

一、关于内存那点事

二、指针的相关概念

三、指针的定义方法

四、指针的分类

五、指针和变量的关系

六、指针和数组元素之间的关系

6.1 数组元素与指针的基本关系

6.2 数组元素的引用方法

七、指针的运算

7.1 指针可以加一个整数

7.2 两个相同类型指针可以比较大小

7.3 两个相同类型的指针可以做减法

7.4 两个相同类型的指针可以相互赋值

八、指针数组

九、指针的指针 -- 二级指针

十、字符串和指针

十一、数组指针

十二、指针与函数的关系

12.1 指针作为函数的参数

12.1.1 复制传参 -- 传数值

12.1.2 地址传参 -- 传地址

12.2 传数组

12.3 指针函数 -- 指针作为函数的返回值

12.4 函数指针 - 指针保存函数的地址

12.4.1 函数指针变量的定义方法

12.4.2 调用函数的方法

12.4.3 函数指针数组

12.4.4 函数指针最常用的地方

十三、经常容易混淆的指针

十四、特殊指针

十五、main函数传参

一、关于内存那点事

存储器：存储数据器件

外存

外存又叫外部存储器，长期存放数据，掉电不丢失数据

常见的外存设备：硬盘、flash、rom、u盘、光盘、磁带

内存

内存又叫内部存储器，暂时存放数据，掉电数据丢失

常见的内存设备：ram、DDR

物理内存：实实在在存在的存储设备

虚拟内存：操作系统虚拟出来的内存，当一个进程被创建的时候，或者程序运行的时候都会分配虚拟内存，虚拟内存和物理内存之间存在映射关系。

操作系统会在物理内存和虚拟内存之间做映射。

在32位系统下，每个进程（运行着的程序）的寻址范围是4G, 0x00 00 00 00 ~ 0xff ff ff ff
在写应用程序的，咱们看到的都是虚拟地址。

在运行程序的时候，操作系统会将 虚拟内存进行分区。

1.堆

在动态申请内存的时候，在堆里开辟内存。

2.栈

主要存放局部变量（在函数内部，或复合语句内部定义的变量）。

3.静态全局区

1）：未初始化的静态全局区

静态变量（定义的时候，前面加static修饰），或全局变量，没有初始化的，存在此区

2）：初始化的静态全局区

全局变量、静态变量，赋过初值的，存放在此区

4.代码区

存放咱们的程序代码

5.文字常量区

存放常量的。

内存以字节为单位来存储数据的，咱们可以将程序中的虚拟寻址空间，看成一个很大的一维的字符数组

本章所接触的内容，涉及到的内存都是虚拟内存，更准确来说是虚拟内存的用户空间

二、指针的相关概念

操作系统给每个存储单元分配了一个编号，从0x00 00 00 00 ~0xff ff ff ff

这个编号咱们称之为地址

指针就是地址

0xffff_fff	
0xffff_fffe	
0xffff_fffd	
...	
...	
...	
0x0000_0003	
0x0000_0002	'\n'
0x0000_0001	'a'
0x0000_0000	100

指针变量：是个变量，是个指针变量，即这个变量用来存放一个地址编号

在32位平台下，地址总线是32位的，所以地址是32位编号，所以指针变量是32位的即4个字节。

注意：

1：无论什么类型的地址，都是存储单元的编号，在32位平台下都是4个字节，
即任何类型的指针变量都是4个字节大小

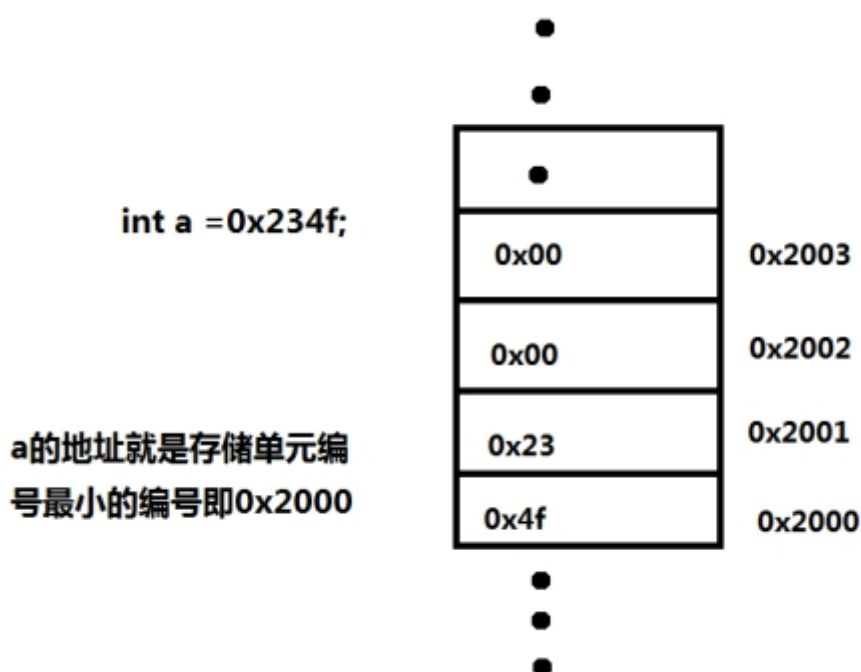
2：对应类型的指针变量，只能存放对应类型的变量的地址

举例：整型的指针变量，只能存放整型变量的地址

扩展：

字符变量 char ch; ch占1个字节，它有一个地址编号，这个地址编号就是ch的地址

整型变量 int a; a占4个字节，它占有4个字节的存储单元，有4个地址编号。



Int a=0x00 00 23 4f

三、指针的定义方法

1.简单的指针

数据类型 * 指针变量名;

