# c与c++的区别

## 1.面向对象的特征

抽象、封装、继承、多态是面向对面的特点，　本质是消息传递。c是面向过程。

## 2.头文件

c++的头文件是是由模板编写而成的，与c的头文件有差别，c是c++的子集，c++是c的超集，如果c头文件与c++头文件都要使用的时候，c头文件放在c++头文件前面。否则可能出现一些问题。

## 3.标准输入输出

标准输出流cout<<，标准输入流cin>>bia变量

endl、ends、\n、fllush

endl=\n+fllush，注意c中\n会刷新缓冲区，但c++中不会

## 4.命名空间

命名空间是用于解决相同命名的对象在使用时发成冲突的问题。原理就是将变量，函数等的作用域限定在某个命名空间中。

C++中定义名称空间的基本格式为：

namespace 名称空间名

{

变量类型 变量；

函数返回类型 函数原型；

}

如果省略空间名，则是匿名空间

c++中使用命名空间的实体的基本格式有：

①作用域限定符：空间名::实体名//每次在名称空间外使用实体都需要使用作用域限定符

②using声明机制：using 空间名::实体名//使得某个实体可见

③using编译指令：using namespace 空间名//使得所有实体可见

定义在名称空间中的变量或者函数都称为实体。名称空间中的实体作用域是全局的, 并不意味着其可见域是全局的。如果不使用作用域限定符和using机制，抛开名称空间嵌套和内部屏蔽（局部变量屏蔽全局变量）的情况，实体的可见域是从实体创建到该名称空间结束，在名称空间外，该实体是不可见的。因而在命名空间外使用实体时候，必须用作用域限定符使其可见：

空间名::实体名

如果不希望在每次使用名称空间中实体时都使用作用域限定符，可使用using声明机制扩展其可见域，using声明的基本格式为：

using 名称空间::实体名; //如果是函数只需声明函数名

如“using A::dispA;”，至于using声明语句将该实体的可见域扩展到什么程度，这取决于using语句的书写位置，换言之，这取决于using语句的可见域。

using声明机制使得某个空间中的特定实体可见，using编译指令比using声明更进一步，使得名称空间中的所有实体都可见，不再需要作用域限定符：

using namespace 名称空间名;

当然除了命名空间外，也有匿名空间，对匿名空间实体访问：

：：实体名

注意：

如果using声明使用不当，很容易引起多重声明错误,比如:

已经定义了全局函数disp，却还使用全局using声明语句“using A::disp”，假设没有屏蔽发生，那么调用disp()时，编译器不确定是全局函数版本还是A::disp()，引发多重声明错误。变量名同样存在这种问题，假设有两个名称空间A、B中都定义了int型变量num，在程序的某处需要使用num，如果写出如下代码：

using A::num;

using B::num;

num=5;

因此，应合理使用using声明机制。c++推荐使用using声明机制。

## 5.const常量关键字

定义常量：

const 数据类型 常量名=初值 //常量定义的时候必须初始化,位于文字常量区

定义常量指针：

const 数据类型 \*指针名=&常量名//指针所指空间可变，但所指空间的内容不可变。

定义指针常量：

数据类型 \* const指针名= 地址//指针所指空间不可变，但所指空间的内容可变

定义指向常量的指针常量：

const 数据类型 \* const指针名= &常量名 //指针所指空间不可变，但所指空间的内容也不可变

宏定义与const的区别：宏定义发生在预编译阶段，没有语法检测，仅仅是简单的字符串替换；const 发生在编译阶段，有语法检测，实质仍然是定义一个变量。

c++中所有的常量或者需加上const关键字

## 6.c++中申请堆空间

开辟单个地址单元：

指针=new 数据类型（初值）

delete 指针

开辟多个地址单元

指针=new 数据类型[下标]

delete []指针名

区别：

（1）malloc和free是c的标准库函数，new和delete是运算符

（2）malloc只负责开辟空间，返回值为void\*，不负责转类型，需要强制转换类型，并需要显式指出空间大小，new开辟空间的时候，会自动转换为所需类型，自动计算空间大小。

（3）new的效率高于malloc

（4）但注意，指针本身是放在栈区的，指向堆区。

## 7.c++支持引用

c++中引用而c中没有

数据类型 &别名=变量名 //变量必须存在

const数据类型 &别名=常量 //对常量的引用必须使用常引用

引用与指针的区别：

（1）引用必须初始化，指针不用

（2）引用一旦定义就不可改变，而指针指向可改变。

（3）对引用的操作就是对原对象的操作，

（4）引用作为函数形参，对形参的操作就是对实参的操作，调用函数时不会开辟形参空间，可以节约传值的时间和空间开销。

指针作为形参，对指针的操作不会改变实参。对指针所指内容的操作会改变实参。但参数传递时会开辟一个指向指针的空间。实质仍是传值。

（5）引用作为函数返回值，从而可以对函数赋值。

int a[3]={1,2,3};

int &func(int idx)

{

return a[idx];

}

func(0)=10

（6）同函数不能返回指向局部变量的指针一样，函数也不能返回局部变量的引用。也不能返回一个指向堆空间变量的引用。

（7）传引用的作用相当于传指针，传指针的引用相当于传二级指针。到底参数传递时要不要使用指针或者引用？取决于传递的参数大小和是否需要在函数中对实参做改变，如果实参很大，或者需要在函数中中实参要改变，则需要使用指针或引用。

