第 1 章 指针

5.1 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。

32bit 32 根

0x00 00 00 00

0xff ff ff ff

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。

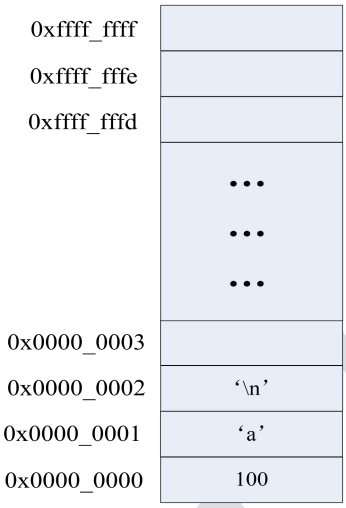
内存以字节为单位来存储数据的，咱们可以将程序中的虚拟寻址空间，看成一个很大的一维的字符数组

5.2 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



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

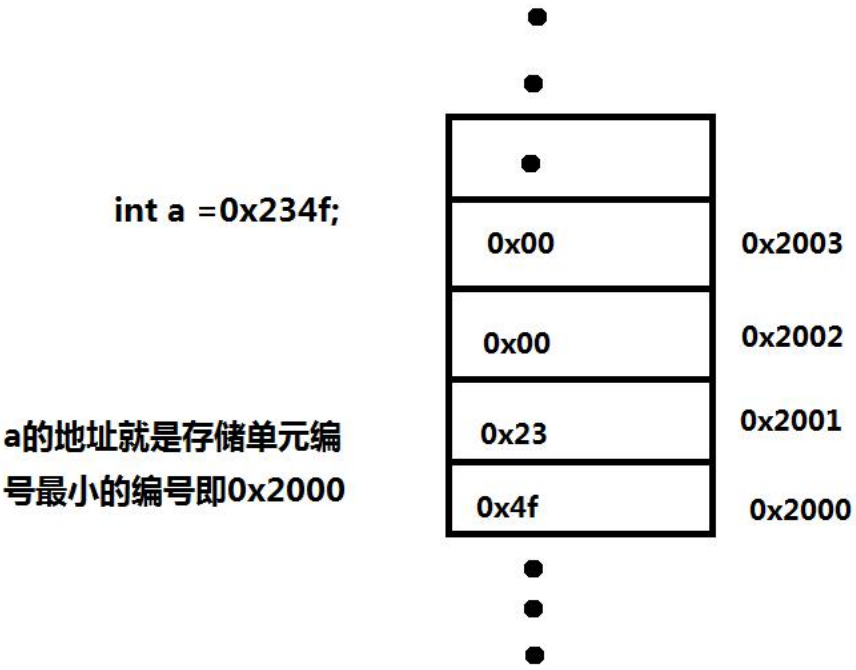
2：对应类型的指针变量，只能存放对应类型的变量的地址

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

5.3 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

例 1 :

```
int a=0x0000234f;
int *p;//在定义指针变量的时候*代表修饰的意思，修饰 p 是个指针变量。
p=&a;//把 a 的地址给 p 赋值 ，&是取地址符，
```

p 保存了 a 的地址，也可以说 p 指向了 a

int num；

num=*p;//注意：在调用的时候 *代表取值得意思 ，*p 就相当于 p 指向的变量，即 a

所以说 num 的值为 0x234f;

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

例 2：

```
int main()
{
	int a= 100, b = 200;
	int *p_1,  *p_2 = &b;  //表示该变量的类型是一个指针变量，指针变量名是 p_1 而不是*p_1.
	//p_1 在定义的时候没有赋初值，p_2 赋了初值
	p_1= &a ;  //p_1 先定义后赋值
	printf("%d\n", a);
	printf("%d\n", *p_1);
	printf("%d\n", b);
	printf("%d\n", *p_2);
	return 0;
}
```

