

1.内存分布

- (1) 代码区
- (2) 全局区
- (3) 栈区(stack)
- (4) 堆区(heap)

2.字符串常量的存储

字符数组与字符指针

3.字符串常量内存释放问题

4.字符串常量生命周期

5.字符串常量定义

6.字符数组

7.字符指针

8.内存图

9.补充

1.内存分布

一个编译的C程序占用的内存分为以下几个部分：

1、栈区（stack）——也称自动类型存储区，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，例如函数调用结束后自动释放。

2、堆区（heap）——也称动态分配内存区，由程序员分配释放，从分配到程序结束为止，若不释放，程序结束时可能由OS回收，比如malloc分配的内存，free释放的内存。

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。

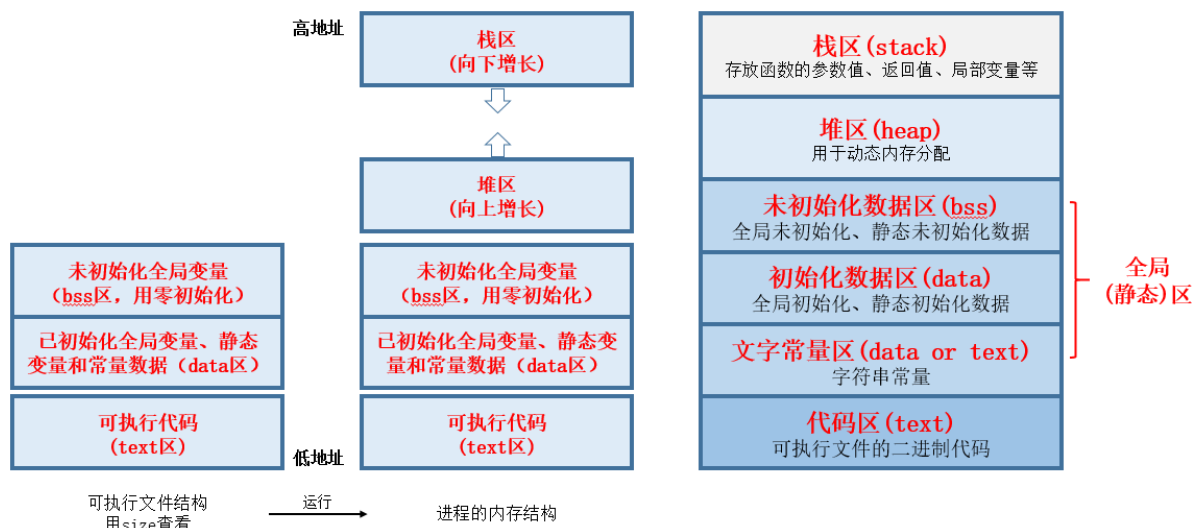
4、文字常量区——常量字符串放在这里，程序结束后由系统释放。

5、程序代码区——编译后的程序代码放在这里。

来看一个具体的C程序

```
1 //来看一个具体的C程序
2 int a=0; //全局初始化区
3 char *p1; //全局未初始化区
4 int main(void)
5 {
6     int b; //栈区
7     char s[]="abc"; //栈区
8     char *p2; //栈区
9     char *p3="123456"; //123456\0在常量区，p3在栈区
10    static int c=0; //加static在全局（静态）初始化区
11    p1 = (char*)malloc(10);
12    p2 = (char*)malloc(20); //malloc分配的区域在堆区
13    strcpy(p1, "123456");
14    //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
```

下图所示为可执行代码存储时结构和运行时结构的对照图。一个正在运行着的C编译程序占用的内存分为代码区、初始化数据区、未初始化数据区、堆区和栈区5个部分。



(1) 代码区

存放 CPU 执行的机器指令。通常代码区是可共享的(即另外的执行程序可以调用它), 使其可共享的目的是对于频繁被执行的程序, 只需要在内存中有一份代码即可。

代码区通常是只读的, 使其只读的原因是防止程序意外地修改了它的指令。另外, 代码区还规划了局部变量的相关信息。

总结: 你所写的所有代码都会放入到代码区中, 代码区的特点是共享和只读。

(2) 全局区

全局区中主要存放的数据有: [全局变量](#)、静态变量、常量(如字符串常量)

