

C++程序设计教程(第二版)

Chapter 9 Object Birth & Death

苏州大学计算机科学与技术学院

主要内容

- ◆ 构造函数设计
- ◆ 构造函数的重载
- ◆ 类成员初始化
- ◆ 构造顺序
- ◆ 拷贝构造函数
- ◆ 析构函数
- ◆ 对象转型与赋值

f0901-point.h

```
#ifndef HEADER_POINT
#define HEADER_POINT
#include<iostream>
using namespace std;
class Point
{
private:
    int x, y;
public:
    void set(int a, int b);
    Point operator+(const Point& d);
    friend ostream& operator<<(ostream& o, const Point& d);
};
inline ostream& operator<<(ostream& o, const Point& d) {
    return o<<' ('<<d.x<<', '<<d.y<<')'<<' \n' ;
}
#endif // HEADER_POINT
```

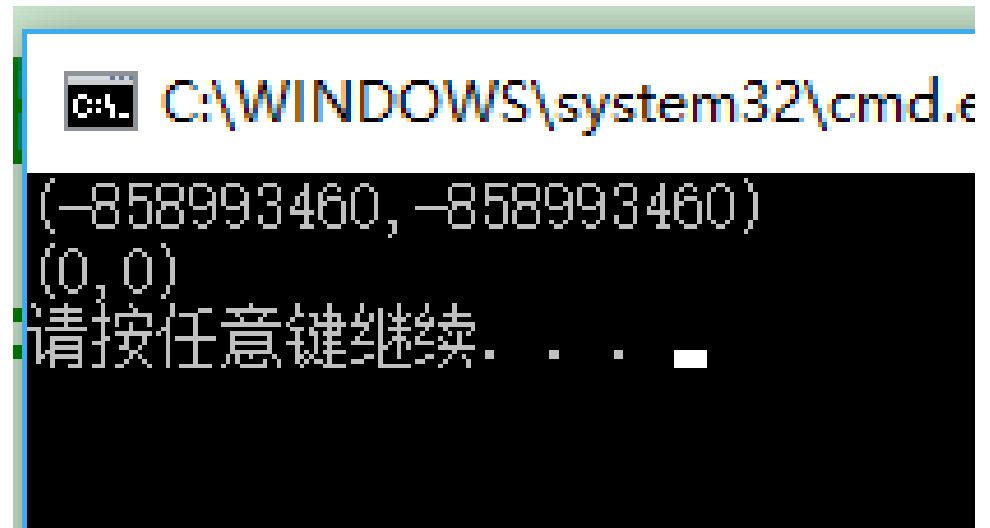
f0901-point.cpp

```
//=====
// point.cpp
//=====
#include"point.h"
//-----
void Point::set(int a, int b) {
    x=a, y=b;
} //-----
Point Point::operator+(const Point& d) {
    Point s;
    s.set(x+d.x, y+d.y);
    return s;
} //-----
```

f0901-driver.cpp

```
//=====
// f0901.cpp
// global and local object values
//=====
#include"point.h"
#include<iostream>
using namespace std;

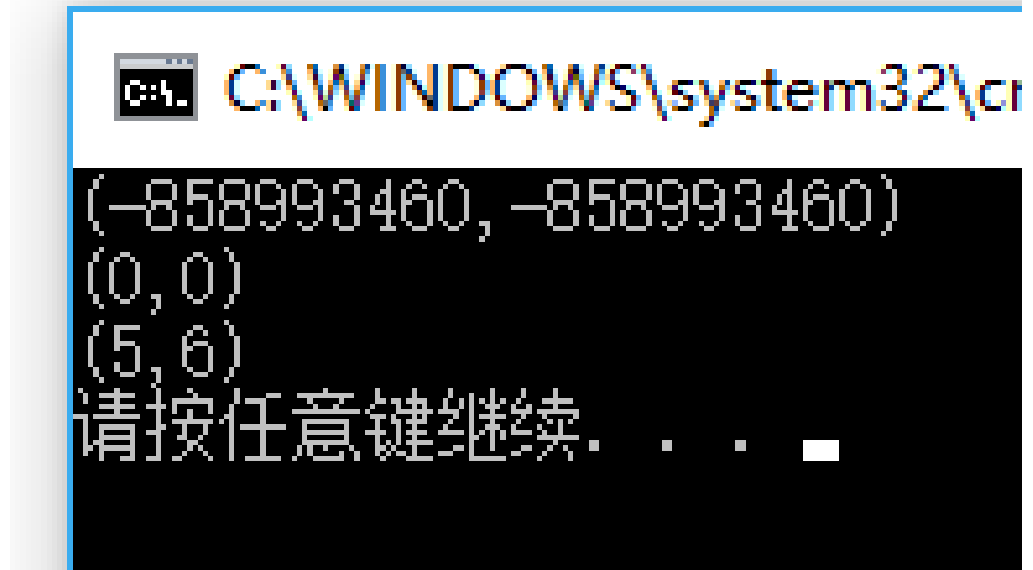
Point t;
int main() {
    Point s;
    cout<<s<<t;
    return 0;
} //=====
```



```
C:\WINDOWS\system32\cmd.e
(-858993460, -858993460)
(0, 0)
请按任意键继续. . .
```

f0901-driver.cpp

```
// f0901.cpp
// global and local object values
#include "point.h"
#include <iostream>
using namespace std;
Point t;
int main() {
    Point s;
    cout << s << t;
    s.set(5, 6);
    cout << s;
    return 0;
}
```



```
C:\WINDOWS\system32\cmd.exe
(-858993460, -858993460)
(0, 0)
(5, 6)
请按任意键继续. . .
```

