如何让 new 操作符不分配内存，只调用构造函数

问题：c++中的 new 操作符　通常完成两个工作　分配内存及调用相应的构造出世核函数。
请问：
1）如何让 new 操作符不分配内存，只调用构造函数？
2) 这样的用法有什么用？

解答：（要求 new 显式调用构造函数，但不分配内存。）

题目要求不能生成内存 还要调用构造函数 说明这个类里面没有对内部操作 但可以对外部操作 比如 static 的数

摘录：如果我是用 new 分配对象的，可以显式调用析构函数吗？
可能不行。除非你使用定位放置 new.

```
class Fred
{public:
    Fred()
    {
        cout<<"fuck";
        }

    };
int main()
{

   Fred*f=new((void*)10000)Fred();
   system("pause");
}
```

其中这个 10000 可以是任意数，但不能为 0

2）
定位放置 new（placement new）有很多作用。最简单的用处就是将对象放置在内存中的特殊位置。这是依靠 new 表达式部分的指针参数的位置来完成的：

```
#include <new>       // 必须 #include 这个，才能使用 "placement new"
#include "Fred.h"    // class Fred 的声明

void someCode()
{
  char memory[sizeof(Fred)];    // Line #1
  void* place = memory;         // Line #2

  Fred* f = new(place) Fred();  // Line #3 (详见以下的“危险”)
```

```
  // The pointers f and place will be equal

  // ...
}
```

Line #1 在内存中创建了一个 sizeof(Fred)字节大小的数组，足够放下 Fred 对象。Line #2 创建了一个指向这块内存的首字节的 place 指针（有经验的 C 程序员会注意到这一步是多余的，这儿只是为了使代码更明显）。Line #3 本质上只是调用了构造函数 Fred::Fred()。Fred 构造函数中的 this 指针将等于 place。因此返回的 f 将等于 place。

Line #3 本质上只是调用了构造函数 Fred::Fred()。

****

placement new 的作用就是：创建对象但是不分配内存，而是在已有的内存块上面创建对象。

用于需要反复创建并删除的对象上，可以降低分配释放内存的性能消耗。

```
#include <iostream>
#include <new>

const intchunk = 16;
class Foo
{
public :
int val( ) { return _val; }
Foo( ) { _val = 0; }

private :
int_val;
};

//预分配内存，但没有 Foo 对象
char*buf = new char[ sizeof(Foo) * chunk ];

int
main( void )
{
//在 buf 中创建一个 Foo 对象
Foo*pb = new (buf) Foo;

//检查一个对象是否被放在 buf 中
if ( pb->val() == 0 )
{
cout <<"new expressio worked!" <<endl;
}

//到这里不能再使用 pb
delete[] buf;

return 0;
```

new 和 malloc 的区别

1、new 是 c++中的操作符，malloc 是 c 中的一个函数

2、new 不止是分配内存，而且会调用类的构造函数，同理 delete 会调用类的析构函数，而 malloc 则只分配内存，不会进行初始化类成员的工作，同样 free 也不会调用析构函数

3、内存泄漏对于 malloc 或者 new 都可以检查出来的，区别在于 new 可以指明是哪个文件的哪一行，而 malloc 没有这些信息。

4、new 和 malloc 效率比较 //总之，new 的效率要比 malloc 稍微低一些
new 有三个字母，malloc 有六个字母
new 可以认为是 malloc 加构造函数的执行。
new 出来的指针是直接带类型信息的。
而 malloc 返回的都是 void 指针。

//在这里，梦梦补充两点在网上查到的信息：
1.malloc 不会抛出异常，但是 new 会
2.你无法重新定义 malloc 失败时的默认行为（返回 null），但你可以重定义 new 失败时的默认行为，比如不让它抛出异常。

一：new delete 是运算符，malloc,free 是函数

//到这里，梦梦想，运算符和函数的区别在哪里呢？
//查了一下资料，没有具体的文章去解释这个问题，倒是论坛上的回答解答了这样的问题：
//运算符和函数，要说区别，大概也就是
// （1）语法形式上会有区别；
// （2）运算符只能重载，不能自定义，函数的名字随便你起，只要是个标志符就行；但运算符

//不行，比如，你无法仿照其它语言的符号，自己定义一个乘方运算符“**”。
// （3）任何函数都可以重载或者覆盖，但通常你不能改变运算符作用于内置类型的行为，比如

