符重载函数作为类成员函数

```
class A {  
    public:  
        int n;  
        A operator+(A &r) {  
            A c;  
            c.n = n + r.n;  
            return c;  
        }  
};
```

重载‘+’运算符

函数名operator+，
其中operator为关键字，
‘+’为运算符

类的运算符重载

运算符重载函数作为类成员函数

```
class A {  
    public:  
        int n;  
        A operator+(A &r) {  
            A c;  
            c.n = n + r.n;  
            return c;  
        }  
};
```

重载‘+’运算符

‘+’是双目运算符，有两个操作数，一个是调用重载函数的类对象(左操作数)，另一个是参数r(右操作数)

类的运算符重载

运算符重载函数作为类成员函数

```
class A {  
    public:  
        int n;  
        A operator+(A &r) {  
            A c;  
            c.n = n + r.n;  
            return c;  
        }  
};
```

重载‘+’运算符

‘+’的行为(函数体)根据需求定义，本例中定义为两个类对象的成员变量n相加

类的运算符重载

运算符重载函数作为类成员函数

```
class A {  
    public:  
        int n;  
        A operator+(A &r) {  
            A c;  
            c.n = n + r.n;  
            return c;  
        }  
};
```

重载‘+’运算符

函数返回一个新的类对象，其成员变量n的值等于两个相加的类对象的n之和

类的运算符重载

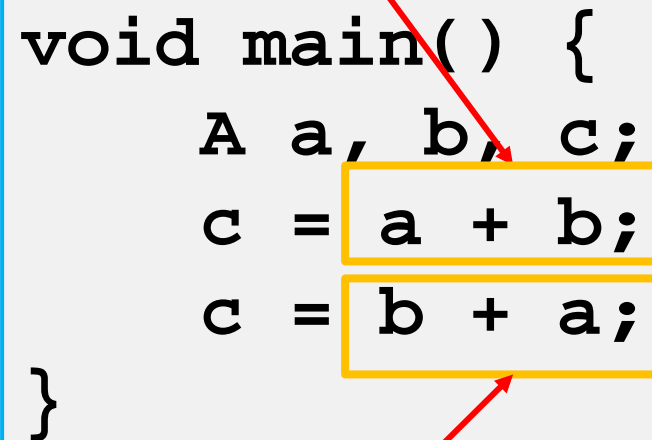
运算符重载函数作为类成员函数

```
class A {  
    public:  
        int n;  
        A operator+(A &r) {  
            A c;  
            c.n = n + r.n;  
            return c;  
        }  
};
```

相当于：

a.operator+(b);

```
void main() {  
    A a, b, c;  
    c = a + b;  
    c = b + a;  
}
```



相当于：

b.operator+(a);

类的运算符重载

运算符重载函数作为类成员函数

```
class A {  
    public:  
        int n;  
    A & operator ++ () {  
        n ++;  
        return *this;  
    }  
}; 前缀++, 返回++之后的值
```

重载‘++’运算符

‘++’是单目运算符，只有一个操作数，就是调用重载函数的类对象本身，函数行为是成员变量n加1

前缀++返回++之后的值，因此通常返回类对象本身（返回类型为引用）

类的运算符重载

运算符重载函数作为类成员函数

```
class A {  
    public:  
        int n;  
    A operator ++ (int) {  
        A s = *this;  
        n ++;  
        return s;  
    }  
};
```

后缀++, 返回++之前的值

重载‘++’运算符

为了区别前缀++和后缀++, C++语言规定, 在后缀++的重载函数的原型参数表中增加一个int 型的无名参数

后缀‘++’需要返回++之前的值, 因此函数通常返回对象的一个copy (返回类型非引用)

类的运算符重载

运算符重载函数作为类成员函数

```
void main() {  
    A a, b;  
    a.n = 0;  
    b.n = 1;  
    cout<<(a++).n;  
    cout<<(++b).n;  
    cout<<a.n<<b.n;  
}
```

相当于：

`a.operator++(int);`

相当于：

`b.operator++();`

输出结果：

0 2 1 2

类的运算符重载

运算符重载函数作为类成员函数

```
class A {  
    public:  
        int *p;  
        A(){p = new int[2];}  
        ~A(){delete []p;}  
};  
  
int main() {  
    A a;  
    A b;  
    a = b;  
    return 0;  
}
```

重载 '=' 运算符

浅拷贝带来的问题:

- (1) a的内存空间丢失;
- (2) b的空间重复析构

类的运算符重载

运算符重载函数作为类成员函数

```
class A {  
    public:  
        int *p;  
        A(){p = new int[2];}  
        ~A(){delete []p;}
```

重载 '=' 运算符

```
    A & operator=(const A &a){  
        for(int i = 0; i < 2; i ++){  
            p[i] = a.p[i]; //拷贝每个数值  
        }  
        return *this;  
    }
```

```
};
```

类的运算符重载

运算符重载函数作为类成员函数

```
int main() {  
    A a, b, c;  
    a = b;  
    a = b = c;  
    (a = b) = c;  
    return 0;  
}
```

相当于：

`a.operator=(b);`

相当于：

`a.operator(b.operator=(c));`

相当于：

`(a.operator(b)).operator=(c);`

类的运算符重载

运算符重载函数作为类成员函数

```
A & operator=(const A &a) {  
    for(int i = 0; i < 2; i++) {  
        p[i] = a.p[i]; //拷贝每个数值  
    }  
    return *this;  
}
```