9.1 构造函数设计

9.1.1 初始化要求

1. 变量的初始化：虽然在定义的同时并不一定需要进行初始化，但无论如何，在参加后续计算前必须先要给定初值

```
int month;
```

```
int month=1;
```

2. 对象与变量的不同在于对象对应于事物，要求从诞生之时起便有明确的意义。
3. 对象若无初始化，将发生以下情况：
 - 全部对象：全0位的模式
 - 局部对象：随机值
4. 对象必须建立一种初始化机制，以满足针对不同条件下的赋初值操作的要求

9.1 构造函数设计

9.1.2 封装性要求

1. 结构体变量初始化时的公有成员特性

```
struct Point;  
Point d={2, 3};  
Point d;    d.x=2;    d.y=3;  // OK!!!
```

2. C++中的类具有封装的特性

```
class Point;  
Point d={2, 3};  // 错误!!!
```

3. 对象创建过程中的特殊性:

- 数据成员一般是私有的;
- 传递的不仅仅是一个简单的值, 可能是一种信息;
- 初始化的过程中除了计算操作以外, 还需要进行必要的验证工作;

4. 封装性要求对象创建过程中按传递的信息进行一个过程化的初始化工作

9.1 构造函数设计

9.1.3 函数形式

1. 初始化的操作应该是一个过程，必须由函数才能完成；
2. 构造函数：类定义中专门用于完成对象创建和初始化的特殊成员函数
3. 构造函数的命名
 - 应该是唯一的；
 - 调用构造函数必须用对象名进行激活，但使用对象名作为构造函数是不现实的；
 - 使用类名成为最佳选择；

9.1 构造函数设计

9.1.4 构造函数的返回值问题

1. 变量创建失败的后果及其处理方法
2. 对象创建失败后的处理方法：
 - 程序捕捉异常
 - 直接终止程序的执行
3. 构造函数不应该有表示任何意义的返回值
4. 构造函数也不是空类型(void)的函数
5. 构造函数中禁止出现return语句

9.1.5 set的缺憾及其改进

1. 两种模式的对比
 - `Date d(2005, 12, 28);`
 - `Date d; d.set(2005, 12, 28);`
2. 改进后的程序举例 f0902(vs演示)

9.1 构造函数设计

9.1.6 一次性对象

1. 在创建对象时不给出对象名，而直接以类名调用构造函数，则将产生一个无名的一次性对象

2. 举例

```
cout << Date(2003, 12, 23);
```

3. 使用场合：一般用在创建后不需要反复使用的场合（如参数传递时）

😊总结：构造函数的特点：

- 1.构造函数不能有返回值
- 2.构造函数不能在程序中显示调用，由系统自动调用。
- 3.构造函数可以重载
- 4.构造函数名字必须和类名字相同

9.2 构造函数的重载

9.2.1 重载构造函数

1. 构造函数可以重载, 也可以参数默认

2. 程序举例: [f0903](#)

3. 无参构造函数调用时的特殊形式

➤ `Date g();` 形式上的歧义, 和函数声明混淆

➤ 为了加以区分, C++语法体系规定采用以下形式:

`Date g;` //去掉括号

➤ 对以下形式的正确理解

`int a;`

`int b();`

`Date g;`

`Date f();`

4. 有参构造函数调用和函数声明在形式上的比较

`Date e(2002);`

`Date e(int y);`



f0903-Date类

```
#include<iostream>
#include<iomanip>
using namespace std;
//-----
class Date{
private:
    int year, month, day;
public:
    Date(int y=2000, int m=1, int d=1); // 设置默认参数
    Date(const string& s);             // 重载
    bool isLeapYear()const;
    friend ostream& operator<<(ostream& o, const Date& d);
};//-----
```

```
Date::Date(const string& s){
    year = atoi(s.substr(0,4).c_str());
    month = atoi(s.substr(5,2).c_str());
    day = atoi(s.substr(8,2).c_str());
}//-----
Date::Date(int y, int m, int d){
    year=y,month=m,day=d; }
bool Date::isLeapYear()const{
    return (year % 4==0 && year % 100 !=0) || year % 400==0;
}//-----
ostream& operator<<(ostream& o, const Date& d){
    o<<setfill('0')<<setw(4)<<d.year<<'-'
    <<setw(2)<<d.month<<'-'
    <<setw(2)<<d.day<<'\n';
    return o;
}//-----
```

f0903-Date类测试

```
int main(){
    Date c("2005-12-28");
    Date d(2003,12,6);
    Date e(2002);           // 默认两个参数
    Date f(2002,12);        // 默认一个参数
    Date g;                 // 默认三个参数
    cout<<c<<d<<e<<f<<g;
} //=====
```



9.2 构造函数的重载

9.2.1 重载构造函数

5. 在构造函数重载中，可以将各构造函数中通用的校验工作的代码分离出来，单独成为一个成员函数并供所有版本的构造函数调用。
6. 举例：[f0904](#)
7. **注意：**f0904中init()函数的定义方式，并理解它被调用的原理！