注意：

在定义 p_1 的时候，因为是个局部变量，局部变量没有赋初值，它的值是随机的，p_1 指向哪里不一定，所以 p_1 就是个野指针。

3.指针大小

例 3：在 32 位系统下，所有类型的指针都是 4 个字节

```
#include <stdio.h>
int main(int argc, char *argv[])
{
	char *p1;
	short int *p2;
	int *p3;
	long int *p4;
	float *p5;
	double *p6;

	printf("%d\n",sizeof(p1));
	printf("%d\n",sizeof(p2));
	printf("%d\n",sizeof(p3));
	printf("%d\n",sizeof(p4));
	printf("%d\n",sizeof(p5));
	printf("%d\n",sizeof(p6));
	return 0;
}
```

例 4：

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
{
    int a=0x1234abcd;
    int *p;
    p=&a;

    printf("&a=%p\n",&a);
    printf("p=%p\n",p);
    return 0;
}
```

5.4 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。

5.5 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

例 5：

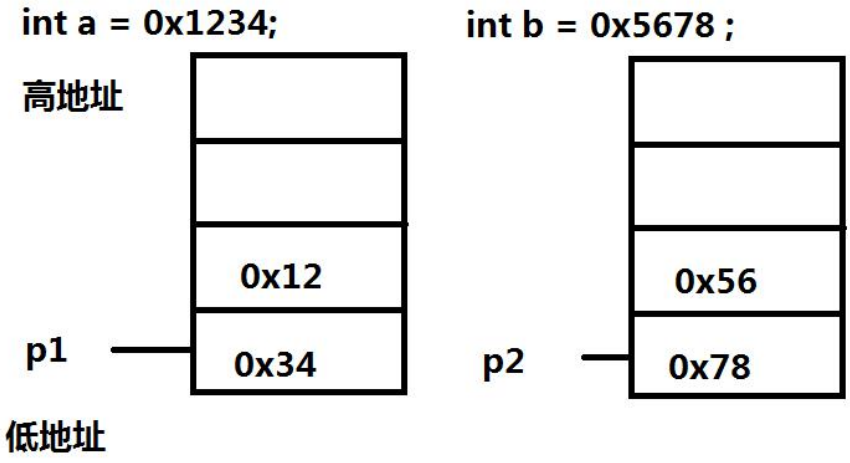
```
#include <stdio.h>
int   main()
{
    int *p1,*p2,temp,a,b;
    p1=&a;
    p2=&b;
    printf("请输入:a b 的值:\n");
    scanf("%d %d",p1,p2);//给 p1 和 p2 指向的变量赋值
    temp = *p1; //用 p1 指向的变量（a）给 temp 赋值
    *p1 = *p2; //用 p2 指向的变量（b）给 p1 指向的变量（a）赋值
    *p2 = temp;//temp 给 p2 指向的变量（b）赋值
    printf("a=%d b=%d\n",a,b);
    printf("*p1=%d *p2=%d\n",*p1,*p2);
```

```
    return 0;
}
```

运行结果：
输入 100　200
运行结果为：
a=200　b=100
a=200　b=100
扩展：
对应类型的指针，只能保存对应类型数据的地址，
如果想让不同类型的指针相互赋值的时候，需要强制类型转换

例 6：

```
#include <stdio.h>
int  main()
{
    int a=0x1234,b=0x5678;
    char *p1,*p2;
    printf("%0x %0x\n",a,b);
    p1=(char *)&a;
    p2=(char *)&b;
    printf("%0x %0x\n",*p1,*p2);
    p1++;
    p2++;
    printf("%0x %0x\n",*p1,*p2);
    return 0;
}
```



结果为：
0x 34　0x78
0x12　0x56

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

5.6 指针和数组元素之间的关系

1、

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。

例 7：

```
int  a[10];
//int *p =&a[0];
int *p;
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
```

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。

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]

例 8：