思考：

如果返回值不是引用， $(a=b)=c$ 会怎样？

$a=b$ 返回的不是 a 本身，而是一个临时变量，那么 $(a=b)=c$ 相当于 c 的值最后没有赋值给 a

类的运算符重载

运算符重载函数作为类成员函数

```
A & operator=(const A &a) {  
    for(int i = 0; i < 2; i++) {  
        p[i] = a.p[i]; //拷贝每个数值  
    }  
    return *this;  
}
```

思考：

如果形参不是const会有什么影响？

如果形参是非const类型的引用，实参必须为左值（即不能是表达式），例如，`a = b+c` 编译会出错

如果形参是const类型的引用，实参可以是非左值，例如，`a = b + c` 正确

类的运算符重载

运算符重载函数作为类的友元函数

```
class A {  
    private:  
        int n;  
    public:  
        friend A operator+(A&, A&);  
};
```

重载‘+’运算符

```
A operator+(A & r1, A & r2) {  
    A c;  
    c.n = r1.n + r2.n;  
    return c;  
}
```

重载函数不是类成员函数，‘+’是双目运算符，有两个操作数，因此重载函数有两个参数(左/右操作数)

类的运算符重载

运算符重载函数作为类的友元函数

```
class A {  
    private:  
        int n;  
    public:  
        friend A operator+(A&, A&);  
};  
A operator+(A & r1, A & r2) {  
    A c;  
    c.n = r1.n + r2.n;  
    return c;  
}
```

重载‘+’运算符

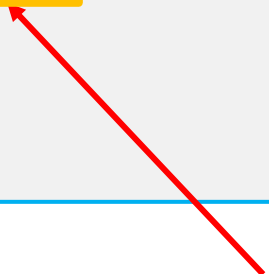
普通函数无法访问类的私有成员，因此，需要将重载函数声明为类的友元

类的运算符重载

运算符重载函数作为类的友元函数

重载‘+’运算符

```
int main() {  
    A a, b, c;  
    c = a + b;  
    return 0;  
}
```



相当于：

`operator+(a, b);`

类的运算符重载

一些规则：

- 类属关系运算符(.)、成员指针运算符(*)、作用域运算符(::)、(sizeof)运算符和三目运算符(?:)不能重载
 - 重载运算符不可以创建新的运算符
 - 重载运算符本质上是函数重载，遵循函数重载的选择原则
-

类的运算符重载

重载时何时使用类的成员函数、友元函数？

- 一般情况下，**单目**运算符最好重载为**类成员函数**，**双目**运算符最好重载为类的**友元函数**
 - 下面这些双目运算符只能重载为类的成员函数：`=`、`()`、`[]`、`->`
 - 若一个运算符的操作需要修改对象的状态，选择重载为成员函数较好，例如 `++`
 - 当需要重载运算符具有可交换性时，选择重载为友元函数，例如 `+`、`*` 等
-


类的运算符重载

若运算符所需的操作数(尤其是第一个操作数)希望有隐式类型转换，则只能选用友元函数

隐式类型转换

```
class A {  
    public:  
        A() {}  
};  
class B {  
    public:  
        B(const A &a) {}  
};  
void f(B b) {}
```

```
int main() {  
    A a;  
    f(a);  
    return 0;  
}
```



a 自动转换成 B 的类对象
，通过带参构造函数：
B(const A&)

类的运算符重载

若运算符所需的操作数(尤其是第一个操作数)希望有隐式类型转换, 则只能选用友元函数

```
class A {  
    int i;  
public:  
    A(int x) {i = x;}  
    A operator+(const A& a) const {  
        return A(i + a.i);  
    }  
    friend A operator-(const A&, const A&);  
};  
A operator-(const A& a1, const A& a2) {  
    return A(a1.i - a2.i);  
}
```

类的运算符重载

若运算符所需的操作数(尤其是第一个操作数)希望有隐式类型转换，则只能选用友元函数

```
int main() {  
    A a(3), b(4);  
    a + b; // ok, a.operator+(b)  
    a + 1; // ok, a.operator+(1), 1隐式转换为A的对象  
    1 + a; // error, operator+的调用者必须为A的对象  
    a - b; // ok, operator-(a, b)  
    a - 1; // ok, operator-(a, 1), 1隐式转换为A的对象  
    1 - a; // ok, operator-(1, a), 1隐式转换为A的对象  
    return 0;  
}
```

类的运算符重载

如果左边的操作数必须是一个不同类的对象，该运算符函数只能作为一个友元函数来实现

```
class A {  
    int i;  
public:  
    A(int x) {i = x;}  
    friend ostream& operator<<(ostream & os,  
const A& a);  
};  
ostream& operator<<(ostream & os, const A& a){  
    return os<<a.i;  
}
```

对类A重载<<运算符

<<是双目运算符，左边的操作数是ostream类的对象，右边是要输出的类对象

类的运算符重载

如果左边的操作数必须是一个不同类的对象，该运算符函数只能作为一个友元函数来实现

```
int main() {  
    A a(3);  
    cout<<a;  
    return 0;  
}
```

cout是ostream类的对象，此句
相当于operator<<(cout, a)

如果把operator<<重载函数作为类A的成员函数，那么左操作数将是A的对象，无法实现cout在<<左侧

END