//你不能通过重载“operator+”，让 3 + 2 产生出 6 来。
//（from csdn：http://topic.csdn.net/t/20050901/17/4245022.html#）

malloc 与 free 是 C++/C 语言的标准库函数，new/delete 是 C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用 maloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。
因此 C++语言需要一个能完成动态内存分配和初始化工作的运算符 new，以及一个能完成清理与释放内存工作的运算符 delete。注意 new/delete 不是库函数。
我们先看一看 malloc/free 和 new/delete 如何实现对象的动态内存管理，见示例。

```
class Obj
{
         public :
                    Obj(void){ cout < < “Initialization” <<
endl; }                                                  //模拟构造函数的功
能
                    ~Obj(void){ cout < < “Destroy” <<
endl; }                                                    //模拟析构函
数的功能
                 void            Initialize(void){ cout < <
“Initialization” << endl; }
                 void            Destroy(void){ cout < < “Destroy” <<
endl; }
};

void UseMallocFree(void)
{
            Obj        *a = (obj *)malloc(sizeof(obj));        // 申请
动态内存
            a->Initialize();
                                          // 初始化
            //…
            a->Destroy();
                                       // 清除工作
            free(a);
                                                    // 释放内存
}

void UseNewDelete(void)
{
            Obj        *a = new
Obj;                                                                  // 申
```

请动态内存并且初始化
//…
delete
a;
// 清除并且释放内存
}

示例用 malloc/free 和 new/delete 如何实现对象的动态内存管理：
类 Obj 的函数 Initialize 模拟了构造函数的功能，函数 Destroy 模拟了析构函数的功能。函数 UseMallocFree 中，由于 malloc/free 不能执行构造函数与析构函数，必须调用成员函数 Initialize 和 Destroy 来完成初始化与清除工作。函数 UseNewDelete 则简单得多。

所以我们不要企图用 malloc/free 来完成动态对象的内存管理，应该用 new/delete。由于内部数据类型的“对象”没有构造与析构的过程，对它们而言 malloc/free 和 new/delete 是等价的。
//即对于内置数据类型来说，二者是等价的（但是梦梦认为这个时候还是 malloc/free 的效率更好一点）。但是对于非内置数据类型，我们只能选择 new/delete 。

既然 new/delete 的功能完全覆盖了 malloc/free，为什么 C++不把 malloc/free 淘汰出局呢？
这是因为 C++程序经常要调用 C 函数，而 C 程序只能用 malloc/free 管理动态内存。
如果用 free 释放“new 创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用 delete 释放“malloc 申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差。所以 new/delete 必须配对使用，malloc/free 也一样。
//即出于兼容性的考虑，C++里保留了 C 语言的 malloc/free 函数

二：new delete 在实现上其实调用了 malloc,free 函数。

三：new operator 除了分配内存，还要调用构造函数。
malloc 函数只是负责分配内存。

new 一维数组
XXX *arr;
int len; // 动态确定该长度值

arr = new XXX[len]; // 动态分配，也可以使用 malloc
...
delete[] arr; //不要忘记释放

new 多维数组　　//话说这部分资料梦梦昨天找了一天，原来姐姐那里有，汗

正确的做法是先声明一个 n 维数组，每个单元是指向 char（DATA_TYPE）的指针，再分别对每个单元分配内存.代码如下

```
char **array=new char*[n];
for(int i=0;i<n;i++)    array=new char[m];
```

注意：上面代码在释放分配的内存时要特别注意。因为这是“深度内存分配”，所以释放时，要对每个单元里的指针指向的内存予以释放。释放内存代码如下：

```
for(int i=0;i<n;i++)            delete[] array;
delete[] *array;
```

在 C++语言中，我们经常会使用 new 给一个对象分配内存空间，而当内存不够会出现内存不足的情况。C++提供了两中报告方式：

1、抛出 bad_alloc 异常来报告分配失败；

2、返回空指针，而不会抛出异常。

C++为什么会采用这两种方式呢？这主要是由于各大编译器公司设计 C++编译器公司的结果，因为标准 C++是提供了异常机制的。例如，VC++6.0 中当 new 分配内存失败时会返回空指针，而不会抛出异常。而 gcc 的编译器对于 C++标准支持比较好，所以当 new 分配内存失败时会抛出异常。

究竟为什么会出现这种情况呢？

首先，C++是在 C 语言的基础之上发展而来，而且 C++发明时是想尽可能的与 C 语言兼容。而 C 语言是一种没有异常机制的语言，所以 C++应该会提供一种没有异常机制的 new 分配内存失败报告机制；（确实是如此，早期的 C++还没有加入异常机制）

其次在返回空指针的实现过程中，C++采用的是 malloc/calloc 等分配内存的函数，该类函数不会抛出异常，但是在分配内存失败时会返回“空指针”。

最后，对于标准的 C++，有着比较完善的异常处理机制，所以对于出现异常时，会抛出响应的异常。对于 new 分配失败时，系统会抛出 bad_alloc 异常。

鉴于以上原因，我们在不同的编译器需要 new 分配失败时做不同的处理。例如：

情况 1：

```
int* p = new int(5);
if ( p == 0 ) // 检查 p 是否空指针
    return -1;
```

```
...
```

情况 2：

```
try {
    int* p = new int(5);
     // 其它代码
} catch ( const bad_alloc& e ) {
    return -1;
}
```