（8）引用和指针都有地址的概念

（9）指针可以赋值为NULL，引用不可以。

## 8.c++支持函数重载

函数名相同而参数不同时，c不能区分，但c++会自动匹配。其原理是：名字改编。当函数名相同时根据返回值类型，参数个数，类型和顺序来进行函数改编。

如果想在c++中以c的方式调用同名函数则需在函数实现前加上extern “C” ，实现c和c++的混合编程，但不能对多个同名函数都以c方式调用，否则会发生冲突。

#ifdef #endif 和#ifndef #endif 避免重复的宏定义冲突，形式如下：

#ifdef \_cplusplus

extern “C”{

#endif

函数定义

#ifdef \_\_cplusplus

}

#endif

函数实现

## 9.变量类型的强制转换

c中的强制转换方式为：（类型名）变量名或者类型名（变量名）,这种强制转换是不安全的。

c++中的强制转换：static\_cast<类型名>（变量名）/const\_cast/dynamic\_cast/reinterpret\_cast

## 10.c++支持bool类型（c99又新增了bool）

c99之前的c语言是没有bool类型的

c++中有bool类型，取值为true和 false，代表0和非零（正数和负数都为真）。其大小为1个字节。

## 11.c++的函数可带默认参数

c++中如果函数的形参有初值，调用函数如果不给定特值则使用初值参与函数调用。如果某个参数是默认参数，那么它后面的参数必须都是默认参数，也就是默认参数的设置必须要从右到左的顺序进行。而函数调用时形参必须从左向右给定，如果有一个参数省略，则后面的参数都必须省略。如果有函数重载时，要注意避免二义性的问题。比如定义了 add（int x，int y）与int add（int x,int y=10,int z=10），此时发生函数调用add（x,y）就会发生function ambiguous的错误。

## 12.inline函数

为了解决程序中函数调用效率问题。

c中可以使用带参数的宏来处理，c++ 中可以在函数之前加上inline关键字。

函数是一种更高级的抽象。它的引入使得编程者只关心函数的功能和使用方法，而不必关心函数功能的具体实现；函数的引入可以减少程序的目标代码，实现程序代码和数据的共享。但是，函数调用也会带来降低效率的问题，因为调用函数实际上将程序执行顺序转移到函数所存放在内存中某个地址，将函数的程序内容执行完后，再返回到转去执行该函数前的地方。这种转移操作要求在转去前要保护现场并记忆执行的地址，转回后先要恢复现场，并按原来保存地址继续执行。因此，函数调用要有一定的时间和空间方面的开销，于是将影响其效率。特别是对于一些函数体代码不是很大，但又频繁地被调用的函数来讲，解决其效率问题更为重要。引入内联函数实际上就是为了解决这一问题。

在程序编译时，编译器将程序中出现的内联函数的调用表达式用内联函数的函数体来进行**替换**。显然，这种做法不会产生转去转回的问题，但是由于在编译时将函数休中的代码被替代到程序中，因此会增加目标程序代码量，进而增加空间开销，而在时间代销上不象函数调用时那么大，可见它是以目标代码的增加为代价来换取时间的节省。

定义内联函数的方法很简单，只要在函数定义的头前加上关键字inline即可。内联函数的定义方法与一般函数一样。如：

inline int add\_int (int x, int y, int z)

{

return x+y+z;

}

在程序中，调用其函数时，该函数在编译时被替代，而不是像一般函数那样是在运行时被调用。

使用内联函数应注意的事项

内联函数具有一般函数的特性，它与一般函数所不同之处公在于函数调用的处理。一般函数进行调用时，要将程序执行权转到被调用函数中，然后再返回到调用它的函数中，；而内联函数在调用时，是将调用表达式用内联函数体来替换。在使用内联函数时，(尽量保证只有顺序语句)应注意如下几点：

1.在内联函数内不允许用循环语句和开关语句。

2.内联函数的定义必须出现在内联函数第一次被调用之前。

此外inline可以解决多次重复定义的问题。

## 13.避免重复定义变量或函数

第一种方法是c中的，在头文件中只进行函数声明，不做定义。

第二种方法是在函数前加关键字inline

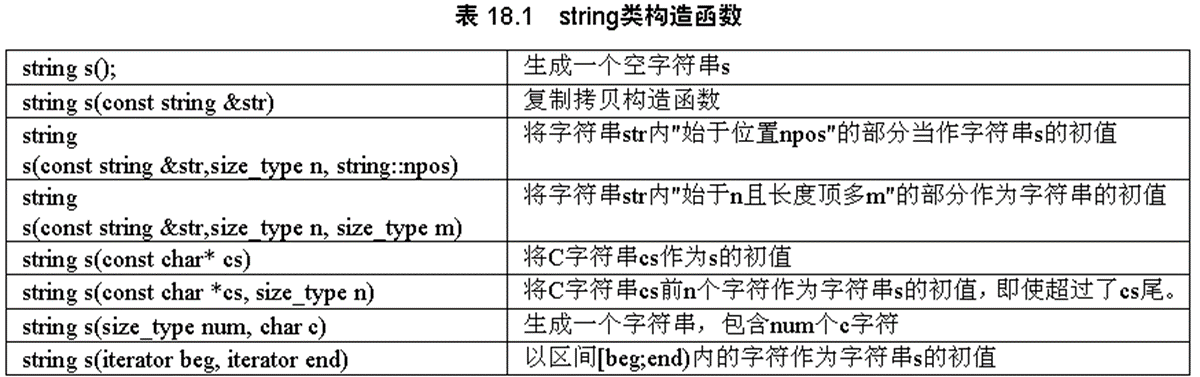
## 14.string类型

c中的字符串都是用字符数组来处理。

c++中的具有字符串类型string。其实质仍然是用字符串数组的模板来实现的。 向下兼容c中的字符串的所有用法。string实质是一个类。

### 14.1. 声明一个字符串

string是字符串类，首先来看一下如何通过构造函数来声明一个字符串，string类的构造函数如所示:



### 14.2. 字符串的输入

C风格字符串的输入方式大致有“>>”、“cin.getline()”和“cin.get()”3种，对string字符串来说，除了重载了“>>”实现输入外，string头文件还定义了getline函数用以输入string字符串，抛弃了“cin.getline()”和“cin.get()”两种输入方式，原因在于getline外部函数能自动调整目标string的大小，使其能恰好存储输入的字符。>>输入同样会自动调整string对象的大小。

外部getline函数第一个参数必须为输入流对象，第二个参数是待输入的string对象，第3个参数是分界符，getline函数将输入流中的字符存储到string变量中，直到满足下列条件之一：

到达文件尾，此时输入流的eofbit置位有效。

遇到分界字符，默认为换行符’\n’，此时，分解符会从流中删除，但并不会存储到string对象中。

读取的字符达到最大允许值（string类中的常量npos决定了string所能存储的最大字符数，一般都很大，不会对输入产生影响，另一个因素是可用内存数，因此，最大允许值便是从npos和可用内存字节数中选取较小的一个），输入流的failbit置位有效。

### 14.3.string字符串和C风格字符串的转换

C风格字符串转换为string字符串相对来说比较简单，通过构造函数即可实现。但由于string字符串实际上是类对象，其并不以空字符’\0’结尾，因此，string字符串向C风格字符串的转化是通过3个成员函数完成的，分别为：

const char \*data(); //以字符数组的形式返回字符串内容，但末尾并不添加’\0’

const char \*c\_str()；//返回一个以‘\0’结尾的字符数组

int copy(char\*s,size\_type n);//字符串的内容复制或写入既有的C风格字符串或字符数组内

在C++11以后，data和c\_str成员函数的功能相同

### 14.4.string 赋值

对string字符串的赋值有两种途径

使用“=”：等号右侧的操作数可以是string、C风格字符串、还可以是单个字符。

使用成员函数assign()：成员函数assgin有多种重载形式，从本质上说，和有参构造函数的参数是相同的。

### 14.5. string 删除

如果对string 清空，直接赋予空值或者用成员函数erase()即可。

此外erase方法用于从字符串中删除字符，

1）iterator erase( iterator First , iterator Last );

删除[First, Last)的字符，返回的迭代器指向最后一个被删除元素的后一个元素。

2）iterator erase( iterator It );

删除string 中 It 所指的字符，返回指向下一个元素的迭代器，如果后面没有其他元素，则返回end()。

3）string& erase( size \_ type pos = 0, size \_ type n = npos );

删除string中从pos位置开始的n个字符或删除到末尾，返回删除后的string 的引用

### **14.6.元素追加与相加**

1）用+号或+=号

2）用append方法

(1)string& append( size\_type count, CharT ch );

(2)string& append( const string& str );

(3)string& append( const string& str, size\_type pos, size\_type count = npos);

(4)string& append( const CharT\* s, size\_type count );

(5)string& append( const CharT\* s );

(6)string& append( InputIt first, InputIt last );

### 14.7.元素插入：

用insert方法：

（1）string& insert( size\_type pos, const char\* s);

（2）string& insert( size\_type pos, const char\* s , size\_type n );

（3）string& insert(size\_type pos, const string& str );

（4）string& insert(size\_type pos,const string& str, size\_type pos1, size\_type n)

（5）string& insert(size\_type pos, size\_type n , char c );

（6）void insert(Iterator pos, Iterator First , Iterator Last );

（7）void insert(Iterator pos,size\_type n,char c);

### 14.8.大小和容量：



### 14.9.元素访问：

可以使用下标操作符[]和函数at()对字符串中包含的字符进行访问，需要注意的是操作符[]并不检查索引是否有效（有效索引0~str.length()），如果索引失效，会引起未定义的行为，而at()会检查，如果使用 at()的时候索引无效，会抛出out\_of\_range异常。比如：对const修饰的string常量str来说，操作符[]对索引值str.length()仍然有效，返回值是’\0’，其他情况下（即str不是const修饰的字符串常量），str.length()索引都是无效的。

### 14.10.字符串比较：

string字符串支持常见的比较操作符（>,>=,<,<=,==,!=），也支持string与C风格字符串的比较。根据字符的字典顺序（字典排序靠前的字符小）从前往后逐一比较，遇到不等的字符就按该位置上的这两个字符的比较结果确定两个字符串的大小，即string(“abcde”)<string(“ac”)。此外，string类还提供了成员函数compare()用于字符串比较，支持多参数处理，支持用索引值和长度定位子串来进行比较，返回一个整数（0：相等、>0：大于、<0：小于）来表示比较结果。

### 14.11.提取子串：

string类提供了substr成员函数用于提取子串，以string str(“98765432112345678”)为例：

（1）str.substr(); //返回str的全部内容

（2）str.substr(9); //截断前九个字符，即“12345678”

（3）str.substr(5,6);//从下标为5的字符开始往后截取，一共截取6个字符，即“432112”

