堆和栈的区别   
一、预备知识—程序的内存分配   
一个由c/C++编译的程序占用的内存分为以下几个部分   
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，呵呵。   
3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后有系统释放   
4、文字常量区—常量字符串就是放在这里的。 程序结束后由系统释放   
5、程序代码区—存放函数体的二进制代码。   
二、例子程序   
这是一个前辈写的，非常详细   
//main.cpp   
int a = 0; 全局初始化区   
char \*p1; 全局未初始化区   
main()   
{   
int b; 栈   
char s[] = "abc"; 栈   
char \*p2; 栈   
char \*p3 = "123456"; 123456\0在常量区，p3在栈上。   
static int c =0； 全局（静态）初始化区   
p1 = (char \*)malloc(10);   
p2 = (char \*)malloc(20);   
分配得来得10和20字节的区域就在堆区。   
strcpy(p1, "123456"); 123456\0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。   
}    
  
一个程序一般分为3段:text段,data段,bss段   
text段:就是放程序代码的,编译时确定,只读,   
data段:存放在编译阶段(而非运行时)就能确定的数据,可读可写   
就是通常所说的静态存储区,赋了初值的全局变量和静态变量存放在这个区域,常量也存放在这个区域   
bss段:定义而没有赋初值的全局变量和静态变量,放在这个区域

[C语言 内存管理详解](http://club.topsage.com/thread-443540-1-1.html)

　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、内存分配方式**  
  
　　内存分配方式有三种：  
  
　　（1）从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。  
  
　　（2）在栈上创建。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。  
  
　　（3） 从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。  
  
　　**2、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”。  
  
　　**3、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。  
  
　　3.1 修改内容  
  
　　示例3-1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。

1. char a[] = “hello”;
2. a[0] = ‘X’;
3. cout << a << endl;
4. char \*p = “world”; // 注意p指向常量字符串
5. p[0] = ‘X’; // 编译器不能发现该错误
6. cout << p << endl;

复制代码

示例3.1 修改数组和指针的内容  
  
　　3.2 内容复制与比较  
  
　　不能对数组名进行直接复制与比较。示例7-3-2中，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。

1. // 数组…
2. char a[] = "hello";
3. char b[10];
4. strcpy(b, a); // 不能用 b = a;
5. if(strcmp(b, a) == 0) // 不能用 if (b == a)
6. …
7. // 指针…
8. int len = strlen(a);
9. char \*p = (char \*)malloc(sizeof(char)\*(len 1));
10. strcpy(p,a); // 不要用 p = a;
11. if(strcmp(p, a) == 0) // 不要用 if (p == a)
12. …

复制代码

示例3.2 数组和指针的内容复制与比较  
  
　　3.3 计算内存容量  
  
　　用运算符sizeof可以计算出数组的容量（字节数）。示例7-3-3（a）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。  
  
　　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。示例7-3-3（b）中，不论数组a的容量是多少，sizeof(a)始终等于sizeof(char \*)。

1. char a[] = "hello world";
2. char \*p = a;
3. cout<< sizeof(a) << endl; // 12字节
4. cout<< sizeof(p) << endl; // 4字节

复制代码

示例3.3（a） 计算数组和指针的内存容量

1. void Func(char a[100])
2. {
3. cout<< sizeof(a) << endl; // 4字节而不是100字节
4. }

复制代码

示例3.3（b） 数组退化为指针  
  
  
**4、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。示例7-4-1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？

1. void GetMemory(char \*p, int num)
2. {
3. p = (char \*)malloc(sizeof(char) \* num);
4. }
5. void Test(void)
6. {
7. char \*str = NULL;
8. GetMemory(str, 100); // str 仍然为 NULL
9. strcpy(str, "hello"); // 运行错误
10. }

复制代码

示例4.1 试图用指针参数申请动态内存  
  
　　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

，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4.2。

1. void GetMemory2(char \*\*p, int num)
2. {
3. \*p = (char \*)malloc(sizeof(char) \* num);
4. }
5. void Test2(void)
6. {
7. char \*str = NULL;
8. GetMemory2(&str, 100); // 注意参数是 &str，而不是str
9. strcpy(str, "hello");
10. cout<< str << endl;
11. free(str);
12. }