全局区的叫法有很多: 全局区、静态区、数据区、全局静态区、静态全局区

这部分可以细分为data区和bss区

2.1 data区

data区里主要存放的是已经初始化的全局变量、[静态变量](#)和常量

2.2 bss区

bss区主要存放的是未初始化的全局变量、静态变量, 这些未初始化的数据在程序执行前会自动被系统初始化为0或者NULL

2.3 常量区

常量区是全局区中划分的一个小区域, 里面存放的是常量, 如const修饰的全局变量、[字符串](#)常量等在VS下运行结果如下:

```

1  #define _CRT_SECURE_NO_WARNINGS
2  #include<stdio.h>
3  #include<string.h>
4  #include<stdlib.h>
5
6  //全局变量 全局区
7  int g_a = 10;
8  int g_b = 10;
9
10 //静态变量 全局区
11 static int s_g_a = 10;
12 static int s_g_b = 10;
13
14 //全局常量 全局区
15 const int g_c_a = 10;
16 const int g_c_b = 10;
17
18 void test01()
19 {
20     printf("全局变量 g_a的地址为: %d\n", &g_a);
21     printf("全局变量 g_b的地址为: %d\n", &g_b);
22
23     printf("全局静态变量 s_g_a的地址为: %d\n", &s_g_a);
24     printf("全局静态变量 s_g_b的地址为: %d\n", &s_g_b);
25
26     //静态局部变量 全局区
27     static int s_a = 10;
28     static int s_b = 20;
29
30     //局部静态变量
31     printf("局部静态变量 s_a的地址为: %d\n", &s_a);
32     printf("局部静态变量 s_b的地址为: %d\n", &s_b);
33
34     //const修饰的全局变量
35     printf("全局常量 g_c_a的地址为: %d\n", &g_c_a);
36     printf("全局常量 g_c_b的地址为: %d\n", &g_c_b);
37
38     //字符串常量 全局区中
39     printf("字符串常量地址: %d\n", &"hello world1");
40     printf("字符串常量地址: %d\n", &"hello world2");
41
42 }

```

```

全局变量 g_a 的地址为: 4292680
全局变量 g_b 的地址为: 4292684
全局静态变量 s_g_a 的地址为: 4292688
全局静态变量 s_g_b 的地址为: 4292692
全局静态变量 s_a 的地址为: 4292696
全局静态变量 s_b 的地址为: 4292700
全局常量 g_c_a 的地址为: 4282456
全局常量 g_c_b 的地址为: 4282460
字符串常量地址: 4282708
字符串常量地址: 4282748
请按任意键继续. . .

```

总结：全局区存放的是全局变量、静态变量和常量

在程序运行后由产生了两个区域，栈区和堆区。

(3) 栈区(stack)

栈是一种先进后出的内存结构，由编译器自动分配释放，存放函数的参数值、返回值、局部变量等。在程序运行过程中实时加载和释放，因此，局部变量的生存周期为申请到释放该段栈空间。

vs运行效果如下

```

44 void test02()
45 {
46
47     //局部变量  栈区
48     int a = 10;
49     int b = 20;
50     printf("局部变量 a的地址为: %d\n", &a);
51     printf("局部变量 b的地址为: %d\n", &b);
52
53     //const修饰局部变量    栈区
54     const int c_a = 10;
55     const int c_b = 10;
56     printf("c_a的地址: %d\n", &c_a);
57     printf("c_b的地址: %d\n", &c_b);
58
59 }

```

```
局部变量 a的地址为: 1637968  
局部变量 b的地址为: 1637956  
c_a的地址: 1637944  
c_b的地址: 1637932  
请按任意键继续. . .
```

(4) 堆区(heap)

堆是一个大容器，它的容量要远远大于栈，但没有栈那样先进后出的顺序。用于动态内存分配。堆在内存中位于BSS区和栈区之间。一般由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收。

vs运行效果如下：

```
61 void test03()  
62 {  
63     //堆区  
64     char * p1 = malloc(64);  
65     char * p2 = malloc(64);  
66     printf("堆区的地址为: %d\n", p1);  
67     printf("堆区的地址为: %d\n", p2);  
68  
69     free(p1);  
70     free(p2);  
71 }  
72
```

```
堆区的地址为: 5339144
堆区的地址为: 5339272
请按任意键继续. . .
```

当我们把几个案例放在一起执行，就可以看到内存将每个区域划分的很有条理。每个区域互不干涉，区域中的数据地址也是非常接近的

```
-----全局区-----
全局变量 g_a 的地址为: 4292680
全局变量 g_b 的地址为: 4292684
全局静态变量 s_g_a 的地址为: 4292688
全局静态变量 s_g_b 的地址为: 4292692
局部静态变量 s_a 的地址为: 4292704
局部静态变量 s_b 的地址为: 4292708
全局常量 g_c_a 的地址为: 4282456
全局常量 g_c_b 的地址为: 4282460
字符串常量地址: 4282708
字符串常量地址: 4282748
-----栈区-----
局部变量 a 的地址为: 1637968
局部变量 b 的地址为: 1637956
c_a 的地址: 1637944
c_b 的地址: 1637932
-----堆区-----
堆区的地址为: 6060040
堆区的地址为: 6060168
请按任意键继续. . .
```

