# 创建与销毁·二

# 创建与销毁·二

By 曹菡雯（计 03）、赵晨阳（软 01）、罗华坤（化 93）、李晨宇（材 01）

readme

这一部分主要是 L5-创建与销毁·二的课堂笔记整理，由小组四名同学共同完成。在课件的基础上，我们尽力做到了对于每一页 PPT 都有完整的解读，同时联系了前后课程的内容与课程作业，对于课件中的操作也有一定的扩展。

最近一次更新：将原文档中所有代码重新进行了编写，有助于阅读。

如果阅读时间不够充足，建议阅读课堂的扩展部分。



# 一、变量与静态变量

## 1.1 四类变量的区别

### 1.1.1 变量按存储区域分

全局变量、静态全局变量和静态局部变量都存放在内存的静态存储区域，局部变量存放在内存的栈区。静态储存区在函数结束后不会销毁，而栈区在函数结束后会退栈而销毁。

### 1.1.2 变量按作用域分

#### 1.1.2.1 全局变量

在整个工程文件内都有效；“在函数外定义的变量”，即从定义变量的位置到本源文件结束都有效。由于同一文件中的所有函数都能引用全局变量的值，因此如果在一个函数中改变了全局变量的值，就能影响到其他函数中全局变量的值。

局部变量：在定义它的函数内有效，但是函数返回后失效。“在函数内定义的变量”，即在一个函数内部定义的变量，只在本函数范围内有效。

注意：全局变量和静态变量如果没有手工初始化，则由编译器初始化为 0。局部变量的值不可知。

#### 1.1.2.2 静态变量

静态决定了两件事，第一就是存储后不会立刻销毁。第二是自带有一定的限定变量作用区域的功能，强化全局/局部的具体作用域。

静态全局变量：只在定义它的文件内有效，效果和全局变量一样，不过就在本文件内部。

静态局部变量: 只在定义它的函数内有效，只是程序仅分配一次内存，函数返回后，该变量不会消失; 静态局部变量的生存期虽然为整个工程，但是其作用仍与局部变量相同,即只能在定义该变量的函数内使用该变量。退出该函数后，尽管该变量还继续存在。(局部决定其无法被其他函数使用只能被同一函数下次再用。静态决定其保存在静态区，无法被立刻销毁)

这里还继续存在意味着很重要的性质，对于这个函数而言，静态局部变量是可以复用的。静态局部变量在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，存储在动态存储区空间(而不是静态存储区空间)，每一次该函数调用结束后即释放。

为静态局部变量赋初值是在编译时进行值的，即只赋初值一次，在程序运行时它已有初值。以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。而为自动变量赋初值，不是在编译时进行的，而是在函数调用时进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。

如果在定义局部变量时不赋初值的话，对静态局部变量来说，编译时自动赋初值 0(对数值型变量)或空字符(对字符型变量)。而对自动变量来说，如果不赋初值，则它的值是一个不确定的值。这是由于每次函数调用结束后存储单元已释放，下次调用时又重新另分配存储单元，而所分配的单元中的值是不确定的。

虽然静态局部变量在函数调用结束后仍然存在,但其他函数是不能引用它的。也就是说，在其他函数中它是“不可见”的。（这个不可见是指静态局部变量的名称不可在其他函数内被操作。但是由于其内存并没有被析构掉，我们已然能够对这块内存进行操作，这一神奇的操作详见《创建与销毁·一》3.2 部分）

静态变量与全局变量最明显的区别就在于: 全局变量在其定义后所有函数都能用，但是静态全局变量只能在一个文件里面使用，而静态局部变量只能在一个函数里使用。

1.1.3 形参变量

只在被调用期间才分配内存单元，调用结束立即释放。

在函数体内，形参的等级最高。例如：

```
#include "stdio.h"
 int Max = 1;
 void add(int Max)
 { Max = 2;}
int main()
{
     add(Max);
   printf("Max = %d",Max);
```

```
getchar();
return 0;}
```

