

指 针

Wang Houfeng

EECS, PKU

wanghf@pku.edu.cn

内容

➤ 地址与指针

- 指针变量的使用与参数传递
- 指针与数组

认识地址与内容

内存中每个单元有一个特定位置-----简称 **地址**

内存

0

⋮

2000

2001

2002

2003

2004

10

20.5

程序中: `int i=10;`

`float f=20.5;`

f 占用4个字节

变量f 的内容为20.5

变量地址由 **&** 表示
如: **&i**, **&f**

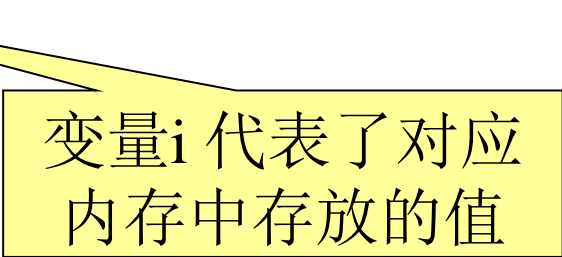
变量f 的地址:
2004 (&f)

直接访问

按变量名存取值的方式称为**直接访问**

```
cout<<i<<endl;
```

```
k=i+j;
```



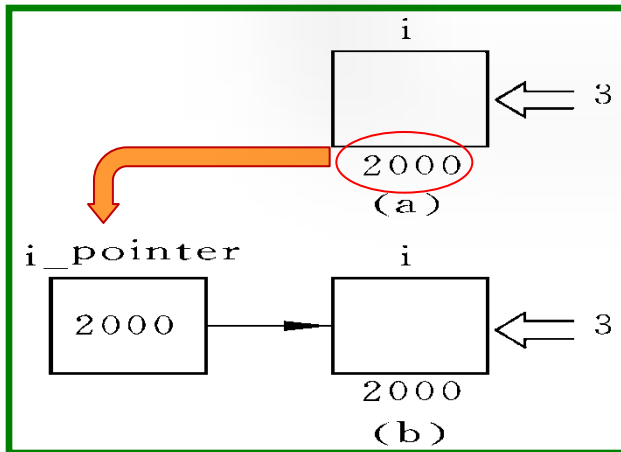
变量*i* 代表了对应
内存中存放的值

变量就是命名的存储单元。

地址与间接访问

按**地址**存取**值**的方式称为**间接访问**

如何表示地址，如何按地址访问？



间接访问是低级程序设计语言（汇编语言）常见的访问方式，称为**间址**：由地址访问内容

如何**区分**变量存放的是内容还是其它单元的地址：

C/C++语言中引入**指针**表示地址！

```
cout<<sizeof(&i)<<endl
```



4

地址由4字节表示

存放地址的变量 —— 指针变量

- 存放**变量地址**的变量称为**指针变量**。
- 声明一个指针变量：


类型标识符 *指针变量名；

例：

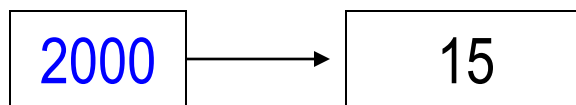
`int i, *i_point; // i_point是指向整型变量的指针变量`

`i_point=&i; //指针变量指向另一变量 i, 使用& (回忆scanf)。`

指针变量不能直接赋整形值，如：

`i_point=2000`  `i_point= &i` ✓
×

i的地址，即，&i 为 2000

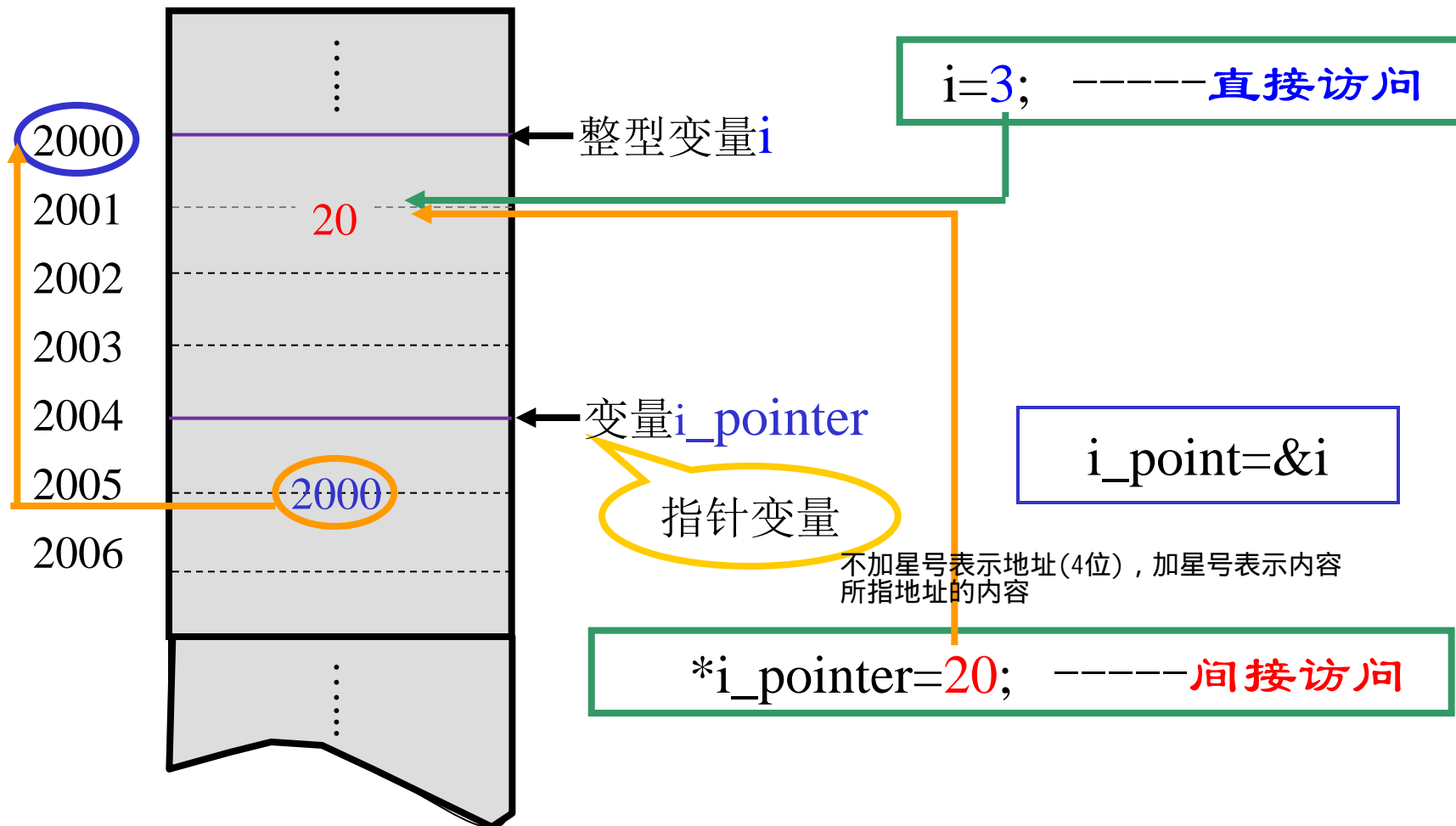


`i_point=2000` × **i=15**

– 直接访问与间接访问

- 直接访问：按变量名存取变量值
- 间接访问：通过存放变量地址的变量去访问值

```
int i;  
int *i_point=&i;
```

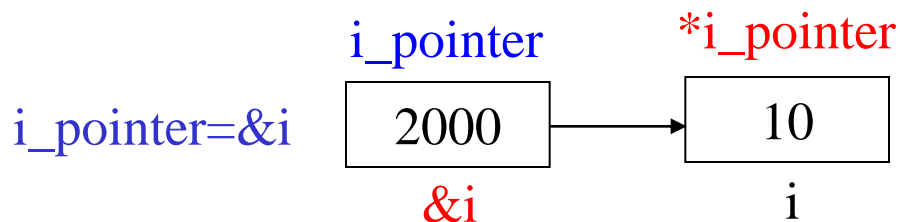


- &与*运算符

含义: 取变量的地址
单目运算符

含义: 取指针所指向变量的内容
单目运算符

- &与*两者关系: 互为逆运算



$i_pointer = \&i = \&(*i_pointer)$

$i = *i_pointer = *(\&i)$

$i_pointer$ -----指针变量, 它的内容是地址量

$*i_pointer$ -----指针所指的**目标变量**, 它的内容是值

$\&i_pointer$ ---指针变量占用内存的地址

“&” 和 “*” 运算符的级别

如果已执行了语句 `pointer_1 = & a ;` ;

