程序设计实习（实验班-2024春）
C++面向对象编程：类与对象

授课教师：姜少峰
助教：冯施源 吴天意

Email: shaofeng.jiang@pku.edu.cn

以下面的问题为例讲解类在C++的实现

- 实现下列与地理信息有关的功能
  - 支持插入一个二维点（每维都是[0,99]整数，不超过1000次插入）
  - 给定一个二维矩形区域的查询，返回当前存在于这里面的点数
  - 返回被查询次数最多的点的坐标

如果一个点在某个矩形区域查询的时候被计数进去了，则记为被查询了一次

```
class GeoDataHandler
{
    private:
    Point arr[1001]; int len; int freq[100][100];
    public:
    GeoDataHandler() {len = 0; memset(freq, 0, sizeof(freq));}
    void append(int x, int y) {
        arr[len++] = Point(x, y);
    }
    int query(int x1, int y1, int x2, int y2) {
        int counter = 0;
        for (int i = 0; i < len; i++) {
            if (arr[i].x >= x1 && arr[i].x <= x2 && arr[i].y >= y1 && arr[i].y <= y2) {
                    counter++; freq[arr[i].x][arr[i].y]++;
            }
        }
        return counter;
    }
    Point HH() {
        int max = -1; Point res;
        for (int i = 0; i < 100; i++)
            for (int j = 0; j < 100; j++)
                if (max < freq[i][j]) {max = freq[i][j]; res = Point(i, j);}
        return res;
    }
}; //注意这个分号
```

访问权限

成员变量

构造函数

成员函数

```
struct Point
{
    int x, y;
    Point() {x = 0; y = 0;}
    Point(int _x, int _y) : x(_x), y(_y) {}
}; //注意这个分号
```

构造函数

访问权限

注意冒号

```
class someClass {
    private:
    // 私有变量和函数
    public:
    // 公有变量和函数
    protected:
    // 保护变量和函数
};
```

访问权限

强行使用会编译报错

此处“内部”也可以理解成实现当前类用的代码中

- private — 私有成员，只能接受来自该类内部的访问
- public — 公有成员，可以接受任何来源的访问
- protected — （之后讲到继承再详细讨论）
- 可在类定义中多次、任意顺序出现
- 作用范围：直到再次遇到下一个访问权限声明或类声明结束

即仅在类内函数的时候可访问；在类外任何作用域都不行

```
class Point {
    private:
    int x, y;
    public:
    int f() {
        return x + y;
    }
};
int main() {
    Point p;
    cout << p.x << endl;
    return 0;
}
```

类内可用private变量，即使在public函数中

这里对于p.x访问报错，因为x是p的private变量

private？为什么不all in public?

- 私有成员变量：隐藏
  - 为了封装，外界不应该看到类自身的数据，对这些数据的访问应该通过对应的成员函数（易于修改和维护）
- 私有成员函数：如工具性的函数，外界不应该调用，但是对于本类实现很有用
  - 例如针对这个if里面的判别条件：

```
if (arr[i].x >= x1 && arr[i].x <= x2 && arr[i].y >= y1 && arr[i].y <= y2)
```

可以写一个private的函数

```
bool isin(Point p, int x1, int y1, int x2, int y2)
```

然后直接if (isin(p, x1, y1, x2, y2))

默认访问权限

- 如果不加任何private/public/protected，则使用默认访问权限
- class的默认访问权限是private
- struct的默认访问权限是public，且这是class与struct仅有的区别！
  - 也就是严格意义上的C的struct是不存在于C++的，是允许定义函数的