int * p;//定义了一个指针变量p

在 定义指针变量的时候 * 是用来修饰变量的，说明变量p是个指针变量。

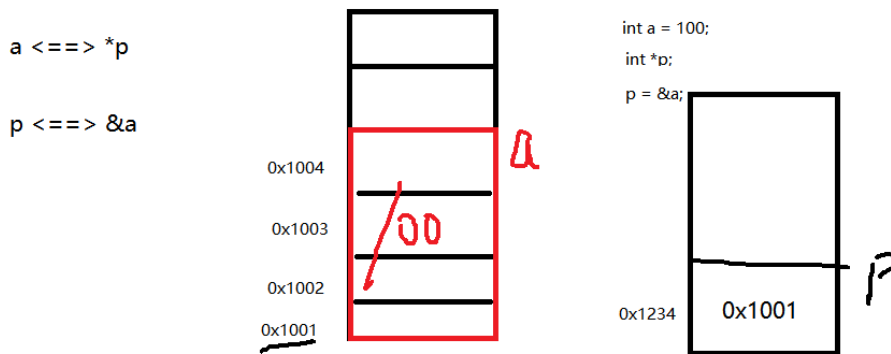
变量名是 p

2.关于指针的运算符

& 取地址 、 *取值

& : 获取一个变量的地址

* : 在定义一个指针变量时，起到标识作用，标识定义的是一个指针变量
除此之外其他地方都表示获取一个指针变量保存的地址里面的内容



```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     //定义一个普通变量
6     int a = 100;
7     //定义一个指针变量
8     int *p;
9
10    //给指针变量赋值
11    //将a的地址保存在p中
12    p = &a;
13
14    printf("a = %d %d\n", a, *p);
15    printf("&a = %p %p\n", &a, p);
16
17    return 0;
18 }
```

执行结果

```
Starting C:\Users\lzx\Desktop\src\build-01_point.exe...
a = 100 100
&a = 0029FEA8 0029FEA8
C:\Users\lzx\Desktop\src\build-01_point.exe exited with code 0
```

扩展：如果在一行中定义多个指针变量，每个指针变量前面都需要加*来修饰

`int *p,*q;`//定义了两个整型的指针变量p和q

`int * p,q;`//定义了一个整型指针变量p，和整型的变量q

3、指针大小

在32位系统下，所有类型的指针都是4个字节

因为不管地址内的空间多大，但是地址编号的长度是一样的，所以在32位操作系统中，地址都是四个字节

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      char *a;
6      short *b;
7      int *c;
8      long *d;
9      float *e;
10     double *f;
11
12     printf("sizeof(a) = %d\n", sizeof(a));
13     printf("sizeof(b) = %d\n", sizeof(b));
14     printf("sizeof(c) = %d\n", sizeof(c));
15     printf("sizeof(d) = %d\n", sizeof(d));
16     printf("sizeof(e) = %d\n", sizeof(e));
17     printf("sizeof(f) = %d\n", sizeof(f));
18
19     return 0;
20 }
21
```

执行结果

```
Starting C:\Users\lzx\Desktop\src\
Debug\debug\02_point_size
sizeof(a) = 4
sizeof(b) = 4
sizeof(c) = 4
sizeof(d) = 4
sizeof(e) = 4
sizeof(f) = 4
C:\Users\lzx\Desktop\src\l
\02_point_size.exe exited
```

四、指针的分类

按指针指向的数据的类型来分

1:字符指针

字符型数据的地址

char *p;//定义了一个字符指针变量，只能存放字符型数据的地址编号

char ch;

p = &ch;

2：短整型指针

short int *p;//定义了一个短整型的指针变量p，只能存放短整型变量的地址

short int a;

p = &a;

3：整型指针

int *p;//定义了一个整型的指针变量p，只能存放整型变量的地址

int a;

p = &a;

注：多字节变量，占多个存储单元，每个存储单元都有地址编号，

c语言规定，存储单元编号最小的那个编号，是多字节变量的地址编号。

4：长整型指针

long int *p;//定义了一个长整型的指针变量p，只能存放长整型变量的地址

long int a;

p = &a;

5：float 型的指针

float *p;//定义了一个float型的指针变量p，只能存放float型变量的地址

float a;

p = &a;

6：double型的指针

```
double *p;//定义了一个double型的指针变量p，只能存放double型变量的地址  
double a;  
p = &a;
```

7：函数指针

8、结构体指针

9、指针的指针

10、数组指针

总结:无论什么类型的指针变量，在32位系统下，都是4个字节，只能存放对应类型的变量的地址编号。

五、指针和变量的关系

指针可以存放变量的地址编号

在程序中，引用变量的方法

1:直接通过变量的名称

```
int a;  
a=100;
```

2:可以通过指针变量来引用变量

```
int *p;//在定义的时候，*不是取值的意思，而是修饰的意思，修饰p是个指针变量  
p=&a;//取a的地址给p赋值，p保存了a的地址，也可以说p指向了a  
*p= 100;//在调用的时候*是取值的意思，*指针变量 等价于指针指向的变量
```

注：指针变量在定义的时候可以初始化

```
int a;  
int *p=&a;//用a的地址，给p赋值，因为p是指针变量  
指针就是用来存放变量的地址的。
```

***+指针变量 就相当于指针指向的变量**

指针变量只能保存开辟好空间的地址，不能随意保存地址