- **`&* pointer_1`** 的含义是什么？

“&” 和 “*” 两个运算符的优先级别相同，但按自右而左方向结合。因此，`&* pointer_1` 与 `& a` 相同，即变量a的地址。

如果有 `pointer_2 = &* pointer_1 ;` 它的作用是将 `& a`（a 的地址）赋给 `pointer_2`，如果 `pointer_2` 原来指向 b，经过重新赋值后它已不再指向 b 了，而指向了 a。

进一步实例说明

运行结果:

a:10

*pa:10

&a:f86(hex)

pa:f8a(hex)

&pa:f8a(hex)

```
int main()
{  int a;
   int *pa=&a;
   a=10;
   cout<<"a:"<<a<<endl;
   cout<<"*pa:"<<*pa<<endl;
   cout<<"&a:"<<&a <<"(hex)"<<endl;
   cout<<"pa:"<<pa <<"(hex)\n"<<endl;
   cout<<"&pa:"<<pa <<"(hex) "<<endl;
   return 0;
}
```

直接

间接

f86

f87

f88

f89

f8a

f8b

f8c

整型变量a

指针变量pa

10

f86

内容

- 地址与指针
 - 指针变量的使用与参数传递
- 指针与数组

指针变量能指向什么对象

例 `int i,j,*p;`
 `float f,*q;`

`p=&i;` (✓)
`q=&j;` (×)
`p=&f` (×)

指针变量的类型和指向的
变量**必须**有相同类型

— 指针变量的初始化说明

一般形式：[存储类型] 数据类型 *指针名=初始地址值；

例 int i;
 int *p=&i;

赋给指针变量，
不是赋给目标变量
也不能是整形值

例 int *p=&i;
 int i;

✗

变量必须已说明过
且类型一致

例 int main()
 { int i;
 static int *p=&i;

 } (✗)

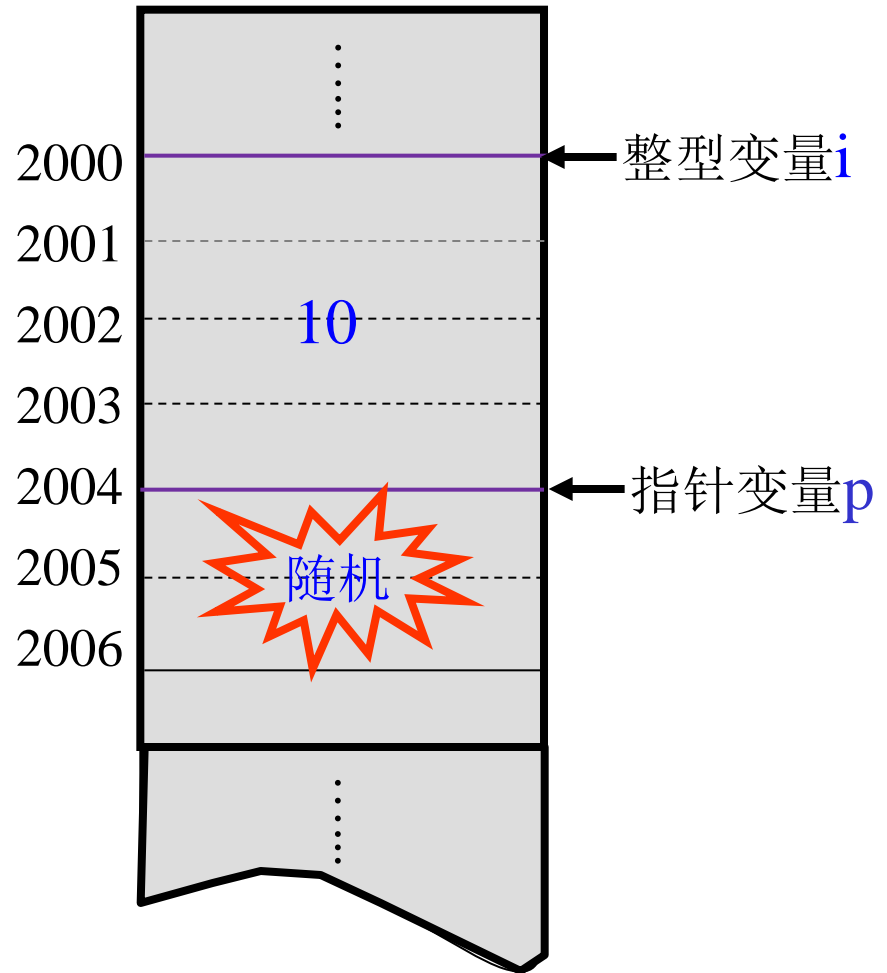
不能用auto变量的地址去初
始化static型指针，因为动
态变量会随时释放

指针变量必须先赋值(地址), 再使用

```
例 int main( )  
{   int i=10;  
    int *p;  
    *p=i; ✗  
    cout<<*p<<endl;  
    return 0;  
}
```

地址! 危险!

```
例 int main( )  
{   int i=10,k;  
    int *p;  
    p=&k;  
    *p=i; ✓  
    cout<<*p<<k;  
    return 0;  
}
```



k 的输出值是多少

零指针与空类型指针

- 零指针：(空指针)

- 定义:指针变量值为零
- 表示: `int *p=0;`

```
#define NULL 0
int *p=NULL;
```

- `p=NULL`与未对`p`赋值不同
- 用途:
 - » 避免指针变量的非法引用
 - » 在程序中常作为状态比较

- `void *`类型指针

- 表示: `void *p;`
- 使用时要进行强制类型转换

`p`指向地址为0的单元，
系统保证该单元不作它用
表示指针变量值没有意义

```
例      int *p;
        .....
        while(p!=NULL)
        {   .....
        }
```

```
例      char *p1;
        void *p2;
        p1=(char *)p2;
        p2=(void *)p1;
```