9.2.2 无参构造函数

1. 对象创建必须而且只能通过构造函数
2. 前面第8章的程序没有任何错误的原因：**默认**的无参构造函数




```
#include<iostream>
#include<iomanip>
using namespace std;
//-----
class Date{
    int year, month, day;
    void init();
public:
    Date(const string& s);
    Date(int y=2000, int m=1, int d=1);
    bool isLeapYear()const;
    friend ostream& operator<<(ostream& o, const Date& d);
};
void Date::init(){
    if(year>5000||year<1||month<1||month>12||day<1||day>31)
        exit(1); // 停机
}
```

f0904-Date类

```
Date::Date(const string& s){
    year = atoi(s.substr(0,4).c_str());
    month = atoi(s.substr(5,2).c_str());
    day = atoi(s.substr(8,2).c_str());
    init();
} //-----

Date::Date(int y, int m, int d){
    year=y, month=m, day=d;
    init();
} //-----

bool Date::isLeapYear()const{
    return (year % 4==0 && year % 100 !=0) || year % 400==0;
} //-----

ostream& operator<<(ostream& o, const Date& d){
    o << setfill('0') << setw(4) << d.year << '-'
    << setw(2) << d.month << '-'
    << setw(2) << d.day << '\n';
    return o;
}
```



f0904-Date类测试

```
int main(){  
    Date c("2005-12-28");  
    Date d(2003,12,6);  
    Date e(2002);  
    Date f(2002,12);  
    Date g;  
    cout<<c<<d<<e<<f<<g;  
}
```



9.2 构造函数的重载

9.2.2 无参构造函数

3. 默认无参构造函数的使用规则：

- 若某类未定义任何一个构造函数，则系统将提供一个默认无参构造函数
- 无参构造函数仅能完成对象空间的申请工作，不能完成其它任何的初始化工作
- **注意：**若某类定义了任何一个构造函数，则系统将不再提供默认无参构造函数；若此时仍然需要，则需用户自己定义
- **注意：**为了方便类的使用，应该而且有必要保证类有一个公有的无参构造函数

```
class Date{  
public:  
    Date(int y, int m, int d);  
    // ...  
};
```

```
int main() {  
    Date d;    // error  
}
```

9.3 类成员初始化

9.3.1 默认调用的无参构造函数

1. 类成员的概念：类中的某个数据成员是另外一个类的对象
2. 类成员初始化时的简单处理方法及其执行过程的分析：[f0905](#)
3. 如果不加特殊处理，则肯定调用类成员的无参构造函数

9.3.2 初始化的困惑

1. 一种企图及其初始化目的的破灭 [f0906](#)
 - ◆ **结论：** 类对象成员的初始化需要用特殊的方式来实现
2. 常量成员和引用成员初始化时也存在一种特殊性
3. 综合举例：L0901 (vs演示)

Goto

f0905

```
//=====
// f0905.cpp
// 对象成员的默认构造
//=====
#include<iostream>
using namespace std;
//-----
class StudentID{
    int value;
public:
    StudentID(){
        static int nextStudentID = 0;
        value = ++nextStudentID;
        cout<<"Assigning student id "<<value<<"\n";
    }
};//-----
```



f0905-续

```
class Student{
    string name;
    StudentID id;
public:
    Student(string n = "noName"){
        cout <<"Constructing student " + n + "\n";
        name = n;
    }
};//-----

int main(){
    Student s("Randy");
    return 0;
};//-----
```

问题：

- StudentID id成员如何初始化？



f0906

```
class Student{
    string name;
    StudentID id;
public:
    Student(string n = "noName", int ssID=0) {
        cout <<"Constructing student " + n + "\n";
        name = n;
        StudentID id(ssID);
    }
};//-----

int main() {
    Student s("Randy", 58);
};//=====
```

问题:

- 能正确初始化 StudentID id成员吗?

➤ NO, 不能!!!

- 如何才能正确初始化 StudentID id成员呢?



9.3 类成员初始化

9.3.2 初始化的困惑(引用成员和常量成员的初始化)

```
class Silly
{
private:
    const int ten;
    int &ra;
public:
    Silly(int x, int &a)
    {
        ten = 10;
        ra = a;
    }
};
```

■ 错误: ten是常量,
不可以作为左值。

■ 错误: 并不能实现a为ra的引用对象, 因此无法对ra进行初始化

9.3 类成员初始化

9.3.3 成员初始化的正确形式

1. **冒号语法**：在构造函数的形参列表的右括号外、花括号前面，使用冒号语法引出构造函数的调用列表，该列表称为**初始化列表**

2. 程序举例：[f0907](#)

3. 冒号语法在初始化成员时可以不出现类的类型声明，这是一种破例

```
PPoint d(3, 2);  
cout<<d(6, 8); //error  
cout<<d.angle( );
```

4. 常量成员、引用成员甚至是普通数据成员也可以通过冒号语法进行初始化

😊 **结论：**

😊 **const** 类型数据成员必须用初始化列表进行初始化

😊 引用数据成员必须用初始化列表进行初始化

😊 类对象成员必须用初始化列表进行初始化

😊 一般的数据成员也可以用初始化列表进行初始化！



```
class Student{
    string name;
    StudentID id;
public:
    Student(string n="no name", int
ssID=0):id(ssID),name(n) {
        cout<<"Constructing student "<<n<<"\n";
    }
};//-----

int main() {
    Student s("Randy", 98);
    Student t("Jenny");
};//=====
```