这个结果当然是 Max=1，因为进入函数后，形参的优先级要高于全局变量了，且函数执行完后，形参释放。

## 1.2 静态变量

### 1.2.0 意义

static 的本质意义在于可控制变量的存储方式和可见性。

在函数内部定义的变量，当程序执行到它的定义处时，编译器为它在栈上分配空间，函数在栈上分配的空间在此函数执行结束时会释放掉。如果想将函数中此变量的值保存至下一次调用时，应该如何实现？ 最容易想到的方法是定义为全局的变量，但定义一个全局变量有许多缺点，最明显的缺点是破坏了此变量的访问范围（使得在此函数中定义的变量，不仅仅只受此函数控制）。static 关键字则可以很好的解决这个问题。

另一方面，在 C++ 中，需要一个数据对象为整个类而非某个对象服务,同时又力求不破坏类的封装性,即要求此成员隐藏在类的内部，对外不可见时，可将其定义为静态数据。

#### 1.2.1 定义

使用 static 修饰的变量。例如 static int i=1;

#### 1.2.2 初始化

初次定义时需要初始化，且只能初始化一次。

#### 1.2.3 静态局部变量

静态局部变量存储在静态存储区，生命周期将持续到整个程序结束。

#### 1.2.4 静态全局变量

静态全局变量具有内部可链接性，用域仅限其声明的源文件，不能被其他源文件所用，可以避免和其他源文件中的同名变量冲突。全局变量一般默认为 static 修型。

即使用 extern 也不能在其他 cpp 文件里用。

#### 1.2.5 extern 修饰符

首先提出的是，**非显式 static 修饰的全局变量和全局函数默认都是非静态的**，即可共享的。

extern 修饰符通常用于当有两个或多个文件希望共享相同的全局变量或函数的时候。一般情况下，没有显式加上 static 修饰的全局变量（譬如 int x）是默认非静态的。如果想在其他的编译单元中使用同一个全局变量，倘若再次定义 int x，必然造成重复定义。于是使用 extern int x 来声明而不再次定义。（注意，不可以 extern 一个显式 static 修饰的全局变量）这样以来，该全局变量对 extern 它的所有的程序文件都可见的。

进一步讲，我们学习过不可以在头文件中定义全局变量，譬如 int x。（这等价于 int x=0） include 的功能很简单，就是复制粘贴。故而对于每个包含了该头文件的 cpp 文件，复制粘贴后相当于都定义了一次非静态的全局变量 int x=0，造成重复定义而链接失败。从而只能在头文件中声明变量，extern int x。

另一方面，**全局函数也是默认非 static 的**。但 extern 修饰对于一个不带修饰的全局函数没有必要，因为当我们在另一 cpp 中声明（而非重定义）这一全局函数时，系统会默认视为 extern。但是，在另一 cpp 中调用这一全局函数，不需要 extern 修饰，但是仍需声明。这样以来，同样的，尽量也不在头文件中定义函数。假设在头文件中定义函数，复制粘贴后还是重定义，也只能在头文件中声明函数。不过对于非显式 extern 修饰的函数而言，不带 extern 的声明会被编译器视为自动带上 extern。

综上，extern 用于实现只声明，不定义。（不分配空间并赋值）

进一步的例子：

```
// func.cpp
int x = 0;
int add(int a) {
    x += a;
    return x;
}
// main.cpp

_____(填空)_____//仅仅只需填写 int add(int)

int main() {
    add(1);
    return 0;
}
```

这里不必然声明 extern int x，因为调用 add 后，add 和 x 在一个 cpp 下，是可以调用的。然而，必须要在 main 当中声明 add 才可以调用。注意到，extern 对于非显式 static 修饰的全局函数没有必要是指不写 extern 也会自动变 extern，而不是说在其他 cpp 中调用这个函数不需要声明。

另外，编译器能够识别 int add(int)和 int add(int x)为同一个函数，名字和形参类型一样就是一个函数。

更进一步：

```
// func.cpp
int x = 0;
int add(int a) {
    x += a;
    return x;
}
// main.cpp
```

```
int x=10 ;
int add(int);
int main() {
   add(1);
   return 0;
}
```

我的两个 x 定义在两个 cpp 里，且对于各自的 cpp 而言而言他们都是非 static 的，故而会 multi-definition。

结论: 对于头文件，尽量只申明函数而不实现函数。尽量只声明全局变量而不定义全局变量。

### 1.2.6 全局变量与局部变量

全局变量和局部变量是从变量的作用域的角度划分。

静态变量和动态变量是从变量的内存分配的角度划分。

全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存储方式上并无不同，区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它。（具体解释就是内部可链接）