2. 字符串常量的存储

首先，毫无疑问，即使是常量（字符串常量）也是要占据空间的。

一般来说，基本类型（整型、字符型等）常量会在编译阶段被编译成立即数，占的是代码段的内存。（代码段是只读的，而且不允许程序员获取代码段的地址，所以在c++中，尽量不为const分配数据段的内存，但是一旦取const的地址，就不得不分配了，但是读const的时候，依然是从代码段读取那个立即数）。当然，占代码段的内存一般不在我们常说的“占内存”范围中。代码段不是寄存器。

而字符串常量或基本类型的常量数组占用的是数据段内存。

程序中但凡出现“XXX”形式，这都是代表字符串常量，是要事先存储在程序的只读数据区的。

```
void test(){
    char name[32] = "hello,world"; // "hello,world" 存储在程序的只读数据区， name 存储在test函数的栈区。
}
```

字符串常量的存储：字符串常量使用一对双引号括起来的字符序列，与基本类型常量的存储相似，字符串常量在内存中的存放位置由系统自动安排。

由于字符串常量是一串字符，通常被看作一个特殊的一位字符数组，与数组的存储类似，**字符串常量中的所有字符在内存中连续存放**。所以，系统在存储一个字符串常量时先给定一个**起始地址**，从该地址指定的存储单元开始，连续存放该字符串中的字符。

显然，该起始地址代表了存放字符串常量首字母的存储单元的地址，被称为字符串常量的值，也就是说，字符串常量实质上是一个指向该字符串首字符的指针常量。

例如字符串"Hello"的值是一个地址，从它指定的存储单元开始连续存放该字符串的6个字符 ('H' 'e' 'l' 'l' 'o' '\n')

字符数组与字符指针

如果定义一个字符指针接收字符串常量的值，该指针就指向字符串的首字符，例如：

```
char sa[]="array ";
char *sp="point ";
printf("%s", sa); //数组名sa作为printf的输出参数
printf("%s", sp); //字符指针sp作为printf的输出参数
printf("%s \n", "string"); //字符串常量作为printf的输出参数
```

输出：array point string

调用printf()函数，以%s的格式输出字符串时，作为输出参数，**数组名sa、指针sp和字符串"string"的值都是地址**，从该地址所指定的单元开始连续输出其中的**内容（字符）**，直到遇到'\0'为止。

由此可见，输出字符串时，输出参数给出的**起始位置（地址）**，'\0'用来控制结束。因此，字符串中其它字符的地址也能作为输出参数。例如：

```
printf("%s", sa+2); //数组元素sa[2]的地址作为输出参数
printf("%s", sp+3); //sp+3作为起始数组
printf("%s \n", "string" +1 ); //t作为起始输出
```

输出：

```
int main() {
    char sa[] = "array ";
    char* sp = "point ";
    printf("%s", sa + 2); //数组元素sa[2]的资质作为输出参数
    printf("%s", sp + 3); //sp+3作为起始数组
    printf("%s\n", "string" + 1); //t作为起始输出
}
```

Microsoft Visual Studio
array nt tring
E:\C语言\比特\C Pr

字符数组与字符指针都可以处理字符串，但两者之间有重要区别，如：

```
char sa[]="This is a string";
char* sp="This is a string";
```

字符数组sa在内存中占用了一块连续的单元，有确定的地址，每个数组元素放字符串的一个字符，字符串就存放在数组中。

字符指针sp只占用一个可以存放地址的内存单元，而不是将字符串放到字符指针变量中去。

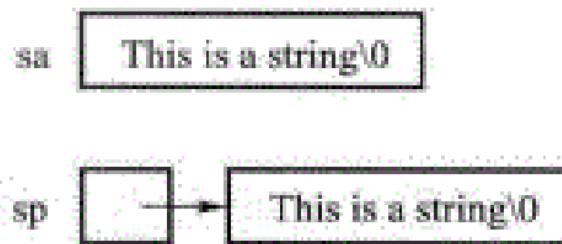


图 8.10 字符数组 sa 和字符指针 sp 的区别示意

CSDN @Llues233

如果要改变数组sa所代表的字符串，只能改变数组元素的内容。

如果要改变指针sp所代表的字符串，通常直接改变指针的值，让它指向新的字符串。因为sp是指针变量，它的值可以改变，转向指向其它单元。

当定义字符指针后，如果没有赋值，指针的值是不确定的，引用未赋值的指针可能出现难以预料的结果：

```
char* s;
scanf("%s", s);
```

没有对指针s赋值，却对s指向的单元赋值，如果该单元已经分配给其他变量，其值就改变了。

所以以下写法str有确定的存储单元，这才是正确的：

```
char* s, str[20];
s=str;
scanf("%s", s);
```

为了避免引用未赋值的指针造成的危害，在定义指针时，可先将它的初值置为空，如：**char* s=NULL;**

一般在使用指针类型后，为避免出现内存泄漏，都需要手动释放内存，如：

```
char* s = new char[128];
delete []s;
s = NULL;
```

