C++内存管理

内存管理是C++最令人切齿痛恨的问题，也是C++最有争议的问题，C++高手从中获得了更好的性能，更大的自由，C++菜鸟的收获则是一遍一遍的检查代码和对C++的痛恨，但内存管理在C++中无处不在，内存泄漏几乎在每个C++程序中都会发生，因此要想成为C++高手，内存管理一关是必须要过的，除非放弃C++，转到Java或者.NET，他们的内存管理基本是自动的，当然你也放弃了自由和对内存的支配权，还放弃了C++超绝的性能。本期专题将从内存管理、内存泄漏、内存回收这三个方面来探讨C++内存管理问题。

1 内存管理

伟大的 Bill Gates 曾经失言：

640K ought to be enough for everybody — Bill Gates 1981

程序员们经常编写内存管理程序，往往提心吊胆。如果不想触雷，唯一的解决办法就是发现所有潜伏的地雷并且排除它们，躲是躲不了的。本文的内容比一般教科书的要深入得多，读者需细心阅读，做到真正地通晓内存管理。

1.1 C++内存管理详解

1.1.1 内存分配方式

1.1.1.1 分配方式简介

在 C++中，内存分成 5 个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。

栈，在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

堆，就是那些由 new 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

自由存储区，就是那些由 malloc 等分配的内存块，他和堆是十分相似的，不过它是用 free 来结束自己的生命的。

全局/静态存储区，全局变量和静态变量被分配到同一块内存中，在以前的 C 语言中，全局变量又分为初始化的和未初始化的，在 C++里面没有这个区分了，他们共同占用同一块内存区。

常量存储区，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改。

1.1.1.2 明确区分堆与栈

在 bbs 上，堆与栈的区分问题，似乎是一个永恒的话题，由此可见，初学者对此往往是混淆不清的，所以我决定拿他第一个开刀。

首先，我们举一个例子：

```
void f() { int* p=new int[5]; }
```

这条短短的一句话就包含了堆与栈，看到 new，我们首先就应该想到，我们分配了一块堆内存，那么指针 p 呢？他分配的是一块栈内存，所以这句话的意思就是：在栈内存中存放了一个指向一块堆内存的指针 p。在程序会先确定在堆中分配内存的大小，然后调用 operator new 分配内存，然后返回这块内存的首地址，放入栈中，他在 VC6 下的汇编代码如下：

```
00401028 push 14h

0040102A call operator new (00401060)

0040102F add esp,4

00401032 mov dword ptr [ebp-8],eax

00401035 mov eax,dword ptr [ebp-8]

00401038 mov dword ptr [ebp-4],eax
```

这里，我们为了简单并没有释放内存，那么该怎么去释放呢？是 delete p 么？澳，错了，应该是 delete []p，这是为了告诉编译器：我删除的是一个数组，VC6 就会根据相应的 Cookie 信息去进行释放内存的工作。

1.1.1.3 堆和栈究竟有什么区别？

好了，我们回到我们的主题：堆和栈究竟有什么区别？

主要的区别由以下几点：

1、管理方式不同；

2、空间大小不同；

3、能否产生碎片不同；

4、生长方向不同；

5、分配方式不同；

6、分配效率不同；

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。

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（好像是，记不清楚了）。当然，我们可以修改：

打开工程，依次操作菜单如下：Project->Setting->Link，在 Category 中选中 Output，然后在 Reserve 中设定堆栈的最大值和 commit。

注意：reserve 最小值为 4Byte；commit 是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。

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队列，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，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构，这里我们就不再一一讨论了。

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。

分配方式：堆都是动态分配的，没有静态分配的堆。栈有 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 alloca 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。

分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是 C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去

增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

从这里我们可以看到，堆和栈相比，由于大量 new/delete 的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址，EBP 和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。

虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。

无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果,就算是在你的程序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候 debug 可是相当困难的：）

1.1.2 控制 C++的内存分配

在嵌入式系统中使用 C++的一个常见问题是内存分配，即对 new 和 delete 操作符的失控。

具有讽刺意味的是，问题的根源却是 C++对内存的管理非常的容易而且安全。具体地说，当一个对象被消除时，它的析构函数能够安全的释放所分配的内存。

这当然是个好事情，但是这种使用的简单性使得程序员们过度使用 new 和 delete，而不注意在嵌入式 C++环境中的因果关系。并且，在嵌入式系统中，由于内存的限制，频繁的动态分配不定大小的内存会引起很大的问题以及堆破碎的风险。

作为忠告，保守的使用内存分配是嵌入式环境中的第一原则。

但当你必须要使用 new 和 delete 时，你不得不控制 C++中的内存分配。你需要用一个全局的 new 和 delete 来代替系统的内存分配符，并且一个类一个类的重载 new 和 delete。

一个防止堆破碎的通用方法是从不同固定大小的内存持中分配不同类型的对象。对每个类重载 new 和 delete 就提供了这样的控制。

1.1.2.1 重载全局的 new 和 delete 操作符

可以很容易地重载 new 和 delete 操作符，如下所示:

```
void * operator new(size_t size)

{

void *p = malloc(size);

return (p);

}

void operator delete(void *p);

{

free(p);

}
```

这段代码可以代替默认的操作符来满足内存分配的请求。出于解释 C++的目的，我们也可以直接调用 malloc() 和 free()。

也可以对单个类的 new 和 delete 操作符重载。这是你能灵活的控制对象的内存分配。

```
class TestClass {

public:

void * operator new(size_t size);

void operator delete(void *p);

// .. other members here ...

};

void *TestClass::operator new(size_t size)

{
```

```
void *p = malloc(size); // Replace this with alternative allocator

return (p);

}

void TestClass::operator delete(void *p)

{

free(p); // Replace this with alternative de-allocator

}
```

所有 TestClass 对象的内存分配都采用这段代码。更进一步，任何从 TestClass 继承的类也都采用这一方式，除非它自己也重载了 new 和 delete 操作符。通过重载 new 和 delete 操作符的方法，你可以自由地采用不同的分配策略，从不同的内存池中分配不同的类对象。

1.1.2.2 为单个的类重载 new[ ]和 delete[ ]

必须小心对象数组的分配。你可能希望调用到被你重载过的 new 和 delete 操作符，但并不如此。内存的请求被定向到全局的 new[ ]和 delete[ ] 操作符，而这些内存来自于系统堆。

C++将对象数组的内存分配作为一个单独的操作，而不同于单个对象的内存分配。为了改变这种方式，你同样需要重载 new[ ] 和 delete[ ]操作符。

```
class TestClass {

public:

void * operator new[ ](size_t size);

void operator delete[ ](void *p);

// .. other members here ..

};

void *TestClass::operator new[ ](size_t size)
```

```
{
void *p = malloc(size);
return (p);
}
void TestClass::operator delete[ ](void *p)
{
free(p);
}
int main(void)
{
TestClass *p = new TestClass[10];
// ... etc ...
delete[ ] p;
}
```

但是注意：对于多数 C++的实现，new[]操作符中的个数参数是数组的大小加上额外的存储对象数目的一些字节。在你的内存分配机制重要考虑的这一点。你应该尽量避免分配对象数组，从而使你的内存分配策略简单。

1.1.3 常见的内存错误及其对策

发生内存错误是件非常麻烦的事情。编译器不能自动发现这些错误，通常是在程序运行时才能捕捉到。而这些错误大多没有明显的症状，时隐时现，增加了改错的难度。有时用户怒气冲冲地把你找来，程序却没有发生任何问题，你一走，错误又发作了。 常见的内存错误及其对策如下：

* 内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为 NULL。如果指针 p 是函数的参数，那么在函数的入口处用 assert(p!=NULL)进行

检查。如果是用 malloc 或 new 来申请内存，应该用 if(p==NULL) 或 if(p!=NULL)进行防错处理。

* 内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。 内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

* 内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多 1”或者“少 1”的操作。特别是在 for 循环语句中，循环次数很容易搞错，导致数组操作越界。

* 忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然死掉，系统出现提示：内存耗尽。

动态内存的申请与释放必须配对，程序中 malloc 与 free 的使用次数一定要相同，否则肯定有错误（new/delete 同理）。

* 释放了内存却继续使用它。

有三种情况：

（1）程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。

（2）函数的 return 语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。

（3）使用 free 或 delete 释放了内存后，没有将指针设置为 NULL。导致产生“野指针”。

【规则 1】用 malloc 或 new 申请内存之后，应该立即检查指针值是否为 NULL。防止使用指针值为 NULL 的内存。

【规则 2】不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。

【规则 3】避免数组或指针的下标越界，特别要当心发生“多 1”或者“少 1”操作。

【规则 4】动态内存的申请与释放必须配对，防止内存泄漏。

【规则 5】用 free 或 delete 释放了内存之后，立即将指针设置为 NULL，防止产生“野指针”。

1.1.4 指针与数组的对比

C++/C 程序中，指针和数组在不少地方可以相互替换着用，让人产生一种错觉，以为两者是等价的。

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。数组名对应着（而不是指向）一块内存，其地址与容量在生命期内保持不变，只有数组的内容可以改变。

指针可以随时指向任意类型的内存块，它的特征是“可变”，所以我们常用指针来操作动态内存。指针远比数组灵活，但也更危险。

下面以字符串为例比较指针与数组的特性。

1.1.4.1 修改内容

下面示例中，字符数组 a 的容量是 6 个字符，其内容为 hello。a 的内容可以改变，如 a[0]= ‘X’。指针 p 指向常量字符串“world”（位于静态存储区，内容为 world），常量字符串的内容是不可以被修改的。从语法上看，编译器并不觉得语句 p[0]= ‘X’有什么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。

```
char a[] = “hello”;
```

```
a[0] = 'X';

cout << a << endl;

char *p = “world”; // 注意 p 指向常量字符串

p[0] = ‘X’; // 编译器不能发现该错误

cout << p << endl;
```

1.1.4.2 内容复制与比较

不能对数组名进行直接复制与比较。若想把数组 a 的内容复制给数组 b，不能用语句 b = a ，否则将产生编译错误。应该用标准库函数 strcpy 进行复制。同理，比较 b 和 a 的内容是否相同，不能用 if(b==a) 来判断，应该用标准库函数 strcmp 进行比较。

语句 p = a 并不能把 a 的内容复制指针 p，而是把 a 的地址赋给了 p。要想复制 a 的内容，可以先用库函数 malloc 为 p 申请一块容量为 strlen(a)+1 个字符的内存，再用 strcpy 进行字符串复制。同理，语句 if(p==a) 比较的不是内容而是地址，应该用库函数 strcmp 来比较。

```
// 数组…

char a[] = "hello";

char b[10];

strcpy(b, a); // 不能用 b = a;

if(strcmp(b, a) == 0) // 不能用 if (b == a)

…

// 指针…

int len = strlen(a);

char *p = (char *)malloc(sizeof(char)*(len+1));

strcpy(p,a); // 不要用 p = a;

if(strcmp(p, a) == 0) // 不要用 if (p == a)

…
```

1.1.4.3 计算内存容量

用运算符 sizeof 可以计算出数组的容量（字节数）。如下示例中，sizeof(a)的值是 12（注意别忘了’’）。指针 p 指向 a，但是 sizeof(p)的值却是 4。这是因为 sizeof(p)得到的是一个指针变量的字节数，相当于 sizeof(char*)，而不是 p 所指的内存容量。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。

```
char a[] = "hello world";

char *p = a;

cout<< sizeof(a) << endl; // 12 字节

cout<< sizeof(p) << endl; // 4 字节
```

注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。如下示例中，不论数组 a 的容量是多少，sizeof(a)始终等于 sizeof(char *)。

```
void Func(char a[100])

{

   cout<< sizeof(a) << endl; // 4 字节而不是 100 字节

}
```

1.1.5 指针参数是如何传递内存的？

如果函数的参数是一个指针，不要指望用该指针去申请动态内存。如下示例中，Test 函数的语句 GetMemory(str, 200)并没有使 str 获得期望的内存，str 依旧是 NULL，为什么？

```
void GetMemory(char *p, int num)

{

   p = (char *)malloc(sizeof(char) * num);

}

void Test(void)
```

```
{
    char *str = NULL;
    GetMemory(str, 100); // str 仍然为 NULL
    strcpy(str, "hello"); // 运行错误
}
```

毛病出在函数 GetMemory 中。编译器总是要为函数的每个参数制作临时副本，指针参数 p 的副本是 _p，编译器使 _p = p。如果函数体内的程序修改了_p 的内容，就导致参数 p 的内容作相应的修改。这就是指针可以用作输出参数的原因。在本例中，_p 申请了新的内存，只是把_p 所指的内存地址改变了，但是 p 丝毫未变。所以函数 GetMemory 并不能输出任何东西。事实上，每执行一次 GetMemory 就会泄露一块内存，因为没有用 free 释放内存。

如果非得要用指针参数去申请内存，那么应该改用“指向指针的指针”，见示例：

```
void GetMemory2(char **p, int num)
{
    *p = (char *)malloc(sizeof(char) * num);
}
void Test2(void)
{
    char *str = NULL;
    GetMemory2(&str, 100); // 注意参数是 &str，而不是 str
    strcpy(str, "hello");
    cout<< str << endl;
    free(str);
}
```

由于“指向指针的指针”这个概念不容易理解，我们可以用函数返回值来传递动态内存。这种方法更加简单，见示例：

```
char *GetMemory3(int num)

{

   char *p = (char *)malloc(sizeof(char) * num);

   return p;

}

void Test3(void)

{

   char *str = NULL;

   str = GetMemory3(100);

   strcpy(str, "hello");

   cout<< str << endl;

   free(str);

}
```

