

对象的生灭：高级篇

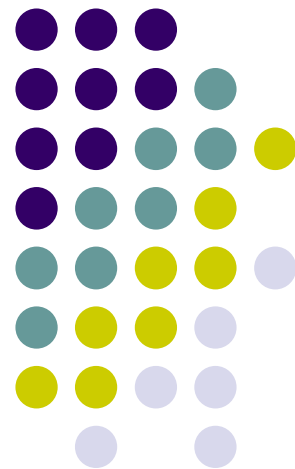
实际讨论：构造函数延伸

1、复制构造函数

2、类型转换的构造函数

吴清锋

最终目标：设计出更好的类



提纲

- 复制构造函数
- 用于类型转换的构造函数

引言：复制（拷贝）构造函数

- 回顾：标准库类型

```
string str1("Hello!"); //构造函数, "Hello!"是实参
string str2(str1); //也是初始化, 有构造函数重载现象, 特殊: 用一个对象对另外一个对象初始化。这是特殊的构造函数:复制构造函数
string str3=str1;
```

- 回顾：之前的代码

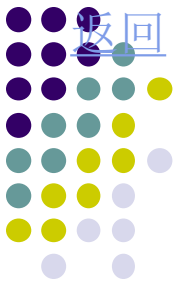
```
class Date {
private:
    int year;
    int month;
    int day;
public:
    ...
}
```

```
Date d1(2016,03,21);
Date d2(d1);
Date d3(d1);
```

- 共性特征：实现对象的克隆，即用一个已有的对象为依据，快速地复制出多个完全相同的同类对象；

类名 对象2(对象1);

- 代码实质：在创建一个新对象时，用另一个同类的对象对其进行初始化，调用特殊的构造函数——拷贝构造函数

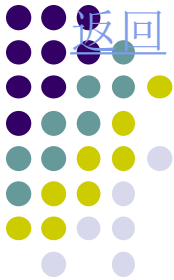


复制构造函数机制的探讨

- 下列语句中：

Date d3(d1);

- 是建立对象的语句，实参是一对象；
- 由于在括号内给定的实参是对象，因此将调用复制构造函数，而不会调用其他构造函数；
- 复制构造函数也是构造函数，是普通构造函数的重载
函数名都是类名
- 复制构造函数的功能：根据实参对象**创造**新的对象,并将实参对象的各数据成员值一一**赋给**新的对象中对应的数据成员；



复制构造函数分类

- 默认复制构造函数
- 自定义复制构造函数

默认复制构造函数

- 如果用户自己未定义复制构造函数，则编译系统会自动提供一个默认的复制构造函数，该函数有参数，函数体中有语句。功能较为简单：复制类中每个数据成员；
- 当对象本体与对象实体不一致时（成员中包含指针关系），便需要自定义拷贝构造函数；
 - 例子1
存在的问题分析
 - 例子2 目标

自定义复制构造函数

● 自定义复制构造函数

- 一旦自定义了拷贝构造函数，默认的拷贝构造函数也就不再起作用；
- 拷贝构造函数原型，其中它有一个参数，这个参数是本类的对象，而且采用对象的**常量引用**形式

<类名> (const <类名>&);

为何用：const

可以省略，有const是为了防止在函数中修改实参对象

为何用：引用&

若不用引用，则会在参数传递上，不停地调用拷贝构造函数
很多编译器会报错

● 在自定义复制构造函数时，往往要完成操作：

- 1、创建空间，并且数据成员指向该空间
- 2、赋值操作

● 若数据成员中有指针，有深拷贝时，不可忽视的析构函数

复制构造函数的应用情形（1）

- 发生以下**三种情况**复制构造函数被调用：

- 用类的对象去初始化该类的另一个对象时；

```
Cat cat1;
```

```
Cat cat2(cat1); //创建cat2时系统自动调用拷贝构造函数，用cat1初始化cat2。
```

注意：用类的对象去初始化另一个对象时的另外一种形式

```
Cat cat2=cat1;
```

```
// 注意并非 Cat cat1, cat2; cat2=cat1;
```

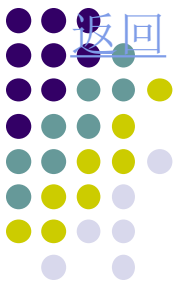
- **对象作为函数的实参传递给函数的形参时；**

传递过程就是将已有对象拷贝到一个新建的形参对象

```
f(Cat a) { } // 定义f函数，形参为cat类对象
```

```
Cat b; // 定义对象b
```

```
f(b); //进行f函数调用时，系统自动调用拷贝构造函数
```

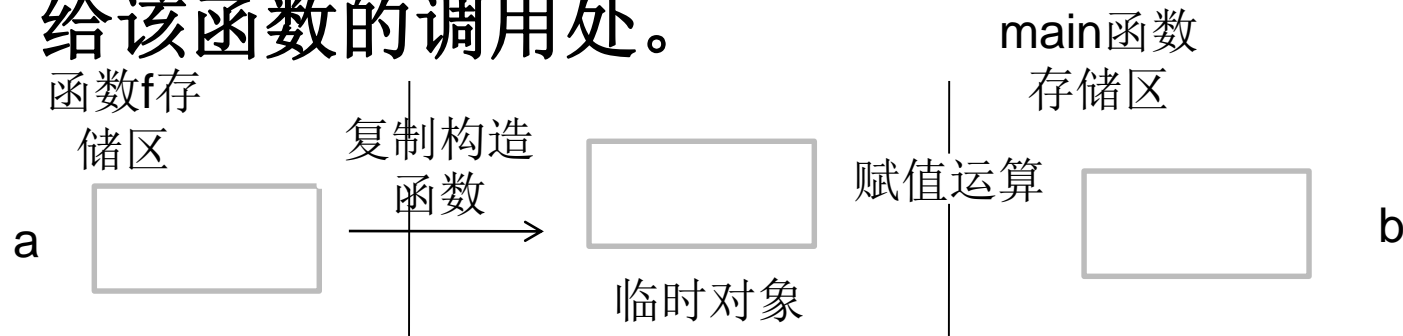
复制构造函数的应用情形（2）

- 发生以下三种情况复制构造函数被调用：
如果函数的返回值是类的对象，函数调用返回时，调用拷贝构造函数。

```
cat f() // 定义f函数，函数的返回值为cat类的对象
{ cat a;
...
return a;
}
```