```
1 #include <stdio.h>  
2  
3 int main(int argc, char *argv[])  
4 {  
5     int *p1,*p2,temp,a,b;  
6     p1=&a;
```

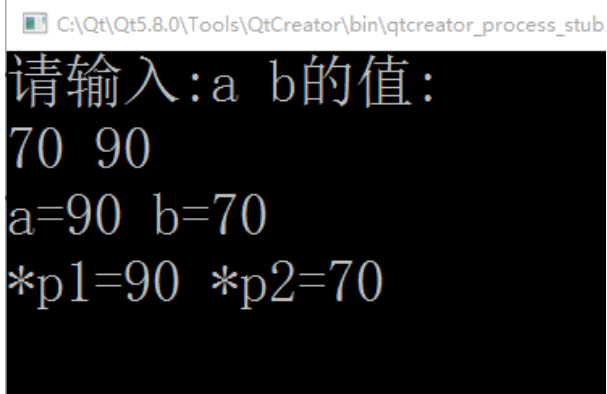


```

7  p2=&b;
8
9  printf("请输入:a b的值:\n");
10 scanf("%d %d", p1, p2); //给p1和p2指向的变量赋值
11
12 temp = *p1; //用p1指向的变量(a)给temp赋值
13 *p1 = *p2; //用p2指向的变量(b)给p1指向的变量(a)赋值
14 *p2 = temp; //temp给p2指向的变量(b)赋值
15
16 printf("a=%d b=%d\n", a, b);
17 printf("*p1=%d *p2=%d\n", *p1, *p2);
18
19 return 0;
20 }

```

执行结果



```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub
请输入:a b的值:
70 90
a=90 b=70
*p1=90 *p2=70

```

扩展：

对应类型的指针，只能保存对应类型数据的地址，
如果能让不同类型的指针相互赋值的时候，需要强制类型转换

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int a=0x1234, b=0x5678;
6      char *p1, *p2;
7      printf("%#x %#x\n", a, b);
8      p1=(char *)&a;
9      p2=(char *)&b;
10     printf("%#x %#x\n", *p1, *p2);
11     p1++;
12     p2++;

```

```

13     printf("%#x %#x\n", *p1, *p2);
14
15     return 0;
16 }

```

执行结果

```

Starting C:\Users\lzx\Desktop\src
Desktop_Qt_5_8_0_MinGW_32bit-Debu
0x1234 0x5678
0x34 0x78
0x12 0x56

```

```

C:\Users\lzx\Desktop\src\build-04
\debug\04 printf.exe

```

注意：

1：*+指针 取值，取几个字节，由指针类型决定的指针为字符指针则取一个字节，指针为整型指针则取4个字节，指针为double型指针则取8个字节。

2：指针++ 指向下个对应类型的数据

字符指针++，指向下个字符数据，指针存放的地址编号加1

整型指针++，指向下个整型数据，指针存放的地址编号加4

六、指针和数组元素之间的关系

6.1 数组元素与指针的基本关系

变量存放在内存中，有地址编号，咱们定义的数组，是多个相同类型的变量的集合，每个变量都占内存空间，都有地址编号

指针变量当然可以存放数组元素的地址。

```

1  int  a[10];
2  //int *p = &a[0];
3  int *p;
4  p = &a[0]; //指针变量p保存了数组a中第0个元素的地址，即a[0]的地址

```

6.2 数组元素的引用方法

方法1：数组名[下标]

```
int a[10];
```

```
a[2]=100;
```

方法2：指针名加下标

```
int a[10];
```

```
int *p;
```

p=a;

p[2]=100;//因为p和a等价

补充：c语言规定：数组的名字就是数组的首地址，即第0个元素的地址，是个常量。

注意：p和a的不同，p是指针变量，而a是个常量。所以可以用等号给p赋值，但不能给a赋值。

例如：int a[10]; a++就是错误的，因为a是数组名是一个地址常量

方法3：通过指针运算加取值的方法来引用数组的元素

int a[10];

int *p;

p=a;

*(p+2)=100;//也是可以的，相当于a[2]=100

解释：p是第0个元素的地址，p+2是 a[2]这个元素的地址。

对第二个元素的地址取值，即a[2]

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int a[5]={0,1,2,3,4};
6      int *p;
7      p=a;
8
9      //只要将数组名赋值给同类型的指针变量，则此时的指针变量与数组名可
10     //以用相同的方法操作数组
11     printf("a[2]=%d\n",a[2]);
12     printf("p[2]=%d\n",p[2]);
13
14     /*(a + n) <==> *(p + n) <==> a[n] <==> p[n]
15     printf("*(p+2) = %d\n",*(p+2));
16     printf("*(a+2) = %d\n",*(a+2));
17
18     printf("p=%p\n",p);
19     printf("p+2=%p\n",p+2);
20     printf("&a[0] = %p\n", &a[0]);
21     printf("&a[2] = %p\n", &a[2]);
22     return 0;
23 }
```

执行结果

```
a[2]=2
p[2]=2
*(p+2) = 2
*(a+2) = 2
p=0029FE98
p+2=0029FEA0
&a[0] = 0029FE98
&a[2] = 0029FEA0
```

七、指针的运算


7.1 指针可以加一个整数

往下指几个它指向的变量，结果还是个地址

前提：指针指向数组的时候，加一个整数才有意义