默认访问权限是public的
因此x和y是public的

```
struct Point
{
    int x, y;
    Point() {x = 0; y = 0;}
    Point(int _x, int _y) : x(_x), y(_y) {}
}; //注意这个分号
```

```
class GeoDataHandler
{
    private:
    Point arr[1001]; int len; int freq[100][100];

    public:
    GeoDataHandler() {len = 0; memset(freq, 0, sizeof(freq));}
    void append(int x, int y) {
        arr[len++] = Point(x, y);
    }
    int query(int x1, int y1, int x2, int y2) {
        int counter = 0;
        for (int i = 0; i < len; i++) {
            if (arr[i].x >= x1 && arr[i].x <= x2 && arr[i].y >= y1 && arr[i].y <= y2) {
                    counter++; freq[arr[i].x][arr[i].y]++;
            }
        }
        return counter;
    }
    Point HH() {
        int max = -1; Point res;
        for (int i = 0; i < 100; i++)
            for (int j = 0; j < 100; j++)
                if (max < freq[i][j]) {max = freq[i][j]; res = Point(i, j);}
        return res;
    }
}; //注意这个分号
```

作用域与this指针

在class外不能直接看到/访问到class内定义的成员

- 在整个class关键字内定义的一切成员的作用域仅限于class内部
- 成员变量的作用域类似于类里面的“全局变量”
  - 如果与局部变量产生歧义，如何消歧义呢？ → this指针
  - **this**指针就是对当前对象的指代

```
struct Point
{
    int x, y;
    Point() {x = 0; y = 0;}
    Point(int _x, int _y) : x(_x), y(_y) {}
    void setValue(int x, int y) {
        this->x = x;
        this->y = y;
    }
};
```

参数中的x和y与成员变量x和y产生了歧义

铺垫：关于引用&

引用

“取地址”操作也是&符号，注意根据上下文区分！

- 引用&是类型的一种附加属性，即int型有int型的引用，X型有X型的引用
- 格式：类型名& 引用名 = 变量名，例如X& x = y; 其中y也是X型
  - 必须要初始化（不能直接 X& x;）；y不能是常量，比如说整数1
- 引用就是初始化变量的别名

为什么

  - 即X& x = y后，在之后的程序中x和y随意互相替换也不会影响结果

```
int s = 5;
int &r = s;
s = 3;
cout << r << endl; // 输出 3
r = 4;
cout << s << endl; // 输出 4
```

引用 (cont.)

- 引用一旦定义（并初始化），就不能改变绑定的对象/变量
- 例如

```
int a = 5;
int& b = a;
int c = 10;
b = c; // 此时a = b = c = 10
b = 5; // 不会改变c，且a = b = 5
```

- 即b = c不会被解释成b重新绑定到c，而是把c的值赋给b
  - 因此int &b = a和b = c这两个里面的“=”是完全不同的含义

引用用途举例

如果不加引用，则为参数的复制，
不会影响外界

- 引用作为函数参数：可以直接修改参数在外面的值

而且此处因为引用而不发生复制！

```
void swap(int& x, int& y)
{
    int t = x;
    x = y;
    y = t;
}
```

```
void func(const vector<int>& V)
{
    // ...
}
```

const引用是一种标准用法

- 利用不发生复制的机制：可以高效将含大量数据的对象以极小代价传入
  - 常常搭配const，由此与非引用变量参数上的行为保持一致（不修改外界）
- 引用作为返回值：原理类似，不发生复制，而是直接把变量本身传出去了

要注意返回值的生命周期：
int& f() {return 1;}就是不安全的

在操作符重载时很常见（之后会讨论）

铺垫：关于const

const变量

大原则：const是一种类型修饰（类似于指针的*）

const int a

- 字面意义上的常量，定义时必须初始化成字面值常量或者其他const常量
- 定义后不可进行更改（赋值操作）

甚至const int a =5; a = 5都不行！

- 常见用途：定义常数，例如const double PI = 3.1416
- 那么可以进行“间接”更改吗？比如int *p = &a; *a = 5;

答案是不可以；int *p = &a就会编译报错；我们下面会详细讨论

const与指针

与const int *a写法等价

pointer to const int

- int const *a：一般指针，指向const int变量

const pointer to int

- int *const a：const指针，指向一般int型变量

const pointer to const int

- const int *const a = int const *const a：const指针，指向const型int变量
- 规律：
  - 看*右面是谁，右面是变量那么const修饰的就是int，是const则修饰的是指针
  - 最外层/左侧的int const顺序不重要（两种写法等价）

const int *

const int *a：一般指针，指向const int型变量

- 初值可以是任何int *，而未必是const int t = 5; const int *a = &t这种
- 但不允许*a = XXX这种修改值的操作
- 例如int f = 7, d = 5; const int *a = &f; a = &d; 这是合法的
- 但是如果再调用*a = 10，就是非法的
- 如果有const int f = 7;想要指到f的指针，就需要是const int *a = &f