cat b; // 定义对象b
b=f(); // 调用函数f，系统自动调用拷贝构造函数，将对象a复制到新创建的临时对象中

- 在函数调用完毕将返回值带回函数调用处时。此时，需要将函数中的对象复制一个临时对象并传给该函数的调用处。



比较：初始化与赋值

- 初始化与赋值有本质区别（赋值出现在两个对象都已经存在，而初始化出现在创建对象时）

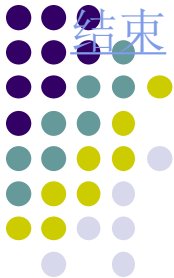
```
class Date {
    private:
        int year;
        int month;
        int day;
    public:
        ...
}
```

初始化发生在对象创建时

```
Date d1(2016,03,21);
Date d2(d1);
Date d3=d1;
```

赋值时，两个对象都已经存在

```
Date d1(2016,03,21);
d2=d1
```




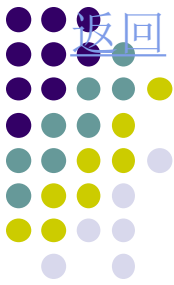
提纲

- 复制构造函数
- 用于类型转换的构造函数

背景:



- 问题的引出: 一个运算中可能涉及多种数据类型, 则在进行运算之前, 经常需要将一种类型转换成另外一种类型:
 - 内置标准数据类型间转换:
 - 隐式转换: 如, 5/8和5.0/8
 - 显式转换
- 现在要讨论的是: **基本类型**  **类类型**
 - 如何实现**其他类型向用户自定义类类型**之间的转换?
必须由程序员通过编程来实现类型的转换, 即定义含一个参数的构造函数. 编译器可以使用这种构造函数把参数的类型**隐式**转换为类类型
 - 类型转换构造函数, 其功能是将一个其他**类型的数据**转换成一个**指定的对象**;



用于类型转换的构造函数

- 转换构造函数只有一个形参，用户根据需要在函数体中指导编译器如何进行转换；

- 例子

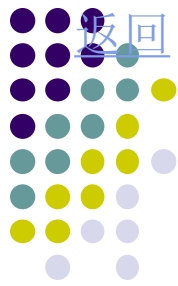


- 代码中的重点讨论：

`box1.compareVolume(50.0)`

`box1.compareVolume(Box(50.0))`

实际是通过构造函数创建一个无名对象，将无名对象复制给函数形参后，系统自动调用析构函数，释放资源



抑制或显示：用于类型转换的构造函数

● 双刃剑

- 抑制由构造函数定义的隐式转化

可以通过将构造函数声明为explicit。例子：

```
class Sales_item {  
    public:
```

```
        explicit Sales_item(const string &book=“ “);  
};
```

注意：explicit关键字只能用于类内部的构造函数声明上，在类的定义体外部所做的定义上不再重复它；

- 为转换而显式地使用构造函数

例子：

```
item.same_isbn(Sales_item(null_book));
```

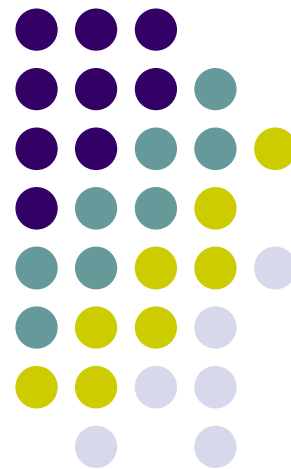
值得一提的是，与例子中隐式调用构造函数不同，在这里构造函数是显式的！显式使用构造函数只是中止了隐式地使用构造函数。任何构造函数都可以用来显式地创建临时对象！

对象与类的进一步讨论

从类设计时的数据成员和
成员函数出发

- 1、`const`
- 2、`static`
- 3、友元

吴清锋
2021年春



提纲

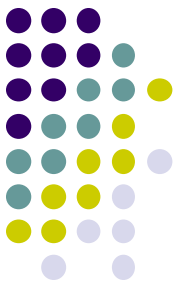
- 常成员：共享数据的保护
- 友元：提高对象私有数据成员的访问效率
- `static`数据成员

const修饰符



- const变量
- const指针变量
- const对象：常对象
- const与迭代器
- 常引用
- 常成员函数
- 常数据成员

共享数据的保护【用const来平衡】



- 数据的安全性与数据的共享性二者是矛盾的；
- 安全性，往往通过private或protected方式，只允许类成员访问，类外不能直接访问；
- 共享性，例如：**实参和形参、变量与其引用、数据和它对应的指针**，可以在不同场合、通过不同途径访问同一个数据。但这有很大风险：破坏数据的安全，因为无法保证在何时、何地数据被修改。
- 期望：在实际编程中，对于某些需要共享，又需要防止随意被改变的数据，可定义为**const**量



const变量

- 格式：const 类型 变量标识符=初始值;
- 例子：const double PI=**3.14**;
- 说明：用**const**定义的常变量，一定要对其初始化。而在说明时进行初始化是對此变量置值的唯一方法。即：程序中无法使用 赋值运算符对这种常变量进行赋值。

```
const double PI=3.14;  
PI=3.1415; //编译无法通过
```