```
1 //指针可以加一个整数,往下指几个它指向的变量，结果还是个地址
2 void test1()
3 {
4     int a[10];
5     int *p, *q;
6     //p和q间隔8个字节，意味着加一个整数最终移动的字节数与指针变量的类型也有关系
7     p = a;
8     q = p + 2;
9
10    printf("p = %p\n", p);
11    printf("q = %p\n", q);
12
13    return ;
14 }
```

执行结果

 选择C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcrea

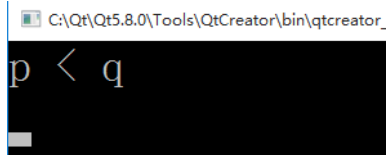
```
p = 0029FE70
q = 0029FE78
```

7.2 两个相同类型指针可以比较大小

前提：只有两个**相同类型的指针指向同一个数组的元素**的时候，比较大小才有意义
指向前面元素的指针 小于 指向后面 元素的指针

```
1 void test2()
2 {
3     int a[10];
4     int *p,*q;
5     p=&a[1];
6     q=&a[6];
7     if(p<q)
8     {
9         printf("p < q\n");
10    }
11    else if(p>q)
12    {
13        printf("p > q\n");
14    }
15    else
16    {
17        printf("p = q\n");
18    }
19 }
```

执行结果



C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_
p < q

7.3 两个相同类型的指针可以做减法

前提：必须是**两个相同类型的指针指向同一个数组的元素**的时候，做减法才有意义
做减法的结果是，两个指针指向的中间有多少个元素

```
1 void test3()
2 {
3     int a[10];
4     int *p,*q;
5     p=&a[0];
6     q=&a[3];
7     printf("%d\n",q-p);
8 }
```

执行结果

选择C:\Qt\Qt5.8.0\Tools\QtC

3

7.4 两个相同类型的指针可以相互赋值

注意:只有相同类型的指针才可以相互赋值 (void *类型的除外)

```
1 void test4()
2 {
3     int a = 100;
4     int *p, *q;
5     p = &a;
6
7     printf("a = %d %d\n", a, *p);
8
9     q = p;
10    printf("*q = %d\n", *q);
11
12    *q = 999;
13    printf("a = %d\n", a);
14 }
```

执行结果

选择C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_proces

```
a = 100 100
*q = 100
a = 999
```

八、指针数组

1、指针和数组的关系

1：指针可以保存数组元素的地址

2：可以定义一个数组，数组中有若干个相同类型指针变量，这个数组被称为指针数组

指针数组的概念：

指针数组本身是个数组，是个指针数组，是若干个相同类型的指针变量构成的集合

注意：一般遇到这样的叠词，本质就是后者

2、指针数组的定义方法：

类型说明符 * 数组名 [元素个数];

```
1 int * p[10]; //定义了一个整型的指针数组p, 有10个元素p[0]~p[9], 每个元素都是int *类型的变量
2 int a;
3 p[1]=&a;
4 int b[10];
5 p[2]=&b[3];
6 p[2]、*(p+2)是等价的, 都是指针数组中的第2个元素。
```

3、指针数组的分类

字符指针数组char *p[10]、短整型指针数组、整型的指针数组、长整型的指针数组

float型的指针数组、double型的指针数组

结构体指针数组、函数指针数组

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     // 大多数情况下, 指针数组都用来保存多个字符串
6     char *name[5] = {"Follow me", "BASIC", "Greatwall", "FORTRAN", "Computer"};
7     int i;
8     for(i=0; i<5; i++)
9     {
10         printf("%s\n", name[i]);
11     }
12
13     return 0;
14 }
15
```

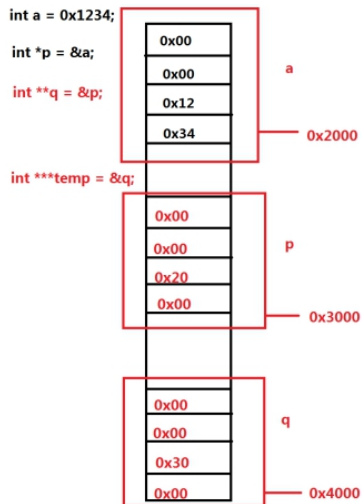
执行结果

```
Starting C:\Users\lzx\Desktop\src\
Debug\debug\07_point_array.exe...
Follow me
BASIC
Greatwall
FORTRAN
Computer
C:\Users\lzx\Desktop\src\build-07.
\debug\07_point_array.exe exited \
```

九、指针的指针 -- 二级指针

指针的指针, 即指针的地址,

咱们定义一个指针变量本身指针变量占4个字节，指针变量也有地址编号



```
int a;
int *p;
p=&a;
*p == a
int **q;
q=&p;
*q == p
**q == *p == a
int ***m;
m=&q;
*(*m) == a
```

注意：

p q m都是指针变量，都占4个字节，都存放地址编号，只不过类型不一样而已

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     int a = 100;
6
7     //定义一个一级指针
8     //一级指针用于保存普通变量的地址
9     int *p = &a;
10
11     //定义一个二级指针
```



```

12 //二级指针用于保存一级指针的地址
13 int **q = &p;
14
15 printf("a = %d %d %d\n", a, *p, **q);
16 printf("&a = %p %p %p\n", &a, p, *q);
17 printf("&p = %p %p\n", &p, q);
18 printf("&q = %p\n", &q);
19
20 return 0;
21 }

```

执行结果