9.4 构造顺序

9.4.1 局部对象（含自动和静态）

1. 对象或变量创建的语句行顺序与运行顺序

```
int a=3;  
if(a==2)  
    Date d;  
Date e;
```

则对象创建的语句行顺序为：

Date d-->Date e

对象创建的运行顺序为：

Date e

- 2. 在C语言中，局部变量的创建是在函数开始执行时统一创建的，创建的顺序为变量的语句行顺序；而C++中，局部对象的创建是在运行时决定的，但静态对象只执行一次
- 3. 程序举例：f0908

f0908

```
//=====
// f0908.cpp
// test local object create order
//=====
#include<iostream>
using namespace std;
//-----
class A{
public:
    A() { cout<<"A->"; }
};//-----
class B{
public:
    B() { cout<<"B->"; }
};//-----
```

f0908-续

```
class C{
public:
    C() { cout<<"C->"; }
};//-----
-----

void func() {
    cout<<"\nfunc: ";
    A a;
    static B b;
    C c;
};//-----
-----
```

```
int main() {
    cout<<"main: ";
    for(int i=1; i<=2; ++i) {
        for(int j=1; j<=2; ++j)
            if(i==2) C c; else A a;
        B b;
    }
    func(); func();
} //=====
```

输出结果:

```
main:  A->A->B->C->C->B
func:  A->B->C->
func:  A->c->
```

9.4 构造顺序

9.4.2 全局对象

1. 全局对象在main函数启动之前生成，而调试则在main函数启动之后。
2. 上述现象的对策：调试时，应先将全局对象作为局部对象来运行观察。或者，在构造函数中添加输出语句来观察运行过程。
3. 同一工程不同代码文件全局对象的创建没有明确顺序规定。
4. 程序举例：[f0909](#)
5. 对策：
 - 不要让不同文件的全局对象互为依赖。因为依赖具有先后性，而其全局对象的创建不能保证该依赖性发挥作用。
 - 最好不要使用全局对象



f0909

```
//=====
// student.h
//=====
#include<iostream>
using namespace std;
//-----
class Student{
    const int id;
public:
    Student(int d):id(d) { cout<<"student\n";
    }
    void print() { cout<<id<<"\n"; }
};//-----
```


f0909

-续

```
class Tutor{
    Student s;

public:
    Tutor(Student& st):s(st) {
        cout<<"tutor\n"; s.print(); }
    };//-----

extern Student ra;
//-----

Tutor je(ra);
int main() {}
//-----

// f0902-2.cpp
#include "student.h"
Student ra(18);
```

可能结果:

tutor

0

student



9.4 构造顺序

9.4.3 成员对象

1. 成员对象的构造顺序按类定义中各成员对象的出现顺序来决定，而与其在初始化列表中的顺序无关，最后执行自身构造函数
2. 程序举例： [f0910](#)

9.4.4 构造位置

1. 全局对象、常对象、静态对象放在全局数据区
2. 局部对象放在栈区
3. 用户申请的对象放在堆区：new操作申请的内存对象



f0910

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

class A{
public:
    A(int x) { cout<<"A:"<<x<<"-
>"; }
};//-----
-----

class B{
public:
    B(int x) { cout<<"B:"<<x<<"-
>"; }
```

f0910-续

```
class C{
    A a;
    B b;
public:
    C(int x, int y):b(x), a(y) {
        cout<<"C\n"; }
};//-----
----

int main() {
    C c(15, 9);
};//=====
```

输出结果:
A:9->B:15->C

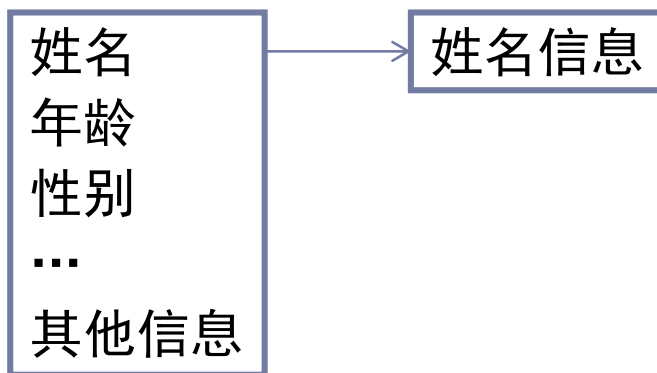


9.5 拷贝构造函数

9.5.1 对象本体与对象实体

1. 对象本体也是对象主体;
2. 对象实体则还包括属于对象的衍生物。如：某个人是人类对象的主体，然而某人还拥有父母，房产等属于某人的世系或资产，描述人的属性不仅仅只是人体数据。
3. 从程序中具体而言，对象除了包括一般的数据成员，还包括指向数据的指针。

个人信息



(对象本体)

9.5 拷贝构造函数

◆ 9.5.1 对象本体与对象实体

4. 更进一步说：在类中包含指针形式的成员变量时，一般在对象创建时还会申请内存空间等形式的资源，从而使得对象实体大于对象本体
5. **重要结论：对象本体与对象实体并不一定等价！**
6. 析构函数的**必要性**：释放创建时申请的空间
7. 程序举例：f0911