常对象



- 格式: `const <类名> <对象名> (初始化);`

```
#include <iostream.h>
```

```
class Sample
```

```
{
```

```
    int n;
```

```
public:
```

```
    Sample(int i){ n=i;};
```

```
    void setValue(int i){ n=i;};
```

```
    void display(){ cout<<"n="<<n<<endl;}
```

```
};
```

```
void main()
```

```
{
```

```
    const Sample a(10);
```

```
    a.setValue(6); //error C2662: 'setValue' : cannot convert
```

```
'this' pointer from 'const class Sample' to 'class Sample &'
```

```
    Conversion loses qualifiers
```

```
    a.display();
```

常对象的成员无法修改!

常对象特性



- 常对象必须进行初始化;
什么是初始化? 在创建对象的同时, 对象有值
- 常对象的数据成员 (无论是公有的或是私有的) 的值在对象的整个生存期内不能被改变;
- 由于无法预料哪些成员函数会改变数据成员的值, 因此规定: 不能通过常对象调用普通的成员函数; 【反思: **对于编程的贡献与思考**】

const与指针



const对指针值或指针本身进行限制

- 1、声明指向常量的指针，即，指针本身可以改变，但不能通过指针来改变所指对象的值

```
const char *name1= “Zhang” ;
```

```
char s[ ]=“Wang”; //char s[ ]是一字符数组
```

```
name1=s;    //正确，name1本身值能改变
```

```
*name1='L'; //错误，不能通过name1改变所指的内容
```

内存的示意图

const与指针



const对指针值或指针本身进行限制

2、声明常指针，即，指针本身值不能改变

```
char * const name1= “Zhang” ;  
name1= “Wang” ; //错误
```

const_iterator (1)

- 使用const_iterator类型定义的迭代器，(迭代器)自身值是可以改变的，但是不能通过该迭代器来改变其所指向的元素的值，只能用于读取容器内元素；

- 例子

```
for (vector<string>::const_iterator iter=text.begin();  
      iter!=text.end();++iter)
```

```
    *iter="Hello"; //error
```

```
for (vector<string>::const_iterator iter=text.begin();  
      iter!=text.end();++iter)
```

```
    cout<<*iter; //借助iter读取元素值,而不改变值
```


const_iterator (2)

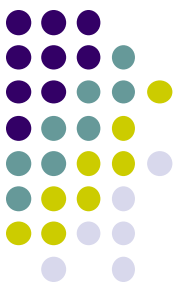
- const_iterator对象与const的iterator对象比较
 - const_iterator对象，它的侧重点在于描述对所指的元素只具有读，而不能具有修改；
 - 定义一个const的迭代器时，必须初始化迭代器；且在运行过程中无法改变(迭代器本身)值；而对于它所指向的元素是否修改没有限制。例子：

```
vector<int> nums(10);
```

```
const vector<int>::iterator cit=nums.begin();
```

```
*cit=1;    //OK,对指向元素的修改
```

```
++cit;     //Error.cit定义时有const修饰符
```



常引用

- 格式： `const 类型 &引用名=初始值;`
- 例子： `int a=5;`
`const int &ref=a;`
 - 说明：定义了常引用，不能通过该引用修改它所引用的变量（对象）
- 一般用：常引用做形参，在函数中不能更新引用所绑定的对象，便不会意外地发生对实参的更改（即：只能访问）
 - 例子：复制构造函数中的常引用
`<类名> (const <类名>&);`

常成员函数声明



- 格式:

<返回类型> <成员函数名>(参数列表) **const**;

```
class Sample
{
    int n;
public:
    Sample(int i){ n=i;};
    void print() const;
};

void main()
{
    const Sample a(10);
    a.print();
}

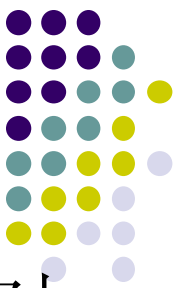
void Sample::print() const
{
    cout<<"2:n="<<n<<endl;
}
```

形式: **const**是函数类型的一个组成部分，因此在函数声明和实现部分中都要带有**const**关键字；若定义常成员函数时丢失了**const**关键字，程序会产生错误。

例子：常成员函数



```
1. class Date {  
2.     public:  
3.         Date(int y=2003,int m=1,int d=1):year(y),month(m),day(d) { }  
4.         int GetYear() const { return year; }  
5.         int Getmonth() const { return month;}  
6.         int GetDay() const;  
7.     private:  
8.         int year,month,day;  
9. };  
  
    inline int Date::GetDay( ) const {  
        return day;  
    }
```



常成员函数特性

```
#include <iostream.h>
class Sample
{
    int n;
public:
    Sample(int i){ n=i;};
    void print() const;
    void setValue(){ n =100;}
};

void main()
{
    const Sample a(10);
    a.print();
}

void Sample::print() const
{
    n=200;
    setValue();

    cout<<"2:n="<<n<<endl;
}
```