复制代码

示例4.2用指向指针的指针申请动态内存  
  
　　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4.3。

1. char \*GetMemory3(int num)
2. {
3. char \*p = (char \*)malloc(sizeof(char) \* num);
4. return p;
5. }
6. void Test3(void)
7. {
8. char \*str = NULL;
9. str = GetMemory3(100);
10. strcpy(str, "hello");
11. cout<< str << endl;
12. free(str);
13. }

复制代码

示例4.3 用函数返回值来传递动态内存  
  
　　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4.4。

1. char \*GetString(void)
2. {
3. char p[] = "hello world";
4. return p; // 编译器将提出警告
5. }
6. void Test4(void)
7. {
8. char \*str = NULL;
9. str = GetString(); // str 的内容是垃圾
10. cout<< str << endl;
11. }

复制代码

示例4.4 return语句返回指向“栈内存”的指针  
  
　　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。  
如果把示例4.4改写成示例4.5，会怎么样？

1. char \*GetString2(void)
2. {
3. char \*p = "hello world";
4. return p;
5. }
6. void Test5(void)
7. {
8. char \*str = NULL;
9. str = GetString2();
10. cout<< str << endl;
11. }

复制代码

示例4.5 return语句返回常量字符串  
  
　　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。  
  
　　**5、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

1. char \*p = NULL;
2. char \*str = (char \*) malloc(100);

复制代码

（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范围。这种情况让人防不胜防，示例程序如下：

1. class A
2. {
3. public:
4. void Func(void){ cout << “Func of class A” << endl; }
5. };
6. void Test(void)
7. {
8. A \*p;
9. {
10. A a;
11. p = &a; // 注意 a 的生命期
12. }
13. p->Func(); // p是“野指针”
14. }

复制代码

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。  
  
  
　　**6、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6。

1. class Obj
2. {
3. public :
4. Obj(void){ cout << “Initialization” << endl; }
5. ~Obj(void){ cout << “Destroy” << endl; }
6. void Initialize(void){ cout << “Initialization” << endl; }
7. void Destroy(void){ cout << “Destroy” << endl; }
8. };
9. void UseMallocFree(void)
10. {
11. Obj \*a = (obj \*)malloc(sizeof(obj)); // 申请动态内存
12. a->Initialize(); // 初始化
13. //…
14. a->Destroy(); // 清除工作
15. free(a); // 释放内存
16. }
17. void UseNewDelete(void)
18. {
19. Obj \*a = new Obj; // 申请动态内存并且初始化
20. //…
21. delete a; // 清除并且释放内存
22. }

复制代码

示例6 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。  
  
　　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。  
  
　　**7、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：

1. void Func(void)
2. {
3. A \*a = new A;
4. if(a == NULL)
5. {
6. return;
7. }
8. …
9. }

复制代码

（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：

1. void Func(void)
2. {
3. A \*a = new A;
4. if(a == NULL)
5. {
6. cout << “Memory Exhausted” << endl;
7. exit(1);
8. }
9. …
10. }

复制代码

（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。

1. void main(void)
2. {
3. float \*p = NULL;
4. while(TRUE)
5. {
6. p = new float[1000000];
7. cout << “eat memory” << endl;
8. if(p==NULL)
9. exit(1);
10. }
11. }

复制代码

示例7试图耗尽操作系统的内存  
  
  
　　**8、malloc/free 的使用要点**  
  
　　函数malloc的原型如下：

1. void \* malloc(size\_t size);

复制代码

用malloc申请一块长度为length的整数类型的内存，程序如下：

1. int \*p = (int \*) malloc(sizeof(int) \* length);

复制代码

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：

1. cout << sizeof(char) << endl;
2. cout << sizeof(int) << endl;
3. cout << sizeof(unsigned int) << endl;
4. cout << sizeof(long) << endl;
5. cout << sizeof(unsigned long) << endl;
6. cout << sizeof(float) << endl;
7. cout << sizeof(double) << endl;
8. cout << sizeof(void \*) << endl;

复制代码

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：

1. void free( void \* memblock );

复制代码

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。  
  
　　**9、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：

1. int \*p1 = (int \*)malloc(sizeof(int) \* length);
2. int \*p2 = new int[length];

复制代码

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

1. class Obj
2. {
3. public :
4. Obj(void); // 无参数的构造函数
5. Obj(int x); // 带一个参数的构造函数
6. …
7. }
8. void Test(void)
9. {
10. Obj \*a = new Obj;
11. Obj \*b = new Obj(1); // 初值为1
12. …
13. delete a;
14. delete b;
15. }