用函数返回值来传递动态内存这种方法虽然好用，但是常常有人把 return 语句用错了。这里强调不要用 return 语句返回指向“栈内存”的指针，因为该内存在函数结束时自动消亡，见示例：

```
char *GetString(void)

{

   char p[] = "hello world";

   return p; // 编译器将提出警告

}

void Test4(void)
```

```
{
    char *str = NULL;
    str = GetString(); // str 的内容是垃圾
    cout<< str << endl;
}
```

用调试器逐步跟踪 Test4，发现执行 str = GetString 语句后 str 不再是 NULL 指针，但是 str 的内容不是“hello world”而是垃圾。

如果把上述示例改写成如下示例，会怎么样？

```
char *GetString2(void)
{
    char *p = "hello world";
    return p;
}
void Test5(void)
{
    char *str = NULL;
    str = GetString2();
    cout<< str << endl;
}
```

函数 Test5 运行虽然不会出错，但是函数 GetString2 的设计概念却是错误的。因为 GetString2 内的“hello world”是常量字符串，位于静态存储区，它在程序生命期内恒定不变。无论什么时候调用 GetString2，它返回的始终是同一个“只读”的内存块。

1.1.6 杜绝“野指针”

“野指针”不是 NULL 指针，是指向“垃圾”内存的指针。人们一般不会错用 NULL 指针，因为用 if 语句很容易判断。但是“野指针”是很危险的，if 语句对它不起作用。
“野指针”的成因主要有两种：

（1）指针变量没有被初始化。任何指针变量刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为 NULL，要么让它指向合法的内存。例如

```
char *p = NULL;

char *str = (char *) malloc(100);
```

（2）指针 p 被 free 或者 delete 之后，没有置为 NULL，让人误以为 p 是个合法的指针。

（3）指针操作超越了变量的作用域范围。这种情况让人防不胜防，示例程序如下：

```
class A

{

   public:

      void Func(void){ cout << “Func of class A” << endl; }

};

void Test(void)

{

   A *p;

   {

      A a;

      p = &a; // 注意 a 的生命期

   }

   p->Func(); // p 是“野指针”

}
```

函数 Test 在执行语句 p->Func()时，对象 a 已经消失，而 p 是指向 a 的，所以 p 就成了“野指针”。但奇怪的是我运行这个程序时居然没有出错，这可能与编译器有关。

1.1.7 有了 malloc/free 为什么还要 new/delete？

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; }

      ~Obj(void){ cout << “Destroy” << endl; }

      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
}
```

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。

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”，结果也会导致程序出错，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。

1.1.8 内存耗尽怎么办？

如果在申请动态内存时找不到足够大的内存块，malloc 和 new 将返回 NULL 指针，宣告内存申请失败。通常有三种方式处理“内存耗尽”问题。

（1）判断指针是否为 NULL，如果是则马上用 return 语句终止本函数。例如：

```
void Func(void)
{
   A *a = new A;
   if(a == NULL)
   {
      return;
   }
   …
}
```

（2）判断指针是否为 NULL，如果是则马上用 exit(1)终止整个程序的运行。例如：

```
void Func(void)
{
   A *a = new A;
   if(a == NULL)
   {
      cout << “Memory Exhausted” << endl;
      exit(1);
   }
   …
}
```

（3）为 new 和 malloc 设置异常处理函数。例如 Visual C++可以用_set_new_hander 函数为 new 设置用户自己定义的异常处理函数，也可以让 malloc 享用与 new 相同的异常处理函数。详细内容请参考 C++使用手册。

上述（1）（2）方式使用最普遍。如果一个函数内有多处需要申请动态内存，那么方式（1）就显得力不从心（释放内存很麻烦），应该用方式（2）来处理。

很多人不忍心用 exit(1)，问：“不编写出错处理程序，让操作系统自己解决行不行？”

不行。如果发生“内存耗尽”这样的事情，一般说来应用程序已经无药可救。如果不用 exit(1) 把坏程序杀死，它可能会害死操作系统。道理如同：如果不把歹徒击毙，歹徒在老死之前会犯下更多的罪。

有一个很重要的现象要告诉大家。对于 32 位以上的应用程序而言，无论怎样使用 malloc 与 new，几乎不可能导致“内存耗尽”。我在 Windows 98 下用 Visual C++编写了测试程序，见示例 7。这个程序会无休止地运行下去，根本不会终止。因为 32 位操作系统支持“虚存”，内存用完了，自动用硬盘空间顶替。我只听到硬盘嘎吱嘎吱地响，Window 98 已经累得对键盘、鼠标毫无反应。

我可以得出这么一个结论：对于 32 位以上的应用程序，“内存耗尽”错误处理程序毫无用处。这下可把 Unix 和 Windows 程序员们乐坏了：反正错误处理程序不起作用，我就不写了，省了很多麻烦。

我不想误导读者，必须强调：不加错误处理将导致程序的质量很差，千万不可因小失大。

```
void main(void)
{
  float *p = NULL;
  while(TRUE)
  {
    p = new float[1000000];
    cout << “eat memory” << endl;
    if(p==NULL)
      exit(1);
```

```
  }

}
```

1.1.9 malloc/free 的使用要点

函数 malloc 的原型如下：

```
void * malloc(size_t size);
```

用 malloc 申请一块长度为 length 的整数类型的内存，程序如下：

```
int *p = (int *) malloc(sizeof(int) * length);
```

我们应当把注意力集中在两个要素上：“类型转换”和“sizeof”。

* malloc 返回值的类型是 void *，所以在调用 malloc 时要显式地进行类型转换，将 void * 转换成所需要的指针类型。

* malloc 函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。我们通常记不住 int, float 等数据类型的变量的确切字节数。例如 int 变量在 16 位系统下是 2 个字节，在 32 位下是 4 个字节；而 float 变量在 16 位系统下是 4 个字节，在 32 位下也是 4 个字节。最好用以下程序作一次测试：

```
cout << sizeof(char) << endl;

cout << sizeof(int) << endl;

cout << sizeof(unsigned int) << endl;

cout << sizeof(long) << endl;

cout << sizeof(unsigned long) << endl;

cout << sizeof(float) << endl;

cout << sizeof(double) << endl;

cout << sizeof(void *) << endl;
```

在 malloc 的“()”中使用 sizeof 运算符是良好的风格，但要当心有时我们会昏了头，写出 p = malloc(sizeof(p))这样的程序来。

函数 free 的原型如下：

```
void free( void * memblock );
```

为什么 free 函数不象 malloc 函数那样复杂呢？这是因为指针 p 的类型以及它所指的内存的容量事先都是知道的，语句 free(p)能正确地释放内存。如果 p 是 NULL 指针，那么 free 对 p 无论操作多少次都不会出问题。如果 p 不是 NULL 指针，那么 free 对 p 连续操作两次就会导致程序运行错误。

1.1.10 new/delete 的使用要点

运算符 new 使用起来要比函数 malloc 简单得多，例如：

```
int *p1 = (int *)malloc(sizeof(int) * length);

int *p2 = new int[length];
```

这是因为 new 内置了 sizeof、类型转换和类型安全检查功能。对于非内部数据类型的对象而言，new 在创建动态对象的同时完成了初始化工作。如果对象有多个构造函数，那么 new 的语句也可以有多种形式。例如

```
class Obj

{

   public :

      Obj(void); //  无参数的构造函数

      Obj(int x); //  带一个参数的构造函数

      ...

}

void Test(void)

{

   Obj *a = new Obj;

   Obj *b = new Obj(1); //  初值为 1
```

```
    ...

    delete a;

    delete b;

}
```

如果用 new 创建对象数组，那么只能使用对象的无参数构造函数。例如：

```
Obj *objects = new Obj[100]; //  创建 100 个动态对象
```

不能写成：

```
Obj *objects = new Obj[100](1);//  创建 100 个动态对象的同时赋初值 1
```

在用 delete 释放对象数组时，留意不要丢了符号‘[]’。例如：

```
delete []objects; //  正确的用法

delete objects; //  错误的用法
```

后者有可能引起程序崩溃和内存泄漏。

1.2 C++中的健壮指针和资源管理

我最喜欢的对资源的定义是："任何在你的程序中获得并在此后释放的东西?quot;内存是一个相当明显的资源的例子。它需要用 new 来获得，用 delete 来释放。同时也有许多其它类型的资源文件句柄、重要的片断、Windows 中的 GDI 资源，等等。将资源的概念推广到程序中创建、释放的所有对象也是十分方便的，无论对象是在堆中分配的还是在栈中或者是在全局作用于内生命的。

对于给定的资源的拥有着，是负责释放资源的一个对象或者是一段代码。所有权分立为两种级别——自动的和显式的（automatic and explicit），如果一个对象的释放是由语言本身的机制来保证的，这个对象的就是被自动地所有。例如，一个嵌入在其他对象中的对象，他的清除需要其他对象来在清除的时候保证。外面的对象被看作嵌入类的所有者。类似地，每个在栈上创建的对象（作为自动变量）的释放（破坏）是在控制流离开了对象被定义的作用域的时候保证的。这种情况下，作用于被看作是对象的所有者。注意所有的自动所有权都是和语言的其他机制相容的，包括异常。无论是如何退出作用域的——正常

流程控制退出、一个 break 语句、一个 return、一个 goto、或者是一个 throw——自动资源都可以被清除。

到目前为止，一切都很好！问题是在引入指针、句柄和抽象的时候产生的。如果通过一个指针访问一个对象的话，比如对象在堆中分配，C++不自动地关注它的释放。程序员必须明确的用适当的程序方法来释放这些资源。比如说，如果一个对象是通过调用 new 来创建的，它需要用 delete 来回收。一个文件是用 CreateFile(Win32 API)打开的，它需要用 CloseHandle 来关闭。用 EnterCritialSection 进入的临界区（Critical Section）需要 LeaveCriticalSection 退出，等等。一个"裸"指针，文件句柄，或者临界区状态没有所有者来确保它们的最终释放。基本的资源管理的前提就是确保每个资源都有他们的所有者。

1.2.1 第一条规则（RAII）

一个指针，一个句柄，一个临界区状态只有在我们将它们封装入对象的时候才会拥有所有者。这就是我们的第一规则：在构造函数中分配资源，在析构函数中释放资源。

当你按照规则将所有资源封装的时候，你可以保证你的程序中没有任何的资源泄露。这点在当封装对象（Encapsulating Object）在栈中建立或者嵌入在其他的对象中的时候非常明显。但是对那些动态申请的对象呢？不要急！任何动态申请的东西都被看作一种资源，并且要按照上面提到的方法进行封装。这一对象封装对象的链不得不在某个地方终止。它最终终止在最高级的所有者，自动的或者是静态的。这些分别是对离开作用域或者程序时释放资源的保证。

下面是资源封装的一个经典例子。在一个多线程的应用程序中，线程之间共享对象的问题是通过用这样一个对象联系临界区来解决的。每一个需要访问共享资源的客户需要获得临界区。例如，这可能是 Win32 下临界区的实现方法。

```
class CritSect
{
   friend class Lock;
   public:
      CritSect () { InitializeCriticalSection (&_critSection); }
      ~CritSect () { DeleteCriticalSection (&_critSection); }
```

```
  private:
    void Acquire ()
    {
      EnterCriticalSection (&_critSection);
    }
    void Release ()
    {
      LeaveCriticalSection (&_critSection);
    }
  private:
    CRITICAL_SECTION _critSection;
};
```

这里聪明的部分是我们确保每一个进入临界区的客户最后都可以离开。"进入"临界区的状态是一种资源，并应当被封装。封装器通常被称作一个锁（lock）。

```
class Lock
{
  public:
    Lock (CritSect& critSect) : _critSect (critSect)
    {
      _critSect.Acquire ();
    }
    ~Lock ()
    {
```

```
        _critSect.Release ();
    }
  private
    CritSect & _critSect;
};
```

锁一般的用法如下：

```
void Shared::Act () throw (char *)
{
  Lock lock (_critSect);
  // perform action —— may throw
  // automatic destructor of lock
}
```

注意无论发生什么，临界区都会借助于语言的机制保证释放。

还有一件需要记住的事情——每一种资源都需要被分别封装。这是因为资源分配是一个非常容易出错的操作，是要资源是有限提供的。我们会假设一个失败的资源分配会导致一个异常——事实上，这会经常的发生。所以如果你想试图用一个石头打两只鸟的话，或者在一个构造函数中申请两种形式的资源，你可能就会陷入麻烦。只要想想在一种资源分配成功但另一种失败抛出异常时会发生什么。因为构造函数还没有全部完成，析构函数不可能被调用，第一种资源就会发生泄露。

这种情况可以非常简单的避免。无论何时你有一个需要两种以上资源的类时，写两个小的封装器将它们嵌入你的类中。每一个嵌入的构造都可以保证删除，即使包装类没有构造完成。

1.2.2 Smart Pointers

我们至今还没有讨论最常见类型的资源——用操作符 new 分配，此后用指针访问的一个对象。我们需要为每个对象分别定义一个封装类吗？（事实上，C++标准模板库已经有了一个模板类，叫做 auto_ptr，其作用就是提供这种封装。我们一会儿在回到

auto_ptr。）让我们从一个极其简单、呆板但安全的东西开始。看下面的 Smart Pointer 模板类，它十分坚固，甚至无法实现。

```
template <class T>

class SmartPointer

{

   public:

      ~SmartPointer () { delete _p; }

      T * operator->() { return _p; }

      T const * operator->() const { return _p; }

   protected:

      SmartPointer (): _p (0) {}

      explicit SmartPointer (T* p): _p (p) {}

      T * _p;

};
```

为什么要把 SmartPointer 的构造函数设计为 protected 呢？如果我需要遵守第一条规则，那么我就必须这样做。资源——在这里是 class T 的一个对象——必须在封装器的构造函数中分配。但是我不能只简单的调用 new T，因为我不知道 T 的构造函数的参数。因为，在原则上，每一个 T 都有一个不同的构造函数；我需要为他定义个另外一个封装器。模板的用处会很大，为每一个新的类，我可以通过继承 SmartPointer 定义一个新的封装器，并且提供一个特定的构造函数。

```
class SmartItem: public SmartPointer<Item>

{

   public:

      explicit SmartItem (int i)

      : SmartPointer<Item> (new Item (i)) {}
```

```
};
```

为每一个类提供一个 Smart Pointer 真的值得吗？说实话——不！他很有教学的价值，但是一旦你学会如何遵循第一规则的话，你就可以放松规则并使用一些高级的技术。这一技术是让 SmartPointer 的构造函数成为 public，但是只是是用它来做资源转换（Resource Transfer）我的意思是用 new 操作符的结果直接作为 SmartPointer 的构造函数的参数，像这样：

```
SmartPointer<Item> item (new Item (i));
```

这个方法明显更需要自控性，不只是你，而且包括你的程序小组的每个成员。他们都必须发誓出了作资源转换外不把构造函数用在人以其他用途。幸运的是，这条规矩很容易得以加强。只需要在源文件中查找所有的 new 即可。

1.2.3 Resource Transfer

到目前为止，我们所讨论的一直是生命周期在一个单独的作用域内的资源。现在我们要解决一个困难的问题——如何在不同的作用域间安全的传递资源。这一问题在当你处理容器的时候会变得十分明显。你可以动态的创建一串对象，将它们存放至一个容器中，然后将它们取出，并且在最终安排它们。为了能够让这安全的工作——没有泄露——对象需要改变其所有者。

这个问题的一个非常显而易见的解决方法是使用 Smart Pointer，无论是在加入容器前还是还找到它们以后。这是他如何运作的，你加入 Release 方法到 Smart Pointer 中：

```
template <class T>

T * SmartPointer<T>::Release ()

{

T * pTmp = _p;

_p = 0;

return pTmp;

}
```

注意在 Release 调用以后，Smart Pointer 就不再是对象的所有者了——它内部的指针指向空。现在，调用了 Release 都必须是一个负责的人并且迅速隐藏返回的指针到新的所有者对象中。在我们的例子中，容器调用了 Release，比如这个 Stack 的例子：

```
void Stack::Push (SmartPointer <Item> & item) throw (char *)

{

if (_top == maxStack)

throw "Stack overflow";

_arr [_top++] = item.Release ();

};
```

同样的，你也可以再你的代码中用加强 Release 的可靠性。

相应的 Pop 方法要做些什么呢？他应该释放了资源并祈祷调用它的是一个负责的人而且立即作一个资源传递它到一个 Smart Pointer？这听起来并不好。

1.2.4 Strong Pointers

资源管理在内容索引（Windows NT Server 上的一部分，现在是 Windows 2000）上工作，并且，我对这十分满意。然后我开始想……这一方法是在这样一个完整的系统中形成的，如果可以把它内建入语言的本身岂不是一件非常好？我提出了强指针（Strong Pointer）和弱指针(Weak Pointer)。一个 Strong Pointer 会在许多地方和我们这个 SmartPointer 相似--它在超出它的作用域后会清除他所指向的对象。资源传递会以强指针赋值的形式进行。也可以有 Weak Pointer 存在，它们用来访问对象而不需要所有对象--比如可赋值的引用。

任何指针都必须声明为 Strong 或者 Weak，并且语言应该来关注类型转换的规定。例如，你不可以将 Weak Pointer 传递到一个需要 Strong Pointer 的地方，但是相反却可以。Push 方法可以接受一个 Strong Pointer 并且将它转移到 Stack 中的 Strong Pointer 的序列中。Pop 方法将会返回一个 Strong Pointer。把 Strong Pointer 的引入语言将会使垃圾回收成为历史。

这里还有一个小问题--修改 C++标准几乎和竞选美国总统一样容易。当我将我的注意告诉给 Bjarne Stroutrup 的时候，他看我的眼神好像是我刚刚要向他借一千美元一样。

然后我突然想到一个念头。我可以自己实现 Strong Pointers。毕竟，它们都很想 Smart Pointers。给它们一个拷贝构造函数并重载赋值操作符并不是一个大问题。事实上，这正是标准库中的 auto_ptr 有的。重要的是对这些操作给出一个资源转移的语法，但是这也不是很难。

```
template <class T>

SmartPointer<T>::SmartPointer (SmartPointer<T> & ptr)

{

_p = ptr.Release ();

}

template <class T>

void SmartPointer<T>::operator = (SmartPointer<T> & ptr)

{

if (_p != ptr._p)

{

delete _p;

_p = ptr.Release ();

}

}
```

使这整个想法迅速成功的原因之一是我可以以值方式传递这种封装指针！我有了我的蛋糕，并且也可以吃了。看这个 Stack 的新的实现：

```
class Stack

{

enum { maxStack = 3 };

public:
```

```
Stack ()

: _top (0)

{}

void Push (SmartPointer<Item> & item) throw (char *)

{

if (_top >= maxStack)

throw "Stack overflow";

_arr [_top++] = item;

}

SmartPointer<Item> Pop ()

{

if (_top == 0)

return SmartPointer<Item> ();

return _arr [--_top];

}

private

int _top;

SmartPointer<Item> _arr [maxStack];

};
```

Pop 方法强制客户将其返回值赋给一个 Strong Pointer,SmartPointer<Item>。任何试图将他对一个普通指针的赋值都会产生一个编译期错误，因为类型不匹配。此外，因为 Pop 以值方式返回一个 Strong Pointer(在 Pop 的声明时 SmartPointer<Item>后面没有&符号)，编译器在 return 时自动进行了一个资源转换。他调用了 operator =来从数组中提取一个 Item,拷贝构造函数将他传递给调用者。调用者最后拥有了指向 Pop 赋值的 Strong Pointer 指向的一个 Item。

我马上意识到我已经在某些东西之上了。我开始用了新的方法重写原来的代码。

1.2.5 Parser

我过去有一个老的算术操作分析器，是用老的资源管理的技术写的。分析器的作用是在分析树中生成节点，节点是动态分配的。例如分析器的 Expression 方法生成一个表达式节点。我没有时间用 Strong Pointer 去重写这个分析器。我令 Expression、Term 和 Factor 方法以传值的方式将 Strong Pointer 返回到 Node 中。看下面的 Expression 方法的实现：

```
SmartPointer<Node> Parser::Expression()

{

// Parse a term

SmartPointer<Node> pNode = Term ();

EToken token = _scanner.Token();

if ( token == tPlus || token == tMinus )

{

// Expr := Term { ('+' | '-') Term }

SmartPointer<MultiNode> pMultiNode = new SumNode (pNode);

do

{

_scanner.Accept();

SmartPointer<Node> pRight = Term ();

pMultiNode->AddChild (pRight, (token == tPlus));

token = _scanner.Token();

} while (token == tPlus || token == tMinus);

pNode = up_cast<Node, MultiNode> (pMultiNode);
```

```
}

// otherwise Expr := Term

return pNode; // by value!

}
```

最开始，Term 方法被调用。他传值返回一个指向 Node 的 Strong Pointer 并且立刻把它保存到我们自己的 Strong Pointer,pNode 中。如果下一个符号不是加号或者减号，我们就简单的把这个 SmartPointer 以值返回，这样就释放了 Node 的所有权。另外一方面，如果下一个符号是加号或者减号，我们创建一个新的 SumMode 并且立刻（直接传递）将它储存到 MultiNode 的一个 Strong Pointer 中。这里，SumNode 是从 MultiMode 中继承而来的，而 MulitNode 是从 Node 继承而来的。原来的 Node 的所有权转给了 SumNode。

只要是他们在被加号和减号分开的时候，我们就不断的创建 terms，我们将这些 term 转移到我们的 MultiNode 中，同时 MultiNode 得到了所有权。最后，我们将指向 MultiNode 的 Strong Pointer 向上映射为指向 Mode 的 Strong Pointer，并且将他返回调用着。

我们需要对 Strong Pointers 进行显式的向上映射，即使指针是被隐式的封装。例如，一个 MultiNode 是一个 Node，但是相同的 is-a 关系在 SmartPointer<MultiNode>和 SmartPointer<Node>之间并不存在，因为它们是分离的类（模板实例）并不存在继承关系。up-cast 模板是像下面这样定义的：

```
template<class To, class From>

inline SmartPointer<To> up_cast (SmartPointer<From> & from)

{

return SmartPointer<To> (from.Release ());

}
```

如果你的编译器支持新加入标准的成员模板（member template）的话，你可以为 SmartPointer<T>定义一个新的构造函数用来从接受一个 class U。

```
template <class T>
```

```
template <class U> SmartPointer<T>::SmartPointer (SPrt<U> & uptr)

: _p (uptr.Release ())

{}
```

这里的这个花招是模板在 U 不是 T 的子类的时候就不会编译成功（换句话说，只在 U is-a T 的时候才会编译）。这是因为 uptr 的缘故。Release()方法返回一个指向 U 的指针，并被赋值为_p，一个指向 T 的指针。所以如果 U 不是一个 T 的话，赋值会导致一个编译时刻错误。

```
std::auto_ptr
```

后来我意识到在 STL 中的 auto_ptr 模板，就是我的 Strong Pointer。在那时候还有许多的实现差异（auto_ptr 的 Release 方法并不将内部的指针清零--你的编译器的库很可能用的就是这种陈旧的实现），但是最后在标准被广泛接受之前都被解决了。

1.2.6 Transfer Semantics

目前为止，我们一直在讨论在 C++程序中资源管理的方法。宗旨是将资源封装到一些轻量级的类中，并由类负责它们的释放。特别的是，所有用 new 操作符分配的资源都会被储存并传递进 Strong Pointer（标准库中的 auto_ptr）的内部。

这里的关键词是传递（passing）。一个容器可以通过传值返回一个 Strong Pointer 来安全的释放资源。容器的客户只能够通过提供一个相应的 Strong Pointer 来保存这个资源。任何一个将结果赋给一个"裸"指针的做法都立即会被编译器发现。

```
auto_ptr<Item> item = stack.Pop (); // ok

Item * p = stack.Pop (); // Error! Type mismatch.
```

以传值方式被传递的对象有 value semantics 或者称为 copy semantics。Strong Pointers 是以值方式传递的--但是我们能说它们有 copy semantics 吗？不是这样的！它们所指向的对象肯定没有被拷贝过。事实上，传递过后，源 auto_ptr 不在访问原有的对象，并且目标 auto_ptr 成为了对象的唯一拥有者（但是往往 auto_ptr 的旧的实现即使在释放后仍然保持着对对象的所有权）。自然而然的我们可以将这种新的行为称作 Transfer Semantics。

拷贝构造函数（copy construcor）和赋值操作符定义了 auto_ptr 的 Transfer Semantics，它们用了非 const 的 auto_ptr 引用作为它们的参数。

```
auto_ptr (auto_ptr<T> & ptr);

auto_ptr & operator = (auto_ptr<T> & ptr);
```

这是因为它们确实改变了他们的源--剥夺了对资源的所有权。

通过定义相应的拷贝构造函数和重载赋值操作符，你可以将 Transfer Semantics 加入到许多对象中。例如，许多 Windows 中的资源，比如动态建立的菜单或者位图，可以用有 Transfer Semantics 的类来封装。

1.2.7 Strong Vectors

标准库只在 auto_ptr 中支持资源管理。甚至连最简单的容器也不支持 ownership semantics。你可能想将 auto_ptr 和标准容器组合到一起可能会管用，但是并不是这样的。例如，你可能会这样做，但是会发现你不能够用标准的方法来进行索引。

```
vector< auto_ptr<Item> > autoVector;
```

这种建造不会编译成功；

```
Item * item = autoVector [0];
```

另一方面，这会导致一个从 autoVect 到 auto_ptr 的所有权转换：

```
auto_ptr<Item> item = autoVector [0];
```

我们没有选择，只能够构造我们自己的 Strong Vector。最小的接口应该如下：

```
template <class T>

class auto_vector

{

public:

explicit auto_vector (size_t capacity = 0);

T const * operator [] (size_t i) const;

T * operator [] (size_t i);
```

```
void assign (size_t i, auto_ptr<T> & p);

void assign_direct (size_t i, T * p);

void push_back (auto_ptr<T> & p);

auto_ptr<T> pop_back ();

};
```

你也许会发现一个非常防御性的设计态度。我决定不提供一个对 vector 的左值索引的访问，取而代之，如果你想设定(set)一个值的话，你必须用 assign 或者 assign_direct 方法。我的观点是，资源管理不应该被忽视，同时，也不应该在所有的地方滥用。在我的经验里，一个 strong vector 经常被许多 push_back 方法充斥着。

Strong vector 最好用一个动态的 Strong Pointers 的数组来实现：

```
template <class T>

class auto_vector

{

private

void grow (size_t reqCapacity);

auto_ptr<T> *_arr;

size_t _capacity;

size_t _end;