```

Starting C:\Users\lzx\Desktop\src\bui
Debug\debug\08_point_point.exe...
a = 100 100 100
&a = 0029FEAC 0029FEAC 0029FEAC
&p = 0029FEA8 0029FEA8
&q = 0029FEA4
C:\Users\lzx\Desktop\src\build-08_poi

```

十、字符串和指针

字符串的概念：

字符串就是以‘\0’结尾的若干的字符的集合

字符串的存储形式：数组、字符串指针、堆

1、char string[100] = "I love C!"

定义了一个字符数组string,用来存放多个字符，并且用"I love C!"给string数组初始化

，字符串"I love C!"存放在string中

2、char *str = "I love C!"

定义了一个指针变量str,只能存放字符地址编号，

所以说I love C! 这个字符串中的字符不能存放在str指针变量中。

str只是存放了字符I的地址编号，"I love C!"存放在文字常量区

3、char *str = (char*)malloc(10*sizeof(char));

动态申请了10个字节的存储空间，首地址给str赋值。

strcpy(str,"I love C"); //将字符串"I love C!"拷贝到str指向的内存里

总结：

字符数组：

在内存（栈、静态全局区）中开辟了一段空间存放字符串

字符串指针：

在文字常量区开辟了一段空间存放字符串，将字符串的**首地址**付给str

堆：

使用malloc函数在堆区申请空间，将字符串拷贝到堆区

注意：

可修改性：

1. 栈和全局区内存中的内容是可修改的

```
char str[100]=" I love C!" ;
```

```
str[0]= 'y' ;//正确可以修改的
```

2. 文字常量区里的内容是**不可修改的**

```
char *str=" I love C!" ;
```

```
*str = ' y' ;//错误，I存放在文字常量区，不可修改
```

3. 堆区的内容是可以修改的

```
char *str =(char*)malloc(10*sizeof(char));
```

```
strcpy(str,"I love C");
```

```
*str= ' y' ;//正确，可以，因为堆区内容是可修改的
```

注意：str指针指向的内存能不能被修改，要看str指向哪里。

str指向文字常量区的时候，内存里的内容不可修改

str指向栈、堆、静态全局区的时候，内存的内容是可以修改

初始化：

字符数组、指针指向的字符串：定义时直接初始化

```
char buf_aver[]="hello world";
```

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

堆中存放的字符串不能初始化、只能使用strcpy、scanf赋值

```
char *buf_heap;
```

```
buf_heap=(char *)malloc(15);
```

```
strcpy(buf_heap,"hello world");
```

```
scanf( "%s" ,buf_heap);
```

使用时赋值

字符数组：使用scanf或者strcpy

```
char buf_aver[128];
```

```
buf_aver="hello kitty"; 错误,因为字符数组的名字是个常量
```

```
strcpy(buf_aver,"hello kitty"); 正确
```

```
scanf("%s",buf_aver); 正确
```

指向字符串的指针：

```
char *buf_point;
```

```
buf_point="hello kitty";
```

 正确, buf_point指向另一个字符串

```
strcpy(buf_point,"hello kitty");
```

 错误, 只读, 能不能复制字符串到buf_piont指向的内存里

取决于buf_point指向哪里。

十一、数组指针

1、二维数组

二维数组，有行，有列。二维数组可以看成有多个一维数组构成的，是多个一维数组的集合，可以认为二维数组的每一个元素是个一维数组。

例：

```
int a[3][5];
```

定义了一个3行5列的一个二维数组。

可以认为二维数组a由3个一维数组构成，每个元素是一个一维数组。

回顾：

数组的名字是数组的首地址，是第0个元素的地址，是个常量，数组名字加1指向下个元素

二维数组a中，a+1 指向下个元素，即下一个一维数组，即下一行。

```
1 #include <stdio.h>
2 int main(int argc, char *argv[])
3 {
4     int a[3][5];
5     printf("a=%p\n", a);
6     printf("a+1=%p\n", a+1);
7     return 0;
8 }
```

2、数组指针的概念：

本身是个指针，指向一个数组，加1跳一个数组，即指向下个数组。

数组指针的作用就是可以保存二维数组的首地址

3、数组指针的定义方法：

指向的数组的类型（*指针变量名）[指向的数组的元素个数]

`int (*p)[5];` //定义了一个数组指针变量p，p指向的是整型的有5个元素的数组
p+1 往下指5个整型，跳过一个有5个整型元素的数组。

```
1 void test1()
2 {
3     int a[3][5]; //定义了一个3行5列的一个二维数组
4     int (*p)[5]; //定义一个数组指针变量p，p+1跳一个有5个元素的整型数组
5
6     printf("a=%p\n", a); //第0行的行地址
7     printf("a+1=%p\n", a+1); //第1行的行地址，a和a +1差20个字节
8
9     p=a;
10
11     printf("p=%p\n", p);
12     printf("p+1=%p\n", p+1); //p+1跳一个有5个整型元素的一维数组
13 }
```

执行结果

```
Starting C:\Users\lxz\Desktop\src\
Debug\debug\09_array_point.exe...
a=0029FE60
a+1=0029FE74
p=0029FE60
p+1=0029FE74
C:\Users\lxz\Desktop\src\build-09
\debug\09_array_point.exe exited
```

数组指针的用法

```
1 //数组指针的用法
2 //可以将二维数组的首地址传递到另一个函数里面，此时函数的形参就需要定义为数组指针
3 void fun(int(*p)[5], int x, int y)
4 {
5     p[0][1]=101;
6 }
7
8 void test2()
9 {
10     int i, j;
11     int a[3][5] = {0};
12     fun(a, 3, 5);
13     for(i=0; i<3; i++)
14     {
15         for(j=0; j<5; j++)
16         {
```

```

17  printf("%d ",a[i][j]);
18  }
19  printf("\n");
20  }
21  }

```

执行结果

```

Starting C:\Users\lzx\Desktop
Debug\debug\09_array_point.exe
0 101 0 0 0
0 0 0 0 0
0 0 0 0 0
C:\Users\lzx\Desktop\src\buil
\debug\09_array_point.exe exi

```

4、各种数组指针的定义：

(1)、一维数组指针，加1后指向下一个一维数组

```
int(*p)[5];
```

配合每行有5个int型元素的二维数组来用

```
int a[3][5]
```

```
int b[4][5]
```

```
int c[5][5]
```

```
int d[6][5]
```

.....

