先看一下两段代码：

1. **char**\* toStr()
2. {
3. **char** \*s = "abcdefghijkl";
4. **return** s;
5. }
6. **int** main()
7. {
8. cout << toStr() << endl;
9. **return** 0;
10. }

和

1. **char**\* toStr()
2. {
3. **char** s[] = "abcdefghijkl";
4. **return** s;
5. }
6. **int** main()
7. {
8. cout << toStr() << endl;
9. **return** 0;
10. }

前一段代码打印出来是字符串，而后一段代码打印出来就是乱码。记得学**[c语言](http://lib.csdn.net/base/c" \o "C语言知识库" \t "_blank)**的时候讲到，字符串是被当做字符数组来处理的。所以字符数组名就相当于指向首地址的指针。那么  
1. char \*s = "abcdefghijkl";  
2. char s[] = "abcdefghijkl";  
这两种表达式似乎是一样的，可是为什么程序结果会不一样呢？原因就是没有对内存分配了解好。当然现在的C语言教材不会讲到的。  
解释：  
程序的意思比较简单，不用解释。  
第一种表达式，指针s是局部变量，他的作用域是函数toStr内。它将其指向的地址返回，返回之后s即被销毁，庆幸s指向的地址被返回了回来。最终打印正确。  
第二种表达式，那么我们会问第二种与第一种的区别在哪，为何错？原因就是第一种指针s虽然是局部变量，被分配在栈空间，作用域是函数内部，但其指向的内容"abcdefghijkl"是常量，被分配在程序的常量区。直到整个程序结束才被销毁。而第二种，s是一数组，分配到栈空间，"abcdefghijkl"作为数组各个元素被放到数组中，一旦函数退出，栈中这块内存就被释放。虽然返回一个地址，可是已经失去它的意义了。

一、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**](http://lib.csdn.net/base/datastructure)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—存放函数体的二进制代码。

从上到下：

Stack——自动分配释放的一个区，存在参数的值，局部变量的值。

|  |  |
| --- | --- |
| Stack | 函数的参数值，局部变量的值 ————stack |
| Heap | 程序结束之后释放——————————链表 Chain |
| Static | 全局变量和Static’全局作用域的东西 （无论初始化与否） |
| 文字常量区 | 文字常量const级别 ，字符串常量 |
| OS |  |

这是一个前辈写的，非常详细

1. //main.cpp
2. **int** a = 0; //全局初始化区  3
3. **char** \*p1; //全局未初始化区  3
5. main()
6. {
7. **int** b; //栈  1
8. **char** s[] = "abc"; //栈  1
9. **char** \*p2; //栈  1
10. **char** \*p3 = "123456"; //123456\\0在常量区，p3在栈上。  1 4
11. **static** **int** c =0;//全局（静态）初始化区  3
12. p1 = (**char** \*)malloc(10);
13. p2 = (**char** \*)malloc(20);//分配得来得10和20字节的区域就在堆区。  2
14. 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。
15. }

这下就对程序的内存分配理解更深入了吧。

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 【自动变量】

heap: 需要程序员自己申请，并指明大小，在c中malloc函数 以及reallco函数

如p1 = (char \*)malloc(10);

在C++中用new运算符

如p2 = (char \*)malloc(10);

但是注意p1、p2本身是在栈中的。

就是他们指向的内存是第四层，但是说他们的指针存在的位置是第一层

# 存储类别 链接和 内存管理

用storage duration 描述对象

可以用scope和linkage来修饰标识符

不同的存储类别 具有不同的storage duration scope 和linkage

Scope作用域：

块作用域 函数作用域 和 函数原型作用域 以及文件作用域

函数的形式参数也是块作用域；

函数原型作用域： 形式参数的定义处到 原型声明结束——也就是说函数的形式参数的名字其实不重要。

文件作用域： 变量定义在函数的外边 【全局变量】

链接——外部链接+内部链接+无链接

块作用域 函数作用域 函数原型作用域 【无链接变量】——变量都是私有的

文件作用域的变量——外部链接或者内部链接——

翻译单元——头文件+源代码文件

Static就是内部链接说明符——static+文件作用域=内部链接的文件作用域（一个cpp only）

而一般的文件作用域变量（就是在main外边声明的变量——cpps ok）【成为全局变量】

Scope ——块 函数 函数原型 文件

Storage duration ——静态存储（static） 线程 自动（一般的局部变量） 动态分配

文件作用域的变量都具有静态存储期，块作用域一般是有着自动存储期的，

自动——自动存储变量占据的工作空间被认为是一个缓冲区感觉。

变长数组的存储期是从声明的位置到块的末尾。

局部变量都都是自动存储期

Storage duration ——scope——linkage——构成了5种存储方式：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Storage duration | Scope | Linkage |  |
| 自动 | 自动 | 块 | 无 | 块 |
| 寄存 | 自从 | 块 | 无 | 块 register |
| 静态内部链接 | 静态 | 文件 | 外部 | Static 外边 |
| 静态外部链接 | 静态 | 文件 | 内部 | 全局变量自动外边 |
| 静态无链接 | 静态 | 块 | 无 | Static 块 |

C中的auto的意思是存储类别声明，而不是一个自动化的变量声明

Static的意思就是静态变量的存储位置原地不变，也就是这个内存的位置直到程序结束之前都不会被分配出去。

Extern 用来声明你的这个东西是一个外部链接的静态存储变量。

外部变量的存储期也是直到程序结束为止。

1. char \*ptr = (char \*)malloc(10); // 分配10个字节的内存空间，用来存放字符

由于申请内存空间时可能有也可能没有，所以需要自行判断是否申请成功，再进行后续操作。

If （！L.elem）也就是来声明到底你的内存访问是不是成功的。

void\* malloc (size\_t size); void\*的意思是不知道返回类型是啥，而不是没有返回类型。

Include <stdlib.h> 也就是一个标准的库函数，包括了malloc函数。

Malloc函数 ，分配成功返回指向该内存的地址，失败则返回 NULL。返回的是一个指针。

注意：函数的返回值类型是 void \*，void 并不是说没有返回值或者返回空指针，而是返回的指针类型未知。所以在使用 malloc() 时通常需要进行强制类型转换，将 void 指针转换成我们希望的类型，例如

Malloc函数获得的内存就是存储在heap里面，然后对应的是程序员手动输入的buffer（）

Char\* buffer；

Buffer = （char\*）malloc（size\_string）；

返回的类型是一个char指针，指向新分配的内存

C库函数 void free(void \*ptr) 由calloc，malloc或realloc调用先前分配的回收内存。

void \*realloc(void \*ptr, size\_t size)

返回值是一个新的指针，然后也要进行验证到底是不是成功的。

C库函数 void \*calloc(size\_t nitems, size\_t size) 分配请求的内存，并返回一个指向它的指针。的malloc和calloc的区别是，malloc不设置内存calloc为零，其中作为分配的内存设置为零。

Calloc函数一般的对象时数组，因为里面的size\_t nitems就是用来判断到底分配多少个元素。