

## 第6章 指针

6.1 指针的概念

6.2 变量与指针

6.3 数组与指针

6.4 字符串与指针

6.5 函数与指针

6.6 返回指针值的函数

6.7 指针数组和指向指针的指针

6.8 有关指针的数据类型和指针运算的小结

\*6.9 引用

## 6.1 指针的概念

为了说清楚什么是指针，必须弄清楚数据在内存中是如何存储的，又是如何读取的。

如果在程序中定义了一个变量，在编译时就给这个变量分配内存单元。系统根据程序中定义的变量类型，分配一定长度的空间。例如，**C++**编译系统一般为整型变量分配**4**个字节，为单精度浮点型变量分配**4**个字节，为字符型变量分配**1**个字节。内存区的每一个字节有一个编号，这图**6.1**就是“地址”。

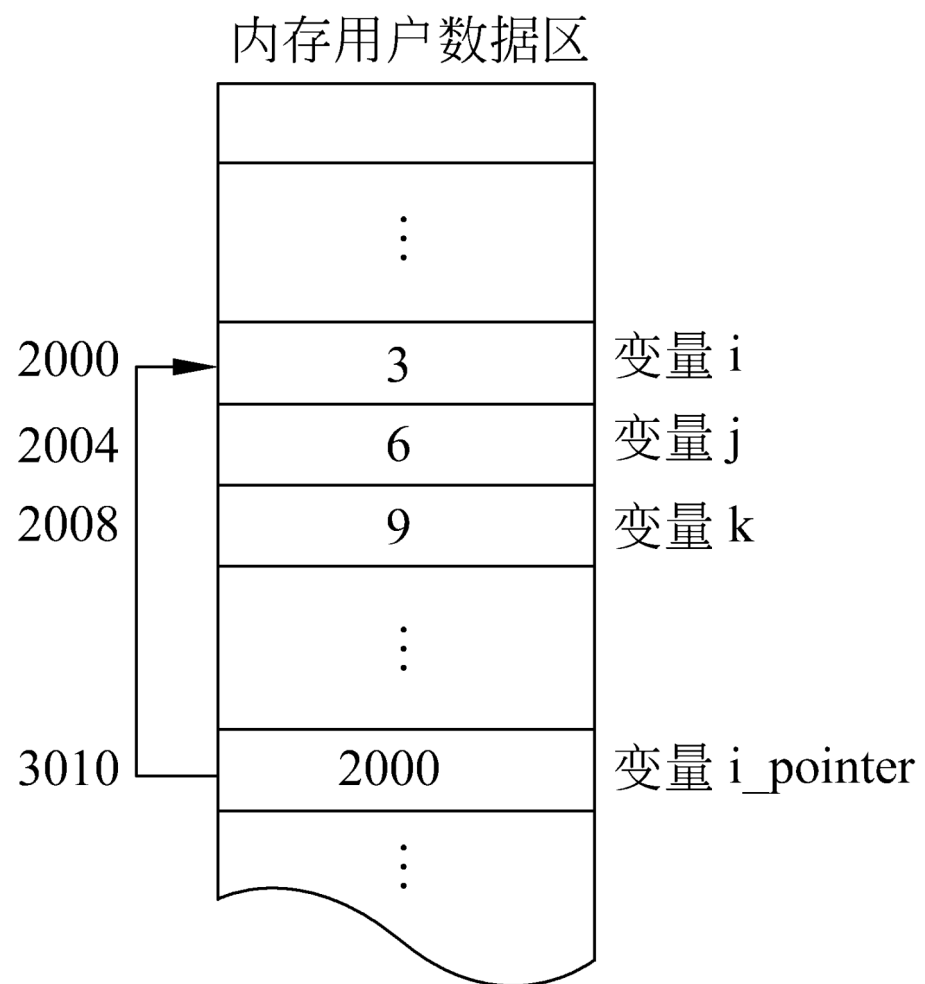


图6.1

请务必弄清楚一个内存单元的地址与内存单元的内容这两个概念的区别。在程序中一般是通过变量名来对内存单元进行存取操作的。其实程序经过编译以后已经将变量名转换为变量的地址，对变量值的存取都是通过地址进行的。这种按变量地址存取变量值的方式称为直接存取方式，或直接访问方式。

