C/C++内存管理详解

By ShinChan

Published Sep 25 2014

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。

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。

内存分配方式

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。

明确区分堆与栈

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，编译器就会根据相应的 `Cookie` 信息去进行释放内存的工作。

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：）

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。

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。

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。

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”。**
那么如何避免产生野指针呢？这里列出了5条规则，平常写程序时多注意一下，养成良好的习惯。

> 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”。

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。

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;
```

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)

…
```

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

}
```

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。

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”的成因主要有三种：

(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 的生命周
    }
    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。

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`delet`e。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。

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"。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。

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);
    }
}
```

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`mallo`c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。

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。

Conclusion

- 越是怕指针，就越要使用指针。不会正确使用指针，肯定算不上是合格的程序员。
- 必须养成使用"调试器逐步跟踪程序"的习惯，只有这样才能发现问题的本质。

Develop C/C++

上一篇：
递归算法详解
下一篇：
简明 Vim 练级攻略

Categories

Algorithm 2

Tags

Python [4] Machine Learning [3] C/C++ [3] NLP [3] Data Mining [2] TextEditor [2] Tools [2] Linux [2] Scrapy [2] Algorithm [2] Text Mining [1] Java [1] Vim [1] Quant [1] Spark [1] Hadoop [1] MySQL [1] MongoDB [1] Markdown [1] Design Pattern [1]

Links

GitHub
PAMI Lab

RSS