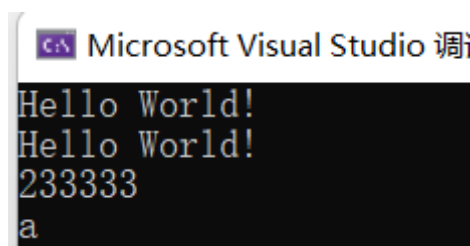
但如果是像 `const char* str` 这种指针，则不需要手动释放内存。

这是因为 `const char* str` 定义的是一个指向常量的指针。

如果str是局部变量，则字符串会随着变量所在的函数的退出而自动释放；如果str是全局变量，则程序退出时才自动释放。

```
#include<stdio.h>
int main(void){
    //字符串指针
    const char* s = "Hello world!";
    printf("%s\n", s);
    //字符串数组
    char ch[] = "Hello world!";
    printf("%s\n", ch);
    //整型变量和指针
    int* i, a = 233333;
    i = &a;
    printf("%d\n", *i);
    //字符型变量和指针
    char* cha, b = 'a';
    cha = &b;
    printf("%c\n", *cha);
    return 0;
}
```

输出结果：



在处理**整型和字符型**等时，**printf()**函数输出时指针变量前**需要**带上 * 号

而处理**字符串**时，输出时指针变量前则**不需要**带 * 号。

3.字符串常量内存释放问题

```
#include <iostream>
using namespace std;
char *str1()
{
    char *str="hello world";
    return str;
}
char *str2()
{
    char str[]="hello world"; //str[]在栈上，子函数结束后自动释放内存，返回的其实是首地址
    return str;    //拷贝后，返回和"hello world"无关，返回存在str[]中的内容
}
char *str3()
{
    static char str[]="hello world";
    return str;
}
int main()
{
    char *str=NULL;
    str=str1();
    cout<<"指针指向内存内容: "<<str<<endl;    //输出hello world
    str=str2();
    cout<<"栈内容: "<<str<<endl;    //输出乱码
    str=str3();
    cout<<"静态存储区内容: "<<str<<endl;    //输出hello world
    return 0;
}
```

字符串常量存放在 静态存储区。

str1返回指针指向内存首地址，由于字符串常量存在静态区，所以内容不变，还是“hello world”

str2将静态存储区的内容拷贝一份到栈中，由于栈在str2结束时释放栈内存，所以输出为乱码

str3返回存在静态存储区的内容，“hello world”

可以返回一个局部变量的值，也可以返回一个局部静态指针的地址，但不应该返回一个局部自动指针的地址

```
int get()
{
    int a=152;
    return a;    //可以正确返回
}
```

```
int *get()
{
    int a=152;
    return & a;    //无意义
}
```

4.字符串常量生命周期

字符串`char *s="hello";`与`char s[]="hello";`，看似都是将hello字符串的地址赋值给指针 `*p`。

但是前面一个表达式是字符串常量的地址赋值给指针，该指针指向的字符串中的字符是不允许被更改的。

而后面一个表达式是将该字符串的每一个字符赋值给数组，该指针指向的数组的首地址，而数组成员是变量，因此可以允许被更改赋值。

下面讨论关于字符串常量在内存中存在生命周期的问题。

问题如下：

假如

```
char *s0="hello";
s0="world";
```

就是说如果开始s0指针指向“hello”这个字符串这个常量的地址，但是当s0指针指向了“world”字符串的时候，那么“hello”这个字符串在内存中的是否会被释放掉，还是会一直存在内存中，直到程序结束。