还可以采用另一种称为间接存取(间接访问)的方式。可以在程序中定义这样一种特殊的变量，它是专门用来存放地址的。

图6.2是直接访问和间接访问的示意图。为了将数值3送到变量中，可以有两种方法：

(1) 直接将数3送到整型变量 i 所标识的单元中。见图6.2(a)。

(2) 将3送到指针变量*i\_pointer*所指向的单元（这就是变量*i*所标识的单元）中。见图6.2(b)。

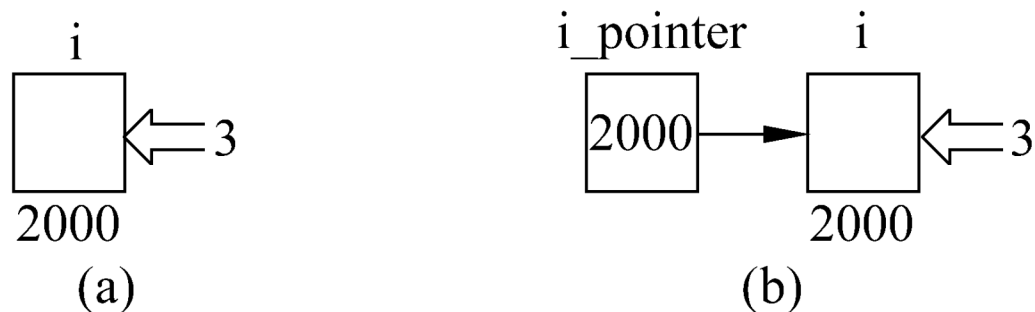


图6.2

所谓指向,就是通过地址来体现的。

由于通过地址能找到所需的变量单元，因此可以说，地址指向该变量单元。因此将地址形象化地称为“指针”。一个变量的地址称为该变量的指针。

如果有一个变量是专门用来存放另一变量地址（即指针）的，则它称为指针变量。指针变量的值（即指针变量中存放的值）是地址（即指针）。

## 6.2 变量与指针

指针变量是一种特殊的变量，它和以前学过的其他类型的变量的不同之处是：用它来指向另一个变量。为了表示指针变量和它所指向的变量之间的联系，在C++中用“\*”符号表示指向，例如，**i\_pointer**是一个指针变量，而**\*i\_pointer**表示**i\_pointer**所指向的变量，见图6.3。

下面两个语句作用相同：

① **i=3;**

② **\*i\_pointer=3;**

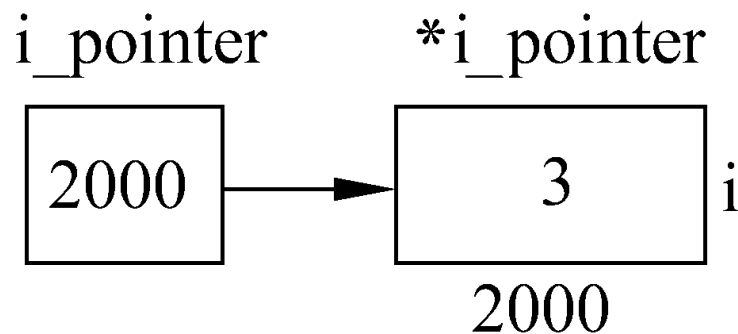


图6.3

## 6.2.1 定义指针变量

**C++**规定所有变量在使用前必须先定义，即指定其类型。在编译时按变量类型分配存储空间。对指针变量必须将它定义为指针类型。先看一个具体例子：

```
int i,j;           //定义整型变量i,j  
int *pointer_1, *pointer_2;    //定义指针变量*pointer_1,*pointer_2
```

第**2**行开头的**int**是指：所定义的指针变量是指向整型数据的指针变量。也就是说，指针变量**pointer\_1**和**pointer\_2**只能用来指向整型数据(例如**i**和**j**)，而不能指向浮点型变量**a**和**b**。这个**int**就是指针变量的基类型。指针变量的基类型用来指定该指针变量可以指向的变量的类型。

定义指针变量的一般形式为

基类型 \*指针变量名;

下面都是合法的定义:

**float \*pointer\_3;**               // **pointer\_3**是指向单精度型数据的指针变量

**char \*pointer\_4;**               // **pointer\_4**是指向字符型数据的指针变量

请注意: 指针变量名是**pointer\_3**和**pointer\_4**, 而不是**\*pointer\_3**和**\*pointer\_4**, 即“\*”不是指针变量名的一部分, 在定义变量时在变量名前加一个“\*”表示该变量是指针变量。

那么, 怎样使一个指针变量指向另一个变量呢? 只需要把被指向的变量的地址赋给指针变量即可。例如:

**pointer\_1=&i;**               //将变量**i**的地址存放到指针变量**pointer\_1**中

**pointer\_2=&j;**               //将变量**j**的地址存放到指针变量**pointer\_2**中



这样，**pointer\_1**就指向了变量**i**，**pointer\_2**就指向了变量**j**。见图6.4。

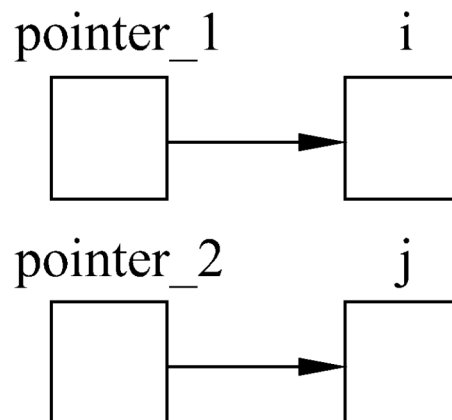


图6.4

一般的**C++**编译系统为每一个指针变量分配**4**个字节的存储单元，用来存放变量的地址。

在定义指针变量时要注意：

- (1) 不能用一个整数给一个指针变量赋初值。
- (2) 在定义指针变量时必须指定基类型。

## 6.2.2 引用指针变量

有两个与指针变量有关的运算符：

(1) **&**取地址运算符。

(2) **\***指针运算符（或称间接访问运算符）。

例如：**&a**为变量**a**的地址，**\*p**为指针变量**p**所指向的存储单元。

## 例6.1 通过指针变量访问整型变量。

```
#include <iostream>
using namespace std;
int main( )
{int a,b;                //定义整型变量a,b
int *pointer_1,*pointer_2;    //定义指针变量
*pointer_1,*pointer_2
a=100;b=10;              //对a,b赋值
pointer_1=&a;             //把变量 a 的地址赋给pointer_1
pointer_2=&b;             //把变量 a 的地址赋给pointer_2
cout<<a<<" "<<b<<endl;    //输出a和b的值
cout<<*pointer_1<<" "<<*pointer_2<<endl; //输出*pointer_1和
*pointer_2的值
return 0;
}
```