```
#include <stdio.h>
int main(int argc, char *argv[])
```

```
{
    int a[5]={0,1,2,3,4};
    int *p;
    p=a;
    printf("a[2]=%d\n",a[2]);
    printf("p[2]=%d\n",p[2]);
    printf("*(p+2)%d\n",*(p+2));
    printf("*(a+2)%d\n",*(a+2));
    printf("p=%p\n",p);
    printf("p+2=%p\n",p+2);
    return 0;
}
```

5.7 指针的运算

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

```
例 9：
int a[10];
int *p;
p=a;
p+2;//p 是 a[0]的地址，p+2 是&a[2]
```

假如 p 保存的地址编号是 2000 的话，p+2 代表的地址编号是 2008

```
例 10：
char buf[10]；
char *q;
q=buf;
q+2 //相当于&buf [2]
```

假如：q 中存放的地址编号是 2000 的话，q+2 代表的地址编号是 2002

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

```
例 11：
#include <stdio.h>
int main(int argc, char *argv[])
{
```

```
    int a[10];
    int *p,*q,n;//如果在一行上定义多个指针变量的，每个变量名前面加*
    //上边一行定义了两个指针 p 和 q ，定义了一个整型的变量 n
    p=&a[1];
    q=&a[6];
    if(p<q)
    {
        printf("p<q\n");
    }
    else if(p>q)
    {
        printf("p>q\n");
    }
    else
    {
        printf("p == q\n");
    }
    return 0;
}
```

结果是 p<q

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

例 12:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a[10];
    int *p,*q;
    p=&a[0];
    q=&a[3];
    printf("%d\n",q-p);
    return 0;
}
```

结果是 3

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）

int *p;

int *q;
int a;
p=&a;//p 保存 a 的地址，p 指向了变量 a
q=p; //用 p 给 q 赋值，q 也保存了 a 的地址，指向 a
注意：如果类型不相同的指针要想相互赋值，必须进行强制类型转换

注意：c 语言规定数组的名字，就是数组的首地址，就是数组第 0 个元素的地址
int *p;
int a[10];
p=a; p=&a[0];这两种赋值方法是等价的

5.8 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**

2、指针数组的定义方法：
类型说明符 * 数组名 [元素个数];
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
int a;
p[1]=&a;
int b[10];
p[2]=&b[3];
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

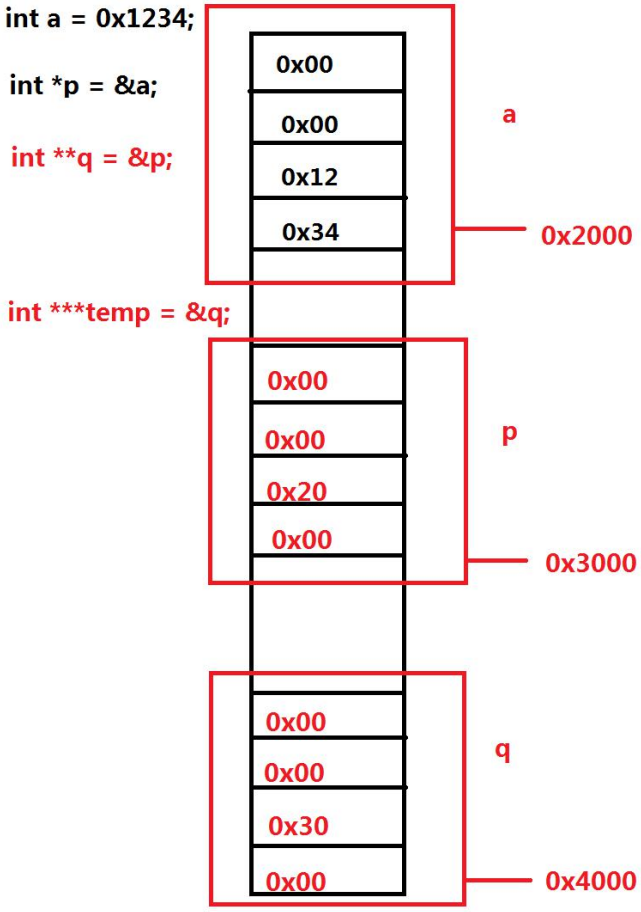
例 13：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    char *name[5] = {"Follw me","BASIC","Greatwall","FORTRAN","Computer"};
    int i;
    for(i=0;i<5;i++)
    {
        printf("%s\n",name[i]);
    }
    return 0;
}
```

5.9 指针的指针

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

5.10 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

- **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。

5.11 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。

例 14：

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

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。

例 15：