```
class Person{
    char* pName;
public:
    Person(char* pN="noName") {
        cout<<"Constructing "<<pN<<"\n";
        pName = new char[strlen(pN)+1];
        if(pName) strcpy(pName, pN);
    }
    ~Person() {
        cout <<"Destructing "<<pName<<"\n";
        delete[] pName;
    }
    /*friend function: 2014-12-1@yws add-----
    -*/
    friend ostream& operator<<(ostream& , const Person&
    obj);
}; //-----
```

f0911

```
/*friend function -----
-----*/

ostream& operator<<(ostream& out, const Person& obj) {
    out<<"The name: "<<obj.pName<<endl;
    return out;
}
```

输出结果:

The two person:

The name:Randy

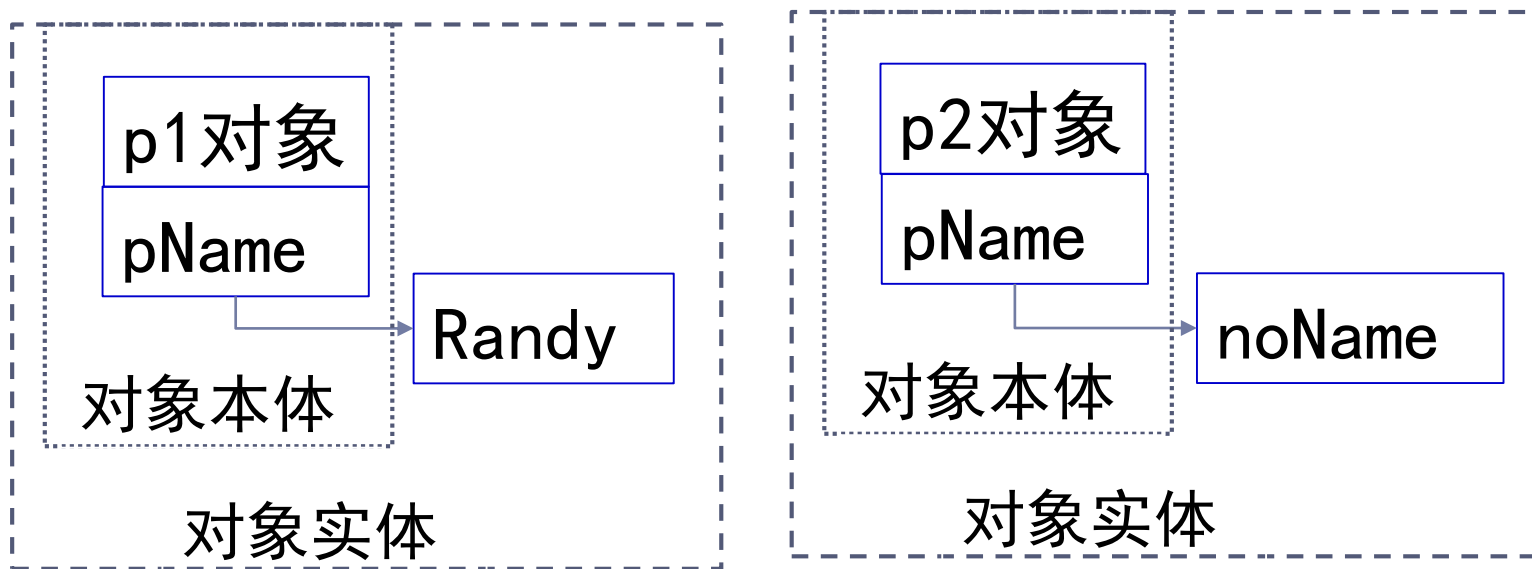
=====

The name: noName

```
int main() {
    Person p1("Randy");
    Person p2;
    cout<<"The two person:\n";
    cout<<p1;
    cout<<"\n=====\\n";
    cout<<p2;
    return 0;
}
```



对象本体和对象实体



9.5 拷贝构造函数

9.5.2 默认拷贝构造函数

1. 类的对象也应该能够和普通变量一样进行同类对象之间的初始化;
2. 若某类中未自定义拷贝构造函数, 则系统将提供一个默认的拷贝构造函数;
3. 程序举例: [f0912-1](#)
4. 默认拷贝构造函数的功能非常弱: 仅能完成新老对象基于位模式的复制, 也就是**只能完成对象本体的复制**。
5. 程序举例: [f0912](#)

f0912

-1

```
/*class definition -----
-----*/
const double pi=3.1415;    // 常量pi
class CCircle {
private:
    double x, y;    // Circle的圆心坐标
    double r;       // Circle的半径
public:
    CCircle(double, double, double); // 带参数的构造函数
    CCircle() ;           // 无参构造函数
    double get_area() const; // 计算圆面积
    /*friend function -----
    -----*/

    friend ostream& operator<<(ostream& ,
const CCircle&);
};
```

f0912 -1续

```
/*class function definition -----  
-----*/  
CCircle::CCircle(double x, double y, double r) {  
    this->x = x;  
    this->y = y;  
    this->r = r;  
}  
CCircle::CCircle() {  
    this->x = 0;  
    this->y = 0;  
    this->r = 0;  
}  
double CCircle::get_area() const {  
    return this->r*this->r*pi;  
}
```

f0912-1续

```
/*friend function definition -----  
-----*/  
ostream& operator<<(ostream& out, const CCircle& obj)  
{  
    out<<"圆心坐标: ("<<obj.x<<","<<obj.y<<")\n"  
        <<"半径: "<<obj.r<<endl;  
    out<<"面积: "<<obj.get_area()<<endl;  
    return out;  
}
```

f0912-1续