运行结果为

100 10

(a和b的值)

100 10

(\*pointer\_1和\*pointer\_2的值)

请对照图6.5分析。

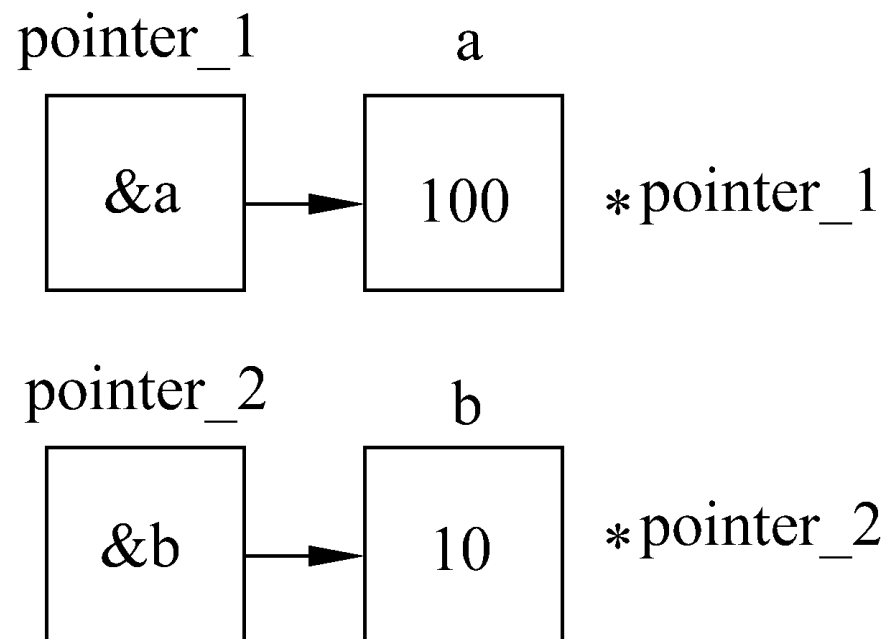


图6.5

下面对“&”和“\*”运算符再做些说明：

(1) 如果已执行了“**pointer\_1=&a;**”语句，请问 **&\*pointer\_1** 的含义是什么？“&”和“\*”两个运算符的优先级别相同，但按自右至左方向结合，因此先进行 **\*pointer\_1** 的运算，它就是变量 **a**，再执行 **&** 运算。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，而也指向了 **a**，见图 6.6。图 6.6(a) 是原来的情况，图 6.6(b) 是执行上述赋值语句后的情况。

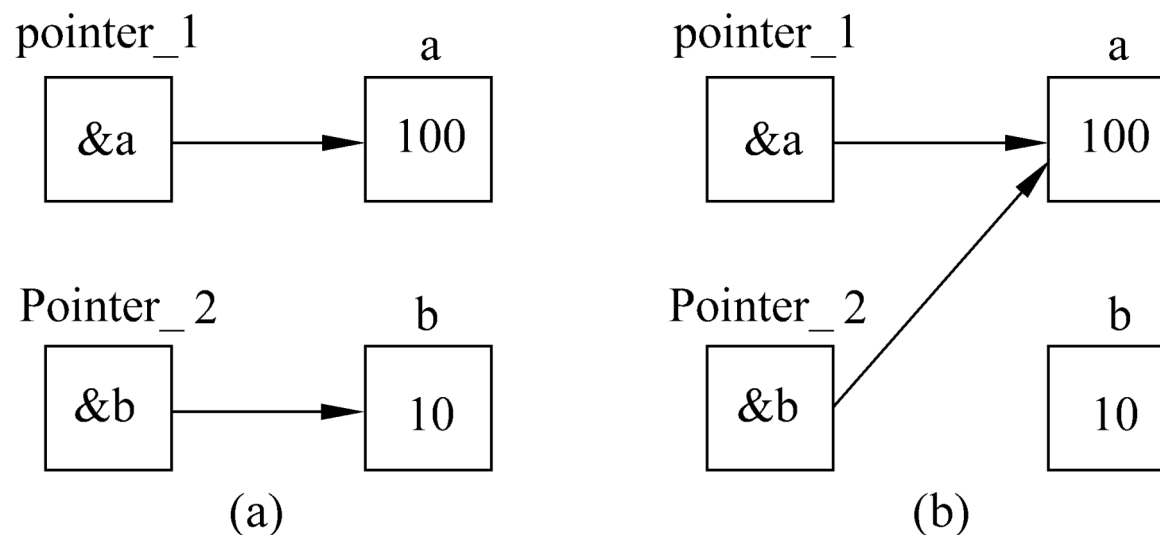


图6.6

(2) **`*&a`**的含义是什么？先进行**`&a`**的运算，得**`a`**的地址，再进行**`*`**运算，即**`&a`**所指向的变量，**`*&a`**和**`*pointer_1`**的作用是一样的（假设已执行了“**`pointer_1=&a;`**”），它们等价于变量**`a`**。即**`*&a`**与**`a`**等价，见图6.7。