表示不指定`p`是指向哪一种
类型数据的指针变量

void类型指针的进一步说明

```
void vobject; //错，不能声明void类型的变量
void *pv;      //对，可以声明void类型的指针
int *pint; int i;
void main()    //void类型的函数没有返回值
{
    pv = &i;    //void类型指针可以指向具体类型变量
    // void指针赋值给int指针需要类型强制转换:
    pint = (int *)pv;
}
```

void可用于为不同类型的变量动态申请空间（后续会介绍）

指针变量的关系运算

- **关系运算**

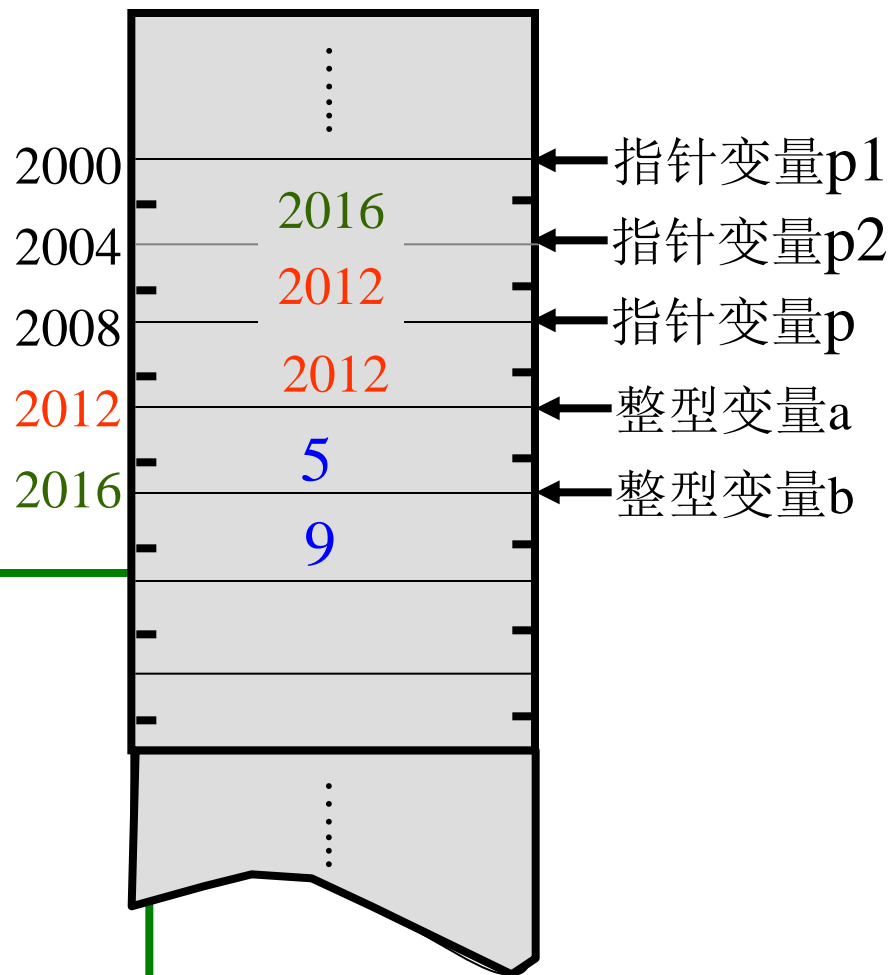
- 指向相同类型数据的指针之间可以进行各种关系运算。
- 指向不同数据类型的指针，以及指针与一般整数变量之间的关系运算是无意义的。
- 指针可以和零之间进行等于或不等于的关系运算。
例如： $p==0$ 或 $p!=0$

- **赋值运算**

- 向指针变量赋的值必须是地址常量或变量，不能是普通整数。但可以赋值为整数0，表示空指针。

程序的功能是什么？

```
int main()
{  int *p1,*p2,*p,a,b;
  cin>>a>>b;
  p1=&a; p2=&b;
  if(a<b)
  { p=p1; p1=p2; p2=p;}
  cout<<"a="<<a<<" a="<<b<<endl;
  cout<<"max="<<*p1<<" min="<<*p2;
  return 0;
}
```



运行结果： a=5,b=9
max=9,min=5

指针变量作为函数参数——地址传递

- 什么时候传值，传值的特点；

```
void swap(int x,int y)
```

```
{  int temp;  
    temp=x;  
    x=y;  
    y=temp;
```

```
}
```

```
int main()
```

```
{  int a,b;  
    cin>>a>>b;  
    if(a<b) swap(a,b);  
    cout<<a<<b<<endl;  
    return 0;
```

```
}
```



输入整数： 5, 9

运行结果： 5, 9

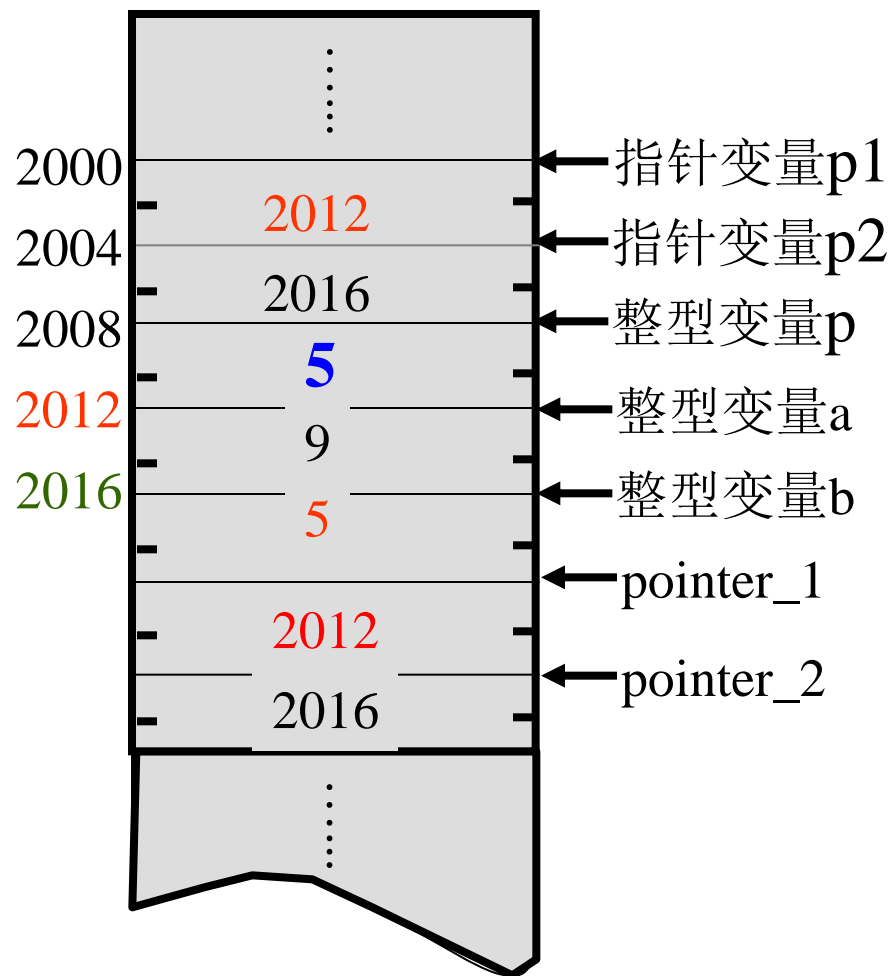
• 传地址的特点;

```
void swap(int *p1, int *p2)
```

```
{  int p;  
    p=*p1;  
    *p1=*p2;  
    *p2=p;  
}
```