```
/*test function  
-*/
```

```
int main() {  
    CCircle circle1, circle2(2, 3);  
    cout<<"两个圆的信息为: \n";  
    cout<<circle1;  
    cout<<"\n===== \n";  
    cout<<circle2;  
    CCircle circle3(circle2);
```

新对象

```
    cout<<"\n===== \n";  
    cout<<"新圆的信息为: \n";  
    cout<<circle3;
```

```
    return 0;
```

C:\WINDOWS\system32\cmd.exe

```
两个圆的信息为:  
Circle的圆心坐标: (0, 0)  
Circle的半径: 0  
Circle的面积: 0
```

```
=====  
Circle的圆心坐标: (2, 3)  
Circle的半径: 5  
Circle的面积: 78.5375
```

```
=====  
新圆的信息为:  
Circle的圆心坐标: (2, 3)  
Circle的半径: 5  
Circle的面积: 78.5375  
请按任意键继续. . .
```

// 利用一个对象构造一个



```
//-----
class Person
{
private:
    char* pName;
public:
    Person(char* pN="noName")    {
        cout<<"Constructing " <<pN<<"\n";
        pName = new char[strlen(pN)+1];
        if(pName) strcpy(pName, pN);
    }
    ~Person();
    void set_name(char*pN);    // 修改姓名的函数
    /*friend function: 2014-12-1@yws add-----
-----*/
    friend ostream& operator<<(ostream& , const Person&
    obj);
} . //-----
```

f0912

续

```
/*class function -----
-----*/
Person::~~Person() {
    cout << "Destructing " << pName << "\n";
    delete [] pName;
}

void Person::set_name(char *pN) {
    if (pName != NULL)
    {
        delete [] pName;
    }
    pName = new char[strlen(pN)+1];
    if (pName)
    {
        strcpy(pName, pN);
    }
}
}
```


f0912-续

```
/*friend function -----  
-----*/  
ostream& operator<<(ostream& out, const Person& obj)  
{  
    out<<"The name of this person:  
"<<obj.pName<<endl;  
    return out;  
}
```

f0912-测试1

```

int main() {
    Person p1("Randy");
    Person p2(p1);    // 调用默认的拷贝构造函数进行初始化

    cout<<"The two person:\n";
    cout<<p1;
    cout<<p2;
    cout<<"\n===== \n";
    p1.set_name("John");
    cout<<"The two person:\n";
    cout<<p1;
    cout<<p2;
    return 0;
}

```

问题：

➤ p1和p2的name值同时发生变化

 C:\WINDOWS\system32\cmd.exe[illegible]

f0912-测试2

发生了什么问题？

```
int main() {  
    Person* pp1 = new Person;  
    Person p2(*pp1);    // 调用默认的拷贝构造函数  
    pp1->set_name("John");  
    cout<<"The two person:\n";  
    cout<<*pp1;  
    cout<<p2;  
    delete pp1;  
    cout << "The two person:\n";  
    cout << p2;  
    return 0;  
}
```

C:\WINDOWS\system32\cmd.exe

```
Constructing noName  
The two person:  
The name of this person: John  
The name of this person: John  
Destructing John  
The two person:  
The name of this person: 葺葺葺葺葺  
Destructing 葺葺葺葺葺  
请按任意键继续. . .
```

9.5 拷贝构造函数

9.5.2 默认拷贝构造函数

1. 结论

- 对象本体和对象实体一致时，可以使用默认拷贝构造函数来通过其他对象实现对象的初始化
- 对象本体和对象实体不一致时，则不可以使用默认拷贝构造函数来通过其他对象实现对象的初始化

2. 浅拷贝和深拷贝

- 对象本体和对象实体一致时，其拷贝称为浅拷贝
- 对象本体和对象实体不一致时，其拷贝称为深拷贝

9.5 拷贝构造函数

9.5.3 自定义拷贝构造函数

1. 在对象本体和实体不一致时，必须要通过正确的自定义拷贝构造函数来避免浅拷贝问题
2. 程序举例：[f0913](#)
3. 自定义拷贝构造函数的形式参数的理解
 - 必须是类对象的**常量引用**：考虑实例：`MyClass c3 = c1+c2;`
 - `c1+c2`返回的是一个临时对象，**只有常量引用才可以引用临时对象**，所以拷贝构造函数的参数必须是常量引用。
 - 引用的目的是为了**避免对象复制过程中的死循环**；
 - 常量限定的目的是为了**保护实参对象**；而且很多时候传递给构造函数的实参本身就是**const对象**
4. 自定义拷贝构造函数被调用的必要条件：
 - `Date obj1=obj2;`
 - `Date obj1(obj2);`
 - 函数参数匹配时
5. 函数的形参为类的对象时为什么需要使用引用形式的原因分析

Goto 