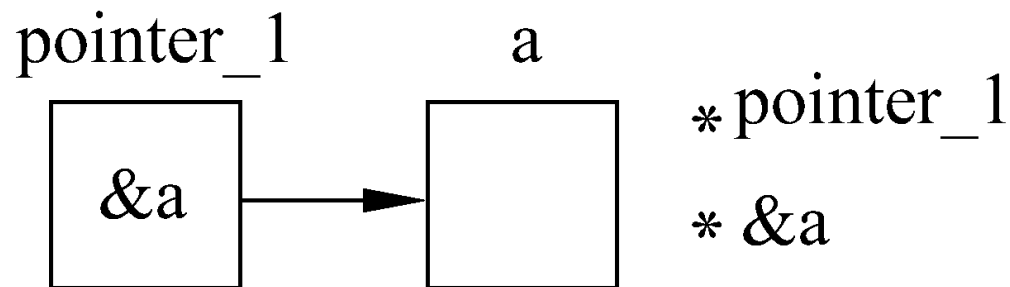


图6.7

**例6.2** 输入**a**和**b**两个整数，按先大后小的顺序输出**a**和**b**(用指针变量处理)。

解此题的思路是：设两个指针变量**p1**和**p2**，使它们分别指向**a**和**b**。使**p1**指向**a**和**b**中的大者，**p2**指向小者，顺序输出**\*p1, \*p2**就实现了按先大后小的顺序输出**a**和**b**。按此思路编写程序如下：

```
#include <iostream>
using namespace std;
int main( )
{
    int *p1,*p2,*p,a,b;
    cin>>a>>b;           //输入两个整数
    p1=&a;                 //使p1指向a
    p2=&b;                 //使p2指向b
    if(a<b)               //如果a<b就使p1与p2的值交换
    {p=p1;p1=p2;p2=p;}    //将p1的指向与p2的指向交换
    cout<<"a="<<a<<" b="<<b<<endl;
    cout<<"max="<<*p1<<" min="<<*p2<<endl;
    return 0;
}
```

运行情况如下：

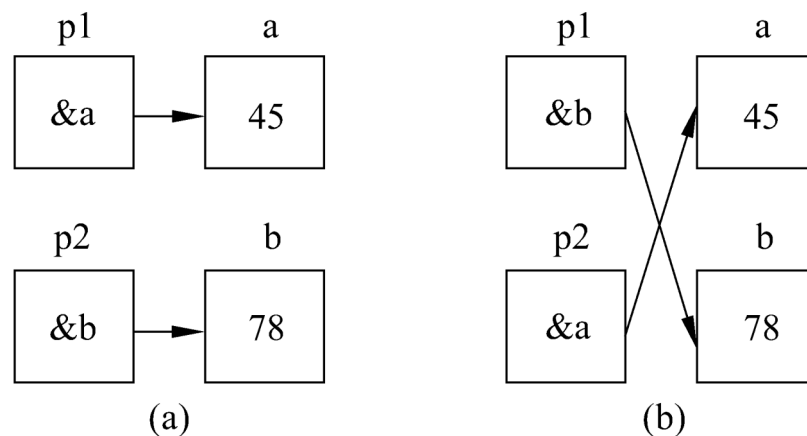


**4578✓**

**a=45 b=78**

**max=78 min=45**

输入**a**的值**45**，**b**的值**78**，由于**a<b**，将**p1**的值和**p2**的值交换，即将**p1**的指向与**p2**的指向交换。交换前的情况见图**6.8(a)**，交换后的情况见图**6.8(b)**。



图**6.8**

请注意，这个问题的算法是不交换整型变量的值，而是交换两个指针变量的值。

### 6.2.3 指针作为函数参数

函数的参数不仅可以是整型、浮点型、字符型等数据，还可以是指针类型。它的作用是将一个变量的地址传送给被调用函数的形参。

**例6.3** 题目同例**6.2**，即对输入的两个整数按大小顺序输出。

这里用函数处理，而且用指针类型的数据作函数参数。

程序如下：

```
#include <iostream>
using namespace std;
int main( )
{ void swap(int *p1,int *p2);    //函数声明
int *pointer_1,*pointer_2,a,b; //定义指针变量pointer_1,pointer_2, 整型变量a,b
cin>>a>>b;
pointer_1=&a;                    //使pointer_1指向a
pointer_2=&b;                    //使pointer_2指向b
if(a<b) swap(pointer_1,pointer_2); //如果a<b, 使*pointer_1和*pointer_2互换
cout<<"max="<<a<<" min="<<b<<endl; //a已是大数, b是小数
return 0;
}

void swap(int *p1,int *p2)        //函数的作用是将*p1的值与*p2的值交换
{ int temp;
temp=*p1;
*p1=*p2;
*p2=temp;
}
```

