创建与销毁·二

[一、 变量与静态变量 2](#_Toc15294)

[1.1 四类变量的区别 2](#_Toc19680)

[1.2 静态变量 4](#_Toc1446)

[1.2.1 定义 4](#_Toc2753)

[1.2.2 初始化 4](#_Toc20830)

[1.2.3 静态局部变量 4](#_Toc3885)

[1.2.4 静态全局变量 4](#_Toc9855)

[1.2.5 全局变量与局部变量 4](#_Toc7482)

[1.2.6 内部可链接与外部可链接 4](#_Toc13856)

[1.3 Static数据成员（类变量） 5](#_Toc6812)

[1.3.0 声明、定义 5](#_Toc11349)

[1.3.1 静态数据成员定义与基本性质 6](#_Toc5008)

[1.3.2 例子 6](#_Toc15068)

[1.4 Static成员函数 7](#_Toc22388)

[1.4.1 定义与基本性质 7](#_Toc3556)

[二、 常量数据成员与函数 7](#_Toc9295)

[2.1常量 7](#_Toc13136)

[2.1.1 常量的定义 7](#_Toc3821)

[2.1.2 常量的性质 7](#_Toc25129)

[2.2常量数据成员 7](#_Toc30319)

[2.2.1 常量数据成员的定义 7](#_Toc3872)

[2.3常量成员函数 8](#_Toc1582)

[2.3.1 常量成员函数的定义 8](#_Toc22672)

[2.3.2 意义 8](#_Toc2556)

[2.3.3 写法 8](#_Toc20082)

[2.4 常量对象 8](#_Toc16273)

[2.5 常量静态变量 8](#_Toc5793)

[2.5.1 常量静态变量的意义 8](#_Toc8280)

[2.5.2 定义方法 8](#_Toc27632)

[2.5.3 访问权限 9](#_Toc20132)

[2.6 常量静态函数不存在 9](#_Toc5893)

[2.7 重载匹配性 9](#_Toc27032)

[三、 构造与析构 9](#_Toc14121)

[3.1常量对象的析构和构造 9](#_Toc25929)

[3.2 静态对象的构造与析构 10](#_Toc8802)

[3.3 类静态对象构造与析构 10](#_Toc12507)

[3.4 参数对象构造与析构 11](#_Toc8084)

[3.4.1 传递形参 11](#_Toc687)

[3.4.2 传递引用与指针 11](#_Toc4735)

[3.4.3 类成员含有指针 12](#_Toc8922)

[3.4.4 传入引用的优点 12](#_Toc12497)

[四、 对象的new和delete 12](#_Toc3712)

[4.1 概述 12](#_Toc20649)

[4.2 图示 12](#_Toc25281)

[4.3 匹配性 13](#_Toc9267)

[4.3.1 搭配使用 13](#_Toc13404)

[4.3.2 搭配不当 13](#_Toc4949)

[4.4.3 例子 13](#_Toc24894)

[五、 友元 14](#_Toc11132)

[5.1 定义与基本性质 14](#_Toc4999)

[5.2 跨类友元 14](#_Toc7942)

[5.2.1 定义 14](#_Toc10711)

[5.2.2 区域无关性 15](#_Toc22759)

[5.2.3 不冲突性 15](#_Toc5854)

[5.3 友元类 15](#_Toc3697)

[5.4 友元的注意事项 15](#_Toc6784)

[5.4.1 非对称性 15](#_Toc18540)

[5.4.2 非传递性 15](#_Toc2320)

[5.4.3 不可继承 15](#_Toc26079)

[5.4.4 不可定义 15](#_Toc14701)

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。

1.1 对四类变量的区别

1.2 静态变量与相关概念的界定

1.2.5 extern修饰符的详细讨论

1.3 类变量与相关概念的界定

1. 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(static)

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。

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，因为static修饰的全局变量的作用域只限于它被定义的那个cpp文件）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(指存储在静态存储区，生命周期持续到整个程序结束），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（所有的cpp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（某个cpp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()

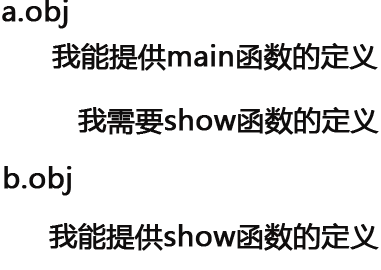
{

    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。



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。 例如非静态全局变量就是外部链接。（只要使用extern关键字），而非静态全局函数也是外部链接。

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（就地初始化），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。