### 1.2.7 内部可链接与外部可链接

编译单元: 简单来说**一个 cpp 文件就是一个编译单元**。当一个 c 或 cpp 文件在编译时，预处理器首先递归包含头文件，形成一个含有所有必要信息的单个源文件，这个源文件就是一个编译单元。

事实上，编译每个编译单元(.cpp)时是相互独立的，即每个 cpp 文件之间是不知道对方的存在的。（不考虑#include “xxx.cpp" 这种奇葩的写法）编译器会分别将每个编译单元(.cpp)进行编译，生成相应的 obj 文件。然后链接器会将所有的 obj 文件进行链接，生成最终可执行文件。

我们知道 C++中声明和定义可以分开。例如我们可以一个函数声明定义放在 b.cpp 中，在 a.cpp 只需再声明一下这个函数，就可以在 a.cpp 中使用这个函数。例如：

```
void show();
int main()
{show();return 0;}//这个是 a.cpp


#include <iostream>void show()
{
   std::cout << "Hello" << std::endl;
}//这个是 b.cpp
```

而通过之前的了解，我们知道每个编译单元间是相互独立不知道彼此的存在的。那么 a.cpp 又是如何知道 show 函数的定义的呢？

其实在编译一个编译单元(.cpp)生成相应的 obj 文件过程中，编译器会将分析这个编译单元(.cpp)，将其所能提供给其他编译单元(.cpp)使用的函数，变量定义记录下来。而将自己缺少的函数，变量的定义也记录下来。所以可以认为 a.obj 和 b.obj 记录了以下的信息。

**a.obj**

**我能提供main函数的定义**

**我需要show函数的定义**

**b.obj**

**我能提供show函数的定义**

然后在链接器连接的时候就会知道 a.obj 需要 show 函数定义，而 b.obj 中恰好提供了 show 函数的定义，通过链接，在最终的可执行文件中我们能看到 show 函数的运行。

内部连接：如果一个名称对编译单元(.cpp)来说是局部的，在链接的时候其他的编译单元无法链接到它且不会与其它编译单元(.cpp)中的同样的名称相冲突。例如 static 函数，inline 函数等（注：用 static 修饰的函数，本限定在本源码文件中，不能被本源码文件以外的代码文件调用。而普通的函数，默认是 extern 的，也就是说，可以被其它代码文件调用该函数。）

外部连接：如果一个名称对编译单元(.cpp)来说不是局部的，而在链接的时候其他的编译单元可以访问它，也就是说它可以和别的编译单元交互。 例如非静态变量就是外部链接， 全局变量。

PPT 第 14 和 15 页有两个比较简单的例子，可以简单看一看。

## 1.3 Static 数据成员（类变量）

### 1.3.0 声明、定义、初始化、赋值

声明（declaration ）指定了一个变量的标识符，用来描述变量的类型，是类型还是对象，或者函数等。声明，用于编译器(compiler)识别变量名所引用的实体。

广义的角度上来讲定义是声明的特例，一般情况下把分配了内存空间的声明称作定义，不需要存储空间的声明称作声明。

对于全局变量，int a;这是定义性声明，或称定义。extern int a;这是引用性声明。对于.h 文件里的 class，仅仅 static int x；这是声明（没有分配内存空间）。如果在 class 里就写上 static int x=0，这里既完成了声明，又完成了定义（分配了内存空间），但是这么写不规范，大多编译器都会报错。（下文会解释）

extern int a;只能全局变量用，只声明但是不分配。但是全局变量 int a 既完成了声明也完成了定义，编译器默认赋值为 0。对于 class，static int a 就起到了只声明不分配（也就是只声明不定义）的作用。

初始化和定义的意义相近，广义上将就是分配了储存空间并完成了赋初始值。而赋值就是给已经完成定义的内存空间赋值。

1.3.1 静态数据成员定义与基本性质

使用 static 修饰的数据成员，是隶属于类的，称为类的静态数据成员，也称“类变量”。该数据成员被该类的所有对象共享，即所有对象中的这个数据域处在同一内存位置，在类实例化对象前已分配内存空间。