```
p=a;
```

```
p=b;
```

```
p=c;
```

```
p=d;
```

都是可以的~~~~

(2)、二维数组指针，加1后指向下一个二维数组

```
int(*p)[4][5];
```

配合三维数组来用，三维数组中由若干个4行5列二维数组构成

```
int a[3][4][5];
```

```
int b[4][4][5];
```

```
int c[5][4][5];
```

```
int d[6][4][5];
```

这些三维数组，有个共同的特点，都是有若干个4行5的二维数组构成。

```
p=a;
```

```
p=b;
```

```
p=c;
```

```
p=d;
```

(3)、三维数组指针，加1后指向下个三维数组

```
int(*p)[4][5][6];
```

p+1跳一个三维数组；

什么样的三维数组啊？

由4个5行6列的二维数组构成的三维数组

配合：

```
int a[7][4][5][6];
```

(4)、四维数组指针，加1后指向下个四维数组，以此类推。。。。

5、容易混淆的内容：

指针**数组**：是个数组，有若干个相同类型的指针构成的集合

```
int *p[10];
```

数组p有10个int *类型的指针变量构成，分别是p[0] ~ p[9]

数组**指针**：本身是个指针，指向一个数组，加1跳一个数组

```
int (*p)[10];
```

P是个指针，p是个数组指针，p加1指向下个数组，跳10个整形。

指针的**指针**：

```
int **p;//p是指针的指针
```

```
int *q;
```

```
p=&q;
```

6、数组名字取地址：变成 **数组指针**

一维数组名字取地址，变成一维数组指针，即加1跳一个一维数组

```
int a[10];
```

a+1 跳一个整型元素，是a[1]的地址

a和a+1 相差一个元素，4个字节

&a就变成了一个一维数组指针,是 int(*p)[10]类型的。

(&a) +1 和&a相差一个数组即10个元素即40个字节。

7、数组名字和指针变量的区别：

```
int a[10];  
int *p;  
p=a;
```

相同点：

a是数组的名字，是a[0]的地址，p=a即p也保存了a[0]的地址，即a和p都指向a[0]，所以在引用数组元素的时候，a和p等价

不同点：

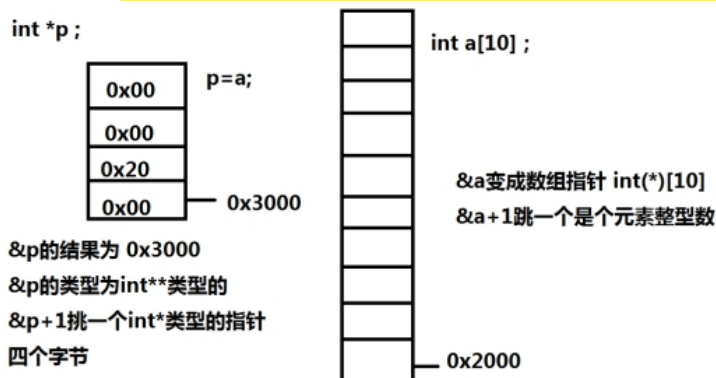
1、a是常量、p是变量

可以用等号'='给p赋值，但是不能用等号给a赋值

2、对a取地址，和对p取地址结果不同

因为a是数组的名字，所以对a取地址结果为数组指针。

p是个指针变量，所以对p取地址（&p）结果为指针的指针。



8、多维数组中指针的转换：

在二维数组中，行地址取*不是取值得意思，而是指针降级的意思，由行地址（数组指针）变成这一行第0个元素的地址。取*前后还是指向同一个地方，但是指针的类型不一样了

```
1 //二维数组的数组名降级问题  
2 //二维数组的数组名默认是一个行指针，加1保存下一行的首地址  
3 //二维数组的数据名取*，表示地址的降级，意味着行指针降级为列指针，加1保存下一个元素的地址  
4  
5 //一维数组的数组名默认是一个列指针，加1保存下一个元素的地址  
6 //一维数组的数组名取&，则是地址的升级，将列指针升级为行指针，加1保存下一行元素的首地址  
7 void test3()  
8 {  
9     int a[3][5];  
10    printf("a=%p\n",a);
```

```

11  printf("a +1=%p\n",a+1);
12
13  printf("*a =%p\n",*a);// *a变成了第0行第0列元素的地址
14  printf("(a)+1 =%p\n",(*a)+1 ); //结果为第0行第1列元素的地址
15  }

```

执行结果