运行情况如下：

45 78 ✓

max=78 min=45

请注意： 不要将**main**函数中的**swap**函数调用写成  
**if(a<b) swap(\*pointer\_1,\*pointer\_2);**

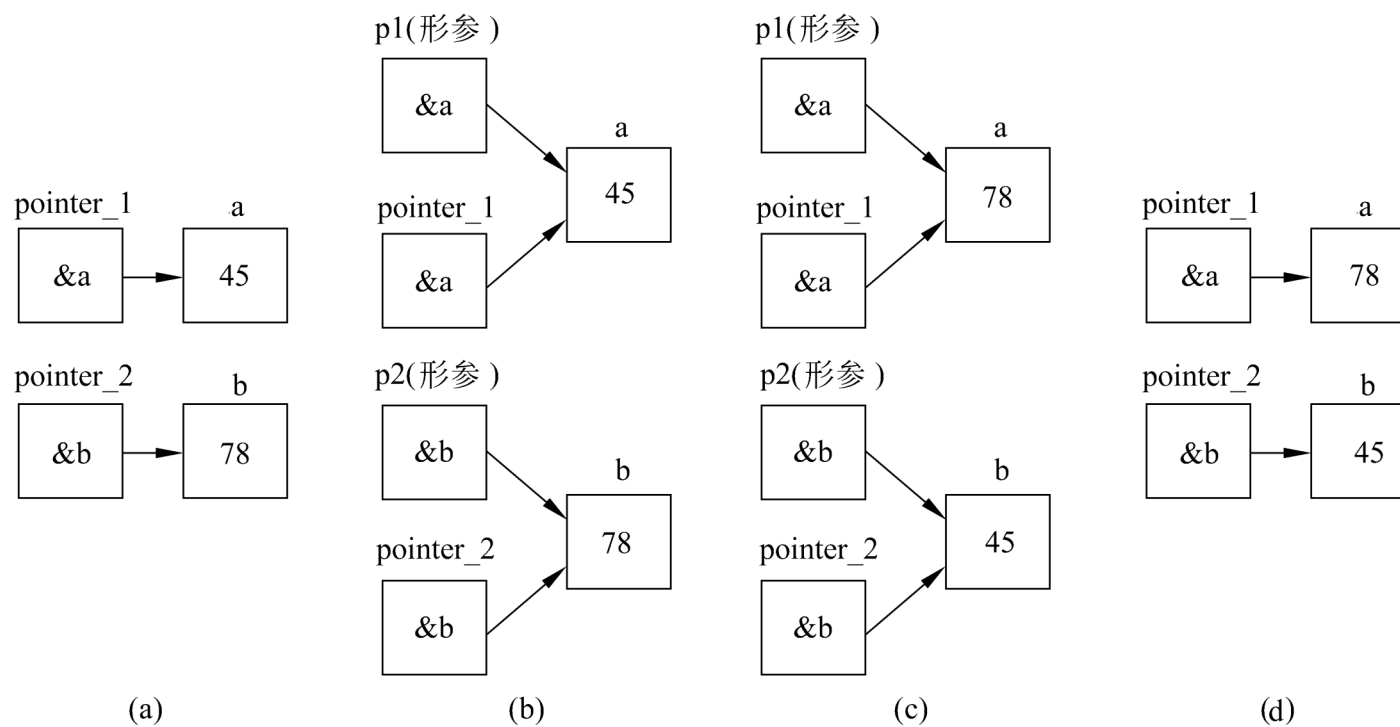


图6.9

请注意交换**\*p1**和**\*p2**的值是如何实现的。如果写成以下这样就有问题了：

```
void swap(int *p1,int *p2)
{int *temp;
 *temp=*p1;           //此语句有问题
 *p1=*p2;
 *p2=*temp;
}
```

本例采取的方法是交换**a**和**b**的值，而**p1**和**p2**的值不变。这恰和例**6.2**相反。

可以看到，在执行**swap**函数后，主函数中的变量**a**和**b**的值改变了。这个改变不是通过将形参值传回实参来实现的。请读者考虑一下能否通过调用下面的函数实现**a**和**b**互换。

```
void swap (int x,int y)
{int temp;
temp=x;
x=y;
y=temp;
}
```

在**main**函数中用“**swap(a,b);**”调用**swap**函数，会有什么结果呢？在函数调用时，**a**的值传送给**x**，**b**的值传送给**y**，如图**6.10(a)**所示。执行完**swap**函数最后一个语句后，**x**和**y**的值是互换了的，但**main**函数中的**a**和**b**并未互换，如图**6.10(b)**所示。也就是说由于虚实结合是采取单向的“值传递”方式，只能从实参向形参传数据，形参值的改变无法回传给实参。