### 14.12.搜索和查找：

string类提供了6种搜索函数，返回值均为size\_type，简要列举如下：

（1）find系列：返回待查找元素或子串在字符串中第一次出现的位置，举例来说，字符串str为“Welcome to C++ World”，使用find函数系列查找单个字符’o’在str中第一次出现位置，返回结果为4，还可查找子串“to”在str中的位置，返回结果为8。

（2）rfind系列：返回待查找元素或子串在字符串中最后一次出现的位置，如使用rfind函数查找单个字符’o’在str中最后一次出现位置，返回结果为16。

（3）find\_first\_of系列：返回待查找子串元素在字符串中第一次出现的位置，如果查找的子串，不是查找整个字符串的匹配，而是搜索子串中的字符首次出现的位置，同样以字符串“Welcome to C++ World”为例，使用find\_first\_of()系列函数查找字符串“abc”在其中首次出现的位置，返回结果为3，此时第一次出现了字符串“abc”中的元素’c’。

（4）find\_last\_of系列：返回待查找子串元素在字符串中最后一次出现的位置，如果查找的子串，不是查找整个字符串的匹配，而是搜索子串中的字符最后出现的位置。

（5）find\_first\_not\_of系列：与find\_first\_of()的工作方式类似，不过搜索的是第一个不位于字符串中的字符第一次出现的位置。

（6）find\_last\_not\_of系列：与find\_first\_of()的工作方式类似，不过搜索的是第一个不位于字符串中的字符最后一次所在位置。

### **14.13.对字符串进行遍历**

（1）用for循环

（2）用迭代器 string：：iterator 迭代器；

for（it =字符串变量.begin（）;迭代器！=字符串标量.end（））

{visit（迭代器）}

获取字符串的子串

# C++程序内存分配方式

## 1. 程序的内存分布

1）栈区（stack）：由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2）堆区（heap）：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3）全局/静态区(static）：全局变量和静态变量的存储是放在一块的，在程序编译时分配。又分为初始化和未初始化区。

4）文字常量区：存放常量字符串

5）程序代码区：存放函数体（类的成员函数、全局函数）的二进制代码.



## 2. 栈与堆的比较

1）申请方式

stack: 由系统自动分配。例如，声明在函数中一个局部变量int b; 系统自动在栈中为b开辟空间

heap: 需要程序员自己申请，并指明大小，在c中malloc函数，如在C++中用new运算符

2）系统响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。另外，对于大多数系统，首地址处会记录这块内存空间中本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

3）大小限制

栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。。由此可见，堆获得的空间比堆的大小受限于计算机系统中有效的虚拟内存较灵活，也比较大。

4）申请效率

栈由系统自动分配，速度较快。但程序员无法控制。

堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。

5）存储内容

栈在函数调用时，第一个进栈的是主函数的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

## 3.栈与堆的区别

1) 管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak

2) 空间大小：一般来讲在32位系统下，堆内存可以达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在VC6下面，默认的栈空间大小是1M

3) 分配方式：堆都是动态分配的，没有静态分配的堆。内存有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由malloc,calloc函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

4) 生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

5） 碎片问题：对于堆来讲，频繁的new/ delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在它上面的后进的栈内容已经被弹出

6) 生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

# 类和对象

## 1.面向对象和面向过程

面向过程：

复杂的大问题->层层分解/模块化->若干子问题->自顶向下，逐步求精

程序 ＝ 变量 980-= + 函数 （程序 ＝ 数据结构 + 算法）

问题：数据结构和算法之间没有直观的联系,理解难 修改难 重用难

面向对象：

认为现实世界是由对象组成的，用程序来模拟现实世界中对象与对象之间的交互，对象由相关的数据和操作组成，对象具有相对独立性，对外提供一定的服务。

程序 ＝ 对象 + 对象 + … + 对象

面向对象的四大特性：抽象、封装、继承、多态

## 2.类和对象的含义

从哲学角度说，先有对象，然后才有类，类和对象是“一般和特殊”这一哲学原理在程序世界中的具体体现。类是对象的抽象，对象是类的实例，相当于数据类型和变量，结构体类型和结构体变量的关系。

对对象的抽象（又叫类的提取）往往是从两个方面来考虑的：

一是特征（C++常称为“属性”，即类的数据成员），

二是功能（C++中常称为“行为或方法”，即操作数据成员的的函数）

## 3.类的定义

C++中使用关键字class定义一个类，其基本形式如下：

class 类名

{

public:

公共数据成员和函数

protected:

保护数据成员和函数

private:

私有数据成员和函数

}; //不要漏写了这个分号;

C++规定，类成员的访问权限默认是private，不加申明的成员默认是private.数据成员的类型前面不可使用auto、extern和register等，也不能在定义时对非静态数据成员进行初始化，如果将float price写成float price = 0，编译器会报错。

类定义中提供的成员函数是函数的原型申明。

## 4.类的实现

类的实现就是定义其成员函数的过程，类的实现有两种方式：

1）在类定义时同时完成成员函数的定义，但要注意类内定义的成员函数都是inline的成员函数。

2）在在类定义内部进行声明，在类定义外部实现。在类定义的外部定义成员函数时，应使用作用域操作符（::）来标识函数所属的类，即有如下形式：

返回值类型 类名::成员函数名(参数列表)

{

函数体

}

其中，返回类型、成员函数名和参数列表必须与类定义时的函数原型一致。

## 5.类的使用

定义类之后，便可以像使用int、double等数据类型定义变量一样，创建该类的对象。由此看来，类的定义实际上是定义了一种类型，类不接收或存储具体的值，只作为生成具体对象的“蓝图”，只有将类实例化，创建对象（定义类的变量）后，系统才为对象分配存储空间。格式如下：