而不能是int *a = &f

int *const

int *const a：const指针，指向一般int型变量

- a存储的地址不能改，但是可以改a指向地址的内容
- 允许：int f = 5; int *const a = &f; *a = 7;
- 不允许：int *const b = &f; a = b;

即使赋值另一个int *const也不行！

更crazy一些……

双重指针（尽量避免使用）

法则：从右往左看，每个*算一个token；越靠右的越是“外层”的类型

- int **: pointer to pointer to int
- int ** const: a const pointer to a pointer to an int
- int * const *: a pointer to a const pointer to an int
- int const **: a pointer to a pointer to a const int
- int * const * const: a const pointer to a const pointer to an int
- …

const成员函数

- 在成员函数声明的结尾加上关键字const

```
struct Point
{
    int x, y;
    Point(int _x, int _y) : x(_x), y(_y) {}
    int getX() const {
        return x;
    }
    int getY() const {
        return y;
    }
}; //注意这个分号
```

const型的对象只能调用const函数（以及构造、析构函数）

- 作用：告知编译器该成员函数禁止修改成员变量

为什么这么设计？C++是编译语言，必须在编译时确保const的promise达到了

我们为什么要用const?

- “不允许修改”是一种封装上的需要
- 多一些具体限定，多一份安全，避免一些不小心修改的bug
- “好”的用法：const引用，const函数
  - 成员函数能加const就应加const，因为有时其他类库可强制只调用const函数
- “坏”的用法：（多重）const指针

尤其是在协作环境里

引用与指针

- 思考：引用与指针作用的比较；用途？谁更泛用？

- 问题：下列哪个与int &的作用最接近？（提示：引用定义后不可改绑定变量）

A. const int *

B. int *const

C. const int *const

构造函数

构造函数

- 类X的构造函数是一个极为特殊的成员函数
  - 没有返回值，名字必须与类名完全相同
  - 参数表可以自己设计，但编译器不允许有X(X)类型的构造函数
  - 在生成X类型的对象时自动被调用，一旦生成就不可再调用
- 一般用于初始化程序内变量，以及做一些预处理

```
GeoDataHandler() {len = 0; memset(freq, 0, sizeof(freq));}
```

```
struct Point
{
    int x, y;
    Point() {x = 0; y = 0;}
    Point(int _x, int _y) : x(_x), y(_y) {}
}; //注意这个分号
```

可以有多个构造

默认构造函数

- 如果无构造函数，编译器会添加默认构造
- 默认构造是一个不接受参数的函数，且不做任何操作！
- 如果定义了任何构造函数，则默认构造不会被添加进去

```
GeoDataHandler() {len = 0; memset(freq, 0, sizeof(freq));}
```

```
struct Point
{
    int x, y;
    Point() {x = 0; y = 0;}
    Point(int _x, int _y) : x(_x), y(_y) {}
}; //注意这个分号
```

- Good practice：尽量写构造函数，不利用默认构造

成员变量的初始化

- 成员变量如果不在声明时写初始值，则编译器会尝试调用无参数的构造
  - 比如这样是可以过编译的：

```
struct Point {
    int x, y;
};
```

- 但如果该变量没有无参数的构造函数呢？
  - 例如：

```
struct Point {
    int x, y;
    Point(int _x, int _y) : x(_x), y(_y) {}
};
struct PointPair {
    Point A, B;
    PointPair() {}
};
```

此处Point类只有双int的构造，没有无参数构造

这里编译报错：编译器不知道如何构造A和B！

- 规定：要么在定义时给出初始值，要么需要在初始化列表给出初始值（下页）

使用初始化列表对成员变量赋值

Good practice：尽可能采取这种方法来初始化成员变量

另外：
若const或引用型的成员变量在声明时未指定初始值，则也要在初始化列表初始化

- 构造函数的初始化列表的格式

构造函数名(参数表0)：成员1(参数表1), …

该参数表可以利用参数表0的内容

- (C++11) 可用此方法调用本类其他构造!