```

class Person{
private:
    char* pName;

public:
    Person(char* pN="noName")    {
        cout<<"Constructing " <<pN<<"\n";
        pName = new char[strlen(pN)+1];
        if(pName) strcpy(pName, pN);

        age = 10;
    }

    Person(const Person& s);    // 拷贝构造函数
    ~Person();

    void set_name(char*pN);    // 修改姓名的函数

    /*friend function: 2014-12-1@yws add-----
    -----*/

    friend ostream& operator<<(ostream& , const Person&
54j);
} //

```

f0913续

```
/*class function -----  
-----*/  
Person::~~Person() {  
    cout <<"Destructing " <<pName<<"\n";  
    delete[] pName;  
}  
Person::Person(const Person& s) {  
    cout<<"copy Constructing " <<s.pName<<"\n";  
    pName = new char[strlen(s.pName)+1];  
    if(pName) strcpy(pName, s.pName);  
    this->age = s.age;  
}
```

```

void Person::set_name(char *pN) {
    if (pName != NULL) {
        delete [] pName;
    }
    pName = new char[strlen(pN)+1];
    if(pName) {
        strcpy(pName, pN);
    }
}

/*friend function -----
-----*/
ostream& operator<<(ostream& out, const Person& obj)
{
    out<<"The name of this person: "<<obj.pName
        << "   age:"<<obj.age<<endl;
    return out;
}

```


f0913-测试1

```
int main() {  
    Person p1("Randy");  
    Person p2(p1);  
    cout<<"The two person:\n";  
    cout<<p1;  
    cout<<p2;  
    cout<<"\n===== \n";  
    p2.set_name("John");  
    cout<<"The two person:\n";  
    cout<<p1;  
    cout<<p2;  
    return 0;  
}
```

```
C:\WINDOWS\system32\cmd.exe  
Constructing Randy  
copy Constructing Randy  
The two person:  
The name of this person: Randy age:10  
The name of this person: Randy age:10  
=====  
The two person:  
The name of this person: Randy age:10  
The name of this person: John age:10  
Destructing John  
Destructing Randy  
请按任意键继续. . .
```

➤ p1和p2的name值不
同时变化

f0913-测试2

```
int main() {  
    Person* pp1 = new Person;  
    Person p2(*pp1);    // 调用自定义的拷贝构造函数  
    pp1->set_name("John");  
    cout << "The two person:\n";  
    cout << *pp1;  
    cout << p2;  
    delete pp1;  
    cout << "The two person:\n";  
    cout << p2;  
    return 0;  
}
```

cmd C:\WINDOWS\system32\cmd.exe

```
Constructing noName  
copy Constructing noName  
The two person:  
The name of this person: John age:10  
The name of this person: noName age:10  
Destructing John  
The two person:  
The name of this person: noName age:10  
Destructing noName  
请按任意键继续. . .
```

9.5 拷贝构造函数

9.5.3 自定义拷贝构造函数

6. 拷贝构造函数并不是必须的，只有当对象本体和对象实体不一致时才是必须的。更明确地说，只有当对象中含有指针形式的成员变量并占用内存资源的时候才需要。而且必须需要，否则浅拷贝而死机

■结论：

- 对象本体与对象实体一致时，可以不自定义拷贝构造函数，直接使用默认拷贝构造函数。
- 对象本体与对象实体不一致时，则必须自定义拷贝构造函数，不允许使用默认拷贝构造函数。
- 自定义拷贝构造函数一般定义为公有成员函数，仍然是构造函数的一个重载函数。
- 一旦自定义拷贝构造函数，则系统不再提供默认拷贝构造函数。

9.6 析构函数

1. 对象结束其生命时，会被系统悄悄地销毁(析构). 即对象本体空间与名字脱离关系.
2. 对象结束生命时，若对象本体与对象实体不同，则需要人为地进行资源释放，以保证对象本体失效之前，资源被收回.
3. 定义析构函数的目的：
由于对象本体与实体不同，所以要进行对象占有资源的释放工作.
4. 一般来说：一个类，若有人为定义的拷贝构造函数，则也应该定义析构函数. 因为对象创建中有资源要获得分配，则对象失效前必应先释放资源.
5. 析构函数的一般语法形式：
~类名（）；
6. 析构函数是在对象生命周期结束时由系统自动调用的

9.6 析构函数

7. 析构函数的说明：

- 析构函数无参数
- 析构函数无返回值
- 析构函数不能重载
- 析构函数的执行顺序和构造函数相反

8. 程序举例： f0914

f0914

```
//-----  
-----  
  
class A{  
public:  
    A() { cout<<"A-> "; }  
    ~A() { cout<<"<-~A "; }  
};//-----  
-----
```

```
  
  
class B{  
public:  
    B() { cout<<"B-> "; }  
    ~B() { cout<<"<-~B "; }  
};//-----  
-----
```

```
class C{  
public:  
    C() { cout<<"C-> "; }  
    ~C() { cout<<"<-~C "; }  
};//-----  
-----
```

```
  
void func() {  
    cout<<"\nfunc: ";  
    A a;  
    cout<<"ok-> ";  
    static B b;  
    C c;  
};//-----  
-----
```

f0914-测试

```
int main() {  
    cout<<"main: ";  
    for(int i=1; i<=2;  
++i) {  
        for(int j=1; j<=2;  
++j)  
            if(i==2) C c;  
else A a;
```

D L.

C:\WINDOWS\system32\cmd.exe

```
}  
main: A-> <-~A A-> <-~A B-> <-~B C-> <-~C C-> <-~C B-> <-~B  
func: A-> ok-> B-> C-> <-~C <-~A  
func: A-> ok-> C-> <-~C <-~A <-~B 请按任意键继续. . .  
} // =
```

9.7 对象转型与赋值

■ 对象转型

一个构造函数, 含有一个其他数据类型的参数, 显然其意义为, 用该参数类型的值可以创建本对象. 从另一方面看, 参数类型的值可以转换为本对象.