```
#include<stdio.h>
int main()
{
    int a[3][5];//定义了一个 3 行 5 列的一个二维数组
    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
    printf("a=%p\n",a);//第 0 行的行地址
    printf("a+1=%p\n",a+1);//第 1 行的行地址，a 和 a +1 差 20 个字节
    p=a;
    printf("p=%p\n",p);
    printf("p+1=%p\n",p+1);//p+1 跳一个有 5 个整型元素的一维数组
    return 0;
}
```

例 16：数组指针的用法 1

```
#include<stdio.h>
void fun(int(*p)[5],int x,int y)
{
    p[0][1]=101;
}
int main()
{
    int i,j;
    int a[3][5];
    fun(a,3,5);
    for(i=0;i<3;i++)
    {
        for(j=0;j<5;j++)
        {
            printf("%d ",a[i][j]);
        }
```

```
            printf("\n");
        }
    }
```

4、各种数组指针的定义：

(1)、一维数组指针，加 1 后指向下个一维数组

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;

5、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];

6、四维数组指针，加 1 后指向下个四维数组，以此类推。。。。

7、注意：

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;

8、**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。

例 18：

```
#include<stdio.h>
int main()
{
    int a[10];
    printf("a=%p\n",a);
    printf("a+1=%p\n",a+1);

    printf("&a=%p\n",&a);
    printf("&a +1=%p\n",&a+1);
    return 0;
}
```

a 是个 int *类型的指针，是 a[0]的地址。
&a 变成了数组指针，加 1 跳一个 10 个元素的整型一维数组

在运行程序时，大家会发现 a 和&a 所代表的地址编号是一样的，即他们指向同一个存储单元，但是 a 和&a 的指针类型不同。

例 19：

```
int  a[4][5];
a+1 跳 5 个整型
```

(&a)+1 跳 4 行 5 列（80 个字节）。

总结：c 语言规定，数组名字取地址，变成了数组指针。加 1 跳一个数组。

9、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

引用数组元素回顾：

a[2]、*(a+2)、p[2]、*(p+2) 都是对数组 a 中 a[2]元素的引用。

不同点：

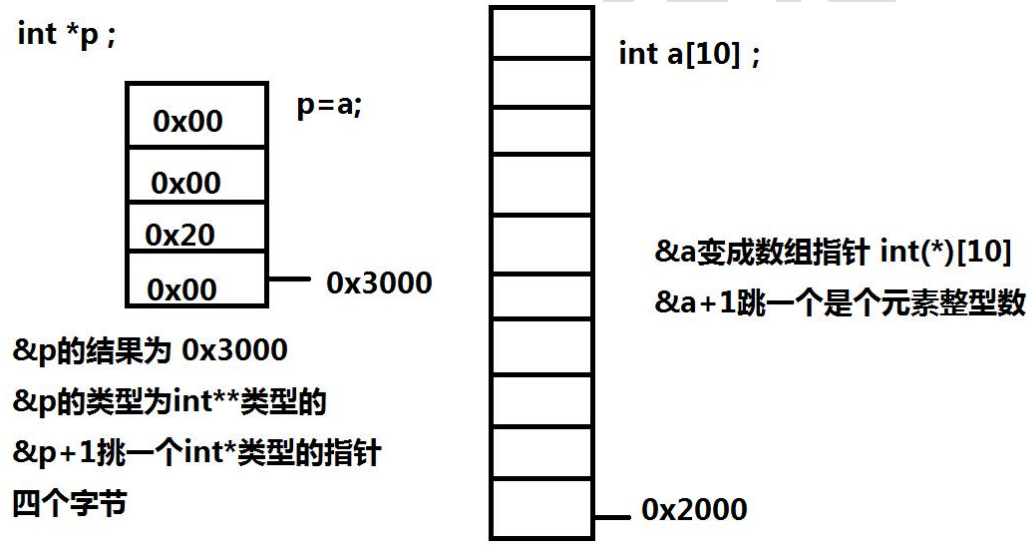
1、 a 是常量、p 是变量

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。



例 20：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a[10];
    int *p;
    p=a;
    printf("a=%p\n",a);
```

```
    printf("&a=%p\n",&a);
    printf("&a +1 =%p\n",&a +1);

    printf("p=%p\n",p);
    printf("&p=%p\n",&p);
    printf("&p +1=%p\n",&p +1);
    return 0;
}
```