我们不妨先试着写几行代码，通过运行结果来进行分析：

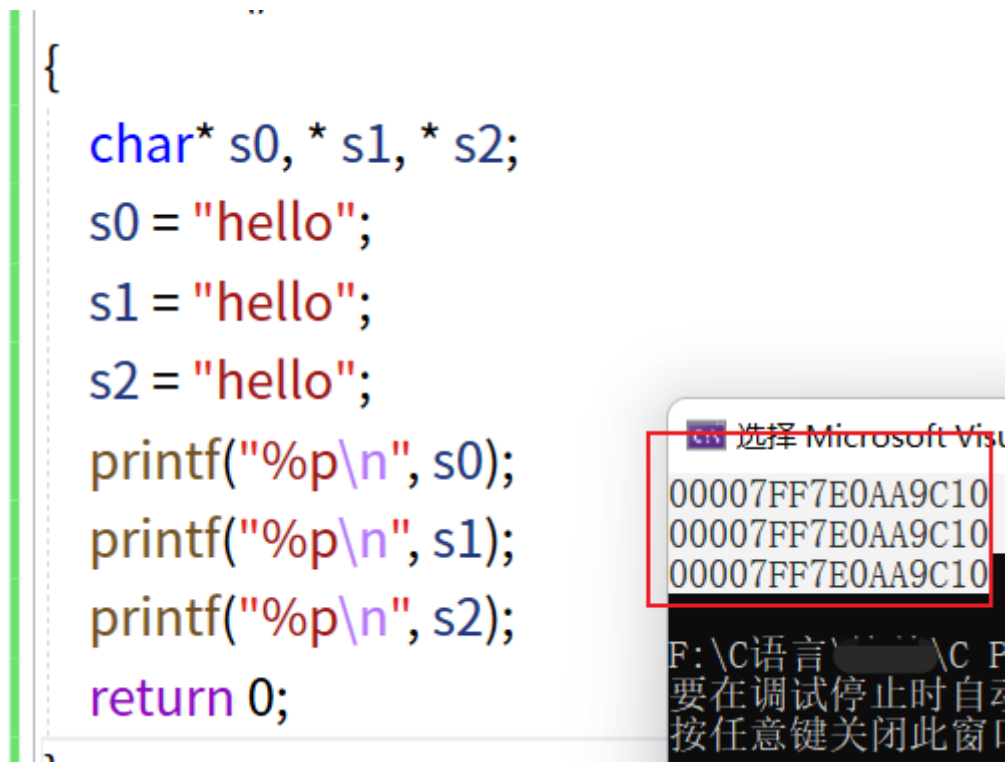
案例 一：

代码：

```
#include<stdio.h>
int main()
{
    char *s0,*s1,*s2;
    s0="hello";
    s1="hello";
    s2="hello";
    printf("%p\n",s0);
    printf("%p\n",s1);
    printf("%p\n",s2);
    return 0;
}
```

运行结果：

三个指针变量的指向的地址都是相同的，如图



分析:

通过运行结果我们可以看出，赋给不同字符指针的相同的字符串，所有的指针都指向了相同的地址。

但是案例一由于三个指针变量都在同一个函数，因此不能看出其生命周期，但是可以为下面的案例解释做好铺垫。

案例二:

代码:

```
#include<stdio.h>
char *s0="hello";
void a(){
    char *s1="hello";
    printf("%p\n",s1);
    s1="world";
}
void b(){
    char *s2="hello";
    printf("%p\n",s2);
    s2="world";
}
int main(){
    char *s3=(char *)0;
    printf("%p\n",s0);
    a();
    b();
    s3="hello";
    printf("%p\n",s3);
    return 0;
```

```
}
```

运行结果：

```
00007FF7B0799C10
00007FF7B0799C10
00007FF7B0799C10
00007FF7B0799C10
```

四个字符指针变量指向的地址还是一样的。

分析：

这就说明问题了，在函数a(), b() 中，局部指针变量s1,s2都指向“hello”字符串，但是在退出之前有改变指针指向其他字符串，但是两个字符串指针打印出来的地址还是一样的，说明在使用了字符串常量后，该字符串常量并没有在随着函数的结束而消失，而是依旧存在于内存中，因此当其他函数中使用一样的字符串常量时，指向的依旧是跟还是一样的地址。

但是也许有人会问到，因为在全局字符指针变量 char *s0 ="hello";指向了该字符串常量，会像全局变量一样，在真个程序运行都不会释放，因此其他函数调使用该字符串常量时才指向了同一地址。确实确实有这种可能，那么我们就可以看下面案例三。

案例三：

```
#include<stdio.h>
void a(){
    char *s1="hello";
    printf("%0x\n",s1);
    s1="world";
}
void b(){
    char *s2="hello";
    printf("%0x\n",s2);
    s2="world";
}
int main(){
    char *s3=(char *)0;
    a();
    b();
    s3="hello";
    printf("%0x\n",s3);
    return 0;
}
```

三个字符串指针变量指向的地址还是一样的，如图：

```
f6a69c10
f6a69c10
f6a69c10
```

分析：

当我们去掉全局指针变量后，其结果依旧是s1,s2,s3 三个指针变量指向的地址依旧是没有变。

在运行完a(),b()两个函数之后，我们再将"hello"赋值给空指针s3指针，其指向地址与s1,s2都一样的。

因此我们可以得出结论：一旦有字符串常量在运行期间创建，就会在内存中一直保持到程序结束，当使用相同的字符串常量的时候，不会再创建字符串常量，而是指向之前的那个。因此字符串常量是贯穿整个程序的生命周期的。