类的静态成员（数据、函数）既可以通过对象来访问，也可以通过类名来访问，如 ClassName::static_var 或者 a.static_var（a 为 ClassName 类的对象）在实现文件(.cpp)中赋初值，格式为：Type ClassName::static_var = Value;

和全局变量一样，类的静态数据成员在程序开始前初始化。应该在 h 文件里声明（不分配内存空间），在 cpp 文件里定义（分配内存空间且初始化）。

最好不要在 h 文件里定义（也就是写 static int x=0）。如果这么做，可能会导致重复定义（重复分配内存空间），故而这么写不太好。可能造成重定义而无法完成链接，编译失败。

1.3.2 例子

```
//Test.h
class Test {
public:
  static int count; //声明静态数据成员
  Test();
  ~Test();
};
```

```
//Test.cpp
#include “Test.h”

int Test::count = 0;//定义静态数据成员
Test::Test() { count ++; }
Test::~Test() { count --; }
```

```
//main.cpp
#include <iostream>
#include “Test.h”
using namespace std;

int main() {
  Test t1[10];
  cout << “Test#: ” << Test::count << “ or ” << t1[0].count << endl;
  //通过类名或对象访问静态数据成员
}
```

运行输出结果

```
Test#: 10
or 10
```

这个例子的意义：注意到我通过构造函数构造了个 Test 数组，含有 10 个对象，每调用一次构造函数就会给 count++；从而加了 10 次。（这其实也体现了 static 数据是整个类共享的）

## 1.4 Static 成员函数

### 1.4.1 定义与基本性质

在返回值前面添加 static 修饰的成员函数，称为类的静态成员函数。

和静态数据成员类似，类的静态成员函数既可以通过对象来访问，也可以通过类名来访问，如 ClassName::static_function 或者 a.static_function(a 为 ClassName 类的对象）

静态成员函数属于整个类，在类实例化对象之前已经分配了内存空间。

类的非静态成员必须在类实例化对象后才分配内存空间。如果使用静态成员函数访问非静态成员，相当于没有定义一个变量却要使用它。

与静态数据成员有区别的是，静态成员函数可以在.h 中实现。（可以但是不推荐，志愿者说希望实现都放到相应的.cpp 里面）

PPT 上的例子较为简单，可自行阅读。

# 二、常量数据成员与函数

## 2.1 常量

### 2.1.1 常量的定义

常量关键字 const 常用于修饰变量、引用/指针、函数返回值。

### 2.1.2 常量的性质

修饰变量时（如 const int n = 1;），必须就地初始化，该变量的值在其生命周期内都不会发生变化。修饰引用/指针时（如 int a=1; const int& b=a;），不能通过该引用/指针修改相应变量的值，常用于函数参数以保证函数体中无法修改参数的值。修饰函数返回值时（如 const int* func() {…}），函数返回值的内容（或其指向的内容）不能被修改。

## 2.2 常量数据成员

### 2.2.1 常量数据成员的定义

使用 const 修饰的数据成员，称为类的常量数据成员，在对象（具体某个对象，而非一个类共有）的整个生命周期里不可更改。

### 2.2.2 初始化

构造函数的初始化列表中被初始化，就地初始化，但是不允许在构造函数的函数体中通过赋值来设置。

### 2.2.3 不能赋值的理解

为什么不能在构造函数里面初始化常量？

构造函数也是函数，常量的意思就是函数不可以改。——cqq

常量只能定义不能赋值，在任何函数体里的都算是赋值语句，构造函数体也算。——cyd

当执行到构造函数的函数体里的时候，实例已经构造完成了（this 指针已经存在），这时就只能修改它的一些非 const 属性了。——单带师

## 2.3 常量成员函数

### 2.3.1 常量成员函数的定义

成员函数也能用 const 来修饰，称为常量成员函数。

### 2.3.2 意义

实现语句不能修改类的数据成员，即不能改变对象状态（内容）

### 2.3.3 写法

ReturnType Func(…) const {…}

注意区别：const ReturnType Func(…) {…}

**后者是返回值为常量**，也就是前文提及的"修饰函数返回值时（如 const int* func() {…}），函数返回值的内容（或其指向的内容）不能被修改。"

## 2.4 常量对象

若对象被定义为常量(const ClassName a;)，则它只能调用以 const 修饰的成员函数，也即是对象中的"数据"不能变。如果调用了非常量的成员函数，那么就有可能改变对象的数据，故而常量对象不可调用非常量成员函数。但是，常量对象可以成为非常量成员函数的参数。