调用默认构造函数定义对象：类名 对象名或 类名 指向对象的指针名

调用自定义构造函数定义对象：类名 对象名（参数表）

成员访问：对象名.成员名 或 指针名->成员名

对象的作用域、可见域和生存期与普通变量，如int型变量的作用域、可见域和生存期并无不同，对象同样有局部、全局和类内之分，对于在代码块中声明的局部对象，在代码块执行结束退出时，对象会被自动撤销，对应的内存会自动释放（当然，如果对象的成员函数中使用了new或malloc申请了动态内存，却没有使用delete或free命令释放，对象撤销时，这部分动态内存不会自动释放，造成内存泄露）。

注意：

数据成员占据不同的内存区域(堆、栈)；成员函数共用同一内存区域(代码段)

无参数的非静态成员函数调用时必须加上空括号。

## 6.class 与 struct的区别

class的定义看上去很像struct定义的扩展，事实上，类定义时的关键字class完全可以替换成struct，也就是说，结构体变量也可以有成员函数。class和struct的唯：一区别在于：struct的默认访问方式是public，而class默认为private。struct 也可以用private关键字

提示：通常使用class来定义类，而把struct用于只表示数据对象、没有成员函数的类。

## 7.public,private,protected

1）类的一个特征就是封装，public和private作用就是实现这一目的。所以：

用户代码（类外）可以访问public成员而不能访问private成员；private成员只能由类成员（类内）和友元访问。

2）类的另一个特征就是继承，protected的作用就是实现这一目的。所以：

protected成员可以被派生类对象访问，不能被用户代码（类外）访问。

继承中的特点：

先记住：不管是否继承，上面的规则永远适用！

有public, protected, private三种继承方式，它们相应地改变了基类成员的访问属性。

1）public继承：基类public成员，protected成员，private成员的访问属性在派生类中分别变成：public, protected, private（原样继承）

2）protected继承：基类public成员，protected成员，private成员的访问属性在派生类中分别变成：protected, protected, private（公有变保护）

3）private继承：基类public成员，protected成员，private成员的访问属性在派生类中分别变成：private, private, private（全变私有）

但无论哪种继承方式，上面两点都没有改变：

1）private成员只能被本类成员（类内）和友元类访问，不能被派生类访问；

2）protected成员可以被派生类访问。

## 8.对象的创建和撤销

C++为类提供了两种特殊的成员函数来完成相反的工作：

一是构造函数，在对象创建时自动调用，用以完成对象数据成员等的初始化及其他操作（如为指针成员动态申请内存空间等）；如果程序员没有显式的定义它，系统会提供一个默认的构造函数。一旦显示的定义了构造函数，系统就不会提供默认的构造函数了。

另一个是析构函数，在对象撤销时自动调用，用以执行一些清理任务，如释放成员函数中动态申请的内存等。如果程序员没有显式的定义它，系统也会提供一个默认的析构函数。

### 8.1.构造函数与初始化表达式

#### 8.1.1.构造函数

当对象被创建时，构造函数自动被调用。构造函数有一些独特的地方：构造函数的函数名与类名相同，没有返回类型和返回值，可以有函数参数或带默认值的参数，并且构造函数支持函数重载。即使是void也不能有。其主要工作有：

给对象一个标识符。

为对象数据成员开辟内存空间。

完成对象数据成员的初始化（函数体内的工作，由程序员完成）。

上述3点也说明了构造函数的执行顺序，在执行函数体之前，构造函数已经为对象的数据成员开辟了内存空间，这时，在函数体内对数据成员的初始化便是顺理成章了。构造函数有如下特点：

（1）如果用户没有显示定义构造函数，编译器自动生成的默认构造函数是无参的；

（2）实际上，构造函数可以接收参数，也可置默认参数，但需遵守相应规则，同样需要注意二义性的问题；

（3）一旦用户定义了构造函数，系统便不再提供默认构造函数。如果还想使用无参的构造函数，必须在类定义中显式定义一个无参构造函数；

（4）构造函数支持重载，在创建对象时，根据传递的具体参数决定采用哪个构造函数；

（5）构造函数是特殊的成员函数，也可以在类内定义或类内声明类外实现；

（6）大多为公有；

（7）构造函数由对象创建时，系统根据参数自动调用,虽然可以由用户显式定义但不能由用户主动调用。

构造函数的形式：

默认构造函数：

类名（）

{

}

自定义构造函数：

类名（参数表）

{

函数体；

}

调用构造函数定义对象：

类名 对象名（参数表）

#### 8.1.2.初始化表达式

除了在构造函数体内用赋值语句初始化数据成员外，还可以通过成员初始化表达式来完成。

成员初始化表可用于初始化类的任意数据成员（后面要介绍的static数据成员除外），该表达式由逗号分隔的数据成员表组成，初值放在一对圆括号中。将成员初始化表达式放在构造函数的头和体之间，并用冒号将其与构造函数的头分隔开，便可实现数据成员表中元素的初始化。

类名（参数表）：数据成员1（初始值）, 数据成员2（初始值），……

{

函数体；

}

每个成员在初始化表中只能出现一次。初始化的顺序不是由数据成员在初始化表中的顺序决定的，而是由数据成员在类中被申明时的顺序决定的。初始化成员列表先执行，构造函数体中的赋值语句后执行。