这行替换成
Point(int x, int y) : x(x), y(y)
也能得到正确结果（这是初始化列表的一个feature），但从可读性上来说不建议使用

对于可以调用无参数构造进行构造的变量：可以但未必要用初始化列表初始化

```
struct Point
{
    int x, y;
    Point(int _x, int _y) : x(_x), y(_y) {}
}; //注意这个分号
struct PointPair {
    Point A, B;
    PointPair(const Point & p) : A(p), B(p) {}
    PointPair() : PointPair(Point(0, 0)) {}
};
```

所有PointPair的构造都必须先用初始化列表初始化A和B

A和B必须在初始化列表初始化，只在函数体内初始化依然编译错误!

可以用这种语法调用本类其他构造函数

构造函数的访问权限

- 一般都把构造函数权限设置成public
  - 如果是private的话，就无法生成任何对象，原因是没有权限调用构造函数！
- 但是private的构造函数也不是完全没用

单例模式 (Singleton Design Pattern)

private构造函数的应用

- 动机：想让某个类只能创建唯一一个对象，并且外界可以获取/使用该对象
  - 概念上类似全局变量，但全局变量不确保唯一性（防止在其他地方创建）
  - 常见用途：日志管理，配置文件管理

一般希望整个程序有统一的日志和配置文件

```
struct Singleton
{
    private:
    Singleton() {}
    static Singleton* instance;

    public:
    static Singleton* getInstance() {
        if (instance == NULL)
            instance = new Singleton();
        return instance;
    }
};

Singleton* Singleton::instance = NULL;
```

如何创建类X的对象

如何调用某个具体的构造函数?

- 直接创建：
  - 调用不含参数的构造：X x; 等价于写成X x = X();
  - 调用某个两参数构造：X x(A, B); 等价于写成X x = X(A, B);
- 指针/new创建：
  - 调用不含参数的构造：X* x = new X;
  - 调用某个两参数构造：X* x = new X(A, B);

不能加括号写成X x()，否则会被当作声明了某个函数

此处也可以加括号写做X* x = new X();

使用成员

- 对象.成员，例如x.f();
- 如果是指针，比如X* x = new X; 则应该先解引用，即(*x).f()
  - 但是C++对此有一个等价的简化的操作符x->f();

复制构造函数

复制构造

- 是一种特殊的构造函数
  - 恰有一个参数，参数的类型为对当前类的引用
- 具体形式可以是X(X&)或者X(const X&)
  - 可两者都定义在同一个类里面
  - 后者相对前者还能处理const型的X对象（前者不可以）

默认复制构造函数

浅复制与深复制

- 当且仅当未给出复制构造，编译器采用默认复制构造
- 默认复制构造会复制类中的每个成员变量
  - 如果遇到非基本类型（主要是其他class），那么会调用对应的复制构造
  - 要小心指针类型的成员变量

```
struct String {
    char *arr;
    String() {arr = new char[100000];}
};
int main() {
    String a;
    String b = a;
    a.arr[0] = 'a';
    cout << b.arr[0] << endl;
    return 0;
}
```

默认情况下只会复制指针存储的地址值，不会复制指针指向的内容

输出'a'

又叫浅复制

深复制：要想复制指针指向的内容，需自定义复制构造

```
struct String {
    char *arr;
    String() {arr = new char[100000];}
    String(const String& s) : String() {
        int i = 0;
        for (i = 0; s.arr[i]; i+
            arr[i] = s.arr[i];
        arr[i] = s.arr[i];
    }
};
```

注意：需要手动调用其他构造确保初始化！

复制构造何时被调用

- 用同样类型的对象初始化时
  - 设instance2是X型
  - 则调用X instance1(instance2)，或者等价的X instance1 = instance2时发生
- 对象作为参数传入某个函数、自动被复制时
  - 例如，设有void f(X x); X x;则调用f(x)时发生
- 对象作为函数返回值返回、自动被复制时