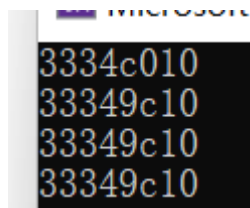
附：既然说到这里了，那就再多说一点。

案例：当将 `char* s0` 改为 `char s0[]` 时。

代码：

```
#include<stdio.h>
char s0[]="hello";
void a(){
    char *s1="hello";
    printf("%0x\n",s1);
    s1="world";
}
void b(){
    char *s2="hello";
    printf("%0x\n",s2);
    s2="world";
}
int main(){
    char *s3=(char *)0;
    printf("%0x\n",s0);
    a();
    b();
    s3="hello";
    printf("%0x\n",s3);
    return 0;
}
```

运行结果：



s0与s1、s2、s3指向的地址不同。

分析：

造成这样的结果的原因是因为 `char s0[]="hello"`，而s0指向的是该数组的首地址，而不是字符串的首地址。

而s1、*s2、*s3都是指向字符串的首地址，因此不同。


```

#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello C-Free!\n");

    //定义一个数组变量，用字符串常量初始化其值 。
    char a[] = "123";
    //定义一个字符指针，再定义一个字符串常量，指针指向的常量首地址
    const char* b = "321";
    const char* c = "321";
    //打印变量在内存里的地址，栈区【高地址->低地址】
    printf("%d,%d,%d\n",&a, &b, &c); //6356772,6356768,6356764
    //打印指针的首地址,b和c相同因为都指向同一常量区
    printf("%d,%d,%d\n",a, b, c);    //6356772,4206607,4206607
    //打印内容
    printf("%s,%s,%s\n",a, b, c);    //123,321,321

    //从b内存拷贝3个字节给a变量，正常
    memcpy(a, b, 3);
    //从a内存拷贝3个字节给a常量指针，编译失败或运行失败，常量不可修改
    memcpy(b, a, 3);

    printf("%s,%s,%s\n",a, b, c);    //123,321,321
    return 0;
}

```

5.字符串常量定义

定义：用双引号("")括起来的0个或者多个字符组成的序列。

存储：每个字符串尾自动加一个 '\0' 作为字符串结束标志。

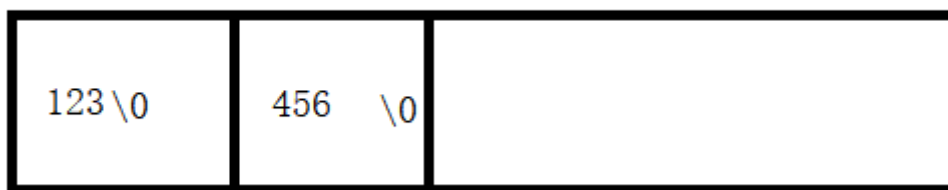
字符串常量在内存的常量存储区是按顺序存储的，如：

```

char* a = "123";
char* b = "456";
char* c = "456";

```

常量存储区



定义a时，存储一个"123\0";

定义b时，判断存储区是否有"456\0"，发现没有则在后面追加"456\0";

定义c时，判断存储区是否有"456\0"，**发现有，则不再存储**，此时b和c两个指针存储的都是"456\0"这片内存地址

既然是常量，那么不可被修改，所以memcpy(b, a, 3);是错误的。

6. 字符数组

```
char a[] = "123";
```

首先声明一个字符数组a，大小没有确定，但是将一个字符串常量"123\0"赋值给了a，故a的length就是4个字节。【注意"123\0"并没有存储在常量区】

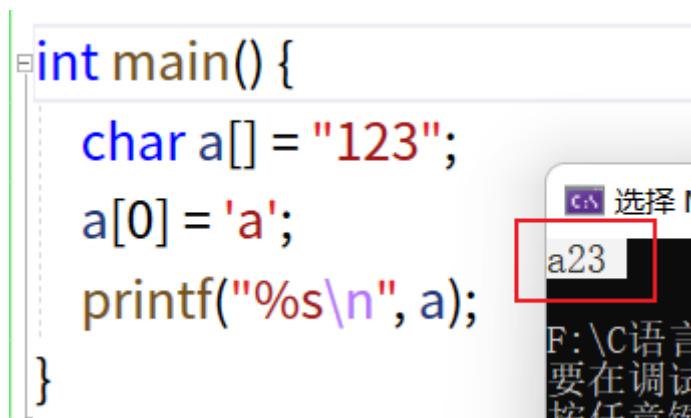
注意a归根结底是一个数组，数组是一个变量，不是指针，虽然可把a当做一个指针，因为它指向数组的首地址，但归根结底不是指针。