};
```

grow 方法申请了一个很大的 auto_ptr<T>的数组，将所有的东西从老的书组类转移出来，在其中交换，并且删除原来的数组。

auto_vector 的其他实现都是十分直接的，因为所有资源管理的复杂度都在 auto_ptr 中。例如，assign 方法简单的利用了重载的赋值操作符来删除原有的对象并转移资源到新的对象：

```
void assign (size_t i, auto_ptr<T> & p)

{

_arr [i] = p;

}
```

我已经讨论了 push_back 和 pop_back 方法。push_back 方法传值返回一个 auto_ptr，因为它将所有权从 auto_vector 转换到 auto_ptr 中。

对 auto_vector 的索引访问是借助 auto_ptr 的 get 方法来实现的，get 简单的返回一个内部指针。

```
T * operator [] (size_t i)

{

return _arr [i].get ();

}
```

没有容器可以没有 iterator。我们需要一个 iterator 让 auto_vector 看起来更像一个普通的指针向量。特别是，当我们废弃 iterator 的时候，我们需要的是一个指针而不是 auto_ptr。我们不希望一个 auto_vector 的 iterator 在无意中进行资源转换。

```
template<class T>

class auto_iterator: public

iterator<random_access_iterator_tag, T *>

{

public:

auto_iterator () : _pp (0) {}

auto_iterator (auto_ptr<T> * pp) : _pp (pp) {}

bool operator != (auto_iterator<T> const & it) const

{ return it._pp != _pp; }
```

```
auto_iterator const & operator++ (int) { return _pp++; }

auto_iterator operator++ () { return ++_pp; }

T * operator * () { return _pp->get (); }

private

auto_ptr<T> * _pp;

};
```

我们给 auto_vect 提供了标准的 begin 和 end 方法来找回 iterator：

```
class auto_vector

{

public:

typedef auto_iterator<T> iterator;

iterator begin () { return _arr; }

iterator end () { return _arr + _end; }

};
```

你也许会问我们是否要利用资源管理重新实现每一个标准的容器？幸运的是，不;事实是 strong vector 解决了大部分所有权的需求。当你把你的对象都安全的放置到一个 strong vector 中，你可以用所有其它的容器来重新安排（weak）pointer。

设想，例如，你需要对一些动态分配的对象排序的时候。你将它们的指针保存到一个 strong vector 中。然后你用一个标准的 vector 来保存从 strong vector 中获得的 weak 指针。你可以用标准的算法对这个 vector 进行排序。这种中介 vector 叫做 permutation vector。相似的，你也可以用标准的 maps, priority queues, heaps, hash tables 等等。

1.2.8 Code Inspection

如果你严格遵照资源管理的条款，你就不会再资源泄露或者两次删除的地方遇到麻烦。你也降低了访问野指针的几率。同样的，遵循原有的规则，用 delete 删除用 new 申请的德指针，不要两次删除一个指针。你也不会遇到麻烦。但是，那个是更好的注意呢？

这两个方法有一个很大的不同点。就是和寻找传统方法的 bug 相比，找到违反资源管理的规定要容易的多。后者仅需要一个代码检测或者一个运行测试，而前者则在代码中隐藏得很深，并需要很深的检查。

设想你要做一段传统的代码的内存泄露检查。第一件事，你要做的就是 grep 所有在代码中出现的 new，你需要找出被分配空间地指针都作了什么。你需要确定导致删除这个指针的所有的执行路径。你需要检查 break 语句，过程返回，异常。原有的指针可能赋给另一个指针，你对这个指针也要做相同的事。

相比之下，对于一段用资源管理技术实现的代码。你也用 grep 检查所有的 new，但是这次你只需要检查邻近的调用：

- 这是一个直接的 Strong Pointer 转换，还是我们在一个构造函数的函数体中？
- 调用的返回知是否立即保存到对象中，构造函数中是否有可以产生异常的代码。？
- 如果这样的话析构函数中时候有 delete?

下一步，你需要用 grep 查找所有的 release 方法，并实施相同的检查。

不同点是需要检查、理解单个执行路径和只需要做一些本地的检验。这难道不是提醒你非结构化的和结构化的程序设计的不同吗？原理上，你可以认为你可以应付 goto，并且跟踪所有的可能分支。另一方面，你可以将你的怀疑本地化为一段代码。本地化在两种情况下都是关键所在。

在资源管理中的错误模式也比较容易调试。最常见的 bug 是试图访问一个释放过的 strong pointer。这将导致一个错误，并且很容易跟踪。

1.2.9 共享的所有权

为每一个程序中的资源都找出或者指定一个所有者是一件很容易的事情吗？答案是出乎意料的，是！如果你发现了一些问题，这可能说明你的设计上存在问题。还有另一种情况就是共享所有权是最好的甚至是唯一的选择。

共享的责任分配给被共享的对象和它的客户（client）。一个共享资源必须为它的所有者保持一个引用计数。另一方面，所有者再释放资源的时候必须通报共享对象。最后一个释放资源的需要在最后负责 free 的工作。

最简单的共享的实现是共享对象继承引用计数的类 RefCounted：

```
class RefCounted

{

public:

RefCounted () : _count (1) {}

int GetRefCount () const { return _count; }

void IncRefCount () { _count++; }

int DecRefCount () { return --_count; }

private

int _count;

};
```

按照资源管理，一个引用计数是一种资源。如果你遵守它，你需要释放它。当你意识到这一事实的时候，剩下的就变得简单了。简单的遵循规则--再构造函数中获得引用计数，在析构函数中释放。甚至有一个 RefCounted 的 smart pointer 等价物：

```
template <class T>

class RefPtr

{

public:

RefPtr (T * p) : _p (p) {}

RefPtr (RefPtr<T> & p)

{

_p = p._p;

_p->IncRefCount ();
```

```
}

~RefPtr ()

{

if (_p->DecRefCount () == 0)

delete _p;

}

private

T * _p;

};
```

注意模板中的 T 不比成为 RefCounted 的后代，但是它必须有 IncRefCount 和 DecRefCount 的方法。当然，一个便于使用的 RefPtr 需要有一个重载的指针访问操作符。在 RefPtr 中加入转换语义学（transfer semantics）是读者的工作。

1.2.10 所有权网络

链表是资源管理分析中的一个很有意思的例子。如果你选择表成为链(link)的所有者的话，你会陷入实现递归的所有权。每一个 link 都是它的继承者的所有者，并且，相应的，余下的链表的所有者。下面是用 smart pointer 实现的一个表单元：

```
class Link

{

// ...

private

auto_ptr<Link> _next;

};
```

最好的方法是，将连接控制封装到一个弄构进行资源转换的类中。

对于双链表呢？安全的做法是指明一个方向，如 forward:

```
class DoubleLink

{

// ...

private

DoubleLink *_prev;

auto_ptr<DoubleLink> _next;

};
```

注意不要创建环形链表。

这给我们带来了另外一个有趣的问题--资源管理可以处理环形的所有权吗？它可以，用一个 mark-and-sweep 的算法。这里是实现这种方法的一个例子：

```
template<class T>

class CyclPtr

{

public:

CyclPtr (T * p)

:_p (p), _isBeingDeleted (false)

{}

~CyclPtr ()

{

_isBeingDeleted = true;

if (!_p->IsBeingDeleted ())

delete _p;

}
```

```
void Set (T * p)

{

_p = p;

}

bool IsBeingDeleted () const { return _isBeingDeleted; }

private

T * _p;

bool _isBeingDeleted;

};
```

注意我们需要用 class T 来实现方法 IsBeingDeleted，就像从 CyclPtr 继承。对特殊的所有权网络普通化是十分直接的。

将原有代码转换为资源管理代码

如果你是一个经验丰富的程序员，你一定会知道找资源的 bug 是一件浪费时间的痛苦的经历。我不必说服你和你的团队花费一点时间来熟悉资源管理是十分值得的。你可以立即开始用这个方法，无论你是在开始一个新项目或者是在一个项目的中期。转换不必立即全部完成。下面是步骤。

（1） 首先，在你的工程中建立基本的 Strong Pointer。然后通过查找代码中的 new 来开始封装裸指针。

（2） 最先封装的是在过程中定义的临时指针。简单的将它们替换为 auto_ptr 并且删除相应的 delete。如果一个指针在过程中没有被删除而是被返回，用 auto_ptr 替换并在返回前调用 release 方法。在你做第二次传递的时候，你需要处理对 release 的调用。注意，即使是在这点，你的代码也可能更加"精力充沛"--你会移出代码中潜在的资源泄漏问题。

（3） 下面是指向资源的裸指针。确保它们被独立的封装到 auto_ptr 中，或者在构造函数中分配在析构函数中释放。如果你有传递所有权的行为的话，需要调用 release 方法。如果你有容器所有对象，用 Strong Pointers 重新实现它们。

（4） 接下来，找到所有对 release 的方法调用并且尽力清除所有，如果一个 release 调用返回一个指针，将它修改传值返回一个 auto_ptr。

（5） 重复着一过程，直到最后所有 new 和 release 的调用都在构造函数或者资源转换的时候发生。这样，你在你的代码中处理了资源泄漏的问题。对其他资源进行相似的操作。

（6） 你会发现资源管理清除了许多错误和异常处理带来的复杂性。不仅仅你的代码会变得精力充沛，它也会变得简单并容易维护。

2 内存泄漏

2.1 C++中动态内存分配引发问题的解决方案

假设我们要开发一个 String 类，它可以方便地处理字符串数据。我们可以在类中声明一个数组，考虑到有时候字符串极长，我们可以把数组大小设为 200，但一般的情况下又不需要这么多的空间，这样是浪费了内存。对了，我们可以使用 new 操作符，这样是十分灵活的，但在类中就会出现许多意想不到的问题，本文就是针对这一现象而写的。现在，我们先来开发一个 String 类，但它是一个不完善的类。的确，我们要刻意地使它出现各种各样的问题，这样才好对症下药。好了，我们开始吧！

```
/* String.h */

#ifndef STRING_H_

#define STRING_H_

class String

{

private:

char * str; //存储数据

int len; //字符串长度

public:

String(const char * s); //构造函数

String(); // 默认构造函数

~String(); // 析构函数

friend ostream & operator＜＜(ostream & os,const String& st);

};

#endif

/*String.cpp*/
```

```
#include ＜iostream＞

#include ＜cstring＞

#include "String.h"

using namespace std;

String::String(const char * s)

{

len = strlen(s);

str = new char[len + 1];

strcpy(str, s);

}//拷贝数据

String::String()

{

len =0;

str = new char[len+1];

str[0]='\0';

}

String::~String()

{

cout＜＜"这个字符串将被删除："＜＜str＜＜'\n';//为了方便观察结果，特留此行代码。

delete [] str;

}

ostream & operator＜＜(ostream & os, const String & st)

{
```

```
os ＜＜ st.str;

return os;

}

/*test_right.cpp*/

#include ＜iostream＞

#include ＜stdlib.h＞

#include "String.h"

using namespace std;

int main()

{

String temp("天极网");

cout＜＜temp＜＜'"n';

system("PAUSE");

return 0;

}
```

运行结果：

```
天极网

请按任意键继续. . .
```

大家可以看到，以上程序十分正确，而且也是十分有用的。可是，我们不能被表面现象所迷惑！下面，请大家用 test_String.cpp 文件替换 test_right.cpp 文件进行编译，看看结果。有的编译器可能就是根本不能进行编译！

test_String.cpp:

```
#include ＜iostream＞

#include ＜stdlib.h＞

#include "String.h"

using namespace std;

void show_right(const String&);

void show_String(const String);//注意，参数非引用，而是按值传递。

int main()

{

String test1("第一个范例。");

String test2("第二个范例。");

String test3("第三个范例。");

String test4("第四个范例。");

cout＜＜"下面分别输入三个范例："n";

cout＜＜test1＜＜endl;

cout＜＜test2＜＜endl;

cout＜＜test3＜＜endl;

String* String1=new String(test1);

cout＜＜*String1＜＜endl;

delete String1;

cout＜＜test1＜＜endl; //在 Dev-cpp 上没有任何反应。

cout＜＜"使用正确的函数："＜＜endl;

show_right(test2);

cout＜＜test2＜＜endl;
```

```
cout＜＜"使用错误的函数："＜＜endl;

show_String(test2);

cout＜＜test2＜＜endl; //这一段代码出现严重的错误！

String String2(test3);

cout＜＜"String2: "＜＜String2＜＜endl;

String String3;

String3=test4;

cout＜＜"String3: "＜＜String3＜＜endl;

cout＜＜"下面，程序结束，析构函数将被调用。"＜＜endl;

return 0;

}

void show_right(const String& a)

{

cout＜＜a＜＜endl;

}

void show_String(const String a)

{

cout＜＜a＜＜endl;