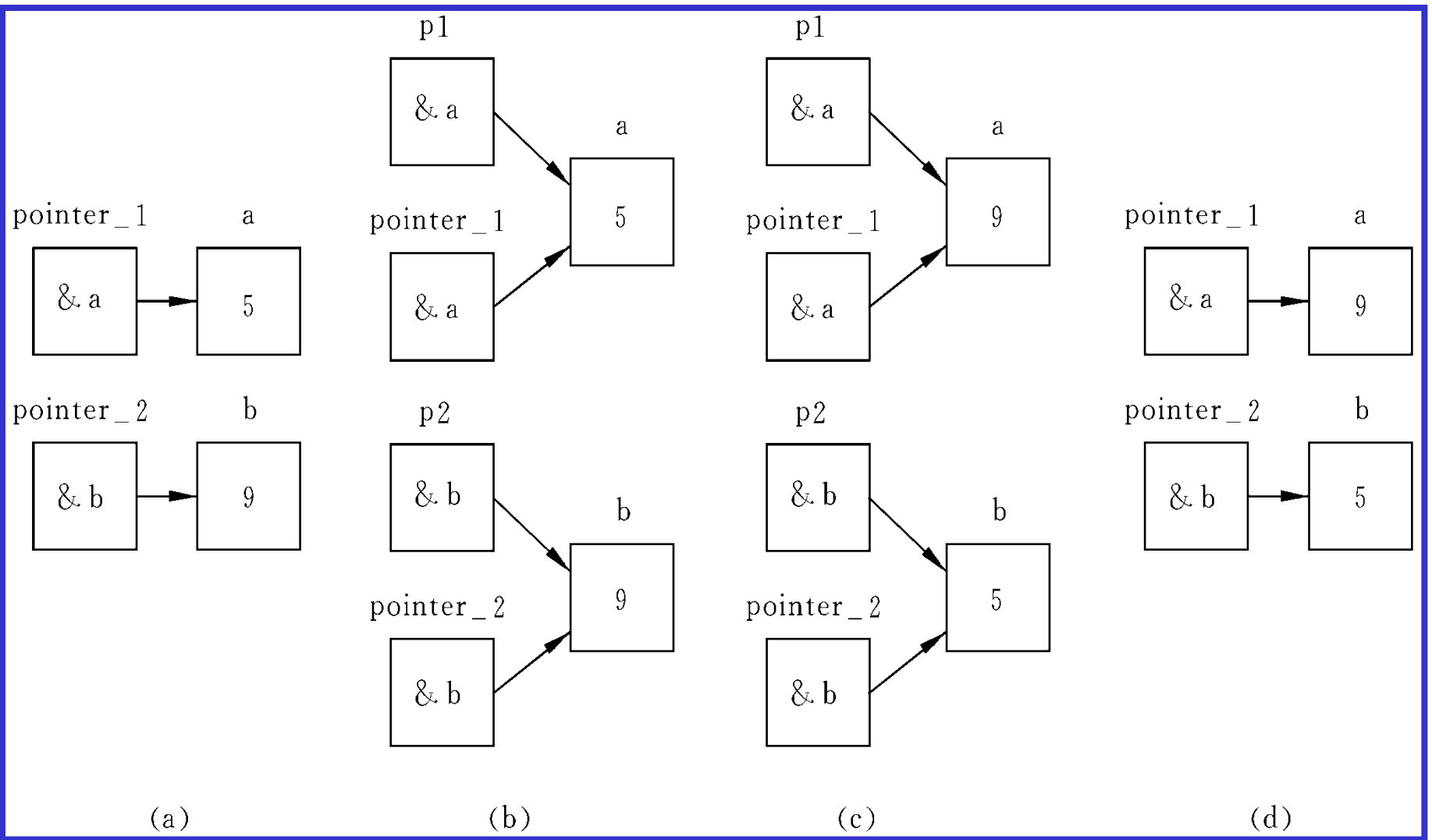
```
int main()
```

```
{  int a,b;  
    int *pointer_1,*pointer_2;  
    cin>>a>>b;  
    pointer_1=&a; pointer_2=&b;  
    if(a<b) swap(pointer_1,pointer_2);  
    cout<<a<<b<<endl;  
    return 0;  
}
```



输入整数: 5, 9

运行结果: ? ?

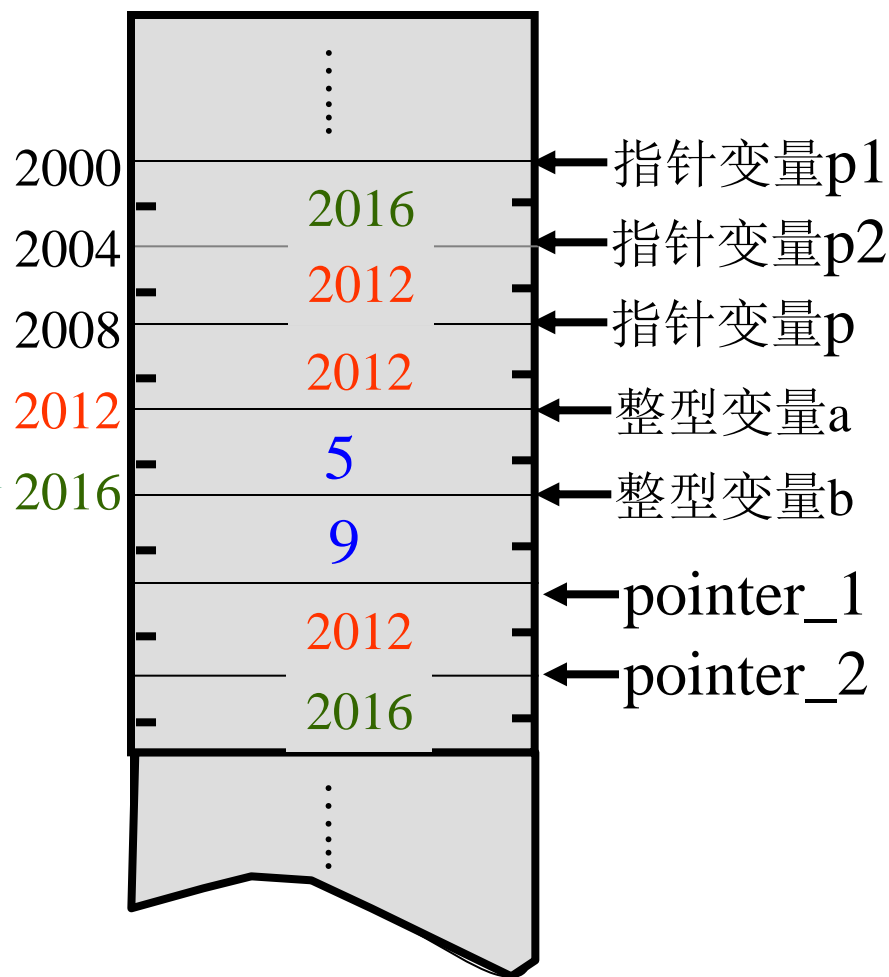


```
void swap(int *p1, int *p2)
```

```
{ int *p;  
  p=p1;  
  p1=p2;  
  p2=p;  
}
```

```
int main()
```

```
{ int a,b;  
  int *pointer_1,*pointer_2;  
  cin>>a>>b;  
  pointer_1=&a; pointer_2=&b;  
  if(a<b) swap(pointer_1,pointer_2);  
  cout<<*pointer_1<<*pointer_2;  
  return 0;  
}
```



输入整数： 5, 9

运行结果： 5, 9

为什么？

返回指针值的函数 — 返回地址

例：返回两个整数
中较大数的地址

注意：不要返回当前函数的
局部变量地址，局部变量会
在函数调用结束时释放

再例：value 为局部变量

```
int *getint(char *str)
{
    int value=strlen(str);
    cout<<str;
    return &value;
}
```

```
int * max(int *m, int *n)
{
    int *p=m;
    if (*m < *n)
        p=n;
    return(p);
}

int main()
{
    int m,n;
    int *p;
    cin>>m>>n;
    p=max(&m,&n);
    cout<<"max("<<m<<n<<") "<<*p);
    return 0;
}
```

返回指针值的函数 — 另一例子

```
#include<iostream>
using namespace std;
int main(){
    int *p,*q;
    p = getInt1();
    q = getInt2();
    cout << *p << endl;
    return 0;
}
```

```
int *getInt1()
{
    int value1 = 20;
    return &value1;
}
int *getInt2()
{
    int value2 = 30;
    return &value2;
}
```