## 2.5 常量静态变量

### 2.5.1 常量静态变量的意义

我们可以定义既是常量也是静态的变量。常量意味着不可改，静量意味着特定的作用区域。我们可以进一步定义常量静态成员数据，作为不可更改的类变量。

### 2.5.2 定义方法

和静态变量一样，在类内只进行声明（不分配空间），在.cpp 里才定义。（完成内存分配并赋初值）

写成 const static 和 static const 没有太多区别。

但有两个例外：**int 和 enum（枚举型）类型可以就地初始化**。（但也不推荐这么写）

故而判断题: 常量静态的成员变量只能在类外进行初始化。是错的，这不是"你可以在类内写的意思，但是不推荐"这意思，而是强调了两个特例。

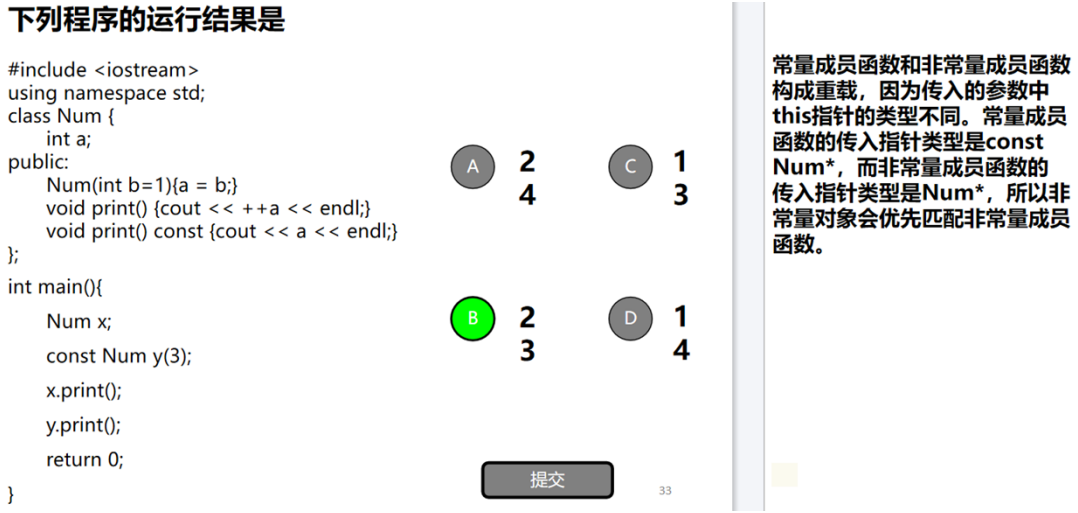
2.5.3 访问权限

常量静态变量和静态变量一样，满足访问权限的任意函数均可访问，但由于 const 的修饰都不能修改。

## 2.6 常量静态函数不存在

常量成员函数依赖于具体的对象，不能修改对象的数据成员。静态成员函数不依赖于对象，它属于整个类，只能调用静态成员。故而对于函数而言，常量和静态是矛盾的。

## 2.7 重载匹配性




常量成员函数和非常量成员函数构成重载时，传入两函数的参数中 this 指针的类型不同。常量成员函数的传入指针类型是 const Num*，而非常量成员函数的传入指针类型是 Num*，所以非常量对象会优先匹配非常量成员函数，而不是进行类型转换，将 this 转换为 const this 再使用。

# 三、 构造与析构

## 3.1 常量对象的析构和构造

常量对象和非常量对象基本相同。

常量全局对象：在 main()函数调用之前进行初始化，在 main()函数执行完 return，程序结束时，对象被析构。

常量局部对象：在程序执行到该局部对象的代码时被初始化。在局部对象生命周期结束、即所在作用域结束后被析构。

## 3.2 静态对象的构造与析构

静态全局对象的构造与析构时机和普通全局对象相同。

今天局部对象在程序执行到该静态局部对象的代码时被初始化，但是离开作用域不析构。第二次执行到该对象代码时，不再初始化，直接使用上一次的对象。（这里和静态局部变量非常相似）