}
```

运行结果：

```
下面分别输入三个范例：

第一个范例。

第二个范例。
```

第三个范例。

第一个范例。

这个字符串将被删除：第一个范例。

使用正确的函数：

第二个范例。

第二个范例。

使用错误的函数：

第二个范例。

这个字符串将被删除：第二个范例。

这个字符串将被删除：?=

?=

String2: 第三个范例。

String3: 第四个范例。

下面，程序结束，析构函数将被调用。

这个字符串将被删除：第四个范例。

这个字符串将被删除：第三个范例。

这个字符串将被删除：?=

这个字符串将被删除：x =

这个字符串将被删除：?=

这个字符串将被删除：

现在，请大家自己试试运行结果，或许会更加惨不忍睹呢！下面，我为大家一一分析原因。

首先，大家要知道，C＋＋类有以下这些极为重要的函数：

一：复制构造函数。

二：赋值函数。

我们先来讲复制构造函数。什么是复制构造函数呢？比如，我们可以写下这样的代码：String test1(test2);这是进行初始化。我们知道，初始化对象要用构造函数。可这儿呢？按理说，应该有声明为这样的构造函数：String(const String &);可是，我们并没有定义这个构造函数呀？答案是，C＋＋提供了默认的复制构造函数，问题也就出在这儿。

（1）：什么时候会调用复制构造函数呢？（以 String 类为例。）

在我们提供这样的代码：String test1(test2)时，它会被调用；当函数的参数列表为按值传递，也就是没有用引用和指针作为类型时，如：void show_String(const String)，它会被调用。其实，还有一些情况，但在这儿就不列举了。

（2）：它是什么样的函数。

它的作用就是把两个类进行复制。拿 String 类为例，C＋＋提供的默认复制构造函数是这样的：

```
String(const String& a)

{

str=a.str;

len=a.len;

}
```

在平时，这样并不会有任何的问题出现，但我们用了 new 操作符，涉及到了动态内存分配，我们就不得不谈谈浅复制和深复制了。以上的函数就是实行的浅复制，它只是复制了指针，而并没有复制指针指向的数据，可谓一点儿用也没有。打个比方吧！就像一个朋友让你把一个程序通过网络发给他，而你大大咧咧地把快捷方式发给了他，有什么用处呢？我们来具体谈谈：

假如，A 对象中存储了这样的字符串：“C＋＋”。它的地址为 2000。现在，我们把 A 对象赋给 B 对象：String B=A。现在，A 和 B 对象的 str 指针均指向 2000 地址。看似可以使用，但如果 B 对象的析构函数被调用时，则地址 2000 处的字符串“C＋＋”已经被从内存中抹去，而 A 对象仍然指向地址 2000。这时，如果我们写下这样的代码：cout＜＜

A＜＜endl;或是等待程序结束，A 对象的析构函数被调用时，A 对象的数据能否显示出来呢？只会是乱码。而且，程序还会这样做：连续对地址 2000 处使用两次 delete 操作符，这样的后果是十分严重的！

本例中，有这样的代码：

```
String* String1=new String(test1);

cout＜＜*String1＜＜endl;

delete String1;
```

假设 test1 中 str 指向的地址为 2000,而 String 中 str 指针同样指向地址 2000，我们删除了 2000 处的数据，而 test1 对象呢？已经被破坏了。大家从运行结果上可以看到，我们使用 cout＜＜test1 时，一点反应也没有。而在 test1 的析构函数被调用时，显示是这样：“这个字符串将被删除：”。

再看看这段代码：

```
cout＜＜"使用错误的函数："＜＜endl;

show_String(test2);

cout＜＜test2＜＜endl;//这一段代码出现严重的错误！
```

show_String 函数的参数列表 void show_String(const String a)是按值传递的，所以，我们相当于执行了这样的代码：String a=test2;函数执行完毕，由于生存周期的缘故，对象 a 被析构函数删除，我们马上就可以看到错误的显示结果了：这个字符串将被删除：?=。当然，test2 也被破坏了。解决的办法很简单，当然是手工定义一个复制构造函数喽！人力可以胜天！

```
String::String(const String& a)
{
len=a.len;
str=new char(len+1);
strcpy(str,a.str);
}
```

我们执行的是深复制。这个函数的功能是这样的：假设对象 A 中的 str 指针指向地址 2000，内容为“I am a C++ Boy!”。我们执行代码 String B=A 时，我们先开辟出一块内

存，假设为 3000。我们用 strcpy 函数将地址 2000 的内容拷贝到地址 3000 中，再将对象 B 的 str 指针指向地址 3000。这样，就互不干扰了。

大家把这个函数加入程序中，问题就解决了大半，但还没有完全解决，问题在赋值函数上。我们的程序中有这样的段代码：

```
String String3;

String3=test4;
```

经过我前面的讲解，大家应该也会对这段代码进行寻根摸底：凭什么可以这样做：String3=test4？？？原因是，C＋＋为了用户的方便，提供的这样的一个操作符重载函数：operator=。所以，我们可以这样做。大家应该猜得到，它同样是执行了浅复制，出了同样的毛病。比如，执行了这段代码后，析构函数开始大展神威^_^。由于这些变量是后进先出的，所以最后的 String3 变量先被删除：这个字符串将被删除：第四个范例。很正常。最后，删除到 test4 的时候，问题来了：这个字符串将被删除：?=。原因我不用赘述了，只是这个赋值函数怎么写，还有一点儿学问呢！大家请看：

平时，我们可以写这样的代码：x=y=z。（均为整型变量。）而在类对象中，我们同样要这样，因为这很方便。而对象 A=B=C 就是 A.operator=(B.operator=(c))。而这个 operator=函数的参数列表应该是：const String& a，所以，大家不难推出，要实现这样的功能，返回值也要是 String&，这样才能实现 A＝B＝C。我们先来写写看：

```
String& String::operator=(const String& a)

{

delete [] str;//先删除自身的数据

len=a.len;

str=new char[len+1];

strcpy(str,a.str);//此三行为进行拷贝

return *this;//返回自身的引用

}
```

是不是这样就行了呢？我们假如写出了这种代码：A=A，那么大家看看，岂不是把 A 对象的数据给删除了吗？这样可谓引发一系列的错误。所以，我们还要检查是否为自身赋

值。只比较两对象的数据是不行了，因为两个对象的数据很有可能相同。我们应该比较地址。以下是完好的赋值函数：

```
String& String::operator=(const String& a)

{

if(this==&a)

return *this;

delete [] str;

len=a.len;

str=new char[len+1];

strcpy(str,a.str);

return *this;

}
```

把这些代码加入程序，问题就完全解决，下面是运行结果：

```
    下面分别输入三个范例：

    第一个范例

    第二个范例

    第三个范例

    第一个范例

    这个字符串将被删除：第一个范例。

    第一个范例

      使用正确的函数：

    第二个范例。

    第二个范例。
```

```
使用错误的函数：

第二个范例。

这个字符串将被删除：第二个范例。

第二个范例。

String2: 第三个范例。

String3: 第四个范例。

下面，程序结束，析构函数将被调用。

这个字符串将被删除：第四个范例。

这个字符串将被删除：第三个范例。

这个字符串将被删除：第四个范例。

这个字符串将被删除：第三个范例。

这个字符串将被删除：第二个范例。

这个字符串将被删除：第一个范例。
```

2.2 如何对付内存泄漏？

写出那些不会导致任何内存泄漏的代码。很明显，当你的代码中到处充满了 new 操作、delete 操作和指针运算的话，你将会在某个地方搞晕了头，导致内存泄漏，指针引用错误，以及诸如此类的问题。这和你如何小心地对待内存分配工作其实完全没有关系：代码的复杂性最终总是会超过你能够付出的时间和努力。于是随后产生了一些成功的技巧，它们依赖于将内存分配（allocations）与重新分配（deallocation）工作隐藏在易于管理的类型之后。标准容器（standard containers）是一个优秀的例子。它们不是通过你而是自己为元素管理内存，从而避免了产生糟糕的结果。想象一下，没有 string 和 vector 的帮助，写出这个：

```
#include<vector>

#include<string>

#include<iostream>
```

```
#include<algorithm>

using namespace std;

int main() // small program messing around with strings

{

    cout << "enter some whitespace-separated words:"n";

    vector<string> v;

    string s;

    while (cin>>s) v.push_back(s);

    sort(v.begin(),v.end());

    string cat;

    typedef vector<string>::const_iterator Iter;

    for (Iter p = v.begin(); p!=v.end(); ++p) cat += *p+"+";

    cout << cat << ’ "n’ ;

}
```

你有多少机会在第一次就得到正确的结果？你又怎么知道你没有导致内存泄漏呢？

注意，没有出现显式的内存管理，宏，造型，溢出检查，显式的长度限制，以及指针。通过使用函数对象和标准算法（standard algorithm），我可以避免使用指针——例如使用迭代子（iterator），不过对于一个这么小的程序来说有点小题大作了。

这些技巧并不完美，要系统化地使用它们也并不总是那么容易。但是，应用它们产生了惊人的差异，而且通过减少显式的内存分配与重新分配的次数，你甚至可以使余下的例子更加容易被跟踪。早在 1981 年，我就指出，通过将我必须显式地跟踪的对象的数量从几万个减少到几打，为了使程序正确运行而付出的努力从可怕的苦工，变成了应付一些可管理的对象，甚至更加简单了。

如果你的程序还没有包含将显式内存管理减少到最小限度的库，那么要让你程序完成和正确运行的话，最快的途径也许就是先建立一个这样的库。

模板和标准库实现了容器、资源句柄以及诸如此类的东西，更早的使用甚至在多年以前。异常的使用使之更加完善。

如果你实在不能将内存分配/重新分配的操作隐藏到你需要的对象中时，你可以使用资源句柄（resource handle），以将内存泄漏的可能性降至最低。这里有个例子：我需要通过一个函数，在空闲内存中建立一个对象并返回它。这时候可能忘记释放这个对象。毕竟，我们不能说，仅仅关注当这个指针要被释放的时候，谁将负责去做。使用资源句柄，这里用了标准库中的 auto_ptr，使需要为之负责的地方变得明确了。

```
#include<memory>

#include<iostream>

using namespace std;

struct S {

    S() { cout << "make an S"n"; }

    ~S() { cout << "destroy an S"n"; }

    S(const S&) { cout << "copy initialize an S"n"; }

    S& operator=(const S&) { cout << "copy assign an S"n"; }

};

S* f()

{

    return new S; //  谁该负责释放这个 S?

};

auto_ptr<S> g()

{

    return auto_ptr<S>(new S); //  显式传递负责释放这个 S

}
```

```
int main()

{

    cout << "start main"n";

    S* p = f();

    cout << "after f() before g()"n";

    // S* q = g(); //  将被编译器捕捉

    auto_ptr<S> q = g();

    cout << "exit main"n";

    // *p 产生了内存泄漏

    // *q 被自动释放

}
```

在更一般的意义上考虑资源，而不仅仅是内存。

如果在你的环境中不能系统地应用这些技巧（例如，你必须使用别的地方的代码，或者你的程序的另一部分简直是原始人类（译注：原文是 Neanderthals，尼安德特人，旧石器时代广泛分布在欧洲的猿人）写的，如此等等），那么注意使用一个内存泄漏检测器作为开发过程的一部分，或者插入一个垃圾收集器（garbage collector）。

2.3 浅谈 C/C++内存泄漏及其检测工具

对于一个 c/c++程序员来说，内存泄漏是一个常见的也是令人头疼的问题。已经有许多技术被研究出来以应对这个问题，比如 Smart Pointer，Garbage Collection 等。Smart Pointer 技术比较成熟，STL 中已经包含支持 Smart Pointer 的 class，但是它的使用似乎并不广泛，而且它也不能解决所有的问题；Garbage Collection 技术在 Java 中已经比较成熟，但是在 c/c++领域的发展并不顺畅，虽然很早就有人思考在 C++中也加入 GC 的支持。现实世界就是这样的，作为一个 c/c++程序员，内存泄漏是你心中永远的痛。不过好在现在有许多工具能够帮助我们验证内存泄漏的存在，找出发生问题的代码。

2.3.1 内存泄漏的定义

一般我们常说的内存泄漏是指堆内存的泄漏。堆内存是指程序从堆中分配的，大小任意的（内存块的大小可以在程序运行期决定），使用完后必须显示释放的内存。应用程序一般使用 malloc，realloc，new 等函数从堆中分配到一块内存，使用完后，程序必须负责相应的调用 free 或 delete 释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了。以下这段小程序演示了堆内存发生泄漏的情形：

```
void MyFunction(int nSize)

{

   char* p= new char[nSize];

   if( !GetStringFrom( p, nSize ) ){

      MessageBox(“Error”);

      return;

   }

   …//using the string pointed by p;

   delete p;

}
```

当函数 GetStringFrom()返回零的时候，指针 p 指向的内存就不会被释放。这是一种常见的发生内存泄漏的情形。程序在入口处分配内存，在出口处释放内存，但是 c 函数可以在任何地方退出，所以一旦有某个出口处没有释放应该释放的内存，就会发生内存泄漏。

广义的说，内存泄漏不仅仅包含堆内存的泄漏，还包含系统资源的泄漏(resource leak)，比如核心态 HANDLE，GDI Object，SOCKET， Interface 等，从根本上说这些由操作系统分配的对象也消耗内存，如果这些对象发生泄漏最终也会导致内存的泄漏。而且，某些对象消耗的是核心态内存，这些对象严重泄漏时会导致整个操作系统不稳定。所以相比之下，系统资源的泄漏比堆内存的泄漏更为严重。

GDI Object 的泄漏是一种常见的资源泄漏：

```
void CMyView::OnPaint( CDC* pDC )