输出结果可能是：

30

Press any key to continue

函数 `getInt1` 被调用后，局部变量 `value1` 的内存被释放，恰好分配给 `getInt2` 的 `value2` 使用，这样，`main` 中 `p` 所指的地方的值隐含这发生了变化。

返回地址需特别谨慎！

函数返回指针值——可返回全局变量的地址

- 返回全局变量的地址，而非局部变量的地址，可确保返回地址有意义

```
#include<iostream.h>
int value1 = 20;
int value2 = 30;
int main()
{   int *p,*q;
    p = getInt1();
    q = getInt2();
    cout << *p << endl;
    return 0;
}
```

```
int *getInt1()
{
    value1-=10;
    return &value1;
}

int *getInt2()
{
    value2*=2
    return &value2;
}
```

输出结果是：

10

Press any key to continue

函数返回指针值——可返回静态局部变量地址

- 返回静态局部变量的地址（静态局部变量不被释放），而非动态局部变量的地址，也可以确保返回地址有意义

```
#include<iostream>
using namespace std;
int main(){
    int *p,*q;
    p = getInt1();
    q = getInt2();
    cout << *p << endl;
    return 0;
}
```

```
int *getInt1()
{
    static int value1 = 10;
    return &value1;
}
int *getInt2()
{
    static int value2 = 30;
    return &value2;
}
```

输出结果是：

10

Press any key to continue

内容

- 地址与指针
 - 指针变量的使用与参数传递
 - 指针与数组

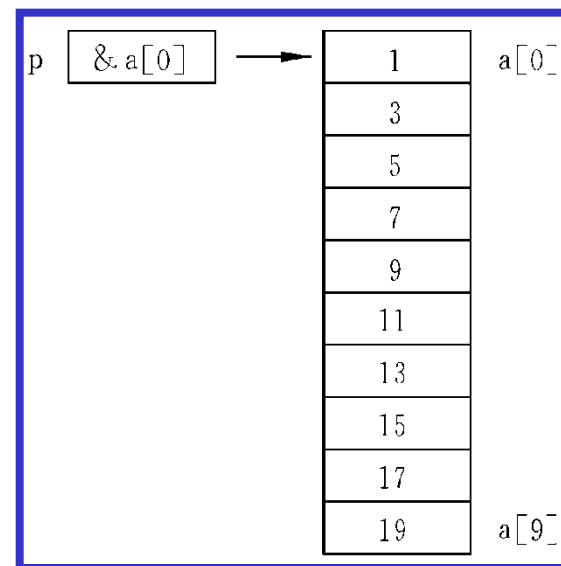
指针与数组

- 指向数组元素的指针变量与前面介绍的指向变量的指针变量相同。

例如：

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

- `p=&a[0];` // 或者 `p=a;`**
`a[0]` 元素的地址也是数组 `a` 的首地址
`&a[0]` 和 `a` 一样，均表示 `a` 的首地址



引用第 `i` 个数组元素，可以用：

- (1) 下标法，可用 **`a[i]` 或 `p[i]`**;
 - (2) 指针法，可用 **`*(a+i)` 或 `*(p+i)`**
- 如果 **`p= &a[3]`**, **`*(p+i)`** 表示什么？

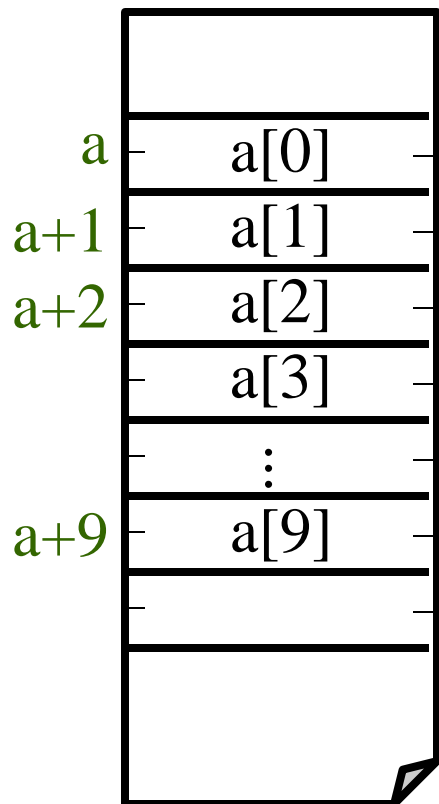
```
for(i=0; i<10; ++i)
    cout<<a[i] <<" "<<p[i]<<endl;
for(i=0; i<10; ++i)
    cout<<*(a+i) <<" "<<*(p+i)<<endl;
```

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

[] 变地址运算符
 $a[i] \Leftrightarrow *(a+i)$

低级程序设计语言
(汇编语言) 中的
变址运算

地址

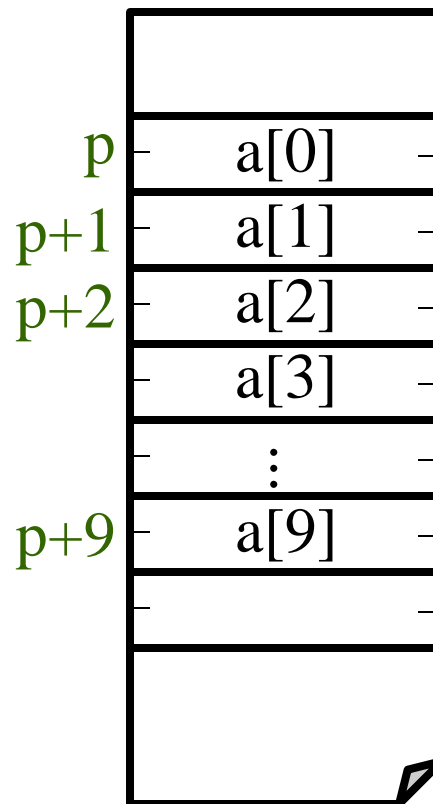


下标法

内容

$a[0]$ $*a$
 $a[1]$ $*(a+1)$
 $a[2]$ $*(a+2)$
 $a[3]$
 $:$
 $a[9]$ $*(a+9)$

地址



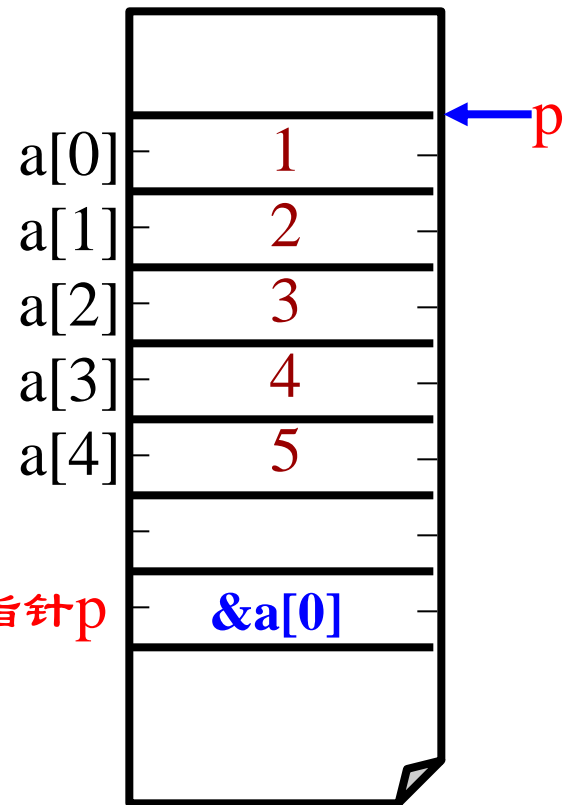
内容

p $*p$
 $p+1$ $*(p+1)$
 $p+2$ $*(p+2)$
 $a[3]$
 $:$
 $p+9$ $*(p+9)$

指针法

结果是什么？

```
int main()
{   int a[5],*p,i;
    for(i=0;i<5;i++)
        a[i]=i+1;
    p=&a[0];    //⇔ p=a;
    for(i=0;i<5;i++)
        cout<<*(p+i)<<endl;
    for(i=0;i<5;i++)
        cout<<*(a+i)<<endl;
    for(i=0;i<5;i++)
        cout<<p[i]<<endl;
    for(i=0;i<5;i++)
        cout<<a[i]<<endl;
    return 0;
}
```