注意：虽然大多数情况下使用赋值语句也可以达到初始化表达式的效果，但在初始化某些特殊的数据成员时却只能使用初始化成员列表。如：引用类型的数据成员。

初始化表达式只能用于构造函数

### 8.2.析构函数

构造函数在创建对象时被系统自动调用，而析构函数在对象被撤销时被自动调用，相比构造函数，析构函数要简单的多。析构函数有如下特点：

（1）与类同名，之前冠以波浪号，以区别于构造函数。

（2）析构函数没有返回类型，也不能指定参数。

（3）析构函数只能有一个，不能被重载。

（4）析构函数也是特殊成员函数，可以在类内定义或者类内声明类外实现；

（5）如果用户没有显式地定义析构函数，编译器将为类生成“缺省析构函数”，缺省析构函数是个空的函数体，只清除类的数据成员所占据的空间，但对类的数据成员通过new和malloc动态申请的内存无能为力，因此，对于动态申请的内存，应在类的析构函数中通过delete或free进行释放，这样能有效避免对象撤销造成的内存泄漏。

对象超出其作用域被销毁时，析构函数会被自动调用。析构函数的形式如下：

~类名（）

{

函数体

}

**显示调用析构函数**

程序员不能显式调用构造函数，但却可以调用析构函数控制对象的撤销，释放对象所占据的内存空间，以更高效地利用内存。虽然可以显式调用析构函数，但不推荐这样做，因为可能带来重复释放指针类型数据成员所指向的内存等问题。

### 8.3构造函数和析构函数的调用时机

对于全局定义的对象，每当程序开始运行，在主函数main(或WinMain)接受程序控制权之前，就调用全局对象的构造函数。整个程序结束时调用析构函数。

对于局部定义的对象，每当程序流程到达该对象的定义处就调用构造函数，在程序离开局部对象的作用域时调用对象的析构函数。（栈对象）

对于关键字static定义的静态局部对象，当程序流程第一次到达该对象定义处调用构造函数，在整个程序结束时调用析构函数。

对于用new运算符创建的对象，每当创建该对象时调用构造函数，当用delete删除该对象时，调用析构函数。（堆对象）

## 9.拷贝构造函数

C++中经常使用一个常量或变量初始化另一个变量，象简单变量的初始化一样，也可以直接用一个对象来初始化另一个对象呢？答案是肯定的，以point类为例：

point pt1(2,3); point pt2=pt1;或point pt2( pt1);

上述语句用pt1初始化pt2，相当于将pt1中每个数据成员的值复制到pt2中，这是表面现象。实际上，系统调用了一个拷贝构造函数。如果类定义中没有显式定义该拷贝构造函数时，编译器会隐式定义一个缺省的拷贝构造函数，它是一个inline、public的成员函数，缺省的拷贝构造函数原型形式为：

类名：：类名（const 类名 &）

{

}

如：point:: point (const point &);

注意：如果不加&则调用构造函数时候会无穷的调用自身，直到函数栈满。推荐加上const,以避免右值为常量时不能满足c++的规则。

**拷贝构造函数在以下三种情况会自动调用**：

（1）当用一个已经存在的对象初始化另一个新的对象时。

（2）当实参和形参都是对象，进行形参和实参的结合时。

（3）当函数的返回值是对象，在函数调用完成时。（在将函数返回值给另一个新对象之前）

默认的拷贝构造函数并非万金油，因为其为浅拷贝（拷贝时只是简单的赋值操作，而没有开辟新空间），在一些情况下，必须由程序员显式定义缺省拷贝构造函数，先来看一段错误代码示例：

其中语句computer comp2(comp1)等价于：

comp2.brand = comp1.brand;

comp2.price = comp1.price;

后一句没有问题，但comp2.brand = comp1.brand却有问题：经过这样赋值后，两个对象的brand指针都指向了同一块内存，当两个对象释放时，其析构函数都要delete[]同一内存块，便造成了2次delete[]，从而引发了错误。

如果类中含有指针型的数据成员、需要使用动态内存，程序员最好显式定义自己的拷贝构造函数（也就是要深拷贝），避免各种可能出现的内存错误，类外实现如下：

类名：：类名（const 类名 &引用名）

{

函数体

}

computer(const computer &cp) //自定义拷贝构造函数

{

//重新为brand开辟和cp.brand同等大小的动态内存

brand = new char[strlen(cp.brand) + 1];

strcpy(brand, cp.brand); //字符串复制

price = cp.price;

}

拷贝构造函数可以看成是一种特殊的构造函数，这里姑且区分为“拷贝构造函数”和“普通构造函数”，因此，它也支持初始化表达式。

创建对象时，只有一个构造函数会被系统自动调用，具体调用哪个取决于创建对象时的参数和调用方式。C++对编译器何时提供缺省构造函数和缺省拷贝构造函数有着独特的规定，如表8.1所示。

拷贝构造函数也支持类内定义或者类内声明类外实现。

拷贝构造函数只有一个。



## 10.运算符函数

### 10.1.运算符函数

运算符函数与普通成员函数类似：

（1）在类内定义运算符函数使其成为inline型，格式为：

class 类名

{

……

返回类型 operator 运算符（参数列表）

{

……

}

……

}

（2）在类内声明，类外实现，但要使用作用域限定符“::”，类外定义的基本格式为：

class 类名

{

……

返回类型 operator 运算符（参数列表）；

……

}

返回类型 类名::operator 运算符（参数列表）

{

…

}

### 10.2赋值运算符函数

赋值运算是一种很常见的运算，如果不重载赋值运算符，编译器会自动为每个类生成一个缺省的赋值运算符重载函数，如下面的语句：

对象1＝对象2;