复制代码

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

1. Obj \*objects = new Obj[100]; // 创建100个动态对象

复制代码

不能写成

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

复制代码

在用delete释放对象数组时，留意不要丢了符号‘[]’。例如

1. delete []objects; // 正确的用法
2. delete objects; // 错误的用法

复制代码

后者相当于delete objects[0]，漏掉了另外99个对象。  
  
　　**10、一些心得体会**  
  
　　我认识不少技术不错的C /C程序员，很少有人能拍拍胸脯说通晓指针与内存管理（包括我自己）。我最初学习C语言时特别怕指针，导致我开发第一个应用软件（约1万行C代码）时没有使用一个指针，全用数组来顶替指针，实在蠢笨得过分。躲避指针不是办法，后来我改写了这个软件，代码量缩小到原先的一半。  
  
　　我的经验教训是：  
  
　　（1）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。  
  
　　（2）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。

C语言程序的内存分配方式  
  
1.内存分配方式  
  
　　内存分配方式有三种：  
　　[1]从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。  
　　[2]在栈上创建。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。  
　　[3]从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由程序员决定，使用非常灵活，但如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，频繁地分配和释放不同大小的堆空间将会产生堆内碎块。  
  
2.程序的内存空间  
  
　　一个程序将操作系统分配给其运行的内存块分为4个区域，如下图所示。  
　　一个由C/C++编译的程序占用的内存分为以下几个部分,  
　　1、栈区（stack）—　 由编译器自动分配释放 ，存放为运行函数而分配的局部变量、函数参数、返回数据、返回地址等。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 。分配方式类似于链表。  
　　3、全局区（静态区）（static）—存放全局变量、静态数据、常量。程序结束后由系统释放。  
　　4、文字常量区 —常量字符串就是放在这里的。 程序结束后由系统释放。  
　　5、程序代码区—存放函数体（类成员函数和全局函数）的二进制代码。  
  
　　下面给出例子程序，  
  
　　int a = 0; //全局初始化区  
　　char \*p1; //全局未初始化区  
　　int main() {  
　　int b; //栈  
　　char s[] = "abc"; //栈  
　　char \*p2; //栈  
　　char \*p3 = "123456"; //123456在常量区，p3在栈上。  
　　static int c =0;//全局（静态）初始化区  
　　p1 = new char[10];  
　　p2 = new char[20];  
　　//分配得来得和字节的区域就在堆区。  
　　strcpy(p1, "123456"); //123456放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。  
　　}  
  
3．堆与栈的比较  
　　  
3.1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，C++中是new运算符。  
　　如p1 = (char \*)malloc(10); p1 = new char[10];  
　　如p2 = (char \*)malloc(10); p2 = new char[20];  
　　但是注意p1、p2本身是在栈中的。  
　　  
3.2申请后系统的响应  
  
　　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。  
　　对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。  
　　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。  
　　  
3.3申请大小的限制  
  
　　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。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。  
　　  
3.4申请效率的比较  
　　栈由系统自动分配，速度较快。但程序员是无法控制的。  
　　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。  
　　另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，他不是在堆，也不是栈，而是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。  
　　  
3.5堆和栈中的存储内容  
  
　　栈：在函数调用时，第一个进栈的是主函数中后的下一条指令（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。堆中的具体内容有程序员安排。  
　　  
3.6存取效率的比较  
  
　　char s1[] = "a";  
　　char \*s2 = "b";  
　　a是在运行时刻赋值的；而b是在编译时就确定的；但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。  
比如：  
　　int　main(){  
　　char a = 1;  
　　char c[] = "1234567890";  
　　char \*p ="1234567890";  
　　a = c[1];  
　　a = p[1];  
　　return 0;  
　　}  
　　对应的汇编代码  
　　10: a = c[1];  
　　00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]  
　　0040106A 88 4D FC mov byte ptr [ebp-4],cl  
　　11: a = p[1];  
　　0040106D 8B 55 EC mov edx,dword ptr [ebp-14h]  
　　00401070 8A 42 01 mov al,byte ptr [edx+1]  
　　00401073 88 45 FC mov byte ptr [ebp-4],al  
　　第一种在读取时直接就把字符串中的元素读到寄存器cl中，而第二种则要先把指针值读到edx中，再根据edx读取字符，显然慢了。  
  
3.7小结  
  
　　堆和栈的主要区别由以下几点：  
　　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。当然，这个值可以修改。  
　　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。  
　　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mallo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。