数组与指针的异同

- 不同点：

- 数组名可以看成地址常量，指针则是地址变量；
因此，指针可以被赋值，进行增1（减1）运算
（如`++p`；`p++`；`--p`），但要小心越界。

- 相同点：

- `a[k]` 和 `p[k]` 都表示起始地址后面的第 k 个元素的内容

- `p[k]` 等价于 `*(p+k)`，同样，`a[k]` 也等价于 `*(a+k)`，`[]` 变址后的内容。

例 `int a[]={1,2,3,4,5,6,7,8,9,10},*p=a,i;`

数组元素地址的正确表示:

(A) `&(a+1)` (B) `a++` (C) `&p[i]` ✓

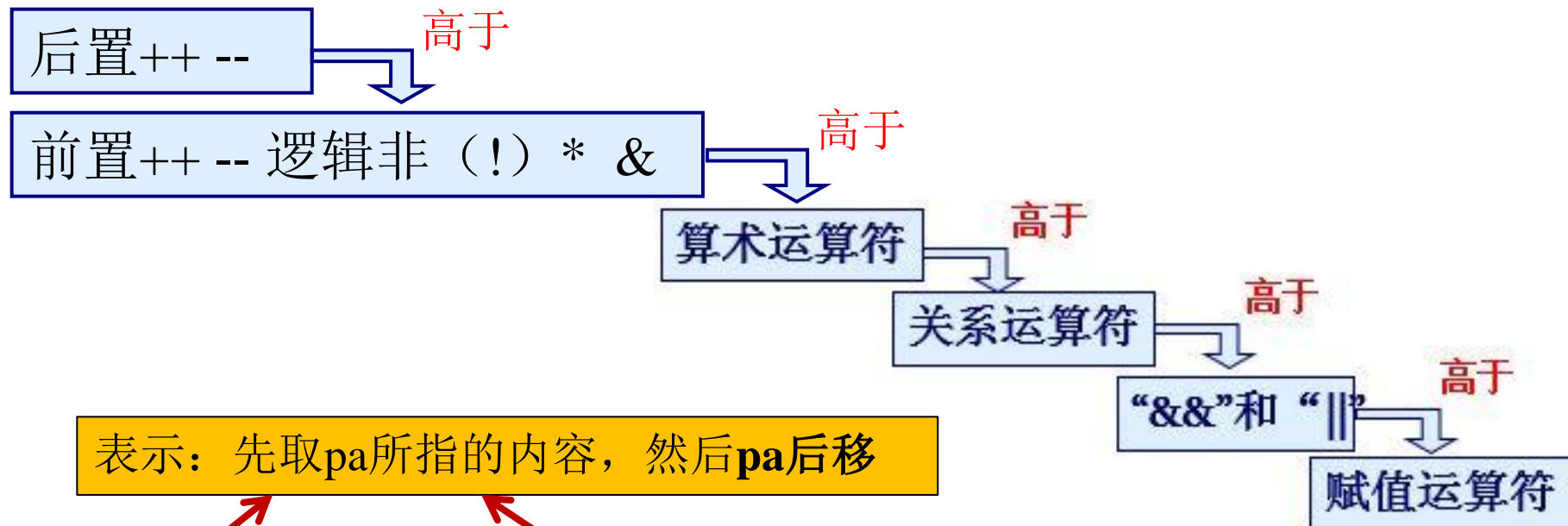
数组名是地址常量

`p++`, `p--` (✓)

`a++`, `a--` (✗)

`a+1`, `*(a+2)` (✓)

指针所指地址与内容



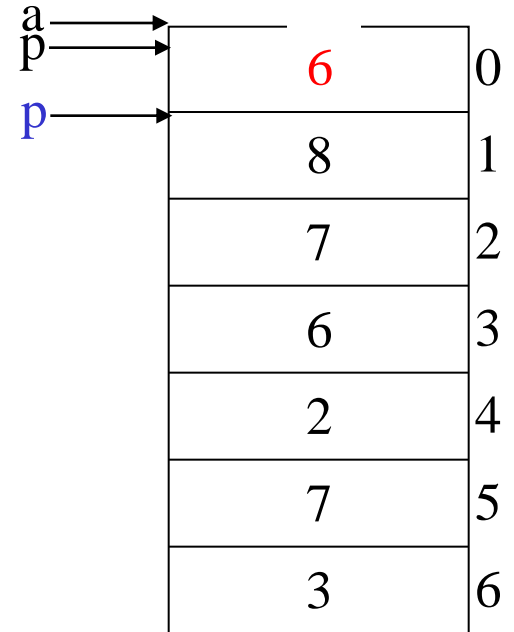
- $*pa++$ 等价于 $*(pa++)$ —— **同级运算右结合优先**，多用 $*(pa++)$ 表示，以便清晰；
- $*(pa++)$ 与 $*(++pa)$ 的效果不同 (? ?) ；
- $(*pa)++$ 表示 pa 所指地址单元的内容相加；

先取所指内容，然后内容增1

表示：pa先后移，再取移后所指的内容

一个有趣的结果

```
例  int main()
    {   int  a[]={5,8,7,6,2,7,3};
        int y,*p=&a[1];
        y=(*--p)++;
        cout<<y<<" " <<a[0]<<endl;
        return 0;
    }
```