赋值运算符函数也是特殊的成员函数，可在类内定义或者类内声明类外实现，格式为：

class 类名

{

……

返回类型 & operator 运算符（const 类名 &引用名）

{

……

}

……

}

赋值运算符函数中：

首先要判断是否是自复制（ 对象1=对象1），判断的条件时this=&ref？如果相等则返回this指针的对象，如果不等且有指针成员时，要进行深拷贝（先释放目标对象的指针成员，再分配空间，最后赋值）。

实际上是完成了由对象2各个成员到对象1相应成员的复制，包括指针成员，这和拷贝构造函数有些类似，如果对象1中含指针成员，并且牵扯到类内指针成员动态申请内存时，问题就会出现。

将已存在的对象赋值给另一个已存在对象，调用赋值运算符函数

## 11.构造函数调用示例

类名 对象1＝对象2; //用已存在对象初始化新对象时，用拷贝构造函数

类名 对象1; 创建对象时，调用默认构造函数

类名 对象1（参数）；//用特定参数创建对象，调用自定义构造函数

类名 名称（）； //返回值为对象的函数声明

对象1＝对象2; //将已存在的对象赋值给另一个已存在对象，调用赋值运算符函数

**示例：**

CTest t0(); //这是函数的声明，不是实例化类

CTest t1; //缺省构造函数

CTest t2(1); //一个参数的构造函数

CTest t3(1, 2); //两个参数的构造函数

CTest t4 = 1; //等价于CTest t4(1);先用1生成一个对象，再用该对象去初始化t4，在每个函数前用explicit关键字禁止隐形转换

CTest t5 = t1; //CTest(t1);

CTest t6 = CTest(); //CTest(1); CTest(1,2);

t6 = CTest(1);

t6 = 1; //首先调用单个参数的构造函数，生成临时对象CTest(1), 然后调用赋值运算符函数

t6 = t1; //调用赋值运算符函数

## 12.this指针

一个类的所有对象共用成员函数代码段，不管有多少个对象，每个成员函数在内存中只有一个版本，那编译器是如何知道是哪个对象在执行操作呢，答案就是“this指针”。对于类的成员函数而言，并不是一个对象对应一个单独的成员函数体，而是此类的所有对象共用这个成员函数体。 当程序被编译之后，此成员函数地址即已确定。而成员函数之所以能把属于此类的各个对象的数据区别开, 就是靠这个this指针。函数体内所有对类数据成员的访问， 都会被转化为this->数据成员的方式。

一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。this作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使你没有写上this指针，编译器在编译的时候也是加上this的，它作为非静态成员函数的隐含形参，对各成员的访问均通过this进行。

this指针是类的一个自动生成、自动隐藏的私有成员，它存在于类的非静态成员函数中，指向被调用函数所在的对象。全局仅有一个this指针，当一个对象被创建时，this指针就存放指向对象数据的首地址。

在 C++ 中，每一个对象都能通过 this 指针来访问自己的地址。this 指针是所有成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。

友元函数没有 this 指针，因为友元不是类的成员。只有成员函数才有 this 指针。

使用this指针的情况：

一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 return \*this；另外一种情况是当参数与成员变量名相同时使用this指针，如this->n = n （不能写成n = n）。

## 13.特殊数据成员的初始化

### 13.1.const数据成员的初始化

数据成员可以由const修饰，这样，一经初始化，该数据成员便具有“只读属性”，在程序中无法对其值修改。

事实上，在构造函数体内或复制构造函数体内初始化const数据成员是非法的， const数据成员只能通过成员初始化表进行初始化

### 13.2.引用类型数据成员的初始化

对于引用类型的数据成员，同样只能通过成员初始化表达式进行初始化。

### 13.3. 类对象成员的初始化

类数据成员也可以是另一个类的对象，比如，一个直线类对象中包含两个point类对象，在直线类对象创建时可以在初始化列表中初始化两个point对象。

### 13.4. static数据成员的初始化

C++允许使用static（静态存储）修饰数据成员，这样的成员在编译时就被创建并初始化的（与之相比，对象是在运行时被创建的），且其实例只有一个，被所有该类的对象共享，就像住在同一宿舍里的同学共享一个房间号一样。静态数据成员和第6章中介绍的静态变量一样，程序执行时，该成员已经存在，一直到程序结束，任何对象都可对其进行访问。

静态数据成员的初始化必须在类定义之外进行，且不再包含static关键字，格式如下：

类型 类名::变量名 = 初始化表达式; //普通变量

类型 类名::对象名(构造参数); //对象变量

如float computer::total\_price = 0;

## 14.特殊成员函数

除了构造函数、拷贝构造函数和析构函数外，其他成员函数被用来提供特定的功能，一般来说，提供给外部访问的函数称为接口，访问权限为public，而一些不供外部访问，仅仅作为内部功能实现的函数，访问权限设为private。本节主要讨论函数成员的一些特殊用法。

### 14.1.static成员函数

成员函数也可以定义成静态的，与静态成员变量一样，系统对每个类只建立一个函数实体，该实体为该类的所有对象共享。

静态成员函数体内不能使用非静态的成员变量和非静态的成员函数；只能调用静态成员数据和静态成员函数。

静态成员函数的参数列表中不含有this指针

### 14.2. const成员函数

前面已经介绍了const在函数中的应用，实际上，const在类成员函数中还有种特殊的用法，把const关键字放在函数的参数表和函数体之间（与放在函数前修饰返回值不同），称为const成员函数，其特点有二：