在 main()函数结束后被析构。

```
void fun(int i, int n) {
    if (i >= n)
        static A static_obj("static");
}
```

比如这段代码，虽然看上去会给 static_obj 多次构造，但是由于其是静态局部对象，故而只会构造一次。之后每次利用上一次剩下部分。（相当于函数里内，第一次以后的对静态局部变量的构造失效）

## 3.3 类静态对象构造与析构

类 A 的对象 a 作为类 B 的静态变量。比如

```
class A {};
class B {
   static A a;
};
```

a 的构造与析构表现和全局对象类似，即在 main()函数调用之前进行初始化，在 main()函数执行完 return，程序结束时，对象被析构。a 作为 b 的类对象，和 B 是否实例化无关。

**常量/静态对象的构造与析构实例**

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

class A {
    const char* s;
public:
    A(const char* str):s(str) {
        cout << s << " A constructing" << endl;
    }
    ~A() {
        cout << s << " A destructing" << endl;
    }
};

class B {
    static A a1;
    const A a2;
public:
    B(const char* str):a2(str) { }
    ~B() { }
};

void fun() {
    static A static_obj("static");
}

const A c_a("const c_a");
static A s_a("static s_a");
A B::a1("static B::a1");

int main() {
    cout << "main starts" << endl;
    static B main_b("static main_b");
    for (int i = 0; i < 4; i++) {
        fun();
    }
    cout << "main ends" << endl;
    return 0;
}
```

运行结果：

```
const c_a A constructing
static s_a A constructing
static B::a1 A constructing
main starts
static main_b A constructing
static A constructing
main ends
static A destructing
static main_b A destructing
static B::a1 A destructing
static s_a A destructing
const c_a A destructing
```

37

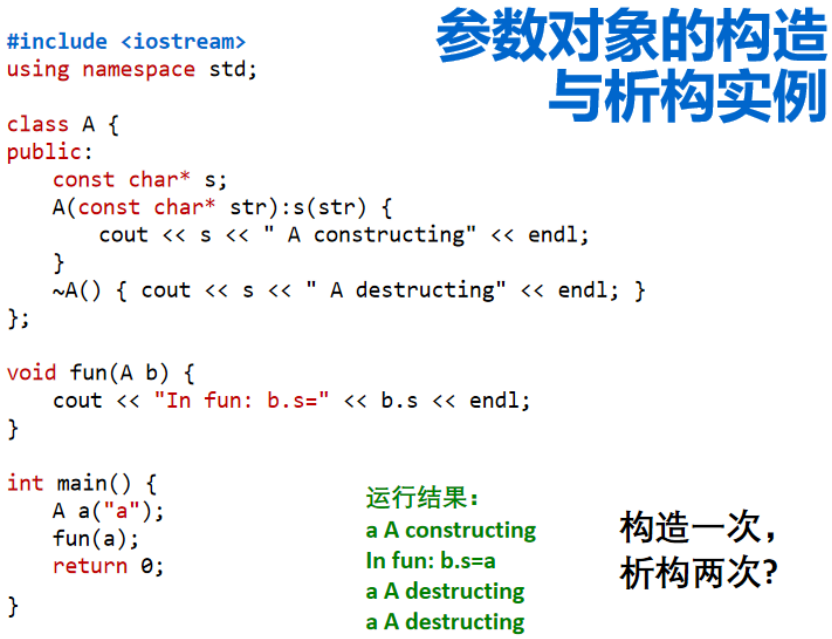
这个例子值得注意的就是，fun 函数体类的静态局部对象只有一次构造，并没有四次。

## 3.4 参数对象构造与析构

### 3.4.1 传递形参

```
void fun(A b) {
   cout << "In fun: b.s=" << b.s << endl;
}
fun(a);
```

在函数被调用时，b 被构造，调用拷贝构造函数（见拷贝构造一节的 PPT）进行初始化。默认情况下，对象 b 的属性值和 a 一致。在函数结束时，调用析构函数，b 被析构。




结合后续学习的拷贝构造和移动构造，我们当然可以确定形参会发生一次拷贝构造，但是为什么没有相应的输出呢？

因为，没有显式定义拷贝构造函数，系统调用了隐式生成的拷贝构造函数，这当然是没有输出的。形参的构造用的是拷贝构造函数，可是析构却是共用的一个析构函数。

### 3.4.2 传递引用与指针