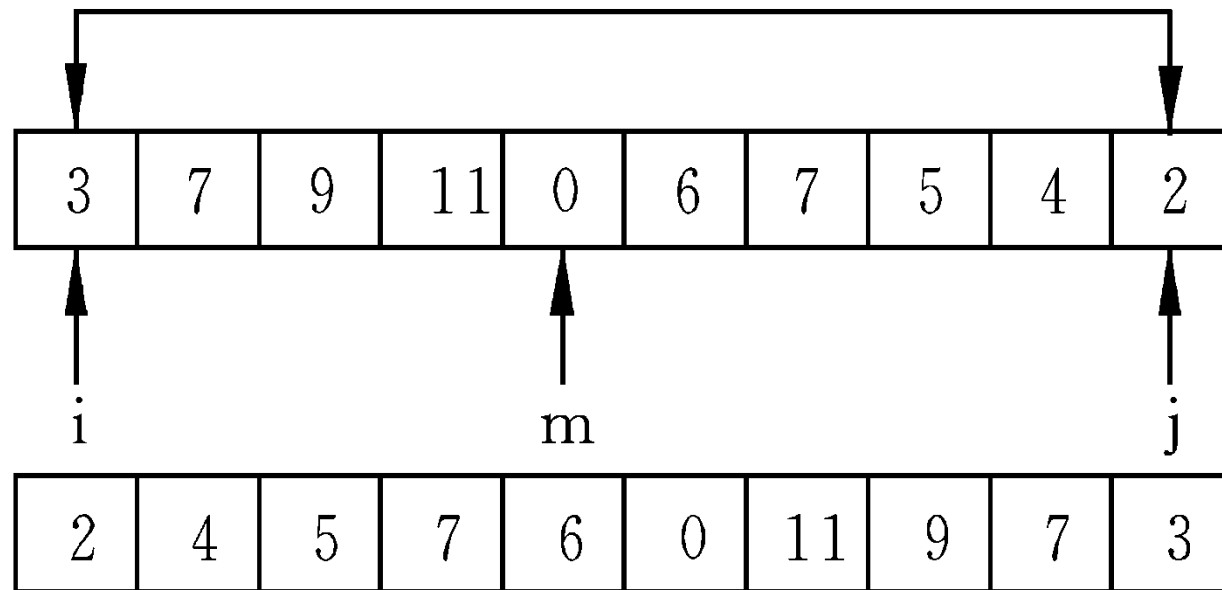
输出： 5 6

指针与数组的关系

- 数组名作函数参数，是地址传递
- 数组名作函数参数，实参与形参的对应关系

实参	形参
数组名	数组名
数组名	指针变量
指针变量	数组名
指针变量	指针变量

例子 将数组中的整数反序



```
void inv(int x[], int n)
{
    int t, i, j;
    for(i=0, j=n-1; i<j; i++,j--)
    {
        t=x[i]; x[i]=x[j]; x[j]=t;
    }
}

int main()
{
    int i,a[10]={3,7,9,11,0,6,7,5,4,2};
    inv(a,10);
    cout<<"The array has been reverted:"<<endl;
    for(i=0;i<10;i++)
        cout<<a[i];
    cout<<endl;
    return 0;
}
```

实参与形参均用数组，因为数组可带回值

```
void inv(int *x, int n)
```

```
{  int t,*i,*j;  
    for(i=x, j=x+n-1; i<j;i++,j--)  
        {  t=*i; *i=*j; *j=t;  }  
}
```

地址

地址比较大小

```
int main()
```

```
{  int i,a[10]={3,7,9,11,0,6,7,5,4,2};
```

```
    inv(a,10);
```

```
    cout<<"The array has been reverted:"<<endl;
```

```
    for(i=0;i<10;i++)
```

```
        cout<<a[i];
```

```
    cout<<endl;
```

```
    return 0;
```

```
}
```

地址移动

实参用数组,形参用指针变量,效果等价
传地址

```
void inv(int *x, int n)
```

```
{  
    for(i=x, j=x+n-1; i<j; i++, j--)  
        { t=*i; *i=*j; *j=t; }  
}
```

```
main()
```

```
{  int i,a[10],*p=a;  
    for(i=0; i<10; i++, p++)
```

为使么再次赋值

```
        cin>>*p;
```

```
        p=a;    inv(p,10);
```

```
    cout<<"The array has been reverted:"<<endl;
```

```
    for(p=a; p<a+10; p++)
```

```
        cout<<*p;
```

```
    return 0;
```

```
}
```

实参与形参均用指针变量

```
void inv(int x[], int n)
```

```
{  int t,i,j;  
    for(i=0,j=n-1; i<j; i++,j--)  
    {  
        t=x[i]; x[i]=x[j]; x[j]=t;  
    }  
}
```

```
main()
```

```
{  int i,a[10],*p=a;  
    for(i=0;i<10;i++,p++)  
        scanf("%d",p);  
    p=a;    inv(p,10);  
    cout<<"The array has been reverted:"<<endl;  
    for(p=a;p<a+10;p++)  
        cout<<*p;  
    return 0;  
}
```

实参用指针变量,形参用数组

地址比较

地址移动

字符串（数组）的不同之处

```
int main()
{
    int a = 5;
    int *pa = &a;

    int b[6] = {1, 2, 3, 4, 5, 6};
    int *pb = b;

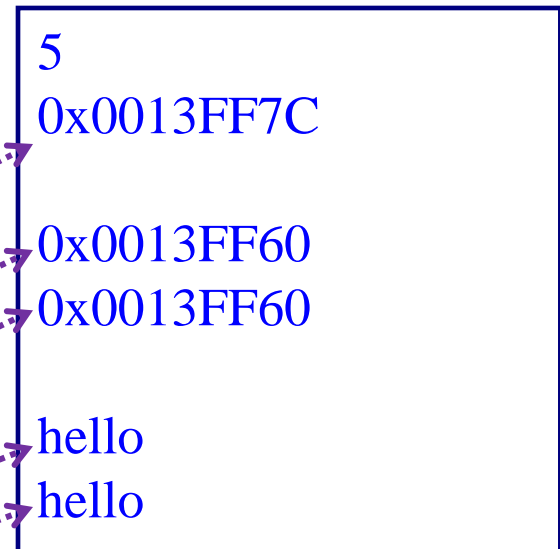
    char c[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
    char *pc = c;

    cout<< a <<endl;
    cout<< pa <<endl<<endl;

    cout<< b <<endl;
    cout<< pb <<endl<<endl;

    cout<< c <<endl;
    cout<< pc <<endl;

    return 0;
}
```



字符指针的特别之处

int *i_point=10;
正确吗?

字符指针**初始化**:把字符串**首地址**赋给string

⇔ char *string;

string="I love China!"; //指针可以这样赋值

例 main()

```
{ char *string="I love China!";
```

```
  cout<<string;
```

```
  string+=7;
```

```
  while(*string)
```

```
  {   cout<<string[0];
```

```
      string++;
```

```
  }
```

```
}
```

string

I
l
o
v
e
C
h
i
n
a
!
\0

*string!=0

结果是??

字符串与字符数组的比较

对字符指针变量赋初值：

```
char * a = " I love China! "; 等价于
```

```
char  * a ;
```

```
a = "I love Chian! ";
```