正因为a是变量，所以a能够修改其存储的值。

比如：

```
int main(){
    char a[]="123";
    a[0]='a';
    printf("%s\n",a);
}
```

输出结果：



7. 字符指针

```
const char* b = "321";
```

既然名字是字符指针，那么它一定是个指针，指针存储地址。

故解读这句就是：

首先声明一个字符指针b，然后定义一个字符串常量"321\0"。

字符串常量存储在常量区，b存储在栈区，b存储的值是字符串常量"321\0"的地址。

这里用到了const，在C语言里不加const也行，C++里不加会有个警告，但不影响编译。但是建议加上const，能够让程序员一眼就知道此指针指向的是常量，也就是最终内容无法修改。

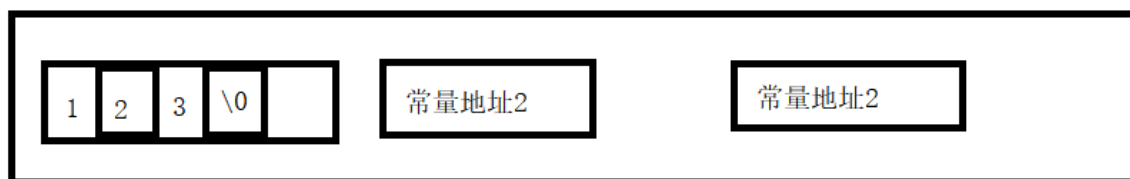
虽然最终指向不能修改，但是指针自身的取值，是可以修改的，即可修改指针指向的地方。

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     printf("Hello C-Free!\n");
6     const int v1 = 100;
7     const int v2 = 200;
8     const int* a = &v1;
9     printf("%d\n", *a);
10    a = &v2;
11    printf("%d\n", *a);
12    return 0;
13 }
```



8.内存图

栈 高地址->低地址

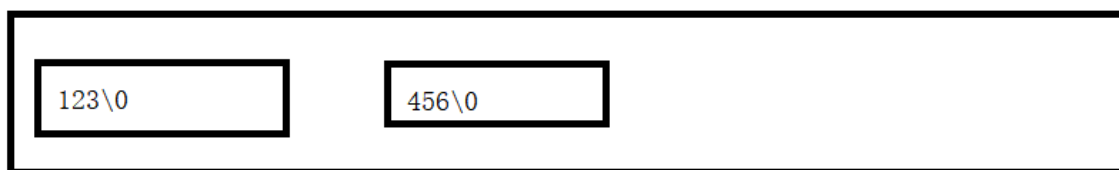


变量a, 数组, 直接存储变量

变量b, 指针, 存储地址

变量c, 指针, 存储地址, 由于编译器优化, 使用的是和b一样的位置

常量区 低地址->高地址



常量地址1

常量地址2

所有存储类型代码区、常量区、静态区（全局区）、堆区、栈区，只有栈是从高地址往低地址存储，其他都是低地址往高地址存储。

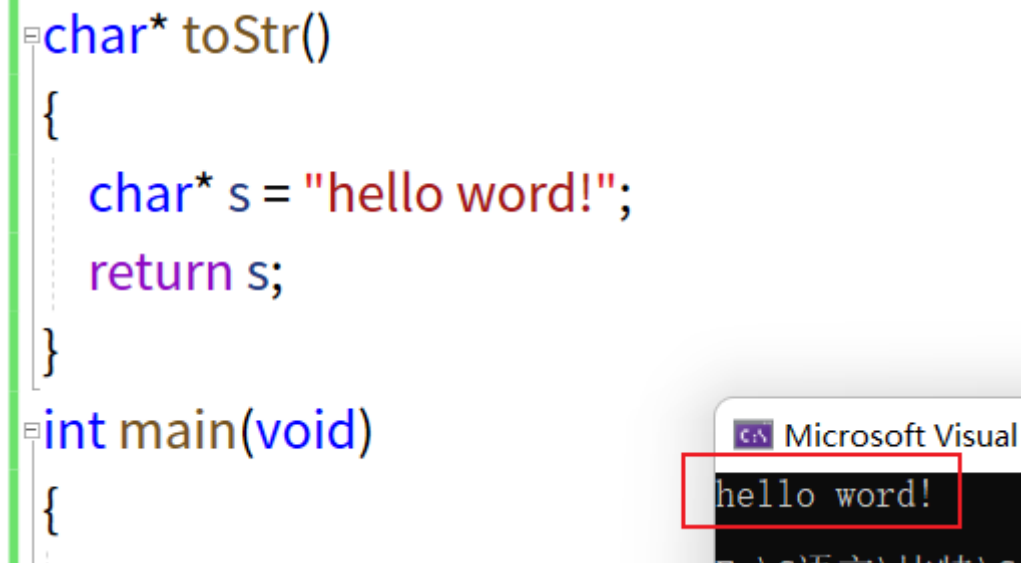
9.补充

来分析几段代码。

代码: test0.c