```

Starting C:\Users\lx\Desktop\src\build-64\debug\09_array_point.exe..
a=0029FE64
a +1=0029FE78
*a =0029FE64
(a)+1 =0029FE68
C:\Users\lx\Desktop\src\build-64\debug\09_array_point.exe exited

```

十二、指针与函数的关系

12.1 指针作为函数的参数

咱们可以给一个函数传一个 整型、字符型、浮点型的数据，也可以 给函数传一个地址。

函数的传参方式：复制传参、地址传参、全局传参（几乎用不到）

12.1.1 复制传参 -- 传数值

```

1  #include <stdio.h>
2
3  //函数的传参方式之复制传参：将实参的值传递给形参，不管形参怎么改变，跟实参都没有关系
4  void myfun1(int a, int b)
5  {
6      int temp;
7      temp = a;
8      a = b;
9      b = temp;
10
11     printf("in fun: a = %d, b = %d\n", a, b);
12     printf("&a = %p, &b = %p\n", &a, &b);
13 }
14
15 void test1()
16 {
17     int a = 100, b = 20;

```



```

18
19     printf("before fun: a = %d, b = %d\n", a, b);
20     printf("&a = %p, &b = %p\n", &a, &b);
21
22     myfun1(a, b);
23
24     printf("after fun: a = %d, b = %d\n", a, b);
25
26 }
27
28 int main(int argc, char *argv[])
29 {
30     test1();
31
32     return 0;
33 }

```

执行结果

**Starting C:\Users\lzx\Desktop\src\build
Debug\debug\10_point_fun.exe...**

before fun: a = 100, b = 20

&a = 0029FE9C, &b = 0029FE98

in fun: a = 20, b = 100

&a = 0029FE80, &b = 0029FE84

after fun: a = 100, b = 20

**C:\Users\lzx\Desktop\src\build-10_point
\10_point_fun.exe exited with code 0**

12.1.2 地址传参 -- 传地址

```

1 //函数的传参方式之地址传参：将实参的地址传递给形参，形参对保存的地址的内容
2 //进行任何操作，实参的值也会跟着改变
3 void myfun2(int *p, int *q)
4 {
5     int temp;
6     temp = *p;
7     *p = *q;
8     *q = temp;
9
10    printf("in fun: *p = %d, *q = %d\n", *p, *q);
11    printf("p = %p, q = %p\n", p, q);
12 }
13
14 void test1()

```

```

15 {
16     int a = 100, b = 20;
17
18     printf("before fun: a = %d, b = %d\n", a, b);
19     printf("&a = %p, &b = %p\n", &a, &b);
20
21     myfun2(&a, &b);
22
23     printf("after fun: a = %d, b = %d\n", a, b);
24
25 }
26
27 int main(int argc, char *argv[])
28 {
29     test1();
30
31     return 0;
32 }

```

执行结果

Starting C:\Users\lzx\Desktop\src\build-10_point_fun.exe...
Debug\debug\10_point_fun.exe...

before fun: a = 100, b = 20

&a = 0029FE9C, &b = 0029FE98

in fun: *p = 20, *q = 100

p = 0029FE9C, q = 0029FE98

after fun: a = 20, b = 100

C:\Users\lzx\Desktop\src\build-10_point_fun.exe exited with code 0

注意：如果实参是一个普通变量，地址传参的话就需要形参是一级指针，

如果实参是一个一级指针，地址传参的话就需要形参是一个二级指针，

以此类推

12.2 传数组

将数组作为参数传递给函数，不存在复制传参和地址传参，本质都是地址传参，所以在函数内部对数组进行改变，则函数执行完毕后，原本的数组也会改变，因为传递给函数的都是数组的地址

```

1 //传一维数组
2 //void fun1(int p[])//形式1
3 void fun1(int *p)//形式2(常用)
4 {

```

```
5  printf("%d\n",p[2]);
6  printf("%d\n",*(p+3));
7  }
8
9  void test2()
10 {
11     int a[10]={1,2,3,4,5,6,7,8};
12     fun1(a);
13 }
14
15 //传二维数组
16 //void fun2( int p[][4] )//形式1
17 void fun2( int (*p)[4] )//形式2: 通过数组指针
18 {
19     //p[x][y] <==> (*(p + x) + y)
20     printf("%d\n", p[0][2]);
21     printf("%d\n", (*(p+1) + 2));
22 }
23
24 void test3()
25 {
26     int a[2][4] = {1, 2, 3, 4,
27     5, 6, 7, 8};
28     fun2(a);
29 }
30
31 //传指针数组
32 void fun3(char **q)
33 {
34     int i;
35     for(i=0;i<3;i++)
36     {
37         printf("%s\n",q[i]);
38     }
39 }
40
41 void test4()
42 {
43     char *p[3]={"hello","world","kitty"};
44     fun3(p);
```

12.3 指针函数 -- 指针作为函数的返回值

指针函数本质是一个函数，只不过函数的返回值是一个指针

```

1 //指针函数：指针作为函数的返回值
2 char *fun4()
3 {
4     //栈区开辟的空间会随着当前代码段的结束而释放空间
5     //char str[100]="hello world";
6
7     //静态区的空间不会随着当前代码段的结束而释放空间
8     static char str[100]="hello world";
9
10    return str;
11 }
12
13 void test5()
14 {
15     char *p;
16     p = fun4();
17     printf("p = %s\n", p);
18 }

```

12.4 函数指针 - 指针保存函数的地址

咱们定义的函数，在运行程序的时候，会将函数的指令加载到内存的代码段，所以函数也有起始地址。

c语言规定：函数的名字就是函数的首地址，即函数的入口地址 咱们就可以定义一个指针变量，

来存放函数的地址，这个指针变量就是函数指针变量。

12.4.1 函数指针变量的定义方法

返回值类型 (*函数指针变量名)(形参列表);