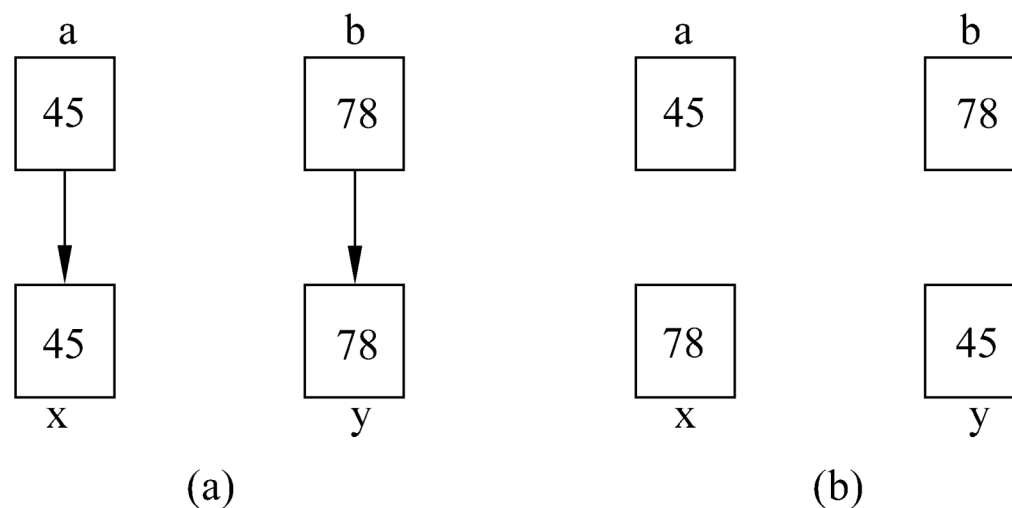


图6.10

为了使在函数中改变了的变量值能被**main**函数所用，不能采取把要改变值的变量作为参数的办法，而应该用指针变量作为函数参数。在函数执行过程中使指针变量所指向的变量值发生变化，函数调用结束后，这些变量值的变化依然保留下来，这样就实现了“通过调用函数使变量的值发生变化，在主调函数中使用这些改变了的值”的目的。

如果想通过函数调用得到**n**个要改变的值，可以采取下面的步骤：①在主调函数中设**n**个变量，用**n**个指针变量指向它们；②编写被调用函数，其形参为**n**个指针变量，这些形参指针变量应当与主调函数中的**n**个指针变量具有相同的基类型；③在主调函数中将**n**个指针变量作实参，将它们的值(是地址值)传给所调用函数的**n**个形参指针变量，这样，形参指针变量也指向这**n**个变量；④通过形参指针变量的指向，改变该**n**个变量的值；⑤在主调函数中就可以使用这些改变了值的变量。

请注意，不能企图通过改变形参指针变量的值而使实参指针变量的值改变。请分析下面程序：



```
#include <iostream>
using namespace std;
int main( )
{ void swap(int *p1,int *p2);
  int *pointer_1,*pointer_2,a,b;
  cin>>a>>b;
  pointer_1=&a;
  pointer_2=&b;
  if(a<b) swap(pointer_1,pointer_2);
  cout<<"max="<<a<<" min="<<b<<endl;
  return 0;
}

void swap(int *p1,int *p2)
{ int *temp;
  temp=p1;
  p1=p2;
  p2=temp;
}
```

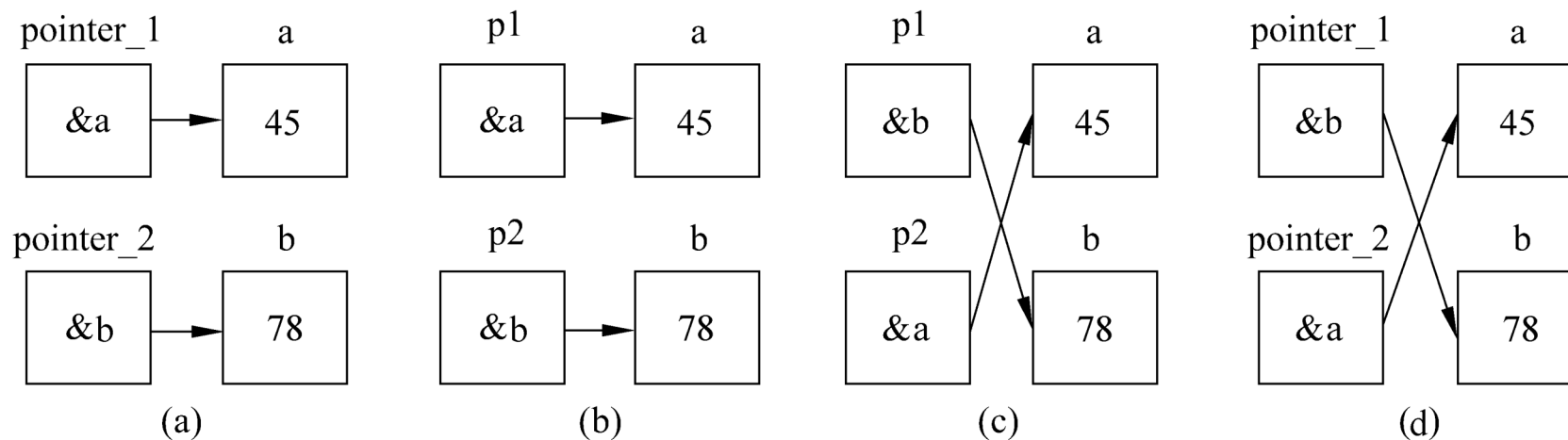


图6.11

实参变量和形参变量之间的数据传递是单向的“值传递”方式。指针变量作函数参数也要遵循这一规则。调用函数时不会改变实参指针变量的值，但可以改变实参指针变量所指向变量的值。