```
void fun(A &b) {
   cout << "In fun: b.s=" << b.s << endl;
}
fun(a);
```

在函数被调用时，b 不需要初始化，因为 b 是 a 的引用。在函数结束时，也不需要调用析构函数，因为 b 只是一个引用，而不是 A 的对象。

### 3.4.3 类成员含有指针

如果传入形参而不是引用或指针，由于拷贝构造不会将被拷贝者置空，故而形参和实参的指针指向了完全相同的地址。但是形参会在函数体结束后析构，

将形参指向的空间释放掉。我的实参又会在主函数结束后析构，这时实参指向的空间将会被再次析构，反复析构，析构空的内存空间，从而报错。

这另一方面启发我们，含有指针的类，一般希望用移动构造而非拷贝构造。

3.4.4 传入引用的优点

尽量使用对象引用作为参数，这样做还可以减少时间开销。

# 四、对象的 new 和 delete

## 4.1 概述

new：生成一个类对象（注意不是类静态对象），并返回地址。（调用默认构造函数，生成了 100 个 A 类对象） A *pA=new A[100]; （调用默认构造函数）A *pA=new A（100）（调用带参数的构造函数，构造了一个 A 类对象）

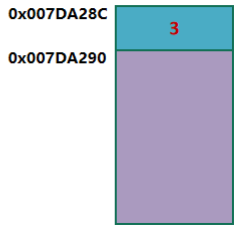
delete：删除该类对象，释放内存资源。（调用析构函数）

delete pA;（释放生成的那个 A 类对象）；delete [] pA;（释放生成的那 100 个 A 类对象）

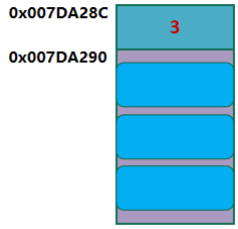
## 4.2 图示

对于大多数编译器，这个过程可以如下概述。

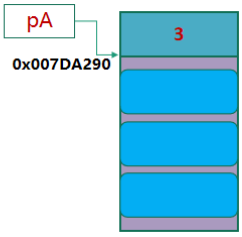
①调用operator new[] 标准库函数来分配足够大的原始未类型化的内存。
注意要多出4个字节来存放数组的大小。

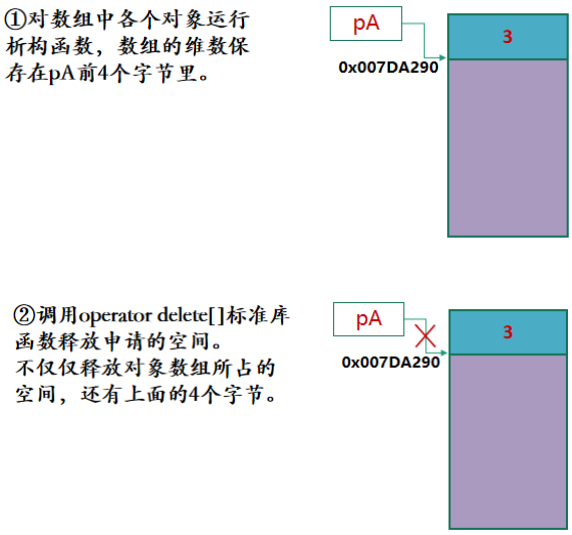



②在刚分配的内存上运行构造函数对新建的对象进行初始化构造。




③返回指向新分配并构造好的对象数组的指针。






## 4.3 匹配性

### 4.3.1 搭配使用

new 和 delete 要配套使用；new 和 delete。new[] 和 delete[]

### 4.3.2 搭配不当

对于大多编译器，如果同时使用 new[]和 delete，会有什么后果？