情况 1 和情况 2 的代码都是对于 new 失败时的处理，而针对不同的编译器，可以这种处理会完全失效。如果在 gcc 编译器采用情况 1,那么 if(p==0)完全是没有意义的，因为不管 new 内存分配成功失败与否，都不会出现 p＝0 的情况。即，如果分配成功，p＝0 完全不可能；而分配失败，new 会抛出异常跳过其后面的代码。而需要采用情况 2 的处理方式，即应该来捕捉异常。

同样，如果在 VC++6.0 中采用情况 2 的代码，那么 new 分配失败时，完全不会抛出异常，那么捕捉异常也是徒劳的。

所以在 new 分配内存的异常处理时要特别小心，可以两种方式联合使用，来解决跨平台跨编译器的难题。

当然情况 2 中的异常处理代码是最简单的处理方式，下面我们为其制定一个客户制定的错误处理函数，即 new－handler。

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw();
```

这里首先定义 new－handler 函数指针类型，然后定义一个 new－handler 函数 set_new_handler，其参数是指向 operator new 无法分配足够内存时应该被调用的函数。其返回指也是一个指针，指向 set_new_handler 被调用前正在执行（但是马上就要被替换）的那个 new－handler 函数。下面设计一个当 operator new 无法分配足够内存时应该被调用的函数：

```
void noMemoryToAlloc()
{
    std::cerr << "unable to satisfy request for memory\n";
     std::abort();
```

```
}
```

使用 noMemoryToAlloc 函数的代码为：

```
int main()
{
    set_new_handler(nomorememory);
    int *pArray = new int[100000000000000L];
...
}
```

当 operator new 无法分配足够空间时，noMemoryToAlloc 就会被调用，于是程序就会发出一个错误信息 cerr 之后，调用 abort 函数结束程序。

如果 operator new 无法分配足够空间时，我们希望不断调用 new－handler 函数，直到找到足够内存为止，那么我们的 operator new 函数就可以设计为：

```
void *operator new(std::size_t size) throw(std::bad_alloc)
{
    if ( size==0 ) {
        size = 1;
    }
    while (true) {
        调用 malloc 等内存分配函数来尝试分配 size 大小的内存；
        if ( 分配成功 )
            return 指向分配得来的内存指针；
        new_handler globalHandler = set_new_handler(0);
        set_new_handle(globalHandler);
        if(globalHandler)
            (*globalHandler)();
        else
            throw std::bad_alloc();
    }
}
```

转自：http://blog.sina.com.cn/s/blog_9f1c09310101953s.html

使用 new 分配内存失败时往往会使用 asert()终止程序，但是这只能在除错模式下 abert 函数才能有效，在生产模式下，abert 只是一个 void 指令，所以连程序都跳不出来。

而当 new 操作失败时，一个好的程序不能简单的终止程序就行了，而是要尝试去释放内存

如何能在 new 操作失败，在抛出异常之前先把相应的处理做了呢？这就要用到 new_handler 了，它是在抛出 exception 调用的。为了指定这个所谓的“内存不足处理函数，new_handler”，client 必须调用 set_new_handler，这是头文件提供的函数，用法：

```
typedef void  (*new_handler) ()
new_handler set_new_handler(new_handler p) throw();
```

简单的程序测试：

```
#include "stdafx.h"
#include
#include
using namespace std;

void NoMoreMemory()
{
cout<<"Unable to statisty request for memory"<<endl;
abort();
}

int _tmain(int argc, _TCHAR* argv[])
{
set_new_handler(NoMoreMemory);
int *p=new int[536870911];
return 0;
}
```

注：当 operator new 无法满足内存需求时，它会不只一次地调用 new_handler 函数（如果 new_handler 没有退出程序的话）；它会不断地调用，直到找到足够的内存为止。因此一个良好设计的 new_handler 函数必须完成下列事情之一：

1）让更多的内存可用。这或许能够让 operator new 的下一次内存配置行为成功。实现此策略的方法之一就是在程序起始时配置一大块内存，然后在 new_handler 第一次被调用时释放之。如此的释放动作常常伴随着某种警告信息，告诉用户目前的内存已经处于低水位，再来的内存需求可能失败，除非有更多的内存回复自由身。

2）安装一个不同的 new_handler，如果目前的 new_handler 无法让更多的内存可用，或许它知道另一个 new_handler 手上我有比较多的资源。果真如此，目前的 new_handler 就可以安装另外一个 new_handler 以取代自己（只要再调用一次 set_new_handler 即可）。当 operator new 下次调用 new_handler 函数时，它会调用最新安装的那个。

3）卸除掉这个 new_handler，也就是说，将 null 指针传给 set_new_handler，一旦没有安装任何 new_handler，operator new 就会在内存配置失败时抛出一个类型为 std::bad_alloc 的 exception。
4）抛出一个 exception，类型为 std::bad_alloc(或者派生类型)。这样的 exception 不会被 operator new 捕获，所以它们就会传动到最初踢出内存需求的那个点上
5）不返回，直接调用 abort 或 exit