但是：对象间用等号赋值（而不是初始化）时，并不会调用复制构造！

相关：转换构造函数

- 只接受一个参数、且是非复制构造的构造函数，例如X(const int&)

```
struct Point
{
    int x, y;
    Point(int _x, int _y) : x(_x), y(_y) {}
    Point(int _x) : x(_x), y(0) {}
}; //注意这个分号
```

```
int main()
{
    Point p(1);
    return 0;
}
```

析构函数

析构函数

- 析构函数可以理解为“反”构造函数，当整个对象声明周期结束时自动调用恰一次
  - 名字为~类名，无返回值
  - 最多只能定义一个析构函数
  - 用于在对象被删除时进行一些cleanup或者其他相关操作
- 当且仅当未定义析构函数，编译器添加默认析构，默认析构什么也不做
- 尤其当成员变量有指针类型并且new出来时，要主动delete掉防止内存泄漏！

一些没有任何其他指针引用的内存如果不被释放，就会不断累积占用内存，直到耗尽机器内存空间！这对于长期运行的程序尤其有害

关于delete

- 删除一般对象/指针
  - X *x = new X; delete x;
- 删除数组
  - int *a = new int[10]; delete [] a;
- delete NULL没关系
- 不要delete一个未new/初始化过的地址，以及不要重复delete

产生未定义行为；多数时候会crash

定义与声明的分离

定义与声明

- 明确一下术语：
  - 声明：只是给出了函数名、参数表和返回值，而不给具体实现
  - 定义：给出具体实现内容

定义与声明的分离

先给声明

可在任何看得到这个class声明的地方进行定义

- 可以在class里面只写声明，定义之后提供

之后可在构造函数中利用初始化列表进行初始化

- 声明中可不提供const和引用型的初始化（即const int a; int& x;是合法的声明）
- 提供定义时，需要写 类名::函数名 来实现作用域的正确指定

这里全部都是声明，未给出定义

```
class GeoDataHandler
{
    private:
    Point arr[1001]; int len; int freq[100][100];
    bool isin(Point p, int x1, int y1, int x2, int y2);

    public:
    GeoDataHandler();
    void append(int x, int y);
    int query(int x1, int y1, int x2, int y2);
    Point HH();
};
```

推迟提供定义

这个 “类名::” 其实就是指定这个函数对应于类里面的函数；要保证实现可被编译器找到，还需确保函数名称、返回值、参数完全一致

```
void GeoDataHandler::append(int x, int y) {
    arr[len++] = Point(x, y);
}
int GeoDataHandler::query(int x1, int y1, int x2, int y2) {
    int counter = 0;
    for (int i = 0; i < len; i++) {
        if (arr[i].x >= x1 && arr[i].x <= x2 && arr[i].y >= y1 && arr[i].y <= y2) {
                counter++;
                freq[arr[i].x][arr[i].y]++;
        }
    }
    return counter;
}
Point GeoDataHandler::HH() {
    int max = -1; Point res;
    for (int i = 0; i < 100; i++)
        for (int j = 0; j < 100; j++)
            if (max < freq[i][j]) {max = freq[i][j]; res = Point(i, j);}
    return res;
}
```

如果有声明的函数没有给出定义/实现，编译时不会报错，但是链接时会报错！

.h与.cpp的分离

- 一个good practice：写一个类X时，把X的声明写在X.h，X的定义写在X.cpp

```
#ifndef _X_H_ // 所谓的“头文件保护符”，防止被重复include
#define _X_H_

class X {
    // ...
    void func();
    // ...
};

#endif
```

```
// X.cpp的内容

#include "X.h" // 需要这条include来看到X的声明

// ...
void X::func() { // ...
}
//...
```

- 注意头文件保护符！

.h与.cpp的分离 (cont.)