{
```

```
    CBitmap bmp;
    CBitmap* pOldBmp;
    bmp.LoadBitmap(IDB_MYBMP);
    pOldBmp = pDC->SelectObject( &bmp );
    ...
    if( Something() ){
        return;
    }
    pDC->SelectObject( pOldBmp );
    return;
}
```

当函数 Something()返回非零的时候，程序在退出前没有把 pOldBmp 选回 pDC 中，这会导致 pOldBmp 指向的 HBITMAP 对象发生泄漏。这个程序如果长时间的运行，可能会导致整个系统花屏。这种问题在 Win9x 下比较容易暴露出来，因为 Win9x 的 GDI 堆比 Win2k 或 NT 的要小很多。

2.3.2 内存泄漏的发生方式

以发生的方式来分类，内存泄漏可以分为 4 类：

1. 常发性内存泄漏。发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。比如例二，如果 Something()函数一直返回 True，那么 pOldBmp 指向的 HBITMAP 对象总是发生泄漏。

2. 偶发性内存泄漏。发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。比如例二，如果 Something()函数只有在特定环境下才返回 True，那么 pOldBmp 指向的 HBITMAP 对象并不总是发生泄漏。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。

3. 一次性内存泄漏。发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块仅且一块内存发生泄漏。比如，在类的构造函数中分配内存，在析构函数中却没有释放该内存，但是因为这个类是一个 Singleton，所以内存泄漏只会发生一次。另一个例子：

```
char* g_lpszFileName = NULL;

void SetFileName( const char* lpcszFileName )

{

   if( g_lpszFileName ){

      free( g_lpszFileName );

   }

   g_lpszFileName = strdup( lpcszFileName );

}
```

如果程序在结束的时候没有释放 g_lpszFileName 指向的字符串，那么，即使多次调用 SetFileName()，总会有一块内存，而且仅有一块内存发生泄漏。

4. 隐式内存泄漏。程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个服务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。举一个例子：

```
class Connection

{

   public:

      Connection( SOCKET s);

      ~Connection();

      ...

   private:
```

```
    SOCKET _socket;
    ...
};
class ConnectionManager
{
  public:
    ConnectionManager(){}
    ~ConnectionManager(){
      list::iterator it;
      for( it = _connlist.begin(); it != _connlist.end(); ++it ){
        delete （*it）;
      }
      _connlist.clear();
    }
    void OnClientConnected( SOCKET s ){
      Connection* p = new Connection(s);
      _connlist.push_back(p);
    }
    void OnClientDisconnected( Connection* pconn ){
      _connlist.remove( pconn );
      delete pconn;
    }
  private:
```

```
    list _connlist;

};
```

假设在 Client 从 Server 端断开后，Server 并没有呼叫 OnClientDisconnected()函数，那么代表那次连接的 Connection 对象就不会被及时的删除（在 Server 程序退出的时候，所有 Connection 对象会在 ConnectionManager 的析构函数里被删除）。当不断的有连接建立、断开时隐式内存泄漏就发生了。

从用户使用程序的角度来看，内存泄漏本身不会产生什么危害，作为一般的用户，根本感觉不到内存泄漏的存在。真正有危害的是内存泄漏的堆积，这会最终消耗尽系统所有的内存。从这个角度来说，一次性内存泄漏并没有什么危害，因为它不会堆积，而隐式内存泄漏危害性则非常大，因为较之于常发性和偶发性内存泄漏它更难被检测到。

2.3.3 检测内存泄漏

检测内存泄漏的关键是要能截获住对分配内存和释放内存的函数的调用。截获住这两个函数，我们就能跟踪每一块内存的生命周期，比如，每当成功的分配一块内存后，就把它的指针加入一个全局的 list 中；每当释放一块内存，再把它的指针从 list 中删除。这样，当程序结束的时候，list 中剩余的指针就是指向那些没有被释放的内存。这里只是简单的描述了检测内存泄漏的基本原理，详细的算法可以参见 Steve Maguire 的<<Writing Solid Code>>。

如果要检测堆内存的泄漏，那么需要截获住 malloc/realloc/free 和 new/delete 就可以了（其实 new/delete 最终也是用 malloc/free 的，所以只要截获前面一组即可）。对于其他的泄漏，可以采用类似的方法，截获住相应的分配和释放函数。比如，要检测 BSTR 的泄漏，就需要截获 SysAllocString/SysFreeString；要检测 HMENU 的泄漏，就需要截获 CreateMenu/ DestroyMenu。（有的资源的分配函数有多个，释放函数只有一个，比如，SysAllocStringLen 也可以用来分配 BSTR，这时就需要截获多个分配函数）

在 Windows 平台下，检测内存泄漏的工具常用的一般有三种，MS C-Runtime Library 内建的检测功能；外挂式的检测工具，诸如，Purify，BoundsChecker 等；利用 Windows NT 自带的 Performance Monitor。这三种工具各有优缺点，MS C-Runtime Library 虽然功能上较之外挂式的工具要弱，但是它是免费的；Performance Monitor 虽然无法标示出发生问题的代码，但是它能检测出隐式的内存泄漏的存在，这是其他两类工具无能为力的地方。

以下我们详细讨论这三种检测工具：

2.3.3.1 VC 下内存泄漏的检测方法

用 MFC 开发的应用程序，在 DEBUG 版模式下编译后，都会自动加入内存泄漏的检测代码。在程序结束后，如果发生了内存泄漏，在 Debug 窗口中会显示出所有发生泄漏的内存块的信息，以下两行显示了一块被泄漏的内存块的信息：

E:"TestMemLeak"TestDlg.cpp(70) : {59} normal block at 0x00881710, 200 bytes long.

Data: <abcdefghijklmnop> 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70

第一行显示该内存块由 TestDlg.cpp 文件，第 70 行代码分配，地址在 0x00881710，大小为 200 字节，{59}是指调用内存分配函数的 Request Order，关于它的详细信息可以参见 MSDN 中_CrtSetBreakAlloc()的帮助。第二行显示该内存块前 16 个字节的内容，尖括号内是以 ASCII 方式显示，接着的是以 16 进制方式显示。

一般大家都误以为这些内存泄漏的检测功能是由 MFC 提供的，其实不然。MFC 只是封装和利用了 MS C-Runtime Library 的 Debug Function。非 MFC 程序也可以利用 MS C-Runtime Library 的 Debug Function 加入内存泄漏的检测功能。MS C-Runtime Library 在实现 malloc/free，strdup 等函数时已经内建了内存泄漏的检测功能。

注意观察一下由 MFC Application Wizard 生成的项目，在每一个 cpp 文件的头部都有这样一段宏定义：

```
#ifdef _DEBUG

#define new DEBUG_NEW

#undef THIS_FILE

static char THIS_FILE[] = __FILE__;

#endif
```

有了这样的定义，在编译 DEBUG 版时，出现在这个 cpp 文件中的所有 new 都被替换成 DEBUG_NEW 了。那么 DEBUG_NEW 是什么呢？DEBUG_NEW 也是一个宏，以下摘自 afx.h，1632 行

```
#define DEBUG_NEW new(THIS_FILE, __LINE__)
```

所以如果有这样一行代码：

```
char* p = new char[200];
```

经过宏替换就变成了：

```
char* p = new( THIS_FILE, __LINE__)char[200];
```

根据 C++的标准，对于以上的 new 的使用方法，编译器会去找这样定义的 operator new：

```
void* operator new(size_t, LPCSTR, int)
```

我们在 afxmem.cpp 63 行找到了一个这样的 operator new 的实现

```
void* AFX_CDECL operator new(size_t nSize, LPCSTR lpszFileName, int nLine)
{
   return ::operator new(nSize, _NORMAL_BLOCK, lpszFileName, nLine);
}
void* __cdecl operator new(size_t nSize, int nType, LPCSTR lpszFileName, int nLine)
{
   ...
   pResult = _malloc_dbg(nSize, nType, lpszFileName, nLine);
   if (pResult != NULL)
      return pResult;
   ...
}
```

第二个 operator new 函数比较长，为了简单期间，我只摘录了部分。很显然最后的内存分配还是通过_malloc_dbg 函数实现的，这个函数属于 MS C-Runtime Library 的 Debug Function。这个函数不但要求传入内存的大小，另外还有文件名和行号两个参数。

文件名和行号就是用来记录此次分配是由哪一段代码造成的。如果这块内存在程序结束之前没有被释放，那么这些信息就会输出到 Debug 窗口里。

这里顺便提一下 THIS_FILE，__FILE 和__LINE__。__FILE__和__LINE__都是编译器定义的宏。当碰到__FILE__时，编译器会把__FILE__替换成一个字符串，这个字符串就是当前在编译的文件的路径名。当碰到__LINE__时，编译器会把__LINE__替换成一个数字，这个数字就是当前这行代码的行号。在 DEBUG_NEW 的定义中没有直接使用__FILE__，而是用了 THIS_FILE，其目的是为了减小目标文件的大小。假设在某个 cpp 文件中有 100 处使用了 new，如果直接使用__FILE__，那编译器会产生 100 个常量字符串，这 100 个字符串都是飧?/SPAN>cpp 文件的路径名，显然十分冗余。如果使用 THIS_FILE，编译器只会产生一个常量字符串，那 100 处 new 的调用使用的都是指向常量字符串的指针。

再次观察一下由 MFC Application Wizard 生成的项目，我们会发现在 cpp 文件中只对 new 做了映射，如果你在程序中直接使用 malloc 函数分配内存，调用 malloc 的文件名和行号是不会被记录下来的。如果这块内存发生了泄漏，MS C-Runtime Library 仍然能检测到，但是当输出这块内存块的信息，不会包含分配它的的文件名和行号。

要在非 MFC 程序中打开内存泄漏的检测功能非常容易，你只要在程序的入口处加入以下几行代码：

```
int tmpFlag = _CrtSetDbgFlag( _CRTDBG_REPORT_FLAG );

tmpFlag |= _CRTDBG_LEAK_CHECK_DF;

_CrtSetDbgFlag( tmpFlag );
```

这样，在程序结束的时候，也就是 winmain，main 或 dllmain 函数返回之后，如果还有内存块没有释放，它们的信息会被打印到 Debug 窗口里。

如果你试着创建了一个非 MFC 应用程序，而且在程序的入口处加入了以上代码，并且故意在程序中不释放某些内存块，你会在 Debug 窗口里看到以下的信息：

```
{47} normal block at 0x00C91C90, 200 bytes long.

Data: < > 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

内存泄漏的确检测到了，但是和上面 MFC 程序的例子相比，缺少了文件名和行号。对于一个比较大的程序，没有这些信息，解决问题将变得十分困难。

为了能够知道泄漏的内存块是在哪里分配的，你需要实现类似 MFC 的映射功能，把 new，maolloc 等函数映射到_malloc_dbg 函数上。这里我不再赘述，你可以参考 MFC 的源代码。

由于 Debug Function 实现在 MS C-RuntimeLibrary 中，所以它只能检测到堆内存的泄漏，而且只限于 malloc，realloc 或 strdup 等分配的内存，而那些系统资源，比如 HANDLE，GDI Object，或是不通过 C-Runtime Library 分配的内存，比如 VARIANT，BSTR 的泄漏，它是无法检测到的，这是这种检测法的一个重大的局限性。另外，为了能记录内存块是在哪里分配的，源代码必须相应的配合，这在调试一些老的程序非常麻烦，毕竟修改源代码不是一件省心的事，这是这种检测法的另一个局限性。

对于开发一个大型的程序，MS C-Runtime Library 提供的检测功能是远远不够的。接下来我们就看看外挂式的检测工具。我用的比较多的是 BoundsChecker，一则因为它的功能比较全面，更重要的是它的稳定性。这类工具如果不稳定，反而会忙里添乱。到底是出自鼎鼎大名的 NuMega，我用下来基本上没有什么大问题。

2.3.3.2 使用 BoundsChecker 检测内存泄漏

BoundsChecker 采用一种被称为 Code Injection 的技术，来截获对分配内存和释放内存的函数的调用。简单地说，当你的程序开始运行时，BoundsChecker 的 DLL 被自动载入进程的地址空间（这可以通过 system-level 的 Hook 实现），然后它会修改进程中对内存分配和释放的函数调用，让这些调用首先转入它的代码，然后再执行原来的代码。BoundsChecker 在做这些动作的时，无须修改被调试程序的源代码或工程配置文件，这使得使用它非常的简便、直接。

这里我们以 malloc 函数为例，截获其他的函数方法与此类似。

需要被截获的函数可能在 DLL 中，也可能在程序的代码里。比如，如果静态连结 C-Runtime Library，那么 malloc 函数的代码会被连结到程序里。为了截获住对这类函数的调用，BoundsChecker 会动态修改这些函数的指令。

以下两段汇编代码，一段没有 BoundsChecker 介入，另一段则有 BoundsChecker 的介入：

```
126: _CRTIMP void * __cdecl malloc (

127: size_t nSize
```

```
128: )
129: {
00403C10 push ebp
00403C11 mov ebp,esp
130: return _nh_malloc_dbg(nSize, _newmode, _NORMAL_BLOCK, NULL, 0);
00403C13 push 0
00403C15 push 0
00403C17 push 1
00403C19 mov eax,[__newmode (0042376c)]
00403C1E push eax
00403C1F mov ecx,dword ptr [nSize]
00403C22 push ecx
00403C23 call _nh_malloc_dbg (00403c80)
00403C28 add esp,14h
131: }
```

以下这一段代码有 BoundsChecker 介入：

```
126: _CRTIMP void * __cdecl malloc (
127: size_t nSize
128: )
129: {
00403C10 jmp 01F41EC8
00403C15 push 0
00403C17 push 1
```

```
00403C19 mov eax,[__newmode (0042376c)]

00403C1E push eax

00403C1F mov ecx,dword ptr [nSize]

00403C22 push ecx

00403C23 call _nh_malloc_dbg (00403c80)

00403C28 add esp,14h

131: }
```

