第二讲 类和对象（二）

一、函数重载

二、初始化列表

三、复合类

四、析构函数

一、函数重载

什么是函数重载？在同一个作用域中，函数名相同，参数列表不同，则多个函数构成重载。参数列表不同分为两种情况：参数个数不同；参数类型不同。如：

```
void f();
void f(int x);
void f(int x,float y);
```

【例1-1】Thing类中重载构造函数

```
/**Thing头文件**/
#ifndef __test__Thing__
#define __test__Thing__
#include <iostream>
using namespace std;
class Thing
{
public:
    Thing();
    Thing(int _x);
    Thing(int _x,int _y);
    //Thing(int _x=0);
    //Thing(int _x=0,int _y=0);

private:
    int x;
    int y;
};
#endif
```

```
/**Thing实现文件**/
#include "Thing.h"
Thing::Thing() {
    x = 0;
    y = 0;
}

Thing::Thing(int _x) {
    x = _x;
    y = 0;
}

Thing::Thing(int _x,int _y) {
    x = _x;
    y = _y;
}

/**测试文件**/
#include <iostream>
#include "Thing.h"
using namespace std;
int main(int argc, const char * argv[]) {
    Thing t;//调用第一个构造函数
    Thing t1(1);//调用第二个构造函数
    Thing t2(1,2);//调用第三个构造
    return 0;
}
```

示例分析:

⑴ 类中可以定义多个构造函数，但是必须能够构成函数重载;

⑵ 类中被注释的两个默认构造函数与对应的两个标准非默认构造无法构成重载，因为函数原型相同;

⑶ 为了避免出现歧义，类中最多只能出现一个默认构造函数;

⑷ 多个默认构造函数歧义原因分析：如示例1-1所示，main函数中声明了三个对象t,t1,t2。声明对象t时没有传递任何参数，所以编译器会调用类中的默认构造函数，如果类中存在多个默认构造函数，则编译器无法确定调用哪一个默认构造函数初始化对象;

二、初始化列表

初始化列表的作用与构造函数体内的作用相同，都是用于初始化成员变量。

【例1-2】初始化列表的使用

```
/**Thing头文件**/

#ifndef __test__Thing__
#define __test__Thing__

#include <iostream>
using namespace std;
class Thing
{
public:
    Thing(int _x,int _y);
    void print();

private:
    int x;
    int y;
};
#endif
```

```
/**Thing实现文件**/

#include "Thing.h"
Thing::Thing(int _x,int _y):x(_x),y(_y){}

void Thing::print() {
    cout<<"("<<x<<","<<y<<")"<<endl;
}

/**测试文件**/

#include <iostream>
#include "Thing.h"
using namespace std;

int main(int argc, const char * argv[]) {
    Thing t(1,1);
    t.print();
    return 0;
}
```

示例分析:

(1) 初始化列表只能出现在构造函数处，具体位置为：构造函数头部之后，以冒号开头，后面紧跟函数体；如：Thing::Thing():x(0),y(0){};

⑵ 初始化列表优先于函数体执行，相比于函数体执行效率更高；

⑶ 有四种情况必须使用初始化列表对成员变量进行初始化：const成员变量；引用类型的成员变量；没有默认构造函数的类类型成员变量；在继承关系中，基类部分的初始化需要使用初始化列表；

三、复合类

一个类的成员变量是另外一个类的对象，那么前者被称为复合类。

【例2-1】复合类示例

```
/**Point头文件**/

#ifndef __test__Point__
#define __test__Point__

#include <iostream>
using namespace std;

class Point
{
public:
    Point();
    Point(float _x,float _y);
    void print();

private:
    float x;
    float y;
};
#endif
```

```
/**Point实现文件**/
#include "Point.h"
Point::Point(){}

Point::Point(float _x,float _y) {
    x = _x;
    y = _y;
}

void Point::print() {
    cout<<"("<<x<<","<<y<<")"<<endl;
}

/**Circle头文件**/
#ifndef __test__Circle__
#define __test__Circle__
#include <iostream>
#include "Point.h"
using namespace std;

class Circle
{
public:
    Circle();
    Circle(float x,float y,float r);
    void setCircle(float r,Point p);
    void print();
```

```
private:
    float radius;
    Point origin;
};
#endif

/**Circle实现文件**/

#include "Circle.h"
Circle::Circle(){}

Circle::Circle(float x,float y,float r):origin(x,y) {
    radius = r;
}

void Circle::setCircle(float r,Point p) {
    radius = r;
    origin = p;
}

void Circle::print() {
    cout<<"radius:"<<radius<<endl;
    origin.print();
}
```

```
/**测试文件**/

#include <iostream>
#include "Point.h"
#include "Circle.h"
using namespace std;
int main(int argc, const char * argv[]) {
    Point p(5,7);
    p.print();
    Circle c;
    c.setCircle(1, p);
    c.print();
    return 0;
}
```

程序运行结果如下：
(5,7)
radius:1
(5,7)

示例分析：

(1) 示例中Circle类内定义了一个Point类型的成员变量origin，那么Circle类为复合类；

(2) 如果构造函数中没有显式使用初始化列表初始化origin，编译器会自动调用Point类中的默认构造对其进行初始化；否则根据实参列表调用匹配的构造函数初始化origin；

四、析构函数

析构函数的作用：当对象生命周期结束时，回收对象所占空间。如：~Circle()。

析构函数有四个特点：

（1）没有返回值；

（2）不能带参数；

（3）不能被重载；

（4）访问权限为公有。

【例3-1】析构函数的简单调用

```
/**Test头文件**/

#ifndef __test__Test__
#define __test__Test__

#include <iostream>
using namespace std;
class Test
{
public:
    Test();
    ~Test();
};

/**Test实现文件**/


#include "Test.h"
Test::Test() {
    cout<<"hello world"<<endl;
}

Test::~Test() {
    cout<<"goodbye world"<<endl;
}
```

```
/**测试文件**/

#include <iostream>
#include "Test.h"
using namespace std;

int main(int argc, const char * argv[]) {
    Test t;
    return 0;
}
```

程序运行结果如下：

hello world

goodbye world

【例3-2】析构函数的实现与使用

```
/**Test头文件**/

#ifndef __test__Test__
#define __test__Test__

#include <iostream>
using namespace std;

class Test
{
public:
    Test();
    ~Test();

private:
    int* _pVar;
};

#endif
```

```
/**Test实现文件**/
#include "Test.h"
Test::Test() {
    _pVar = new int[10];
    cout<<"hello world"<<endl;
}
Test::~Test() {
    if (_pVar != NULL)
    {
        delete []_pVar;
        _pVar = NULL;
    }
    cout<<"goodbye world"<<endl;
}

/**测试文件**/
#include <iostream>
#include "Test.h"
using namespace std;

int main(int argc, const char * argv[]) {
    Test t;
    return 0;
}
```

程序运行结果如下:

hello world

goodbye world

示例分析:

(1) 析构函数会由编译器自动调用。调用时机分三种：对象的作用域结束、销毁对象时以及delete指向对象的指针时，析构函数都会被调用;

(2) 如果类中没有定义析构函数，编译器会提供一个空析构;

(3) 如果类中定义了指针成员变量，并且指向了堆区空间，为了避免内存泄漏，必须在类中定义析构函数;