# 第四讲 类和对象（四）

## 一、类的静态成员

## 二、友元类与友元函数

# 一、类的静态成员

【例1-1】静态局部变量回顾

```
#include <iostream>
using namespace std;
void show_average(double x) {//输入数字并计算平均值
    static double num = 0;
    static double sum = 0;
    num += 1;
    sum += x;
    cout<<"num="<<num<<" sum="<<sum<<" avg="<<sum/num<<endl;
}
int main(int argc, const char * argv[]) {
    double entry = 0;
    for (; ; ) {
        cout<<"Enter a number: ";
        cin>>entry;
        if (entry < 0) {
            break;
        }
        show_average(entry);
    }
    return 0;
}
```

程序运行结果如下:

```
Enter a number: 1
num=1 sum=1 avg=1
Enter a number: 10
num=2 sum=11 avg=5.5
Enter a number: 20
num=3 sum=31 avg=10.3333
Enter a number: -1
```

示例分析：

⑴ 静态局部变量存储在静态存储区；

⑵ 静态局部变量在函数结束后内存空间不会被回收，下次使用时可以保留上一次的计算结果，这与全局变量很相似；

⑶ 静态局部变量如果未进行初始化，会被编译器初始化为0或NULL；

**静态全局变量：**

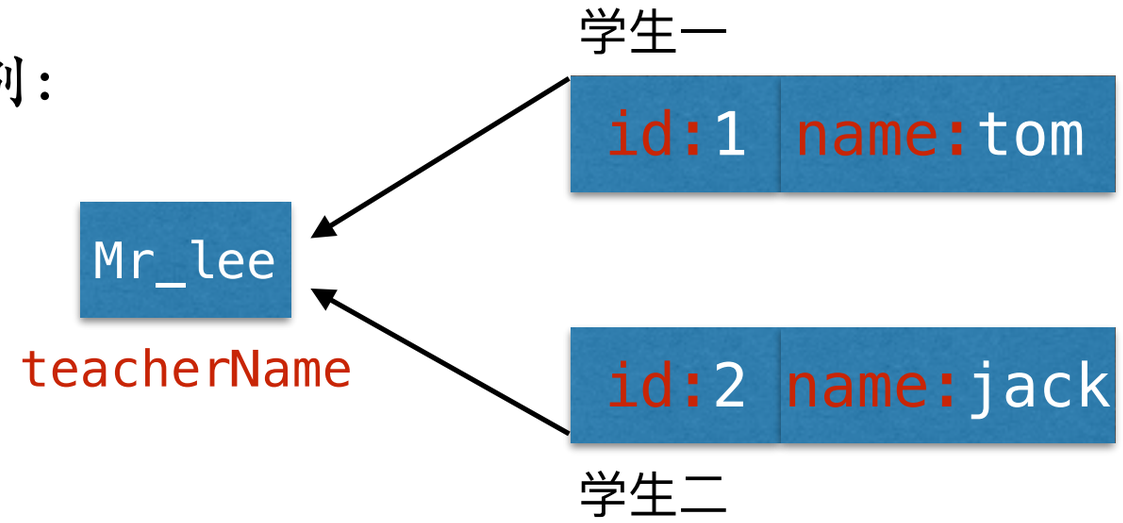
静态全局变量只能在它所声明的文件中使用，不能被extern关键字引用。

**类的静态成员：**

目前我们使用的所有的数据成员和成员函数都是被某个特定的对象调用，但是一个类也可能有数据成员和成员函数属于整个类，这样的成员称之为类的静态成员。

**类的静态成员变量和非静态成员变量的区别：**

非静态成员变量：每个对象都会为自己的非静态成员变量，在内存中保留一块空间，来存储它的值。而对于非静态成员变量，整个类只有一份拷贝，不需要每个对象都保留一份拷贝。




```
class Student
{
    int id;
    char *name;
    static char *teacherName;
};
```

所有学生对象共享同一个teacherName

上述图解中，构造一个学生类，有学生id、name、teacherName三个属性，而我们知道每一个学生都应该有自己的id和name，但是对于teacherName则不需要每一个对象都有一块空间来保存teacherName，因为所有的学生的teacherName都一样，因此只需要在整个类中有一块空间来存储这个teacherName即可，所以将teacherName声明为静态成员变量，既可以节省空间也可以方便修改值。

**类的静态成员函数：**

一个类的成员函数也可以被指定为静态成员函数，如static int totalPerson();

**类的静态成员访问：**

类的静态成员可以通过类名::静态成员名直接访问(Student::teacherName)，或使用对象名.静态成员（Student s1;s1.teacherName），因为静态成员不属于某一个特定的对象，属于整个类的所有对象共有，所以推荐使用第一种方式来访问。