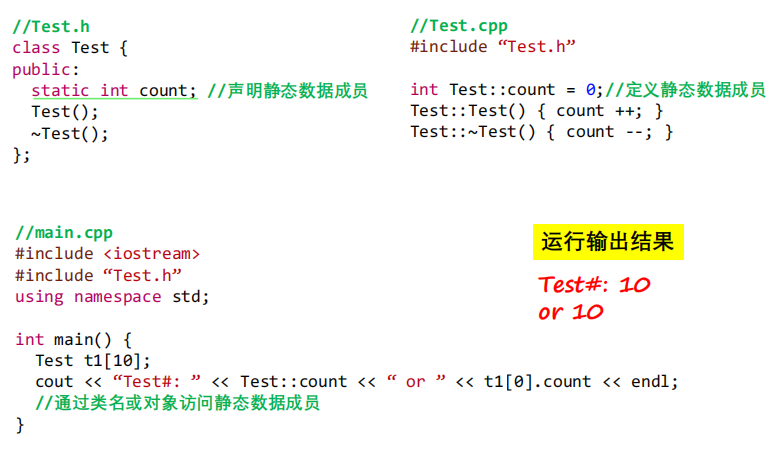
类的静态成员（数据、函数）既可以通过对象来访问，也可以通过类名来访问，如ClassName::static\_var或者a.static\_var（a为ClassName类的对象）通过类名访问的特点更加强调了静态成员是属于类的，而不是属于某个具体对象的。

类的静态成员在实现文件(.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

 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。

1. 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。(当参数类型是const&时，既可以避免拷贝构造参数的成本，又可以避免函数体修改参数。）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（const）

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）

（回顾：static变量要在cpp里声明，在.h里定义；而const类型要在初始化列表里初始化或者就地初始化）

写成const static和static const没有区别。

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，但是不推荐”这意思，而是强调了两个特例。

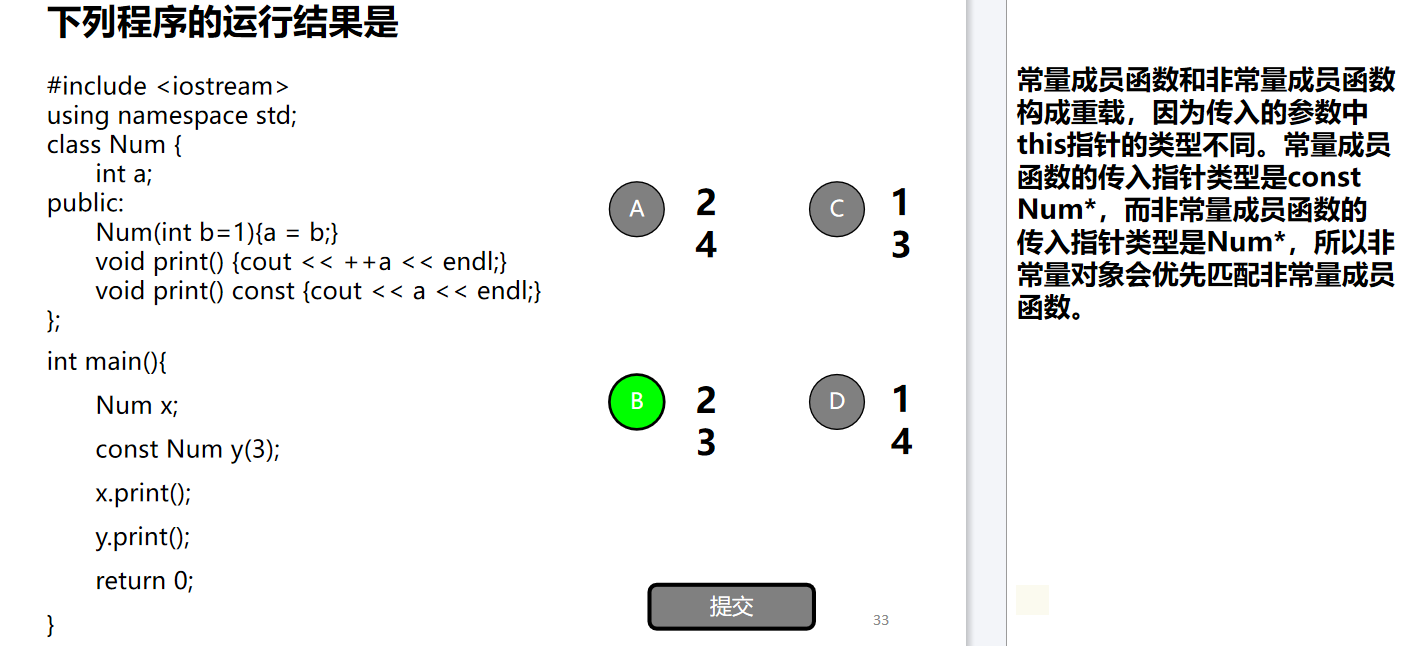
2.5.3 访问权限

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。

1. 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。

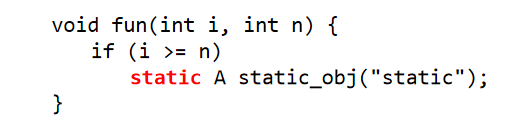
常量局部对象：在程序执行到该局部对象的代码时被初始化。在局部对象生命周期结束、即所在作用域结束后被析构。