[**c++内存分配[整理]**](http://blog.csdn.net/llhyy17/article/details/5357790)

2010-03-08 20:48 8人阅读 [评论](http://blog.csdn.net/llhyy17/article/details/5357790#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/llhyy17/article/details/5357790#report)

一、内存基本构成  
可编程内存在基本上分为这样的几大部分：静态存储区、堆区和栈区。他们的功能不同，对他们使用方式也就不同。  
静态存储区：内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。它主要存放静态数据、全局数据和常量。  
栈区：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。  
堆区：亦称动态内存分配。程序在运行的时候用malloc或new申请任意大小的内存，程序员自己负责在适当的时候用free或delete释放内存。动态内存的生存期可以由我们决定，如果我们不释放内存，程序将在最后才释放掉动态内存。但是，良好的编程习惯是：如果某动态内存不再使用，需要将其释放掉，否则，我们认为发生了内存泄漏现象。  
  
二、三者之间的区别  
我们通过代码段来看看对这样的三部分内存需要怎样的操作和不同，以及应该注意怎样的地方。  
例一：静态存储区与栈区  
  
char\* p = “Hello World1”;  
  
char a[] = “Hello World2”;  
  
p[2] = ‘A’;  
  
a[2] = ‘A’;  
  
char\* p1 = “Hello World1;”  
  
  
这个程序是有错误的，错误发生在p[2] = ‘A’这行代码处，为什么呢，是变量p和变量数组a都存在于栈区的（任何临时变量都是处于栈区的，包括在main（）函数中定义的变量）。但是，数据 “Hello World1”和数据“Hello World2”是存储于不同的区域的。  
  
因为数据“Hello World2”存在于数组中，所以，此数据存储于栈区，对它修改是没有任何问题的。因为指针变量p仅仅能够存储某个存储空间的地址，数据“Hello World1”为**字符串常量**，所以**存储在静态存储区**。虽然通过p[2]可以访问到静态存储区中的第三个数据单元，即字符‘l’所在的存储的单元。但是因为数据“Hello World1”为**字符串常量**，不可以改变，所以在程序运行时，会报告内存错误。并且，如果此时对p和p1输出的时候会发现p和p1里面保存的地址是完全相同的。换句话说，在数据区只保留一份相同的数据（见图1－1）。  
  
例二：栈区与堆区  
  
char\* f1()  
  
{  
  
char\* p = NULL;  
  
char a;  
  
p = &a;  
  
return p;  
  
}  
  
char\* f2()  
  
{  
  
char\* p = NULL:  
  
p =(char\*) new char[4];  
  
return p;  
  
}  
  
这两个函数都是将某个存储空间的地址返回，二者有何区别呢？f1()函数虽然返回的是一个存储空间，但是此空间为临时空间。也就是说，此空间只有短暂的生命周期，它的生命周期在函数f1()调用结束时，也就失去了它的生命价值，即：此空间被释放掉。所以，当调用f1()函数时，如果程序中有下面的语句：  
  
  
  
char\* p ;  
  
p = f1();  
  
\*p = ‘a’;  
  
此时，编译并不会报告错误，但是在程序运行时，会发生异常错误。因为，你对不应该操作的内存（即，已经释放掉的存储空间）进行了操作。但是，相比之下， f2()函数不会有任何问题。因为，new这个命令是在堆中申请存储空间，一旦申请成功，除非你将其delete或者程序终结，这块内存将一直存在。也可以这样理解，堆内存是共享单元，能够被多个函数共同访问。如果你需要有多个数据返回却苦无办法，堆内存将是一个很好的选择。但是一定要避免下面的事情发生：  
  
  
  
void f()  
  
{  
  
…  
  
char \* p;  
  
p = (char\*)new char[100];  
  
…  
  
}  
  
  
这个程序做了一件很无意义并且会带来很大危害的事情。因为，虽然申请了堆内存，p保存了堆内存的首地址。但是，此变量是临时变量，当函数调用结束时p变量消失。也就是说，再也没有变量存储这块堆内存的首地址，我们将永远无法再使用那块堆内存了。但是，这块堆内存却一直标识被你所使用（因为没有到程序结束，你也没有将其delete，所以这块堆内存一直被标识拥有者是当前您的程序），进而其他进程或程序无法使用。我们将这种不道德的“流氓行为”（我们不用，却也不让别人使用）称为内存泄漏。这是我们C++程序员的大忌！！请大家一定要避免这件事情的发生。  
  