10、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

```
例 21 :
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a[3][5];
    printf("a=%p\n",a);
    printf("a +1=%p\n",a+1);

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

5.12 指针和函数的关系

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。

例：

```
int num;
scanf("%d",&num);
```

函数传参：

(1)、传数值：

```
例 22:
void swap(int x,int y)
{
```

```
    int temp;
    temp=x;
    x=y;
    y=temp;
}
int main()
{
    int a=10,b=20;
    swap(a,b);
    printf("a=%d b=%d\n",a,b);//a=10   b=20
}
```

实参：调用函数时传的参数。
形参：定义被调函数时，函数名后边括号里的数据
结论：给被调函数传数值，只能改变被调函数形参的值，不能改变主调函数实参的值

(2)、传地址：

例 23：

```
void swap(int *p1,int *p2)
{
    int temp;
    temp= *p1;
    *p1=*p2;// p2 指向的变量的值，给 p1 指向的变量赋值
    *p2=temp;
}
int main()
{
    int a=10,b=20;
    swap(&a,&b);
    printf("a=%d b=%d\n",a,b);//结果为 a=20 b=10
}
```

结论：调用函数的时候传变量的地址，在被调函数中通过*+地址来改变主调函数中的变量的值

例 24：

```
void swap(int *p1,int *p2)//&a   &b
{
    int *p;
    p=p1;
    p1=p2;//p1 =&b 让 p1 指向 main 中的 b
    p2=p;//让 p2  指向  main 函数中  a
```

```
}//此函数中改变的是 p1 和 p2 的指向，并没有给 main 中的 a 和 b 赋值
int main()
{
    int a=10,b=20;
    swap(&a,&b);
    printf("a=%d b=%d\n",a,b);//结果为 a=10 b=20
}
```

总结一句话：要想改变主调函数中变量的值，必须传变量的地址，
而且还得通过*+地址 去赋值。无论这个变量是什么类型的。

```
例 25：
void fun(char *p)
{
    p="hello kitty";
}
int main()
{
    char *p="hello world";
    fun(p);
    printf("%s\n",p);//结果为： hello world
}
```

答案分析：

在 fun 中改变的是 fun 函数中的局部变量 p，并没有改变 main 函数中的变量 p，所以 main 函数中的，变量 p 还是指向 hello world。

```
例 26：
void fun(char **q)
{
    *q="hello kitty";
}
int main()
{
    char *p="hello world";
    fun(&p);
    printf("%s\n",p);//结果为：hello kitty
}
```

总结一句话：要想改变主调函数中变量的值，必须传变量的地址，而且还得通过*+地址 去赋值。无论这个变量是什么类型的。

(3)、传数组：

给函数传数组的时候，没法一下将数组的内容作为整体传进去。

只能传数组的地址。

例 27：传一维数组

```
//void fun(int p[])//形式 1
void fun(int *p)//形式 2
{
	printf("%d\n",p[2]);
	printf("%d\n",*(p+3));
}
int main()
{
	int a[10]={1,2,3,4,5,6,7,8};
	fun(a);
	return 0;
}
```

例 28：传二维数组