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。（即：在main()调用前初始化，在main（）执行完return后被析构）

静态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。



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

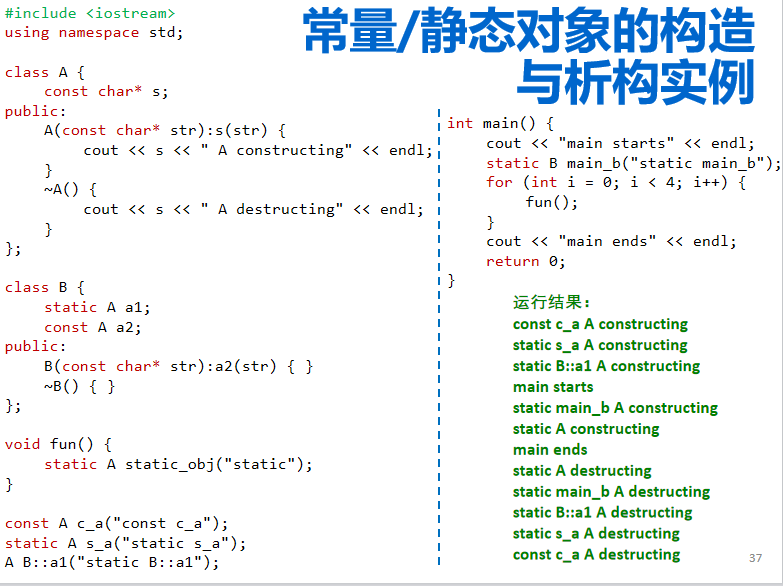
class A {};

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。（这个和类里面的默认类型的静态变量,比如B里面的对象static int c;是一样的）



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

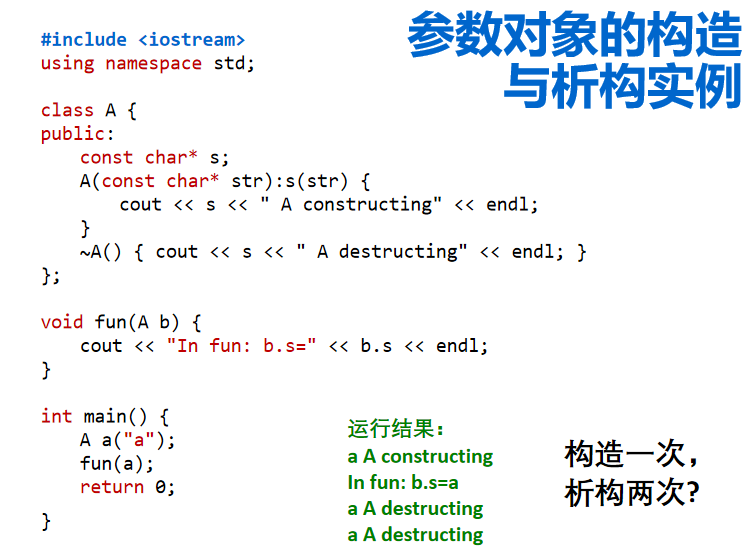
void fun(A b) {

    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不需要被拷贝构造，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。（因为省去了拷贝构造形参的时间）

1. 对象的new和delete

4.1 概述

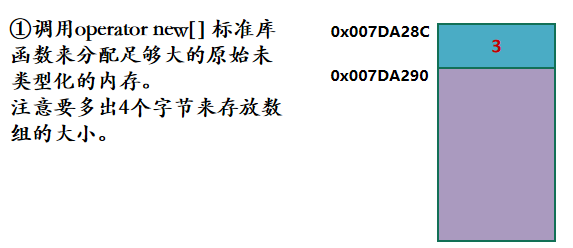
new：生成一个类对象（注意不是类静态对象），并返回地址。（调用默认构造函数，生成了100个A类对象） A \*pA=new A[100]; （调用默认构造函数）A \*pA=new A（100）（调用带参数的构造函数，构造了一个A类对象）