【例1-2】类的静态成员示例

```
/** Employee.h **/
#ifndef __test__Employee__
#define __test__Employee__

#include <iostream>
using namespace std;

class Employee
{
public:
    Employee(char * name_str);
    static void compute_totalpay(float f);
    void print();

private:
    char   name[50];
    int    id;
    static int next_id;
    static float total_pay;
};
#endif
```

```
/** Employee.cpp **/
#include “Employee.h"

int Employee::next_id = 0;//类的静态成员变量必须在类外初始化
float Employee::total_pay = 0;

Employee::Employee(char * name_str) {
    next_id++;     //静态成员变量
    id = next_id; //可以保证每一个员工id唯一
    strcpy(name, name_str);
}

void Employee::compute_totalpay(float f) {
    //cout<<name<<id<<endl;  //error静态成员函数中不能访问非静态成员
    //print();               //error
    total_pay += f;
    cout<<"total_pay:"<<total_pay<<endl;
}

void Employee::print(){
    cout<<"name:"<<name<<" id"<<id<<endl;
}
```

```cpp
/** main.cpp **/
#include <iostream>
#include "Employee.h"
using namespace std;

int main(int argc, const char * argv[]) {
    Employee lily("lily");
    Employee jack("jack");
    Employee rose("rose");
    lily.print();//调用print函数对lily的信息进行打印
    lily.acc_total(10000);
    jack.acc_total(20000);
    rose.acc_total(30000);
    Employee::acc_total(40000);
    Employee::acc_total(50000);
    Employee::acc_total(60000);
    return 0;
}
```

```
程序运行结果如下:
    name:lily id1
    total_pay:10000
    total_pay:30000
    total_pay:60000
    total_pay:100000
    total_pay:150000
    total_pay:210000
```

注意：静态成员函数不能访问非静态成员变量和非静态成员函数。而普通成员函数，既可以访问静态成员也可以访问非静态成员。

# 二、友元类和友元函数

友元函数可以访问一个类的私有成员，但注意元友函数并不是类的成员函数。一个类也可以作为另一个类的友元类，也就意味着这个类中的所有的成员函数是另一个类的友元函数。

【例2-1】友元的使用

```
/** Point头文件 **/
#ifndef __test__Point__
#define __test__Point__
class Circle;//前向声明,表示有一个类为Circle,但此时编译器并不知道该类的具体声明和实现
class Point{
public:
    Point();
    Point(float a,float b);
    friend class Circle;//为Point类声明了友元类
    friend const Point middle(const Point& u,const Point& v);//声明友元函数
    void print()const;
private:
    float x;
    float y;
};
#endif
```

```
/** Point.cpp **/
#include “Point.h”

//默认构造函数
Point::Point():x(0.0),y(0.0) {
    cout<<"Default constructor is called"<<endl;
}

//普通构造函数
Point::Point(float a,float b):x(a),y(b) {}

//打印函数
void Point::print()const {
    cout<<"x:"<<x<<" y:"<<y<<endl;
}

//友元函数
const Point middle(const Point& u,const Point& v) {
    return Point((u.x+v.x)/2.0,(u.y+v.y)/2.0);
}
```

下面的Circle类是一个复合类，将Point类的对象center作为成员变量，Circle类再扩展一个半径radius即可构成圆类。

```
/** Circle.h **/
#ifndef __test__Circle__
#define __test__Circle__

#include <iostream>
#include "Point.h"
using namespace std;
class Circle
{
public:
    Circle();
    Circle& move(float a, float b);
    void print()const;

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

```
/** Circle.cpp **/
#include "Circle.h"
Circle::Circle():radius(0.0) {
    center.x = 0.0;
    center.y = 0.0;
}

//Circle是Point的友元类，可以直接访问Point中的私有成员
Circle& Circle::move(float a, float b) {
    center.x += a;
    center.y += b;
    return *this;
}

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

```
/** main.cpp **/
#include <iostream>
#include "Point.h"
#include "Circle.h"
using namespace std;

int main()
{
    Point p(1.0,2.0);
    Point p1(3.0,4.0);
    p.print();
    p1.print();
    middle(p, p1).print();

    Circle c;
    c.print();
    c.move(10.0, 10.0);
    c.print();
    return 0;
}
```

程序运行结果如下：

```
x:1.0 y:2.0
x:3.0 y:4.0
x:2.0 y:3.0
Default constructor is called
x:0.0 y:0.0
radius:0.0
x:10.0 y:10.0
radius:0.0
```

示例分析:

⑴ middle函数为Point类的友元函数，则函数体内可以直接通过对象名来访问类中的私有成员;

⑵ 友元函数并非类的成员函数，所以其本身不受访问权限修饰符的限制;

⑶ Circle类为Point类的友元类，Circle类中所有的成员函数都可以访问Point类中的私有成员;

⑷ 友元函数可以直接访问类的私有成员，提高了性能，但同时破坏了封装性;