指针更灵活

而对数组的初始化：

```
char str[14]= { " I love China! " } ;
```

不能等价于

```
char str[14];
```

```
str[14]="I love China! "; // 不正确写法
```

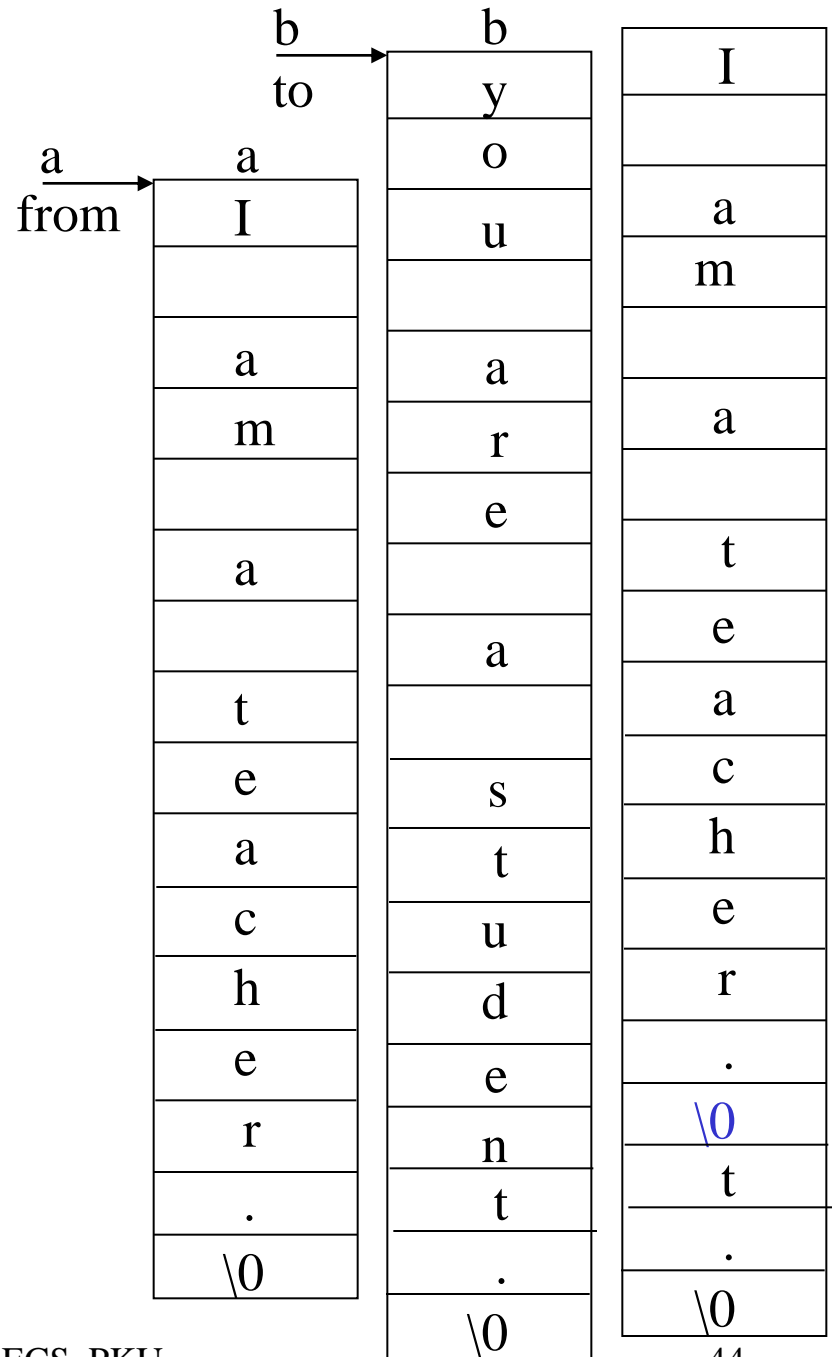
```
str= "I love China! "; // 不正确写法，如何写？
```

字符串指针作函数参数

```
void copy_string(char *from, char *to)
{ for(; *from != '\0'; from++, to++)
  *to = *from;
  *to = '\0';
}

main()
{ char *a = "I am a teacher.";
  char *b = "You are a student.";
  cout << "string_a=" << a << endl;
  cout << "string_b=" << b << endl;
  copy_string(a, b);
  cout << "string_a=" << a << endl;
  cout << "string_b=" << b << endl;
}
```

结果？



比较：字符串（数组）作函数参数

```
void copy_string(char from[],char to[])
```

```
{  int i=0;
    while(from[i]!='\0')
    {  to[i]=from[i];
        i++;
    }
    to[i]='\0';
}
```

```
main()
```

```
{  char a[]="I am a teacher.";
    char b[]="You are a student.";
    cout<<"string_a="<<a<<endl;
    cout<<"string_b="<<b<<endl;
    copy_string(a,b);
    cout<<"string_a="<<a<<endl;
    cout<<"string_b="<<b<<endl;
}
```

比较：字符指针变量与字符数组

- 赋值

`char *cp;` 与 `char str[20];`

– 字符数组只有在初始化时可以赋值：

`char str[20]= "I love China! ";`

– 字符指针可以在任何地方赋值：

`char *cp= "I love China! ";`

`char *cp;`

`cp= "I love China! ";`

比较：字符指针变量与字符数组

- 字符串用一维字符数组存放
- 字符数组具有一维数组的所有特点
 - » 数组名是指向数组首地址的**地址常量**
 - » 数组元素的引用方法可用指针法和下标法
 - » 数组名作函数参数是地址传递等

char str[]={“Hello!”};	(✓)
char str[]=“Hello!”;	(✓)
char str[]={‘H’,‘e’,‘l’,‘l’,‘o’,‘!’};	(✓)
char *cp=“Hello”;	(✓)
int a[]={1,2,3,4,5};	(✓)
int *p={1,2,3,4,5};	(✗)
char *cpoint=‘H’,‘e’}	(✗)

char str[10],*cp;	
int a[10],*p;	
str=“Hello”;	(✗)
cp=“Hello!”;	(✓)
a={1,2,3,4,5};	(✗)
p={1,2,3,4,5};	(✗)

指针数组简介

- 指针数组的特点：
 - 多个元素构成数组
 - 数组中的每个元素为指针变量
 - 常用于处理多个一维数组和多个字符串

定义形式：[**存储类型**] **数据类型** ***数组名** [**数组长度说明**];

例 int *p[4];

指针所指向变量的数据类型

int *p[4]表示有 4 个指针指向整型变量

指针数组的初始化和赋值

赋值:

```
main()
```

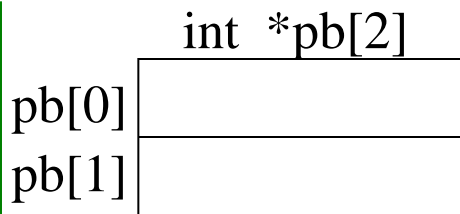
```
{ int b[2][3], *pb[2];
```

```
  pb[0]=b[0];
```

```
  pb[1]=b[1];
```

```
  .....
```

```
}
```



`int b[2][3]`

1
2
3
2
4
6

也是地址

初始化:

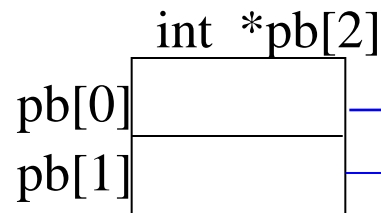
```
main()
```

```
{ int b[2][3], *pb[ ]={b[0],b[1]};
```

```
  .....
```

```
}
```

隐含为2



`int b[2][3]`

1
2
3
2
4
6

指针数组的初始化和赋值

初始化:

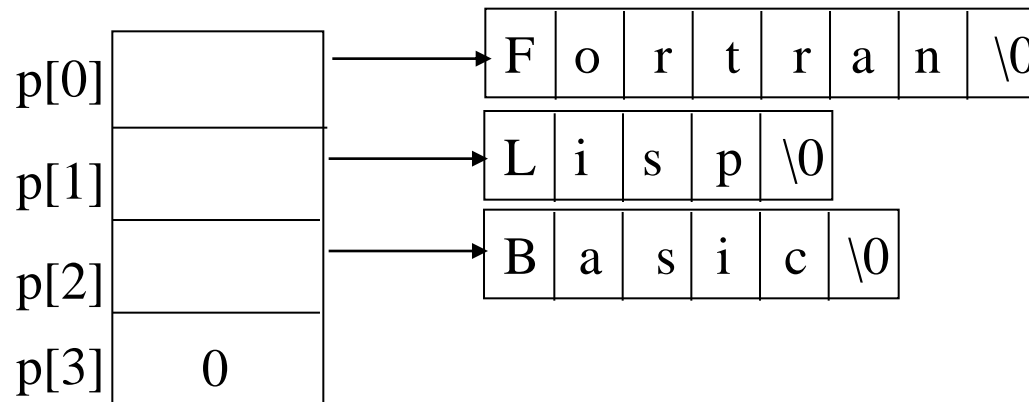
main()

```
{ char a[]="Fortran";  
  char b[]="Lisp";  
  char c[]="Basic";  
  char *p[4]={a,b,c,NULL};  
  .....  
}
```

赋值:

main()

```
{ char *p[4];  
  p[0]= "Fortran";  
  p[1]= "Lisp";  
  p[2]= "Basic";  
  p[3]=NULL;  
  .....  
}
```



指针数组的初始化和赋值

另一种初始化:

```
main()
```

```
{ char *p[]={"Fortran", "Lisp", "Basic", NULL};
```

```
.....
```

```
}
```