当 BoundsChecker 介入后，函数 malloc 的前三条汇编指令被替换成一条 jmp 指令，原来的三条指令被搬到地址 01F41EC8 处了。当程序进入 malloc 后先 jmp 到 01F41EC8，执行原来的三条指令，然后就是 BoundsChecker 的天下了。大致上它会先记录函数的返回地址（函数的返回地址在 stack 上，所以很容易修改），然后把返回地址指向属于 BoundsChecker 的代码，接着跳到 malloc 函数原来的指令，也就是在 00403c15 的地方。当 malloc 函数结束的时候，由于返回地址被修改，它会返回到 BoundsChecker 的代码中，此时 BoundsChecker 会记录由 malloc 分配的内存的指针，然后再跳转到到原来的返回地址去。

如果内存分配/释放函数在 DLL 中，BoundsChecker 则采用另一种方法来截获对这些函数的调用。BoundsChecker 通过修改程序的 DLL Import Table 让 table 中的函数地址指向自己的地址，以达到截获的目的。

截获住这些分配和释放函数，BoundsChecker 就能记录被分配的内存或资源的生命周期。接下来的问题是如何与源代码相关，也就是说当 BoundsChecker 检测到内存泄漏，它如何报告这块内存块是哪段代码分配的。答案是调试信息（Debug Information）。当我们编译一个 Debug 版的程序时，编译器会把源代码和二进制代码之间的对应关系记录下来，放到一个单独的文件里(.pdb)或者直接连结进目标程序，通过直接读取调试信息就能得到分配某块内存的源代码在哪个文件，哪一行上。使用 Code Injection 和 Debug Information，使 BoundsChecker 不但能记录呼叫分配函数的源代码的位置，而且还能记录分配时的 Call Stack，以及 Call Stack 上的函数的源代码位置。这在使用像 MFC 这样的类库时非常有用，以下我用一个例子来说明：

```
void ShowXItemMenu()
```

```
{
   ...
   CMenu menu;
   menu.CreatePopupMenu();
   //add menu items.
   menu.TrackPropupMenu();
   ...
}
void ShowYItemMenu( )
{
   ...
   CMenu menu;
   menu.CreatePopupMenu();
   //add menu items.
   menu.TrackPropupMenu();
   menu.Detach();//this will cause HMENU leak
   ...
}
BOOL CMenu::CreatePopupMenu()
{
   ...
   hMenu = CreatePopupMenu();
   ...
```

```
}
```

当调用 ShowYItemMenu()时，我们故意造成 HMENU 的泄漏。但是，对于 BoundsChecker 来说被泄漏的 HMENU 是在 class CMenu::CreatePopupMenu()中分配的。假设的你的程序有许多地方使用了 CMenu 的 CreatePopupMenu()函数，如 CMenu::CreatePopupMenu()造成的，你依然无法确认问题的根结到底在哪里，在 ShowXItemMenu()中还是在 ShowYItemMenu()中，或者还有其它的地方也使用了 CreatePopupMenu()？有了 Call Stack 的信息，问题就容易了。BoundsChecker 会如下报告泄漏的 HMENU 的信息：

```
Function

File

Line

CMenu::CreatePopupMenu

E:"8168"vc98"mfc"mfc"include"afxwin1.inl

1009

ShowYItemMenu

E:"testmemleak"mytest.cpp

100
```

这里省略了其他的函数调用

如此，我们很容易找到发生问题的函数是 ShowYItemMenu()。当使用 MFC 之类的类库编程时，大部分的 API 调用都被封装在类库的 class 里，有了 Call Stack 信息，我们就可以非常容易的追踪到真正发生泄漏的代码。

记录 Call Stack 信息会使程序的运行变得非常慢，因此默认情况下 BoundsChecker 不会记录 Call Stack 信息。可以按照以下的步骤打开记录 Call Stack 信息的选项开关：

1. 打开菜单：BoundsChecker|Setting…

2. 在 Error Detection 页中，在 Error Detection Scheme 的 List 中选择 Custom

3. 在 Category 的 Combox 中选择 Pointer and leak error check

4. 钩上 Report Call Stack 复选框

5. 点击 Ok

基于 Code Injection，BoundsChecker 还提供了 API Parameter 的校验功能，memory over run 等功能。这些功能对于程序的开发都非常有益。由于这些内容不属于本文的主题，所以不在此详述了。

尽管 BoundsChecker 的功能如此强大，但是面对隐式内存泄漏仍然显得苍白无力。所以接下来我们看看如何用 Performance Monitor 检测内存泄漏。

2.3.3.3 使用 Performance Monitor 检测内存泄漏

NT 的内核在设计过程中已经加入了系统监视功能，比如 CPU 的使用率，内存的使用情况，I/O 操作的频繁度等都作为一个个 Counter，应用程序可以通过读取这些 Counter 了解整个系统的或者某个进程的运行状况。Performance Monitor 就是这样一个应用程序。

为了检测内存泄漏，我们一般可以监视 Process 对象的 Handle Count，Virutal Bytes 和 Working Set 三个 Counter。Handle Count 记录了进程当前打开的 HANDLE 的个数，监视这个 Counter 有助于我们发现程序是否有 Handle 泄漏；Virtual Bytes 记录了该进程当前在虚地址空间上使用的虚拟内存的大小，NT 的内存分配采用了两步走的方法，首先，在虚地址空间上保留一段空间，这时操作系统并没有分配物理内存，只是保留了一段地址。然后，再提交这段空间，这时操作系统才会分配物理内存。所以，Virtual Bytes 一般总大于程序的 Working Set。监视 Virutal Bytes 可以帮助我们发现一些系统底层的问题; Working Set 记录了操作系统为进程已提交的内存的总量，这个值和程序申请的内存总量存在密切的关系，如果程序存在内存的泄漏这个值会持续增加，但是 Virtual Bytes 却是跳跃式增加的。

监视这些 Counter 可以让我们了解进程使用内存的情况，如果发生了泄漏，即使是隐式内存泄漏，这些 Counter 的值也会持续增加。但是，我们知道有问题却不知道哪里有问题，所以一般使用 Performance Monitor 来验证是否有内存泄漏，而使用 BoundsChecker 来找到和解决。

当 Performance Monitor 显示有内存泄漏，而 BoundsChecker 却无法检测到，这时有两种可能：第一种，发生了偶发性内存泄漏。这时你要确保使用 Performance Monitor 和使用 BoundsChecker 时，程序的运行环境和操作方法是一致的。第二种，发生了隐式的内存泄漏。这时你要重新审查程序的设计，然后仔细研究 Performance Monitor 记录的 Counter 的值的变化图，分析其中的变化和程序运行逻辑的关系，找到一些可能的原因。

这是一个痛苦的过程，充满了假设、猜想、验证、失败，但这也是一个积累经验的绝好机会。

3 探讨 C++内存回收

3.1 C++内存对象大会战

如果一个人自称为程序高手，却对内存一无所知，那么我可以告诉你，他一定在吹牛。用 C 或 C++写程序，需要更多地关注内存，这不仅仅是因为内存的分配是否合理直接影响着程序的效率和性能，更为主要的是，当我们操作内存的时候一不小心就会出现问题，而且很多时候，这些问题都是不易发觉的，比如内存泄漏，比如悬挂指针。笔者今天在这里并不是要讨论如何避免这些问题，而是想从另外一个角度来认识 C++内存对象。

我们知道，C++将内存划分为三个逻辑区域：堆、栈和静态存储区。既然如此，我称位于它们之中的对象分别为堆对象，栈对象以及静态对象。那么这些不同的内存对象有什么区别了？堆对象和栈对象各有什么优劣了？如何禁止创建堆对象或栈对象了？这些便是今天的主题。

3.1.1 基本概念

先来看看栈。栈，一般用于存放局部变量或对象，如我们在函数定义中用类似下面语句声明的对象：

```
Type stack_object ;
```

stack_object 便是一个栈对象，它的生命期是从定义点开始，当所在函数返回时，生命结束。

另外，几乎所有的临时对象都是栈对象。比如，下面的函数定义：

```
Type fun(Type object);
```

这个函数至少产生两个临时对象，首先，参数是按值传递的，所以会调用拷贝构造函数生成一个临时对象 object_copy1 ，在函数内部使用的不是使用的不是 object，而是 object_copy1，自然，object_copy1 是一个栈对象，它在函数返回时被释放；还有这个函数是值返回的，在函数返回时，如果我们不考虑返回值优化（NRV），那么也会产生一个临时对象 object_copy2，这个临时对象会在函数返回后一段时间内被释放。比如某个函数中有如下代码：

```
Type tt ,result ; //生成两个栈对象
```

```
tt = fun(tt); //函数返回时，生成的是一个临时对象 object_copy2
```

上面的第二个语句的执行情况是这样的，首先函数 fun 返回时生成一个临时对象 object_copy2 ，然后再调用赋值运算符执行

```
tt = object_copy2 ; //调用赋值运算符
```

看到了吗？编译器在我们毫无知觉的情况下，为我们生成了这么多临时对象，而生成这些临时对象的时间和空间的开销可能是很大的，所以，你也许明白了，为什么对于“大”对象最好用 const 引用传递代替按值进行函数参数传递了。

接下来，看看堆。堆，又叫自由存储区，它是在程序执行的过程中动态分配的，所以它最大的特性就是动态性。在 C++中，所有堆对象的创建和销毁都要由程序员负责，所以，如果处理不好，就会发生内存问题。如果分配了堆对象，却忘记了释放，就会产生内存泄漏；而如果已释放了对象，却没有将相应的指针置为 NULL，该指针就是所谓的“悬挂指针”，再度使用此指针时，就会出现非法访问，严重时就导致程序崩溃。

那么，C++中是怎样分配堆对象的？唯一的方法就是用 new（当然，用类 malloc 指令也可获得 C 式堆内存），只要使用 new，就会在堆中分配一块内存，并且返回指向该堆对象的指针。

再来看看静态存储区。所有的静态对象、全局对象都于静态存储区分配。关于全局对象，是在 main()函数执行前就分配好了的。其实，在 main()函数中的显示代码执行之前，会调用一个由编译器生成的_main()函数，而_main()函数会进行所有全局对象的的构造及初始化工作。而在 main()函数结束之前，会调用由编译器生成的 exit 函数，来释放所有的全局对象。比如下面的代码：

```
void main（void）

{

  … …// 显式代码

}
```

实际上，被转化成这样：

```
void main（void）
```

```
{
   _main（）; //隐式代码，由编译器产生，用以构造所有全局对象
   … … // 显式代码
   … …
   exit（） ; // 隐式代码，由编译器产生，用以释放所有全局对象
}
```

所以，知道了这个之后，便可以由此引出一些技巧，如，假设我们要在 main()函数执行之前做某些准备工作，那么我们可以将这些准备工作写到一个自定义的全局对象的构造函数中，这样，在 main()函数的显式代码执行之前，这个全局对象的构造函数会被调用，执行预期的动作，这样就达到了我们的目的。 刚才讲的是静态存储区中的全局对象，那么，局部静态对象了？局部静态对象通常也是在函数中定义的，就像栈对象一样，只不过，其前面多了个 static 关键字。局部静态对象的生命期是从其所在函数第一次被调用，更确切地说，是当第一次执行到该静态对象的声明代码时，产生该静态局部对象，直到整个程序结束时，才销毁该对象。

还有一种静态对象，那就是它作为 class 的静态成员。考虑这种情况时，就牵涉了一些较复杂的问题。

第一个问题是 class 的静态成员对象的生命期，class 的静态成员对象随着第一个 class object 的产生而产生，在整个程序结束时消亡。也就是有这样的情况存在，在程序中我们定义了一个 class，该类中有一个静态对象作为成员，但是在程序执行过程中，如果我们没有创建任何一个该 class object，那么也就不会产生该 class 所包含的那个静态对象。还有，如果创建了多个 class object，那么所有这些 object 都共享那个静态对象成员。

第二个问题是，当出现下列情况时：

```
class Base
{
   public:
      static Type s_object ;
}
```

```
class Derived1 : public Base / /  公共继承

{

   … …// other data

}

class Derived2 : public Base / /  公共继承

{

   … …// other data

}

Base example ;

Derivde1 example1 ;

Derivde2 example2 ;

example.s_object = …… ;

example1.s_object = …… ;

example2.s_object = …… ;
```

请注意上面标为黑体的三条语句，它们所访问的 s_object 是同一个对象吗？答案是肯定的，它们的确是指向同一个对象，这听起来不像是真的，是吗？但这是事实，你可以自己写段简单的代码验证一下。我要做的是来解释为什么会这样？ 我们知道，当一个类比如 Derived1，从另一个类比如 Base 继承时，那么，可以看作一个 Derived1 对象中含有一个 Base 型的对象，这就是一个 subobject。一个 Derived1 对象的大致内存布局如下：

让我们想想，当我们将一个 Derived1 型的对象传给一个接受非引用 Base 型参数的函数时会发生切割，那么是怎么切割的呢？相信现在你已经知道了，那就是仅仅取出了 Derived1 型的对象中的 subobject，而忽略了所有 Derived1 自定义的其它数据成员，然后将这个 subobject 传递给函数（实际上，函数中使用的是这个 subobject 的拷贝）。

所有继承 Base 类的派生类的对象都含有一个 Base 型的 subobject（这是能用 Base 型指针指向一个 Derived1 对象的关键所在，自然也是多态的关键了），而所有的

subobject 和所有 Base 型的对象都共用同一个 s_object 对象，自然，从 Base 类派生的整个继承体系中的类的实例都会共用同一个 s_object 对象了。上面提到的 example、example1、example2 的对象布局如下图所示：