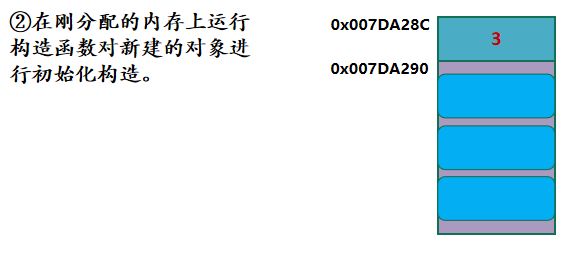
delete：删除该类对象，释放内存资源。（调用析构函数）

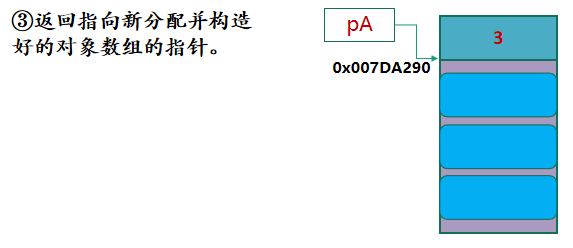
delete pA;（释放生成的那个A类对象）；delete [] pA;（释放生成的那100个A类对象）

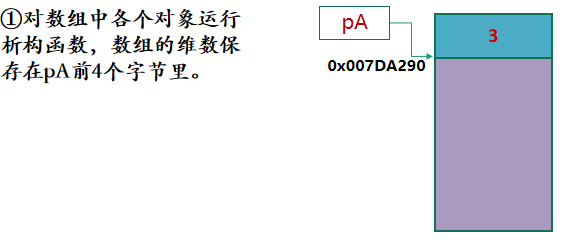
4.2 图示

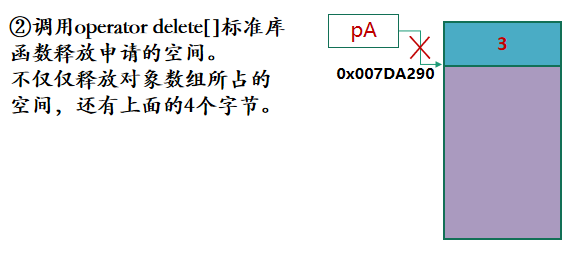
对于大多数编译器，这个过程可以如下概述。











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？

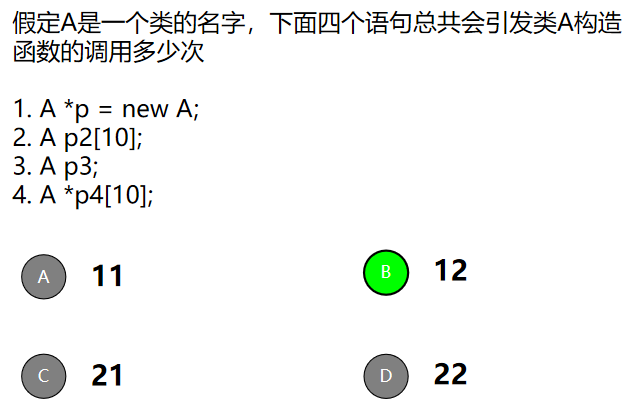
A \*pA = new A[3];

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



new是会调用类A的构造函数的。（过程：分配内存-初始化-返回指针）

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。

当然，如果我们调用 A\*p=new A[5],那么会调用5次A的构造函数。

1. 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);

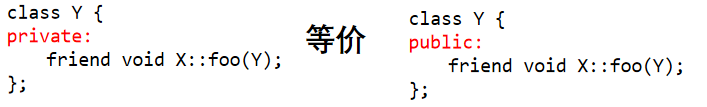
    friend X::X(Y), X::~X();

};

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。



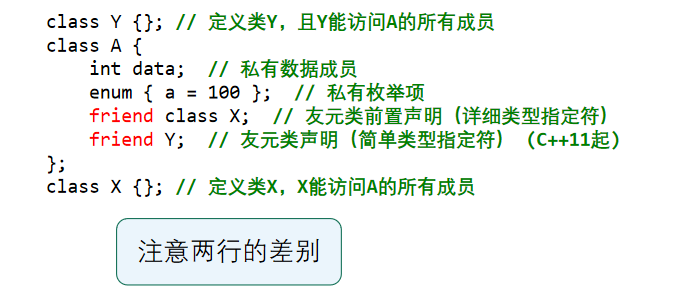
5.2.3 不冲突性

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。

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。



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

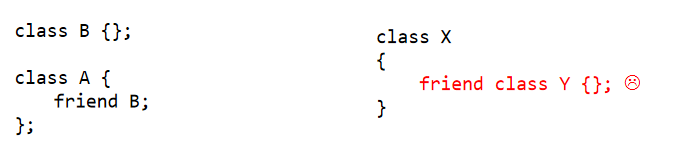
类A中声明B是A的友元类，则B可以访问A的私有成员，但A不能访问B的私有成员。

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



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。（不过friend class Y倒是可以，即使so far还没有定义Y）