- 常成员函数不能修改更新对象的数据成员
- 常成员函数只能调用const的常成员函数

场景： 用于读取数据成员或是可能出现常对象的情况

```
//error C2166: l-value specifies const object
//error C2662: 'setValue' : cannot convert 'this' pointer
//from 'const class Sample' to 'class Sample &'
```



const成员函数可重载

```
#include <iostream.h>
```

```
class Sample
```

```
{
```

```
    int n;
```

```
public:
```

```
    Sample(int i){ n=i;;}
```

```
    void print() { cout<<"1:n="<<n<<endl;}
```

```
    void print() const{ cout<<"2:n="<<n<<endl;}
```

```
};
```

```
void main()
```

```
{
```

```
    const Sample a(10);
```

```
    Sample b(20);
```

```
    a.print();
```

```
    b.pirnt();
```

```
}
```

- **const**关键字可以用于参与对重载函数的区分。原则是：常对象调用常成员函数，一般对象调用一般成员函数

常对象

- 常对象的常成员
- 例子
- 总结

```
#include <iostream.h>
class Sample
{
    int n;
public:
    Sample(int i){ n=i;};
    void print() { cout<<"1:n="<<n<<endl;};
};

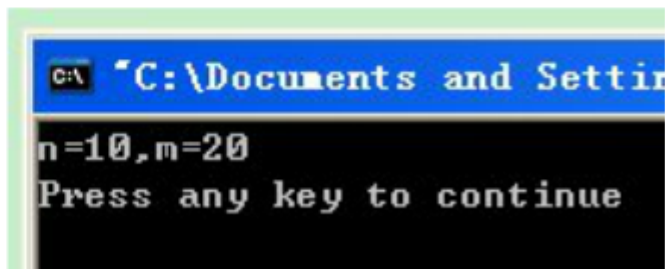
void main()
{
    const Sample a(10);
    a.print(); //error C2662: 'print' : cannot convert 'this' pointer
               //from 'const class Sample' to 'class Sample &'
               //Conversion loses qualifiers
}
```

| 类型 | 普通数据成员 | 常数据成员 | 常对象 |
|--------|-------------|------------|----------|
| 普通成员函数 | 可以访问，也可以修改值 | 可以访问，但不能修改 | 不允许 |
| 常成员函数 | 可访问，不能修改 | 可访问，不能修改 | 可访问，不能修改 |

const数据成员

- 常数据成员必须进行初始化，并且不能被更新
- 常数据成员的初始化只能通过构造函数的成员初始化列表显式进行。

```
#include <iostream.h>
class Sample
{
    const int n,m;
public:
    Sample(int i,int j): n(i),m(j){}
    void print()
    {
        cout<<"n="<<n<<","m="<<m<<endl;
    }
};
void main()
{
    Sample a(10,20);
    a.print();
}
```

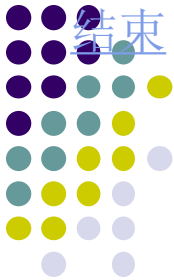


```
C:\ Documents and Settings
n=10,m=20
Press any key to continue
```




值得思考的问题：

- 现在仅仅是：从语法视角描述有const
- 更为关键的是，如何从特性出发反思：
 - 问题域的什么情景，你会主动选择考虑使用



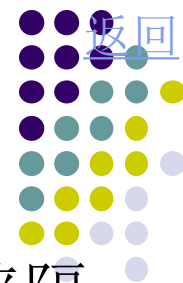
提纲

- 常成员：共享数据的保护
- 友元：提高对象私有数据成员的访问效率
- `static`数据成员

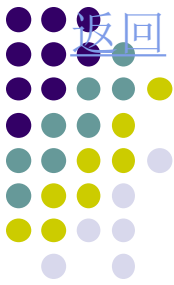
背景：封装与高效访问之间的平衡

- 类的封装性，一般将数据成员声明为私有成员，在外不能直接访问，只能通过类的公有成员函数对私有成员进行访问。有时，操作上，需要频繁调用成员函数来访问私有成员，这就存在系统开销。
- 从高效的角度，**提供友元机制**，使被声明为友元的全局函数或是其他类可以直接访问当前类中的私有成员，又不改变其私有成员的访问权限。
- **【重要的应用场景】友元和运算符重载有密切联系；两个类要共享数据的时。**

友元概述



- 封装性和信息隐藏使对象和外界以一堵不透明的墙隔开，友元在这堵墙上开了一个小孔；
- 友元机制：类外的一般函数或是另一个类的成员函数能够对该类体中的私有成员和其他成员能直接访问；
- 友元是以牺牲信息隐藏原则，削弱封装性来提高编程效率，增大了类与类之间的耦合度；



友元的种类

- 友元的种类:

- 一般函数：类外的函数

- 另一个类的成员函数

友元函数

- 另一个类：其他类全部的成员函数

友元类

- 特点：都是在本类外

友元不是被访问类的成员，所以它不受类的访问权限影响。

概述：类的友元函数

- 在本类以外的其他地方定义了一个函数（可以是不属于任何类的一般函数，也可以是其他类的成员函数），在对本类进行声明时在类体中用friend对该函数进行声明，此函数就成为本类的友元函数。
- 友元函数说明格式：
friend <类型> <函数名>(<参数列表>)
- 特性：可以访问与其有好友关系的类中的私有成员；

类的友元函数

弱弱地问：为何不直接写成成员函数就好了？难道就要怎么折腾？

有没有可能，对于类而言，只能表示成一般函数？

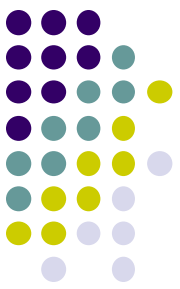
- 根据函数的定义位置：
 - 类别1：将普通函数声明为友元函数
 - [例子1](#) 【注意：一般函数需要在本类中声明为友元】
 - [代码分析](#)
 - 类别2：友元成员函数，该函数可以是另一个类中的成员函数
 - [例子2](#)
 - [代码分析](#)
- 一个函数（普通函数或成员函数）可以被多个类声明为友元，因此可以引用多个类中的私有数据