```
#include <stdio.h>
#include <stdlib.h>
char* toStr()
{
    char *s = "hello word!";
    return s;
}
int main(void)
{
    printf("%s\n", toStr());
}
```

运行结果：



运行没有任何问题，因为“hello world”是一个字符串常量，存放在文字常量区，把该字符串常量存放的地址赋值给了指针 s，toStr 函数退出时，该该字符串常量所在内存不会被回收，故能够通过指针顺利无误的访问。

代码：test1.c

```
#include <stdio.h>
#include <stdlib.h>
char* toStr()
{
    char s[] = "hello word!";
    return s;
}
int main(void)
{
    printf("%s\n", toStr());
}
```

编译运行结果：

```
cat@cai-virtual-machine:~/c_notebook$ gcc test1.c
test1.c: In function 'toStr':
test1.c:7:9: warning: function returns address of local variable [-Wreturn-local-addr]
   7 |     return s;
     |           ^
cat@cai-virtual-machine:~/c_notebook$
cat@cai-virtual-machine:~/c_notebook$ ./a.out
段错误 (核心已转储)
```

调用 toStr 函数，定义了一个局部变量(char [] 型数组)，该局部变量存放在栈中，当 toStr 函数退出时，栈要清空，局部变量的内存也被清空了，所以这时的函数返回的是一个已被释放的内存地址，出现段错误。

如果函数的返回值非要是一个局部变量的地址，那么该局部变量一定要申明为**static**类型。

代码:test2.c

```
#include <stdio.h>
char *returnStr()
{
    static char p[]="hello world!";
    return p;
}
int main()
{
    char *str=NULL;
    str=returnStr();
    printf("%s\n", str);

    return 0;
}
```

运行结果：

```
hello world!
```

综合性代码：

```
#include <stdio.h>

//返回的是局部变量的地址，该地址位于动态数据区，栈里

char *s1()
{
    char p[] = "Hello world!";

    printf("in s1 p=%p\n", p);
    printf("in s1: string's address: %p\n", &("Hello world!"));

    return p;
}

//返回的是字符串常量的地址，该地址位于静态数据区
```

```

char *s2()
{
    char *q = "Hello world!";

    printf("in s2 q=%p\n", q);
    printf("in s2: string's address: %p\n", &("Hello world!"));

    return q;
}

//返回的是静态局部变量的地址，该地址位于静态数据区
char *s3()
{
    static char r[] = "Hello world!";

    printf("in s3 r=%p\n", r);
    printf("in s3: string's address: %p\n", &("Hello world!"));
    return r;
}

int main()
{
    char *t1, *t2, *t3;

    t1 = s1();
    t2 = s2();
    t3 = s3();

    printf("\nin main:\n");
    printf("p = %p\nq = %p\nr = %p\n", t1, t2, t3);

    // printf("%s\n", t1);    /* 运行会出现段错误，先屏蔽 */
    printf("%s\n", t2);
    printf("%s\n", t3);

    return 0;
}

```

运行结果：

```

in s1 p=000000B0E074FA78
in s1: string's address: 00007FF6AB739C18
in s2 q=00007FF6AB739C18
in s2: string's address: 00007FF6AB739C18
in s3 r=00007FF6AB73C050
in s3: string's address: 00007FF6AB739C18

in main:
p = 000000B0E074FA78
q = 00007FF6AB739C18
r = 00007FF6AB73C050
Hello world!
Hello world!

```


根据运行结果我们就很清楚了，这里不再赘述。

✎ 部分参考文章：

[\(37条消息\) 关于字符串常量在内存中的生命周期_w 16822的博客-CSDN博客](#)

[\(39条消息\) 【C语言入门】笔记十\(指针中\)Liues233的博客-CSDN博客假设字符数组sa为一个英文字符串,设计程序,当用户用键盘输入字符串sa后,将](#)

[\(40条消息\) C语言—内存的管理和释放^不加糖^的博客-CSDN博客c语言释放内存](#)

欢迎关注，一位喜欢慢慢生活的博主。

自述文件



雨翼轻尘

110

3.3w

2.1w

原创内容

作者排名

粉丝数量



CSDN