这里用<X.h>也是可以的

- 在别处代码用到类X的时候，只需要 #include “X.h”，不需要include X.cpp
- 但是要注意：编译整个项目的时候要把X.cpp也编译进去，否则会连接时错误

- 在编译器/程序员看来：不需要考虑X具体实现，只根据声明来写程序/检查程序
- 这是团队分工协作的常用方法：大家先约定好所有的.h，之后分头实现.cpp

这种具体实现与声明分开的思想是面向对象的核心思想之一，
在之后的内容中也会再次看到

相关重要问题：如何定义两个互相依赖的类?

问题：比如有Stduent类和Class类，Class要存学生列表，学生要存所属Class

- 具体来说，学生存一个Class对象成员，Class存一个Student对象的数组成员

难点在哪?

- C++只支持前向定义，即Class想用Student对象那么必须Student先定义过
- 但同理，Student想用Class对象也必须先定义Class：这产生无解的循环依赖!

解决方法：两个类都先声明后定义，且对方类型成员都是指针型

```h
#ifndef _CLASS_H_
#define _CLASS_H_

#include <string>

class Student;
struct Class {
    Student* arr[1000]; // student list
    std::string className; int len;
    Class() {className = "cssyb"; len = 0;}
    void addStudent(Student *);
    void printStudentList();
};

#endif
```

要先声明Student类，否则后面的Student*无法解析

“先声明后定义”也不绝对：只要调用对方变量和函数就行

class.h

```h
#ifndef _STUDENT_H_
#define _STUDENT_H_

#include <string>

class Class;
struct Student {
    Class *inWhichClass;
    std::string studentName;
    Student(std::string, Class*);
    void printClass();
};

#endif
```

student.h

```cpp
#include "student.h"
#include "class.h"

int main()
{
    Class * cls = new Class;
    cls->addStudent(new Student("a", cls));
    cls->addStudent(new Student("b", cls));
    cls->printStudentList();
    return 0;
}
```

不需要（也不推荐）include对应的.cpp文件

main.cpp

在定义的.cpp中，要同时include两个.h，从而同时看到两个类的声明

```cpp
#include "student.h"
#include "class.h"
#include <iostream>

void Class::printStudentList() {
    for (int i = 0; i < len; i++)
        std::cout << arr[i]->studentName << std::endl;
}

void Class::addStudent(Student *t) {
    arr[len++] = t;
}
```

class.cpp

```cpp
#include "student.h"
#include "class.h"
#include <iostream>

void Student::printClass() {
    std::cout << inWhichClass->className << std::endl;
}

Student::Student(std::string n, Class *cls) : studentName(n), inWhichClass (cls) {}
```

student.cpp

编译指令（以最简单的g++为例）： g++ main.cpp student.cpp class.cpp

注意student.cpp和class.cpp需要在main.cpp之外额外编译/链接

为何只能用指针来定义Class和Student的对象?

- 对编译器来说，指针已经是完整的类型，不再需要继续关心指向类的内容
- 这避开了提供Student和Class的内容/定义才能完整解析类型
- 对比：Student stu;和Student *stu的区别?

前者要求知道Student完整定义，且（隐式）调用了Student构造函数

为何必须让定义出现在声明后面?

- 定义在后才可以同时看到两个类各自的声明，满足依赖

静态成员

静态成员

- 加static关键字可将成员声明成静态的

```
struct Point
{
    int x, y;
    static int memberCounter;
    static double getDist(Point A, Point B) {
        double dx = A.x - B.x, dy = A.y - B.y;
        return sqrt(dx * dx + dy * dy);
    }
    Point() {x = 0; y = 0;}
    Point(int _x, int _y) : x(_x), y(_y) {}
}; //注意这个分号
```

静态成员变量

静态成员函数

每个对象各自有其他成员变量的copy

- 静态成员变量：只有全局的一份，为所有创建出来的对象共享
- 静态成员函数：不可以访问本类非静态的成员（函数和变量）

静态成员变量的初始化

- 静态成员变量不允许在类里声明的时候初始化，除非是const型

```
struct Point
{
    int x, y;
    static int memberCounter = 0; // 错误，不能在声明处初始化
}; //注意这个分号
```

- 需要在之后进行初始化：int Point::memberCount = 0且不能在函数内写

不加static，但要加 “类名::”

如果采用了类的.h/.cpp分离的方式，那么可以放在.cpp里

- C++这么设计的原因?