友元类概述

- 可以将一个类声明为另一个类的友元，即友元类。
- 特性：若有**B**类是**A**类的友元类,友元类**B**中所有函数都是**A**类的友元函数,可以访问**A**类中的所有成员；
- 如何声明

```
friend class 类名;  
例: class Screen {  
    friend class Window_Mgr;  
    ...      //Screen类的其他内容  
            };      //谁能访问谁的私有成员?
```


例子：友元类

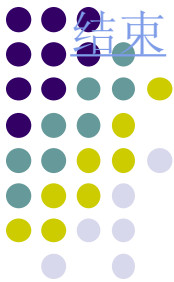


- class DATE {
- public:
- DATE(int y=2003,int m=1,int d=1):year(y),month(m),day(d) { }
- **void DateTime(const TIME &t);**
- private:
- int year,month,day;
- };
- class Time {
- **friend DATE;**
- public:
- TIME(int h=0,int m=0,int s=0):hour(h),mintue(m),second(s){ }
- private:
- int hour,mintue,second;
- };

```
void DATE::DateTime(const Time &t) {  
    cout<<year<<month<<day<<endl;  
    cout<<t.hour<<t.minute<<t.second<<endl;  
}
```

友元类特性

- 友元的关系是单向的；
- 友元的关系不能传递；
B类是A类的友元，C类是B类的友元，那么：
C类和A类之间，若没有声明，就没有任何友元关系
- 友元的利弊分析：
 - 弊：友元可以访问其他类中的私有成员，是对封装性原则的破坏；
 - 利：实现数据共享，提高程序的效率；



提纲

- 常成员：共享数据的保护
- 友元：提高对象私有数据成员的访问效率
- **static**数据成员

定义

初始化

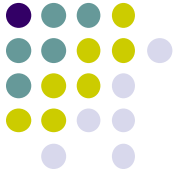
应用



记忆中的static

- 变量的修饰符
 - auto
 - static : 初始化一次; 定义在特殊位置; 可保留值
 - register
 - extern

static 声明局部变量



```
1. #include <stdio.h>
2. void main( ) {
3.     int f(int);
4.     int a=2,i;
5.     for (i=0;i<3;i++) //通过循环构造出多次调用
6.         printf("%d",f(a));
7. }
8. int f(int a) {
9.     auto int b=0;    static int c=3;
10.    b=b+1;           c=c+1;
11.    return(a+b+c);
12. }
```

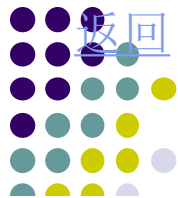
问题的导引

- 问题的引出：某个类，若有 n 个对象，那么每一个对象都分别有自己的数据成员，不同对象的数据成员各自有值，即：对象的属性值是不同的。

但，有时希望有某一个或几个数据成员为所有对象所共有，以便实现数据共享【类中共享】。

- 回顾下，数据共享的方式：
 - 全局变量，能够实现数据共享。但是，由于各个函数都可以自由修改全局变量的值，使得安全性得不到保证。
 - 类似，是否可以使用全局对象？可以，但是也不安全！

static类成员——概述



- C++ 提供static关键字来声明静态成员，生成“**类属性**”，即：该属性为整个类所共有，不属于任何一个具体对象 **【以此实现共享】**；
- 静态成员在每个类中只有一个副本，由该类的所有对象共同维护和使用；
- 使用**静态成员**，来解决同一个类的不同对象之间的数据与函数共享问题，是类的所有对象共享的成员，而不是某个对象的成员，分为：
 - 静态数据成员
 - 静态成员函数

静态数据成员定义

- 静态数据成员是一种特殊的数据成员，它以关键字**static**开头；
- 语句格式：

```
class 类名 {
    ...
    static 类型说明符 成员名;
    ... };
```

- 定义**static**数据成员，能够实现类的所有对象在类的范围内共享某个数据。一般用于表示：
 - 1、统计计数器,如:程序的任意一点总共创建多少类对象
 - 2、同类所有对象某个数据成员值相同，如储户类利率

静态数据成员的特性

- 类的**静态数据成员**视为**该类类型的全局变量**
 - 静态数据成员在**每个类对象中并不占有**存储空间，是对**每个类**分配有存储空间；
 - 静态数据成员只有一份，它为各对象所共有【不是独立拥有】，并不只属于某个对象，所有对象都可以引用它，实现同类中不同对象之间的数据共享；
 - **在定义对象之前，就有静态数据成员；**
 - 对于所有对象而言，静态数据成员的值是一样的；
- 对于**非静态数据成员**，每个类对象中都有**自己的副本**；

静态数据成员初始化

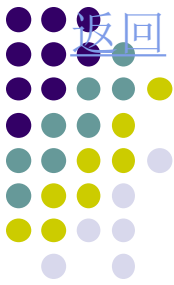
- 在类的声明中仅是对静态数据成员进行声明，还需要对静态数据成员进行定义和初始化。形式：
数据类型 类名::静态数据成员名=初值;
- 注意：
 - 初始化不能在类声明或是类的构造函数中进行【相对对象，静态数据成员是独立的】，**只能在类体外进行**
 - 此时不再写**static**关键字，需要使用域运算符::
- 若未对静态数据成员赋初值,则自动赋予初值（根据类型不同，值也不同）
- 静态数据成员是在所有对象之外单独开辟空间。只要在类中定义了静态数据成员，**即使不定义对象，也为静态数据成员分配空间，可以被引用**