3.1.2 三种内存对象的比较

栈对象的优势是在适当的时候自动生成，又在适当的时候自动销毁，不需要程序员操心；而且栈对象的创建速度一般较堆对象快，因为分配堆对象时，会调用 operator new 操作，operator new 会采用某种内存空间搜索算法，而该搜索过程可能是很费时间的，产生栈对象则没有这么麻烦，它仅仅需要移动栈顶指针就可以了。但是要注意的是，通常栈空间容量比较小，一般是 1MB～2MB，所以体积比较大的对象不适合在栈中分配。特别要注意递归函数中最好不要使用栈对象，因为随着递归调用深度的增加，所需的栈空间也会线性增加，当所需栈空间不够时，便会导致栈溢出，这样就会产生运行时错误。

堆对象，其产生时刻和销毁时刻都要程序员精确定义，也就是说，程序员对堆对象的生命具有完全的控制权。我们常常需要这样的对象，比如，我们需要创建一个对象，能够被多个函数所访问，但是又不想使其成为全局的，那么这个时候创建一个堆对象无疑是良好的选择，然后在各个函数之间传递这个堆对象的指针，便可以实现对该对象的共享。另外，相比于栈空间，堆的容量要大得多。实际上，当物理内存不够时，如果这时还需要生成新的堆对象，通常不会产生运行时错误，而是系统会使用虚拟内存来扩展实际的物理内存。

接下来看看 static 对象。

首先是全局对象。全局对象为类间通信和函数间通信提供了一种最简单的方式，虽然这种方式并不优雅。一般而言，在完全的面向对象语言中，是不存在全局对象的，比如 C#，因为全局对象意味着不安全和高耦合，在程序中过多地使用全局对象将大大降低程序的健壮性、稳定性、可维护性和可复用性。C++也完全可以剔除全局对象，但是最终没有，我想原因之一是为了兼容 C。

其次是类的静态成员，上面已经提到，基类及其派生类的所有对象都共享这个静态成员对象，所以当需要在这些 class 之间或这些 class objects 之间进行数据共享或通信时，这样的静态成员无疑是很好的选择。

接着是静态局部对象，主要可用于保存该对象所在函数被屡次调用期间的中间状态，其中一个最显著的例子就是递归函数，我们都知道递归函数是自己调用自己的函数，如果在递归函数中定义一个 nonstatic 局部对象，那么当递归次数相当大时，所产生的开销也是

巨大的。这是因为 nonstatic 局部对象是栈对象，每递归调用一次，就会产生一个这样的对象，每返回一次，就会释放这个对象，而且，这样的对象只局限于当前调用层，对于更深入的嵌套层和更浅露的外层，都是不可见的。每个层都有自己的局部对象和参数。

在递归函数设计中，可以使用 static 对象替代 nonstatic 局部对象（即栈对象），这不仅可以减少每次递归调用和返回时产生和释放 nonstatic 对象的开销，而且 static 对象还可以保存递归调用的中间状态，并且可为各个调用层所访问。

3.1.3 使用栈对象的意外收获

前面已经介绍到，栈对象是在适当的时候创建，然后在适当的时候自动释放的，也就是栈对象有自动管理功能。那么栈对象会在什么会自动释放了？第一，在其生命期结束的时候；第二，在其所在的函数发生异常的时候。你也许说，这些都很正常啊，没什么大不了的。是的，没什么大不了的。但是只要我们再深入一点点，也许就有意外的收获了。

栈对象，自动释放时，会调用它自己的析构函数。如果我们在栈对象中封装资源，而且在栈对象的析构函数中执行释放资源的动作，那么就会使资源泄漏的概率大大降低，因为栈对象可以自动的释放资源，即使在所在函数发生异常的时候。实际的过程是这样的：函数抛出异常时，会发生所谓的 stack_unwinding（堆栈回滚），即堆栈会展开，由于是栈对象，自然存在于栈中，所以在堆栈回滚的过程中，栈对象的析构函数会被执行，从而释放其所封装的资源。除非，除非在析构函数执行的过程中再次抛出异常——而这种可能性是很小的，所以用栈对象封装资源是比较安全的。基于此认识，我们就可以创建一个自己的句柄或代理来封装资源了。智能指针（auto_ptr）中就使用了这种技术。在有这种需要的时候，我们就希望我们的资源封装类只能在栈中创建，也就是要限制在堆中创建该资源封装类的实例。

3.1.4 禁止产生堆对象

上面已经提到，你决定禁止产生某种类型的堆对象，这时你可以自己创建一个资源封装类，该类对象只能在栈中产生，这样就能在异常的情况下自动释放封装的资源。

那么怎样禁止产生堆对象了？我们已经知道，产生堆对象的唯一方法是使用 new 操作，如果我们禁止使用 new 不就行了么。再进一步，new 操作执行时会调用 operator new，而 operator new 是可以重载的。方法有了，就是使 new operator 为 private，为了对称，最好将 operator delete 也重载为 private。现在，你也许又有疑问了,难道创建栈对象不需要调用 new 吗？是的，不需要，因为创建栈对象不需要搜索内存，而是直接调整堆

栈指针，将对象压栈，而 operator new 的主要任务是搜索合适的堆内存，为堆对象分配空间，这在上面已经提到过了。好，让我们看看下面的示例代码：

```
#include <stdlib.h> //需要用到 C 式内存分配函数

class Resource ; //代表需要被封装的资源类

class NoHashObject

{

   private:

      Resource* ptr ;//指向被封装的资源

      ... ... //其它数据成员

      void* operator new(size_t size) //非严格实现，仅作示意之用

      {

         return malloc(size) ;

      }

      void operator delete(void* pp) //非严格实现，仅作示意之用

      {

         free(pp) ;

      }

   public:

      NoHashObject()

      {

         //此处可以获得需要封装的资源，并让 ptr 指针指向该资源

         ptr = new Resource() ;

      }
```

```
    ~NoHashObject()
    {
       delete ptr ; //释放封装的资源
    }
};
```

NoHashObject 现在就是一个禁止堆对象的类了，如果你写下如下代码：

```
NoHashObject* fp = new NoHashObject() ; //编译期错误!
delete fp ;
```

上面代码会产生编译期错误。好了，现在你已经知道了如何设计一个禁止堆对象的类了，你也许和我一样有这样的疑问，难道在类 NoHashObject 的定义不能改变的情况下，就一定不能产生该类型的堆对象了吗？不，还是有办法的，我称之为“暴力破解法”。C++是如此地强大，强大到你可以用它做你想做的任何事情。这里主要用到的是技巧是指针类型的强制转换。

```
void main(void)
{
   char* temp = new char[sizeof(NoHashObject)] ;
   //强制类型转换，现在 ptr 是一个指向 NoHashObject 对象的指针
   NoHashObject* obj_ptr = (NoHashObject*)temp ;
   temp = NULL ; //防止通过 temp 指针修改 NoHashObject 对象
   //再一次强制类型转换，让 rp 指针指向堆中 NoHashObject 对象的 ptr 成员
   Resource* rp = (Resource*)obj_ptr ;
   //初始化 obj_ptr 指向的 NoHashObject 对象的 ptr 成员
   rp = new Resource() ;
   //现在可以通过使用 obj_ptr 指针使用堆中的 NoHashObject 对象成员了
```

```
    ... ...

    delete rp ;//释放资源

    temp = (char*)obj_ptr ;

    obj_ptr = NULL ;//防止悬挂指针产生

    delete [] temp ;//释放 NoHashObject 对象所占的堆空间。

}
```

上面的实现是麻烦的，而且这种实现方式几乎不会在实践中使用，但是我还是写出来路，因为理解它，对于我们理解 C++内存对象是有好处的。对于上面的这么多强制类型转换，其最根本的是什么了？我们可以这样理解：

某块内存中的数据是不变的，而类型就是我们戴上的眼镜，当我们戴上一种眼镜后，我们就会用对应的类型来解释内存中的数据，这样不同的解释就得到了不同的信息。

所谓强制类型转换实际上就是换上另一副眼镜后再来看同样的那块内存数据。

另外要提醒的是，不同的编译器对对象的成员数据的布局安排可能是不一样的，比如，大多数编译器将 NoHashObject 的 ptr 指针成员安排在对象空间的头 4 个字节，这样才会保证下面这条语句的转换动作像我们预期的那样执行：

```
Resource* rp = (Resource*)obj_ptr ;
```

但是，并不一定所有的编译器都是如此。

既然我们可以禁止产生某种类型的堆对象，那么可以设计一个类，使之不能产生栈对象吗？当然可以。

3.1.5 禁止产生栈对象

前面已经提到了，创建栈对象时会移动栈顶指针以“挪出”适当大小的空间，然后在这个空间上直接调用对应的构造函数以形成一个栈对象，而当函数返回时，会调用其析构函数释放这个对象，然后再调整栈顶指针收回那块栈内存。在这个过程中是不需要 operator new/delete 操作的，所以将 operator new/delete 设置为 private 不能达到目的。当然从上面的叙述中，你也许已经想到了：将构造函数或析构函数设为私有的，这样系统就不能调用构造/析构函数了，当然就不能在栈中生成对象了。

这样的确可以，而且我也打算采用这种方案。但是在此之前，有一点需要考虑清楚,那就是，如果我们将构造函数设置为私有，那么我们也就不能用 new 来直接产生堆对象了，因为 new 在为对象分配空间后也会调用它的构造函数啊。所以，我打算只将析构函数设置为 private。再进一步，将析构函数设为 private 除了会限制栈对象生成外，还有其它影响吗？是的，这还会限制继承。

如果一个类不打算作为基类，通常采用的方案就是将其析构函数声明为 private。

为了限制栈对象，却不限制继承，我们可以将析构函数声明为 protected，这样就两全其美了。如下代码所示：

```
class NoStackObject

{

   protected:

      ~NoStackObject() { }

   public:

      void destroy()

      {

         delete this ;//调用保护析构函数

      }

};
```

接着，可以像这样使用 NoStackObject 类：

```
NoStackObject* hash_ptr = new NoStackObject() ;

... ... //对 hash_ptr 指向的对象进行操作

hash_ptr->destroy() ;
```

呵呵，是不是觉得有点怪怪的，我们用 new 创建一个对象，却不是用 delete 去删除它，而是要用 destroy 方法。很显然，用户是不习惯这种怪异的使用方式的。所以，我决定将构造函数也设为 private 或 protected。这又回到了上面曾试图避免的问题，即不用

new，那么该用什么方式来生成一个对象了？我们可以用间接的办法完成，即让这个类提供一个 static 成员函数专门用于产生该类型的堆对象。（设计模式中的 singleton 模式就可以用这种方式实现。）让我们来看看：

```
class NoStackObject

{

   protected:

      NoStackObject() { }

      ~NoStackObject() { }

   public:

      static NoStackObject* creatInstance()

      {

         return new NoStackObject() ;//调用保护的构造函数

      }

      void destroy()

      {

         delete this ;//调用保护的析构函数

      }

};
```

现在可以这样使用 NoStackObject 类了：

```
NoStackObject* hash_ptr = NoStackObject::creatInstance() ;

... ... //对 hash_ptr 指向的对象进行操作

hash_ptr->destroy() ;

hash_ptr = NULL ; //防止使用悬挂指针
```

现在感觉是不是好多了，生成对象和释放对象的操作一致了。

3.2 浅议 C++ 中的垃圾回收方法

许多 C 或者 C++ 程序员对垃圾回收嗤之以鼻，认为垃圾回收肯定比自己来管理动态内存要低效，而且在回收的时候一定会让程序停顿在那里，而如果自己控制内存管理的话，分配和释放时间都是稳定的，不会导致程序停顿。最后，很多 C/C++ 程序员坚信在 C/C++ 中无法实现垃圾回收机制。这些错误的观点都是由于不了解垃圾回收的算法而臆想出来的。

其实垃圾回收机制并不慢，甚至比动态内存分配更高效。因为我们可以只分配不释放，那么分配内存的时候只需要从堆上一直的获得新的内存，移动堆顶的指针就够了；而释放的过程被省略了，自然也加快了速度。现代的垃圾回收算法已经发展了很多，增量收集算法已经可以让垃圾回收过程分段进行，避免打断程序的运行了。而传统的动态内存管理的算法同样有在适当的时间收集内存碎片的工作要做，并不比垃圾回收更有优势。

而垃圾回收的算法的基础通常基于扫描并标记当前可能被使用的所有内存块，从已经被分配的所有内存中把未标记的内存回收来做的。C/C++ 中无法实现垃圾回收的观点通常基于无法正确扫描出所有可能还会被使用的内存块，但是，看似不可能的事情实际上实现起来却并不复杂。首先，通过扫描内存的数据，指向堆上动态分配出来内存的指针是很容易被识别出来的，如果有识别错误，也只能是把一些不是指针的数据当成指针，而不会把指针当成非指针数据。这样，回收垃圾的过程只会漏回收掉而不会错误的把不应该回收的内存清理。其次，如果回溯所有内存块被引用的根，只可能存在于全局变量和当前的栈内，而全局变量(包括函数内的静态变量)都是集中存在于 bss 段或 data 段中。

垃圾回收的时候，只需要扫描 bss 段, data 段以及当前被使用着的栈空间，找到可能是动态内存指针的量，把引用到的内存递归扫描就可以得到当前正在使用的所有动态内存了。

如果肯为你的工程实现一个不错的垃圾回收器，提高内存管理的速度，甚至减少总的内存消耗都是可能的。如果有兴趣的话，可以搜索一下网上已有的关于垃圾回收的论文和实现了的库，开拓视野对一个程序员尤为重要。