```
A *pA = new A[3];
delete pA;
```

该 delete 命令做了两件事：调用一次 pA 指向的对象的析构函数，释放 pA 地址的内存。因为，只调用了一次析构函数，故而如果类对象中有大量申请内存的操作，那么因为没有调用析构函数，这些内存无法被释放，造成内存泄漏。

此外，直接释放 pA 指向的内存空间，这个会造成严重的段错误，程序必然会崩溃。因为分配空间的起始地址是 pA-4byte。（delete[] pA 的释放地址自动转换为 pA-4byte）。

### 4.4.3 例子

假定A是一个类的名字，下面四个语句总共会引发类A构造
函数的调用多少次

1. A *p = new A;
2. A p2[10];
3. A p3;
4. A *p4[10];

A 11　　B 12

C 21　　D 22

注意到这里第四个，我实际上是构造了 A 的指针数组，构造了 10 个指针，但是没调用构造函数。

# 五、友元

## 5.1 定义与基本性质

A 类声明 B 为友元类或者友元函数，则 B 具有访问 A 的 private 及 protected 成员的访问权限，即可以访问 A 的一切成员。

友元的声明只能在类内进行。

```
#include <iostream>
using namespace std;
class Test {
    int id;
public:
    Test(int i) : id(i) { cout << "obj_" << id << " created\n"; }

    friend istream& operator>> (istream& in, Test& dst);
    friend ostream& operator<< (ostream& out, const Test& src);
};
istream& operator>> (istream& in, Test& dst) {
    in >> dst.id;
    return in;
}
ostream& operator<< (ostream& out, const Test& src) {
    out << src.id;
    return out;
}  // 以上类中声明了 Test 类的两个友元函数 —— 全局流运算符重载函数，

    // 使这两个函数在实现时可以访问对象的私有成员（如 int id）.

    //并且，这两个流运算符任然是全局函数
```

流运算符重载往往会声明为友元函数，因为经常需要输出私有成员数据。

被友元声明的函数一定不是当前类的成员函数，即使该函数的定义写在当前类内。（注意事项一定不是，而不是不一定是）当前类的成员函数也不需要友元修饰。（因为成员函数本身就可以访问所有数据成员）

## 5.2 跨类友元

### 5.2.1 定义

可以声明别的类的成员函数，为当前类的友元。其他类的构造函数、析构函数也可以是友元。

```
class Y {
   int data;
   friend void X::foo(Y);
   friend X::X(Y), X::~X();
};
```

X 的构造函数 X::X()和析构函数 X::~X()为 Y 的友元函数，则在它们的函数体内可直接访问/修改 Y 的私有成员。

### 5.2.2 区域无关性

友元的声明与当前所在域是否为 private 或 public 无关。

```
class Y {
private:
    friend void X::foo(Y);
};
```

等价

```
class Y {
public:
    friend void X::foo(Y);
};
```

### 5.2.3 不冲突性

一个普通函数可以是多个类的友元函数。

## 5.3 友元类

可对 class/struct/union 进行友元声明，代表该类的所有成员函数均为友元函数。

可对 class/struct/union 进行友元声明，代表该类的所有成员函数均为友元函数。

对基础类型的友元声明会被忽略（因为没有实际价值）。编译器可能会发出警告，但不会认为是错误。

```
class Y {}; // 定义类Y，且Y能访问A的所有成员
class A {
    int data;  // 私有数据成员
    enum { a = 100 };  // 私有枚举项
    friend class X;  // 友元类前置声明（详细类型指定符）
    friend Y;  // 友元类声明（简单类型指定符）（C++11起）
};
class X {}; // 定义类X，X能访问A的所有成员
```

注意两行的差别

两行的区别：friend Y 必须先定义 class Y，但是 friend class X 可以不用先定义 class X。

## 5.4 友元的注意事项

### 5.4.1 非对称性

类 A 中声明 B 是 A 的友元类，则 B 可以访问 A 的私有成员，但 A 不能访问 B 的私有成员。

### 5.4.2 非传递性

朋友的朋友不是你的朋友

### 5.4.3 不可继承

### 5.4.4 友元声明不能定义新的 class

```
class B {};

class A {
    friend B;
};
```

```
class X
{
    friend class Y {}; ☹
}
```

左图是我先有了 B 才能定义 B 为 A 的友元。而右边企图在 X 类内定义 Y 为友元类，并给出 Y 的定义，这是不合法的。