函数的调用可以（而且只可以）得到一个返回值（即函数值），而使用指针变量作函数参数，就可以通过指针变量改变主调函数中变量的值，相当于通过函数调用从被调用的函数中得到多个值。如果不用指针变量是难以做到这一点的。

**例6.4** 输入**a,b,c** 3个整数，按由大到小的顺序输出。用上面介绍的方法，用**3**个指针变量指向**3**个整型变量，然后用**swap**函数来实现互换**3**个整型变量的值。

程序如下：

```
#include <iostream>
using namespace std;
int main( )
{ void exchange(int *,int *,int *); //对exchange函数的声明
  int a,b,c,*p1,*p2,*p3;
  cin>>a>>b>>c;                //输入3个整数
```

```
p1=&a;p2=&b;p3=&c;           //指向3个整型变量
exchange(p1,p2,p3);          //交换p1,p2,p3指向的3个整型变量的值
cout<<a<<" "<<b<<" "<<c<<endl; //按由大到小的顺序输出3个整数
}
```

```
void exchange(int *q1,int *q2,int *q3)
{void swap(int *,int *);      //对swap函数的声明
if(*q1<*q2) swap(q1,q2);      //调用swap,将q1与q2所指向的变量的值互换
if(*q1<*q3) swap(q1,q3);      //调用swap,将q1与q3所指向的变量的值互换
if(*q2<*q3) swap(q2,q3);      //调用swap,将q2与q3所指向的变量的值互换
}
```

```
void swap(int *pt1,int *pt2)   //将pt1与pt2所指向的变量的值互换
{int temp;
temp=*pt1;
*pt1=*pt2;
*pt2=temp;
}
```

运行情况如下:

12 -56 87✓  
87 12 -56

## 6.3 数组与指针

### 6.3.1 指向数组元素的指针

一个变量有地址，一个数组包含若干元素，每个数组元素都在内存中占用存储单元，它们都有相应的地址。指针变量既然可以指向变量，当然也可以指向数组元素（把某一元素的地址放到一个指针变量中）。所谓数组元素的指针就是数组元素的地址。

```
int a[10];      //定义一个整型数组a，它有10个元素
```

```
int *p;        //定义一个基类型为整型的指针变量p
```

```
p=&a[0];       //将元素a[0]的地址赋给指针变量p，使p指向a[0]
```

在C++中，数组名代表数组中第一个元素(即序号为0的元素)的地址。因此，下面两个语句等价：

```
p=&a[0];
```

```
p=a;
```

在定义指针变量时可以给它赋初值:

```
int *p=&a[0];      //p的初值为a[0]的地址
```

也可以写成

```
int *p=a;          //作用与前一行相同
```

可以通过指针引用数组元素。假设**p**已定义为一个基类型为整型的指针变量，并已将一个整型数组元素的地址赋给了它，使它指向某一个数组元素。如果有以下赋值语句：

```
*p=1;              //对p当前所指向的数组元素赋予数值1
```

如果指针变量**p**已指向数组中的一个元素，则**p+1**指向同一数组中的下一个元素。

如果 $p$ 的初值为 $\&a[0]$ ，则：

(1)  $p+i$ 和 $a+i$ 就是 $a[i]$ 的地址，或者说，它们指向 $a$ 数组的第 $i$ 个元素，见图6.12。

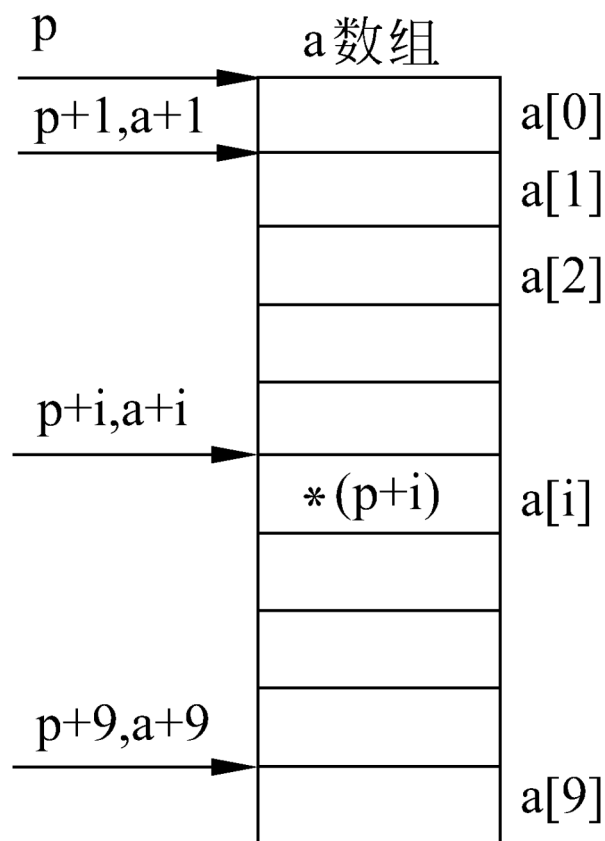


图6.12

(2)  $*(p+i)$ 或 $*(a+i)$ 是 $p+i$ 或 $a+i$ 所指向的数组元素，即 $a[i]$ 。

可以看出， $[]$ 实际上是变址运算符。对 $a[i]$ 的求解过程是：先按 $a+i \times d$ 计算数组元素的地址，然后找出此地址所指向的单元中的值。