静态成员函数

- 静态函数因为不属于任何对象，因此不能用this指针
- 但这并不代表静态函数不能使用/创建对象（甚至可以创建自己类的对象）

```
struct Point
{
    int x, y;
    static Point add(Point A, Point B) {
        return Point(A.x + B.x, A.y + B.y);
    }
    Point() {x = 0; y = 0;}
    Point(int _x, int _y) : x(_x), y(_y) {}
}; //注意这个分号
```

访问静态成员

静态成员不需要创建对象来访问（但对象可以访问静态成员）

- 不创建对象时，可通过 类名::成员名访问
  - 例如 Point::memberCounter
  - Point::getDist(A, B)

```
struct Point
{
    int x, y;
    static int memberCounter;
    static double getDist(Point A, Point B) {
        double dx = A.x - B.x, dy = A.y - B.y;
        return sqrt(dx * dx + dy * dy);
    }
    Point() {x = 0; y = 0;}
    Point(int _x, int _y) : x(_x), y(_y) {}
}; //注意这个分号
```

为何需要静态成员？

小知识：sizeof不计静态成员；为什么？

- 静态变量是“全局”的，可以作为某种通信/统计目的
  - 例如利用静态变量可以进行对象计数，即每次创建对象就counter++
- 静态函数一般用来实现类里面的纯面向过程算法
  - 经常可以看作是“工具”代码，utility

“违反规则”的
mutable和friend

mutable类型变量

mutable是修饰成员变量的，一个mutable成员变量可在const成员函数修改

```
class GeoDataHandler {
    private:
    Point arr[1001]; int len; mutable int freq[100][100];

    public:
    int query(int x1, int y1, int x2, int y2) const
    {
        int counter = 0;
        for (int i = 0; i < len; i++) {
            if (arr[i].x >= x1 && arr[i].x <= x2 && arr[i].y >= y1 && arr[i].y <= y2) {
                    counter++;
                    freq[arr[i].x][arr[i].y]++;
            }
        }
        return counter;
    }
}; //注意这个分号
```

友元friend

- friend可以修饰某个其他类或者（其他类）的函数
- 作用是让其他类及（其他类的）函数可以访问当前类private成员（变量+函数）
- 例子：friend修饰其他类

```
class Point {
    private:
    int x, y;
    Point() {}
    Point(int _x, int _y) : x(_x), y(_y) {}
    friend class Geom;
}; //注意这个分号
```

整个Geom类都可以访问Point的所有private成员

```
class Geom {
    private:
    Point arr[1000]; int len = 0;

    public:
    void addPoint(const Point& t) {arr[len++] = t;}
    int diam() {
        int res = 0;
        for (int i = 0; i < len; i++)
            for (int j = 0; j < len; j++) {
                int dx = arr[i].x - arr[j].x;
                int dy = arr[i].y - arr[j].y;
                res = max(res, dx * dx + dy * dy);
            }
        return res;
    }
};
```

直接访问Point的private成员

友元friend (cont.)

```
class GeoData {
    private:
    string content;
    Point *location;

    public:
    void print();
};

class Point {
    private:
    int x, y;
    Point() {}
    Point(int _x, int _y) : x(_x), y(_y) {}
    void test() {}
    friend void GeoData::print();
}; //注意这个分号

void GeoData::print()
{
    cout << content << endl;
    cout << location->x << " " << location->y << endl;
}
```

只允许GeoData::print访问Point的private成员（而不是整个GeoData类）

因为有friend声明，此处访问合法

互为友元

- 两个类A和B互为对方友元是可以做到的
- 但：对于函数，如何声明A::fa()为B的friend，且B::fb()为A的friend?
  - 问题所在：在声明B时，无法看到A::fa()；同理声明A时也无法看到B::fb()
  - 因此这是不可能（直接）做到的!

慎用

上述的mutable和friend应该谨慎使用

- 都是作为const和private规则的“补丁”存在的
- 用多了会导致const和private丧失存在的意义，与他们本身的意义相矛盾
- 好处是灵活，且有时可显著降低代码量