```
//void fun( int p[][4] )//形式 1
void fun( int (*p)[4] )//形式 2
{
}
int main()
{
	int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
	fun(a);
	return 0;
}
```

例 29：传指针数组

```
void fun(char **q) // char *q[]
{
	int i;
	for(i=0;i<3;i++)
		printf("%s\n",q[i]);
}
int main()
{
	char *p[3]={"hello","world","kitty"}; //p[0] p[1] p[2] char *
	fun(p);
```

```
        return 0;
    }
```

2、指针函数：指针作为函数的返回值

一个函数可以返回整型数据、字符数据、浮点型的数据，

也可以返回一个指针。

例 30：

```
char * fun()
{
    char str[100]="hello world";
    return str;
}
int main()
{
    char *p;
    p=fun();
    printf("%s\n",p);//
}
```

//总结，返回地址的时候，地址指向的内存的内容不能释放

如果返回的指针指向的内容已经被释放了，返回这个地址，也没有意义了。

例 31：返回静态局部数组的地址

```
char * fun()
{
    static char str[100]="hello world";
    return str;
}
int main()
{
    char *p;
    p=fun();
    printf("%s\n",p);//hello world
}
```

原因是，静态数组的内容，在函数结束后，亦然存在。

例 32：返回文字常量区的字符串的地址

```
char * fun()
{
```

```
    char *str="hello world";
    return str;
}
int main()
{
    char *p;
    p=fun();
    printf("%s\n",p);//hello world
}
```

原因是文字常量区的内容，一直存在。

例 33：返回堆内存的地址

```
char * fun()
{
    char *str;
    str=(char *)malloc(100);
    strcpy(str,"hello world");
    return str;
}
int main()
{
    char *p;
    p=fun();
    printf("%s\n",p);//hello world
    free(p);
}
```

原因是堆区的内容一直存在，直到 free 才释放。

总结：返回的地址，地址指向的内存的内容得存在，才有意义。

3、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。

(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);

int (*p)(int,int);//定义了一个函数指针变量 p,p 指向的函数
必须有一个整型的返回值，有两个整型参数。

```
int max(int x,int y)
{

}

int min(int x,int y)
{

}
```

可以用这个 p 存放这类函数的地址。

```
p=max;
p=min;
```

(2) **:调用函数的方法**

1.通过函数的名字去调函数（最常用的）

```
int max(int x,int y)
{

 }

 int main()
 {
      int num;
      num=max(3,5);
 }
```

2.可以通过函数指针变量去调用

```
int max(int x,int y)
{

}
 int main()
{
      int num;
      int (*p)(int ,int);
      p=max;
      num=p(3,5);
}
```

(3):函数指针数组

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。

(4):函数指针最常用的地方

给函数传参

```
#include<stdio.h>
int add(int x,int y)
{
	return x+y;
}
int sub(int x,int y)
{
	return x-y;
}
int mux(int x,int y)
{
	return x*y;
}
int dive(int x,int y)
{
	return x/y;
}
int process(int (*p)(int ,int),int a,int b)
{
	int ret;
	ret = (*p)(a,b);
	return ret;
}
int main()
{
	int num;
	num = process(add,2,3);
	printf("num =%d\n",num);

	num = process(sub,2,3);
	printf("num =%d\n",num);

	num = process(mux,2,3);
```

```
    printf("num =%d\n",num);

    num = process(dive,2,3);
    printf("num =%d\n",num);
    return 0;
}
```

5.13 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

p=&q[0]; 等价于 p=q;

例 34：

```
void fun(char**p)
{
    int i;
    for(i=0;i<3;i++)
    {
        printf("%s\n",p[i]);//*（p+i）
```

```
        }
    }
    int main()
    {
        char *q[3]={"hello","world","China"};
        fun(q);
        return 0;
    }
```

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。

5.14 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。

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。

5.15 main 函数传参：

例 35：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    printf("argc=%d\n",argc);
    for(i=0;i<argc;i++)
    {
        printf("argv[%d]=%s\n",i,argv[i]);
    }
    return 0;
}
```