静态数据成员应用

- 如何**使用**静态数据成员？（**具有访问属性限制**）
 - 在类的**public**部分说明的静态数据成员
通过对象名或是类名，即：
对象名.静态数据成员 **或** 类名::静态数据成员
这可传达：静态数据成员并不是属于或依赖某个对象,而是属于类的，但同类的对象可以共享引用它。
 - 在类的非**public**部分说明的静态数据成员
只能由类的成员函数访问 **//与全局变量相比的优势！**
- 例子
- 通过静态数据成员，实现数据共享，因此可以不使用全局变量。全局变量破坏了封装的原则！
思考：公用静态数据成员与全局变量的差异
静态数据成员的作用域只限定于定义该类的作用域内。

静态成员函数定义

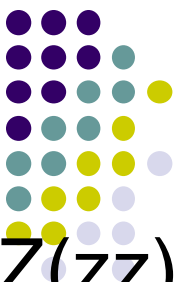
- 在类中声明函数的前面加**static**就成了静态成员函数.
- 语句格式: `class 类名 {`
`...`
`static 类型 函数名(形参) { 函数体 }`
`... };`
- 若是在类外定义**static**成员函数, 则无须重复指定**static**关键字;
- **static**成员函数不能被声明为**const**;



静态成员函数特性和应用

- static成员函数特性：
 - 是类的一部分，而不是对象的一部分。
 - 它可以直接访问所属类的static成员，但由于没有this形参，不能直接使用非static成员（原因：由于调用时可不捆绑具体对象，被调用时，无当前对象信息。若要访问非静态成员时【访问对象的私有空间】，必须通过参数传递的方式得到相应的对象，再通过对象来访问。）；
- 静态成员函数的使用：
 - 在类外调用静态成员函数
 - 类名::静态成员函数 或 对象名.静态成员函数
- 例子

例子



```
1. class Point {  
2.     public:  
3.         Point(int xx=0,int yy=0,int zz=0):X(xx),Y(yy),Z(zz)  
4.             { count++; } //用来统计生成的当前对象个数  
5.         static void Display(Point & p);  
6.     private:  
7.         int X,Y,Z;  
8.         static int count;  
9. };  
10. void Point::Display(Point &p) {  
11.     cout<<p.X<<p.Y<<p.Z<<endl; }  
12. int Point::count=0;
```

```
Point p1(1,2,3),p2(4,5,6);  
Point::Display(p1);  
Point::Display(p2);
```

如何凸显静态数据成员特性？

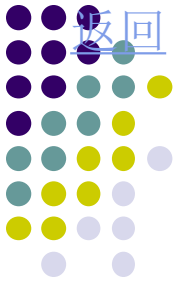
静态成员函数特性

- 引进静态成员函数的**核心**目的，不是为了对象之间的沟通（不是访问某对象），而是**访问类中的静态数据成员**；
- 静态数据成员与静态成员函数的关系：
 - 对于**static**的数据成员不是非要**static**的成员函数才能访问，使用普通的成员函数也是能访问的；**只是使用静态成员函数去访问是更合适的！**
 - 如果想让静态成员函数也能访问对象中的私有数据，则需要向静态成员函数传递一个对象【因为无**this**指针】；



建议大家去总结

- 类的特殊成员
 - 静态成员
 - const**成员
 - 内联函数
 - 友元
- 去分析：
 - 产生背景
 - 如何定义或声明
 - 如何使用

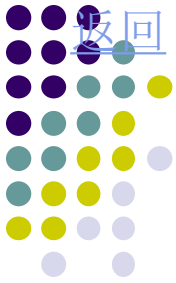


例子：构造函数创建实体

- 代码

```
class Person {  
    private:  
        char* pName;  
    public:  
        Person(char *pN="noName") { //构造函数  
            cout<<"构造中  " <<pN<<"\n";  
            pName=new char[strlen(pN)+1];  
            if (pName) strcpy(pName,pN);  
            }...  
};
```

```
Person  p1("June"); //初始化操作
```

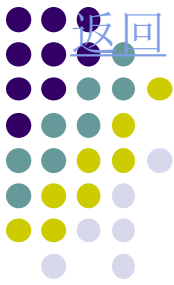


例子：对象的赋值

- 代码

```
class Date {  
    private:  
        int year,month,day;  
    public:  
        Date():year(2000),month(1),day(1) { };  
        Date(int year_,int month_,int day_)  
            { year=year_;month=month_;day=day_;}  
        void prin() {cout<<year<<month<<day;}  
}; ....
```

```
Date d1(2009,4,1);  
Date d2;  
d2.prin();  
d2=d1;  
d2.prin();
```