```
class Student{
public:
    Student(const string& n);
    // ...
};
void fn(Student& s);
int main(){
    string t="jenny";
    fn(t); // 参数为string, 却能匹配Student类型
}
```


9.7 对象转型与赋值

■ 对象转型的规则：

- 只会尝试含有一个参数的构造函数
- 如果有二义性，则会放弃尝试
- 推导是一次性的，不允许步推导

fn("Jenny")不能匹配

void fn(const Student& s);

因为： " Jenny"

-> string

-> Student

经历了两步.

9.7 对象转型与赋值

9.7.2 对象赋值

1. 对象赋值即对象拷贝：两个已经存在的对象之间的复制
`Person d, g;`
`d = g; // 对象赋值`
2. 对象赋值便是使用类中的赋值操作符。
3. 如果类中没有定义赋值操作符，则系统悄悄地定义一个默认的赋值操作符：
`Person& operator=(const Person& p)`
`{`
 `memcpy(*this, *p, sizeof(p));`
`}`
4. 默认的赋值运算符在对象本体和对象实体不一致时将会产生浅拷贝问题
5. 如对象本体和实体不一致时，必须要自定义赋值运算符

9.7 对象转型与赋值

定义赋值操作符:

- 排除客体对象与本对象同一的情况
- 申请客体对象相同大小的资源空间
- 拷贝客体对象的资源到本对象
- 释放本对象的资源

```
class Person{  
    char* pName;  
public:  
    Person(char* pN="noName");  
    Person(const Person& s);  
    Person& operator=(const Person& s){  
        if(this==&s) return s;  
        delete[] pName;  
        pName = new char[strlen(s.pName)+1];  
        if(pName) strcpy(pName,s.pName);  
        return *this;  
    }  
    ~Person(){  
        delete[] pName;  
    }  
};
```

9.7 对象转型与赋值

6. 赋值运算符的重载和拷贝构造函数是完全不同的概念，其功能也是完全不同的
7. 赋值运算符和拷贝构造函数不同的调用场合举例：
 - `DATE Obj1=Obj2;`
 - `DATE Obj1(Obj2);`
 - `DATE Obj1,Obj2; Obj1=Obj2;`
 - `void fun(DATE Obj); DATE Obj1; fun(obj1); //拷贝构造`
 - `void fun(DATE &Obj); DATE Obj1; fun(obj1); // 都不调用`
 - `DATE fun(); DATE Obj1; Obj1 = fun(); // 拷贝构造`
 - `DATE& fun(); DATE Obj1; Obj1 = fun(); //赋值运算符`
 - `DATE& fun(); DATE Obj1 = fun(); // 拷贝构造`
 - `DATE& fun(); DATE &Obj1 = fun(); // 都不调用`

本章小结

◆ 构造函数

- 函数名：类名
- 不允许有返回值
- 可以重载
- 不允许显示调用，由系统自动调用

◆ 默认构造函数/无参构造函数

- 如果不为类定义构造函数，则系统提供一个默认构造函数
- **建议**为类提供一个无参构造函数，取代默认构造函数的功能。
- 功能：用于定义不带初始值的对象

◆ 带参数的构造函数/带默认参数值的构造函数

- 带默认参数值的构造函数可以取代无参构造函数

本章小结

◆ 特殊对象的构造

- 类对象成员、常量成员、引用成员：必须使用冒号法来初始化
- 对象的构造顺序：
 - ◆ C语言是在程序运行前构造所有变量
 - ◆ C++是在程序运行过程根据需要来构造所有对象
 - ◆ 类成员的构造顺序是成员在类中的定义顺序，与成员在冒号法中的顺序无关。

本章小结

◆ 拷贝构造函数

● 对象本体

- ◆ 也就是对象主体，指由对象成员数据组成的主体部分。

● 对象实体

- ◆ 包含对象主体部分以及对象的外延数据，对象实体一般大于等于对象本体

● 默认拷贝构造函数

- ◆ 只能实现对象本体的按位拷贝赋值，称为**浅拷贝**
- ◆ 只能应用于对象本体和对象实体一致的类中。

● 自定义拷贝构造函数

- ◆ 对象本体和对象实体不一致时，必须自定义拷贝构造函数
- ◆ 实现对象实体拷贝的操作称为**深拷贝**

本章小结

◆析构函数

- 函数名：~类名()
- 不允许有返回值
- 不允许有参数
- 不允许重载
- 不允许显示调用，由系统自动调用
- 功能：是否对象占有的资源，比如释放动态内存、关闭文件等

◆转型构造函数

- 只有一个参数
- 仍然是构造函数，具有构造函数的特点
- 只进行一次转换推理

综合练习

将堆栈作为一种抽象数据类型，定义成如下类： f0919

```
class MyStack
{
    int *data; //A stack of characters
    int top;
    int MaxLength;
public :
    MyStack(int len);
    MyStack(const MyStack &p);
    ~MyStack();
    bool Push(char n);
    int Pop(char& r);
    int IsEmpty(void) { return top == 0; }
    int IsFull(void) { return top >= MaxLength; }
};
```

它实现一个拥有 n 个整数的堆栈，试编写该类的成员函数程序代码，写成一个完整的可运行源程序。