总之，对于堆区、栈区和静态存储区它们之间最大的不同在于，栈的生命周期很短暂。但是堆区和静态存储区的生命周期相当于与程序的生命同时存在（如果您不在程序运行中间将堆内存delete的话），我们将这种变量或数据成为全局变量或数据。但是，对于堆区的内存空间使用更加灵活，因为它允许你在不需要它的时候，随时将它释放掉，而静态存储区将一直存在于程序的整个生命周期中。  
我们此专题仅仅是简要的分析了内存基本构成以及使用它们时需要注意的问题。对内存的分析和讨论将一直贯穿于我们以后所有的专题，这也就是为什么把它作为第一讲的原因。

1.以字符串形式出现的，编译器都会为该字符串自动添加一个0作为结束符，如在代码中写  
  "abc",那么编译器帮你存储的是"abc/0"

2."abc"是常量吗？答案是有时是，有时不是。

  不是常量的情况："abc"作为字符数组初始值的时候就不是，如  
                  char str[] = "abc";  
    因为定义的是一个字符数组，所以就相当于定义了一些空间来存放"abc"，而又因为  
    字符数组就是把字符一个一个地存放的，所以编译器把这个语句解析为  
    char str[3] = {'a','b','c'};  
                  又根据上面的总结1，所以char str[] = "abc";的最终结果是  
    char str[4] = {'a','b','c','/0'};  
    做一下扩展，如果char str[] = "abc";是在函数内部写的话，那么这里  
    的"abc/0"因为不是常量，所以应该被放在栈上。  
    
  是常量的情况：  把"abc"赋给一个字符指针变量时，如  
                  char\* ptr = "abc";  
    因为定义的是一个普通指针，并没有定义空间来存放"abc"，所以编译器得帮我们  
    找地方来放"abc"，显然，把这里的"abc"当成常量并把它放到程序的常量区是编译器  
    最合适的选择。所以尽管ptr的类型不是const char\*，并且ptr[0] = 'x';也能编译  
    通过，但是执行ptr[0] = 'x';就会发生运行时异常，因为这个语句试图去修改程序  
    常量区中的东西。  
    记得哪本书中曾经说过char\* ptr = "abc";这种写法原来在c++标准中是不允许的，  
    但是因为这种写法在c中实在是太多了，为了兼容c，不允许也得允许。虽然允许，  
    但是建议的写法应该是const char\* ptr = "abc";这样如果后面写ptr[0] = 'x'的  
    话编译器就不会让它编译通过，也就避免了上面说的运行时异常。  
    又扩展一下，如果char\* ptr = "abc";写在函数体内，那么虽然这里的"abc/0"被  
    放在常量区中，但是ptr本身只是一个普通的指针变量，所以ptr是被放在栈上的，  
    只不过是它所指向的东西被放在常量区罢了。

3.数组的类型是由该数组所存放的东西的类型以及数组本身的大小决定的。  
  如char s1[3]和char s2[4]，s1的类型就是char[3]，s2的类型就是char[4]，  
  也就是说尽管s1和s2都是字符数组，但两者的类型却是不同的。

4.字符串常量的类型可以理解为相应字符常量数组的类型，  
  如"abcdef"的类型就可以看成是const char[7]

5.sizeof是用来求类型的字节数的。如int a;那么无论sizeof(int)或者是sizeof(a)都  
  是等于4，因为sizeof(a)其实就是sizeof(type of a)

6.对于函数参数列表中的以数组类型书写的形式参数，编译器把其解释为普通  
  的指针类型，如对于void func(char sa[100],int ia[20],char \*p)  
  则sa的类型为char\*，ia的类型为int\*，p的类型为char\*

7.根据上面的总结，来实战一下：  
  对于char str[] = "abcdef";就有sizeof(str) == 7,因为str的类型是char[7]，  
  也有sizeof("abcdef") == 7，因为"abcdef"的类型是const char[7]。  
  对于char \*ptr = "abcdef";就有sizeof(ptr) == 4，因为ptr的类型是char\*。  
  对于char str2[10] = "abcdef";就有sizeof(str2) == 10，因为str2的类型是char[10]。  
  对于void func(char sa[100],int ia[20],char \*p);  
  就有sizeof(sa) == sizeof(ia) == sizeof(p) == 4，  
  因为sa的类型是char\*，ia的类型是int\*，p的类型是char\*。