```

1 int (*p)(int,int);//定义了一个函数指针变量p,p指向的函数
2 //必须有一个整型的返回值，有两个整型参数。
3 int max(int x,int y) { }
4 int min(int x,int y) { }
5 //可以用这个p存放这类函数的地址。
6 p=max; p=min;

```

12.4.2 调用函数的方法

1. 通过函数的名字去调函数（最常用的）

```
1 int max(int x,int y) { }
2 int main()
3 {
4     int num;
5     num=max(3,5);
6 }
```

2. 可以通过函数指针变量去调用

```
1 int max(int x,int y) { }
2 int main()
3 {
4     int num;
5     int (*p)(int ,int);
6     p=max;
7     num=p(3,5);
8 }
```

12.4.3 函数指针数组

函数指针数组：本质是一个数组，数组里面的每一个元素都是一个函数指针

返回值类型 (*函数指针变量名[函数指针的个数])(形参列表);

```
int(*p[10])(int,int);
```

定义了一个函数指针数组，有10个元素p[0] ~p[9]，每个元素都是函数指针变量，指向的函数，必须有整型的返回值，两个整型参数。

12.4.4 函数指针最常用的地方

函数指针最常用的地方在于将一个函数作为参数传递给另一个函数的时候要使用函数指针
将一个函数作为参数传递给另一个函数，将这个函数称之为回调函数

```
1 #include <stdio.h>
2
3 int add(int x,int y)
4 {
5     return x+y;
6 }
```

```
7 int sub(int x,int y)
8 {
9     return x-y;
10 }
11 int mux(int x,int y)
12 {
13     return x*y;
14 }
15 int dive(int x,int y)
16 {
17     return x/y;
18 }
19
20 int process(int (*p)(int ,int),int a,int b)
21 {
22     int ret;
23     ret = (*p)(a,b);
24     return ret;
25 }
26
27 int main(int argc, char *argv[])
28 {
29     int num;
30     num = process(add,2,3);
31     printf("num = %d\n",num);
32
33     num = process(sub,2,3);
34     printf("num = %d\n",num);
35
36     num = process(mux,2,3);
37     printf("num = %d\n",num);
38
39     num = process(dive,2,3);
40     printf("num = %d\n",num);
41
42     return 0;
43 }
```

执行结果

```
Starting C:\Users\lzx\Desktop\src\bu
Debug\debug\11_fun_point.exe...
num = 5
num = -1
num = 6
num = 0
C:\Users\lzx\Desktop\src\build-11_fu
\11_fun_point.exe exited with code 0
```

十三、经常容易混淆的指针

第一组：

1、 `int *a[10];`

这是个指针数组，数组a中有10个整型的指针变量
a[0]~a[9]

2、 `int (*a)[10];`

数组指针变量，它是个指针变量。它占4个字节，存地址编号。
它指向一个数组，它加1的话，指向下个数组。

3、 `int **p;`

这个是个指针的指针，保存指针变量的地址。
它经常用在保存指针的地址：

常见用法1：

```
int **p
int *q;
p=&q;
```

常见用法2：

```
int **p;
int *q[10];
```

分析：q是指针数组的名字，是指针数组的首地址，是q[0]的地址。
q[0]是个int *类型的指针。 所以q[0]指针变量的地址，是int **类型的

第二组：

1、 `int *f(void);`

注意：*f没有用括号括起来
它是个函数的声明，声明的这个函数返回值为int *类型的。

2、 `int (*f)(void);`

注意*f用括号括起来了，*修饰f说明，f是个指针变量。

f是个函数指针变量，存放函数的地址，它指向的函数，必须有一个int型的返回值，没有参数。

十四、特殊指针

1、空类型的指针 (void *)

char * 类型的指针指向char型的数据

int * 类型的指针指向int型的数据

float* 类型的指针指向float型的数据

void * 难道是指向void型的数据吗？

不是，因为没有void类型的变量

回顾：对应类型的指针只能存放对应类型的数据地址

void* 通用指针，任何类型的指针都可以给void*类型的指针变量赋值。主要也是用在函数的参数和返回值的位置

```
int *p;
```

```
void *q;
```

q=p 是可以的，不用强制类型转换

举例子：

有个函数叫memset

```
void * memset(void *s,int c,size_t n);
```

这个函数的功能是将s指向的内存前n个字节，全部赋值为 c。

Memset可以设置字符数组、整型数组、浮点型数组的内容，所以第一个参数，就必须是个通用指针

它的返回值是s指向的内存的首地址，可能是不同类型的地址。所以返回值也得是通用指针

注意：void*类型的指针变量，也是个指针变量，在32为系统下，占4个字节

2、NULL

空指针:

```
char *p=NULL;
```

咱们可以认为p哪里都不指向，也可以认为p指向内存编号为0的存储单位。

在p的四个字节中，存放的是0x00 00 00 00

一般NULL用在给指针初始化。

十五、main函数传参

`int main(int argc, char *argv[])`

argc：是一个int类型的变量，标识命令终端传入的参数的个数

argv：是一个指针数组，用于保存每一个命令终端传入的参数

```
1 #include <stdio.h>
2 int main(int argc, char *argv[])
3 {
4     int i;
5     printf("argc=%d\n",argc);
6     for(i=0;i<argc;i++)
7     {
8         printf("argv[%d]=%s\n",i,argv[i]);
9     }
10
11     return 0;
12 }
```