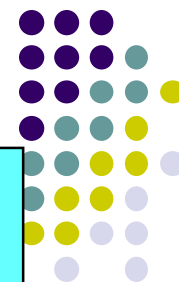


例子：复制构造函数创建实体

- 代码

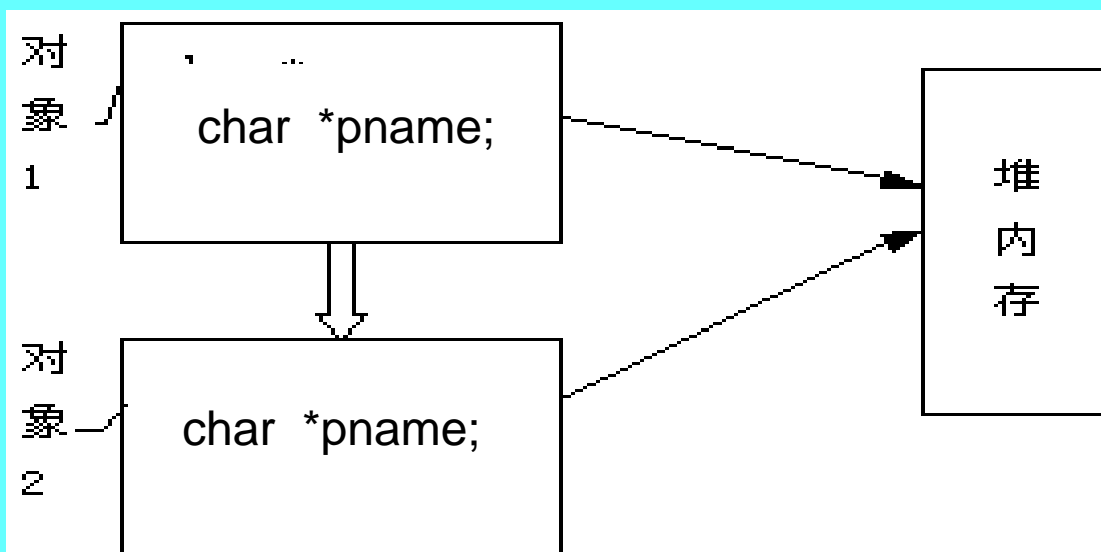
```
class Person {  
    private:  
        char* pName;  
    public:  
        Person(char *pN="noName") {  
            cout<<"构造中  " <<pN<<"\n";  
            pName=new char[strlen(pN)+1]; //在堆区创建  
            if (pName) strcpy(pName,pN);  
        }...  
}; ...  
Person p1("John"); //使用构造函数来创建p1  
  
Person p2(p1); //生成对象p2,调用默认的复制构造函数  
              //数，仅拷贝了对象本体！
```

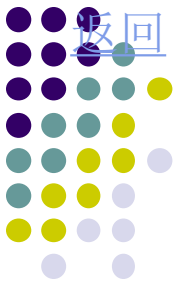
存在的不足



Person p2(p1);