只能读取类数据成员，而不能修改之

只能调用const成员函数，不能调用非const成员函数

如Point类中的打印函数：

void print(Point \* const this)

void print() const

void print(const Point \* const this) const

其基本定义格式为：

（1）类内定义时：

类型 函数名(参数列表) const

{

函数体

}

（2）类外定义时，共分两步：

类内声明：

类型 函数名(参数列表) const；

类外定义

类型 类名::函数名(参数列表) const

{

函数体

}

## 15. 对象的组织

有了自己定义的类，或者使用别人定义好的类创建对象，其机制与使用int等创建普通变量几乎完全一致，同样可以const对象、创建指向对象的指针，创建对象数组，还可使用new和delete等创建动态对象。

### 15.1.const对象

类对象也可以声明为const对象，一般来说，能作用于const对象的成员函数除了构造函数和析构函数，便只有const成员函数了，因为const对象只能被创建、撤销以及只读访问，改写是不允许的。

### 15.2.指向对象的指针

对象占据一定的内存空间，和普通变量一致，C++程序中采用如下形式声明指向对象的指针：

类名\* 指针名 [=初始化表达式];

初始化表达式是可选的，既可以通过取地址（&对象名）给指针初始化，也可以通过申请动态内存给指针初始化，或者干脆不初始化（比如置为NULL），在程序中再对该指针赋值。

指针中存储的是对象所占内存空间的首地址。

定义point类如备注中的代码

针对上述定义，则下列形式都是合法的：

point pt; //默认构造函数

point \*ptr = NULL; //空指针

point \*ptr = &pt; //取某个对象的地址

point \*ptr = new point(1, 2.2);//动态分配内存并初始化

point \*ptr = new point[5]; //动态分配5个对象的数组空间

使用指针对象：

ptr->print();(\*ptr).print();都是合法的。

### 15.3.对象的大小(sizeof)

对象占据一定大小的内存空间。总的来说，对象在内存中是以结构形式（只包括非static数据成员）存储在数据段或堆中，类对象的大小（sizeof）不一定是类中所有非static成员的大小之和，这和结构体一样涉及内存对齐的问题。在程序编译期间，就已经为static变量在静态存储区域分配了内存空间，并且这块内存在程序的整个运行期间都存在。而类中的成员函数存在于代码段中，不管多少个对象都只有一个副本。

计算对象大小有一些特殊之处需要强调：

C++将类中的引用成员当成“指针”来维护，占据4个内存字节。

如果类中有虚函数(后面课程将会介绍)时，虚析构函数除外，还会额外分配一个指针用来指向虚函数表（vtable），因此，这个时候对象的大小还要加4。

指针成员和引用成员要根据系统的字长来确定。

就算类没有一个成员（空类），其大小也不为0。因为如果为0，则无法区分不同的空类。

### 15.4.对象数组

对象数组和标准类型数组的使用方法并没有什么不同，也有声明、初始化和使用3个步骤。

(1)对象数组的声明：

类名 数组名[对象个数];

这种格式会自动调用无参或所有参数都有缺省值的构造函数，类定义要符合该要求，否则编译报错。

(2)对象数组的初始化：可以在声明时初始化。

对于point(int ix,int iy){}这种没有缺省参数值的构造函数：

point pt[2]={point(1,2), point(3,4)}; //#1 正确

point pt[ ]={point(1,2), point(3,4)}; //#2 正确

point pt[5]={point(1,2), point(3,4)}; //#3 错误

语句#1和#2是正确的，但语句#3错误，因为pt的后3个元素会自动调用无参的或者所有参数都有缺省值的构造函数，但这样的构造函数不存在。

### 15.5. 为对象动态分配内存

和把一个简单变量创建在动态存储区一样，可以用new和delete为对象分配动态存储区，在拷贝构造函数一节中已经介绍了为类内的指针成员分配动态内存的相关范例，本节主要讨论如何为对象和对象数组动态分配内存。

#### 15.5.1为单个对象申请和释放动态内存

int main()

{

point\* p = new point(4,5);

//动态申请一块内存，存储point类对象，并将地址赋值给point型指针p

p->print();//使用指针加－>调用成员函数

delete p; //释放动态申请的内存，防止内存泄露

p=NULL; //养成良好习惯，防止野指针

return 0;

}

此时，如果写了析构函数，并且是用new动态分配数组的内存空间的话，那么还会多开辟4个字节的内存空间，用来存放分配数组的大小。

#### 15.5.2.为对象数组申请和释放动态内存

使用new为对象数组分配动态空间时，不能显式调用对象的构造函数，因此，对象要么没有定义任何形式的构造函数（由编译器缺省提供），要么显式定义了一个（且只能有一个）所有参数都有缺省值的构造函数。

malloc/free无法满足动态对象的要求，因为malloc和free无法像new/delete及new/delete[]那样自动调用对象的构造函数和析构函数

## 16.单例设计模式（singleton)

单例设计模式：类在内存中只能有1个对象

实现步骤：

（1）将构造函数私有化。

（2）在类中定义一个静态的指针对象（可为私有，可为公有），并在类外初始化为空

（3）定义一个返回值为类指针的静态成员函数，如果2中的指针对象为空，则初始化对象，以后再有对象调用该静态成员函数的时候，不再初始化对象，而是直接返回对象，保证类在内存中只有一个实例。