**有关于char数组与char\*关于内存所在位置的探究**

Diagram

Description automatically generated

问题引入：  
在实习过程中发现了一个以前一直默认的错误，同样char \*c = "abc"和char c[]="abc",前者改变其内

容程序是会崩溃的，而后者完全正确。  
程序演示：  
测试环境Devc++  
代码  
运行结果  
2293628 4199056 abc  
2293624 2293624 abc  
2293620 4199056 abc

#include <iostream>  
using namespace std;

main()  
{  
   char \*c1 = "abc";  
   char c2[] = "abc";  
   char \*c3 = ( char\* )malloc(3);  
   c3 = "abc";  
   printf("%d %d %s\n",&c1,c1,c1);  
   printf("%d %d %s\n",&c2,c2,c2);  
   printf("%d %d %s\n",&c3,c3,c3);  
   getchar();  
}   

参考资料：  
首先要搞清楚编译程序占用的内存的分区形式：  
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  
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所向"123456"优化成一个地方。  
}  
二、堆和栈的理论知识  
2.1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  
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。  
2.2  
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，另外，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。另外，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。  
2.3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。  
2.4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Virtual Alloc分配内存，他不是在堆，也不是在栈,而是直接在进

程的地址空间中保留一块内存，虽然用起来最不方便。但是速度快，也最灵活。  
2.5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。堆中的具体内容由程序员安排。  
2.6存取效率的比较  
char s1[]="aaaaaaaaaaaaaaa";  
char \*s2="bbbbbbbbbbbbbbbbb";  
aaaaaaaaaaa是在运行时刻赋值的；  
而bbbbbbbbbbb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：  
#include  
voidmain()  
{  
char a=1;  
char c[]="1234567890";  
char \*p="1234567890";  
a = c[1];  
a = p[1];  
return;  
}  
对应的汇编代码  
10:a=c[1];  
004010678A4DF1movcl,byteptr[ebp-0Fh]  
0040106A884DFCmovbyteptr[ebp-4],cl  
11:a=p[1];  
0040106D8B55ECmovedx,dwordptr[ebp-14h]  
004010708A4201moval,byteptr[edx+1]  
004010738845FCmovbyteptr[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，在根据

edx读取字符，显然慢了。  
2.7小结：  
堆和栈的区别可以用如下的比喻来看出：  
使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会

切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。  
使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

自我总结：  
char \*c1 = "abc";实际上先是在文字常量区分配了一块内存放"abc",然后在栈上分配一地址给c1并指向

这块地址，然后改变常量"abc"自然会崩溃

然而char c2[] = "abc",实际上abc分配内存的地方和上者并不一样，可以从  
4199056  
2293624 看出，完全是两块地方，推断4199056处于常量区，而2293624处于栈区

2293628  
2293624  
2293620 这段输出看出三个指针分配的区域为栈区，而且是从高地址到低地址

2293620 4199056 abc 看出编译器将c3优化指向常量区的"abc"

继续思考：  
代码：  
  
输出：  
2293628 4199056 abc  
2293624 2293624 abc  
2293620 4012976 gbc  
写成注释那样，后面改动就会崩溃  
可见strcpy(c3,"abc");abc是另一块地方分配的，而且可以改变，和上面的参考文档说法有些不一定，

#include <iostream>  
using namespace std;

main()  
{  
   char \*c1 = "abc";  
   char c2[] = "abc";  
   char \*c3 = ( char\* )malloc(3);  
   //  \*c3 = "abc" //error  
   strcpy(c3,"abc");  
   c3[0] = 'g';  
   printf("%d %d %s\n",&c1,c1,c1);  
   printf("%d %d %s\n",&c2,c2,c2);  
   printf("%d %d %s\n",&c3,c3,c3);  
   getchar();  
}