实际是将p1中pname的值（地址）复制一份给p2中pname，两个对象指向同一个空间





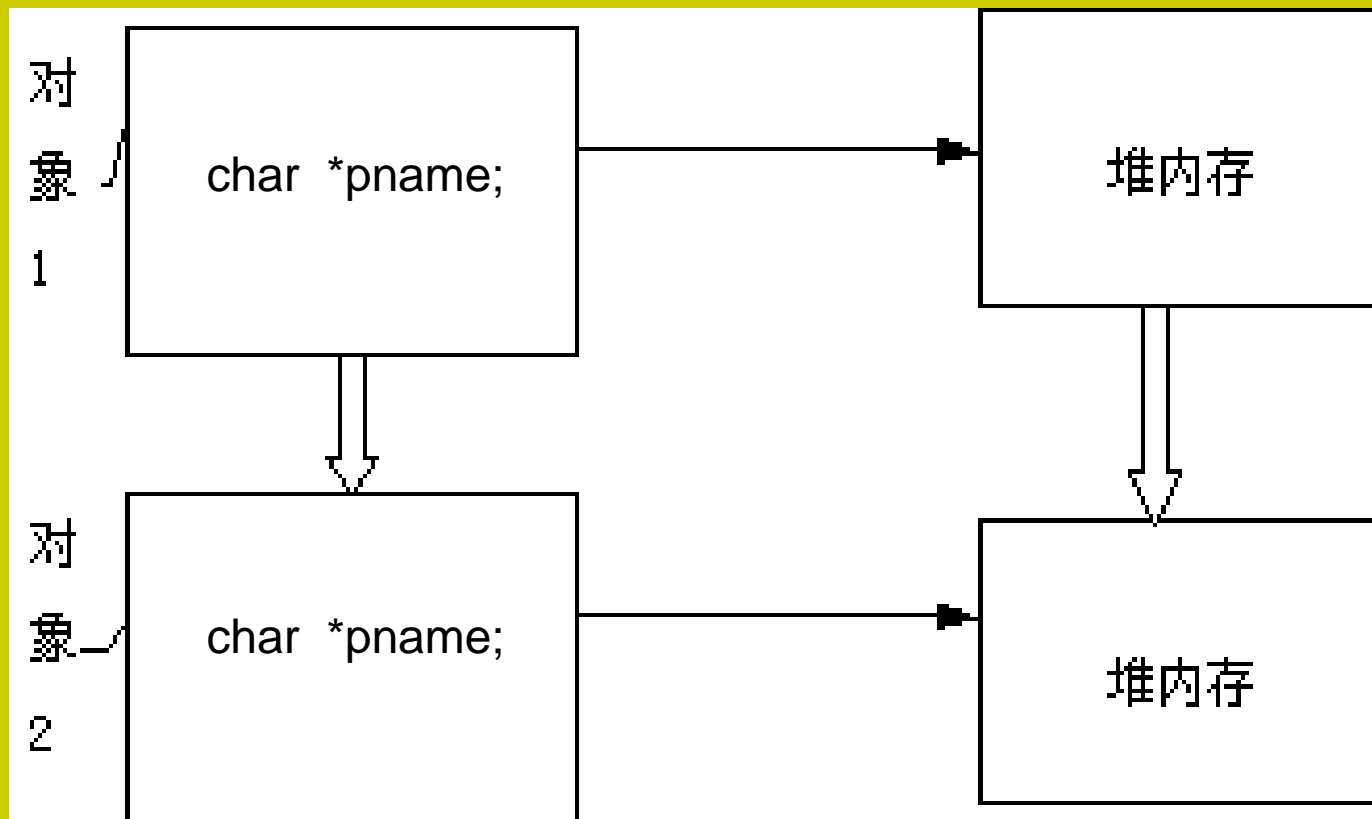
例子：自定义复制构造函数

- 代码

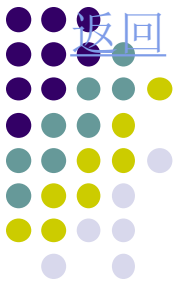
```
class Person {  
    private:  
        char* pName;  
    public:  
        Person(char *pN="noName") {  
            cout<<"构造中  " <<pN<<"\n";  
            pName=new char[strlen(pN)+1];  
            if (pName) strcpy(pName,pN);  
        }  
        Person(const Person &s) { //自定义复制构造函数  
            cout<<"复制构造函数  " <<s.pName<<"\n";  
            pName=new char[strlen(s.pName)+1];  
            if (pName) strcpy(pName,s.pName);  
        } ...  
};
```

Person P1("Zhang");

Person P2(P1);



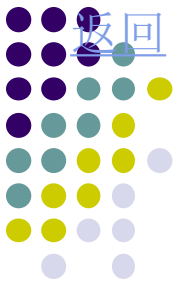
深拷贝



例子：普通函数声明为友元函数

若干关键点分析：

- **display** 是一个在类外定义的且未使用类 **Time** 作域限定的函数，它是非成员函数，不属于任何类；
- 若未在 **Time** 类的定义体中未声明 **display** 函数为 **friend**，它是无法引用 **Time** 的私有成员；
- 注意在引用私有数据成员时，必须加上对象名，因为 **display** 函数并不是 **Time** 类的成员函数，无法默认使用 **Time** 类的数据成员，必须指定要访问的对象。



例子：友元成员函数

若干关键点分析：

- 前向声明

- 允许“前向声明”，即在正式声明一个类之前，先声明一个类名，表示此类将在稍后声明；前向声明，只包含类名，不包含类体；
- 即使有了前向声明，还是必须只有在正式声明后，才可以定义类对象；
- 思考，是否可以使用提前引用声明来定义对象的引用呢？可以，因为引用并没有具体的空间！

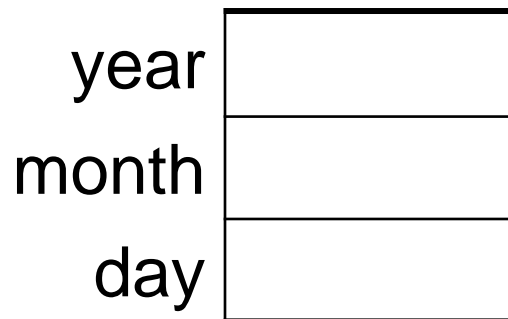
- 友元成员函数

- 一般情况下，两个不同的类是互不相干的。在本例中，由于在**Date**类中声明了**Time**类中的**display**成员函数是**Date**类的“朋友”，因此该函数可以引用**Date**类中所有的数据。

对象本体与实体（1）

- 声明了类，定义了对对象，在内存中就为开辟了一块对象空间。例：

```
class Date{  
    private:  
        int year,month,day;  
    public:  
        ... };
```



- 对象空间中一个复杂的情景：成员中包含指针关系

对象本体与实体（2）

- 对象空间中一个复杂的情景

```

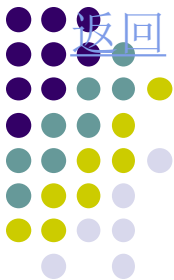
class Student {
    private:
        string name;
        int age;
        bool sex;
        char* resume;
        Person* father;
        Person* mother;
    public:
        .. };
    
```

例子：类表示学生的个人信息，该信息中可能包含长度无法确定的个人简历、父母亲个人信息等成员,具体：

- 由于个人简历长度不确定，用指针才能充分利用存储空间；
- 父母亲信息本身也是一个完整的对象,考虑使用对象的指针.

- 例子

此时，就不仅仅是对象本体的创建，而是需要依赖于构造函数，在函数体中对整个对象实体进行充分创建。即，需要分配动态内存空间给对象中的指针成员



对象本体与实体不一致下的问题

- 析构函数 ~
 - 用户自定义析构函数
- 赋值运算 =
 - 对于深拷贝，需要对运算符进行重载
- 对象复制
 - 对于深拷贝，需要自定义拷贝构造函数

对象的赋值

- 对象的赋值

- 如果对一个类已经定义了两个或多个对象，则在这些同类的对象间可以互相赋值，即将对象中所有数据成员的值赋给另一个同类的对象；
- 对象之间的赋值通过“=”来实现；

- 例子

- 注意：

- 在缺省情况下，“=”执行的是对象成员之间的拷贝
- 但是，默认的赋值操作符只管本体的复制。若要实现对象之间的深拷贝，则必须自己定义赋值操作符；

私有和公有

- 类中public和private成员的特性，在类之外可以访问公用成员，且只有本类中的函数可以访问本类的私有成员.

```
class Time {  
    private: //实现数据的封装，达到数据保护目的  
        int hour; int minute; int second;  
    public: //此处省略好多字  
        void prin() { cout<<hour<<minute<<second;  
        }  
    Time t;  
    t.prin();  
    cout<<t.hour; //?
```

类成员对象下的复制构造函数调用