(3) 指向数组元素的指针变量也可以带下标，如 $p[i]$ 与 $*(p+i)$ 等价。

根据以上叙述，引用一个数组元素，可用以下方法：

(1) 下标法，如 $a[i]$ 形式；

(2) 指针法，如 $*(a+i)$ 或 $*(p+i)$ 。其中 $a$ 是数组名， $p$ 是指向数组元素的指针变量。如果已使 $p$ 的值为 $a$ ，则 $*(p+i)$ 就是 $a[i]$ 。可以通过指向数组元素的指针找到所需的元素。使用指针法能使目标程序质量高。



## 例6.5 输出数组中的全部元素。

假设有一个整型数组**a**，有**10**个元素。要输出各元素的值有**3**种方法：

### (1) 下标法

```
#include <iostream>
using namespace std;
int main( )
{ int a[10];
  int i;
  for(i=0;i<10;i++)
    cin>>a[i];           //引用数组元素a[i]
  cout<<endl;
  for(i=0;i<10;i++)
    cout<<a[i]<<" ";     //引用数组元素a[i]
  cout<<endl;
  return 0;
}
```

运行情况如下：

9 8 7 6 5 4 3 2 1 0 ✓ (输入10个元素的值)

9 8 7 6 5 4 3 2 1 0 (输出10个元素的值)

## (2) 指针法

将上面程序第7行和第10行的“**a[i]**”改为“**\*(a+i)**”，运行情况与(1)相同。

## (3) 用指针变量指向数组元素

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

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

```
int i,*p=a;      //指针变量p指向数组a的首元素a[0]
```

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

```
cin>>*(p+i);    //输入a[0]~a[9]共10个元素
```

```
cout<<endl;
```

```
for(p=a;p<(a+10);p++)  
    cout<<*p<<" ";    //p先后指向a[0]~a[9]  
cout<<endl;  
return 0;  
}
```

运行情况与前相同。请仔细分析**p**值的变化和**\*p**的值。

对**3**种方法的比较：

方法**(1)**和**(2)**的执行效率是相同的。第**(3)**种方法比方法**(1)**、**(2)**快。这种方法能提高执行效率。

用下标法比较直观，能直接知道是第几个元素。用地址法或指针变量的方法都不太直观，难以很快地判断出当前处理的是哪一个元素。

在用指针变量指向数组元素时要注意：指针变量**p**可以指向有效的数组元素，实际上也可以指向数组以后的内存单元。如果有

```
int a[10],*p=a;           //指针变量p的初值为&a[0]
```

```
cout<<*(p+10);           //要输出a[10]的值
```

在使用指针变量指向数组元素时，应切实保证指向数组中有效的元素。

指向数组元素的指针的运算比较灵活，务必小心谨慎。下面举几个例子：

如果先使**p**指向数组**a**的首元素(即**p=a**),则：

(1) **p++**（或**p+=1**）。使**p**指向下一元素，即**a[1]**。

如果用**\*p**，得到下一个元素**a[1]**的值。

(2) **\*p++**。由于++和\*同优先级，结合方向为自右而左，因此它等价于**\*(p++)**。作用是：先得到**p**指向的变量的值(即**\*p**)，然后再使**p**的值加1。例6.5(3)程序中最后一个**for**语句：

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

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

可以改写为

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

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

(3) **\*(p++)**与**\*(++p)**作用不同。前者是先取**\*p**值，然后使**p**加1。后者是先使**p**加1，再取**\*p**。若**p**的初值为**a**（即**&a[0]**），输出**\*(p++)**得到**a[0]**的值，而输出**\*(++p)**则得到**a[1]**的值。

**(4) (\*p)++**表示**p**所指向的元素值加 1，即 **(a[0])++**，如果**a[0]=3**，则**(a[0])++**的值为**4**。注意：是元素值加**1**，而不是指针值加**1**。

**(5)** 如果**p**当前指向**a[i]**，则

**\*(p--)** 先对**p**进行“**\***”运算，得到**a[i]**，再使**p**减**1**，**p**指向**a[i-1]**。

**\*(++p)** 先使**p**自加**1**，再作**\***运算，得到**a[i+1]**。

**\*(--p)** 先使**p**自减**1**，再作**\***运算，得到**a[i-1]**。

将**++**和**--**运算符用于指向数组元素的指针变量十分有效，可以使指针变量自动向前或向后移动，指向下一个或上一个数组元素。例如，想输出 **a** 数组 **100**个元素，可以用以下语句：

**p=a;**

**while(p<a+100)**    或

**cout<<\*p++;**

**p=a;**

**while(p<a+100)**

**{cout<<\*p;**

**p++; }**

在用**\*p++**形式的运算时，很容易弄错，一定要十分小心，弄清楚先取**p**值还是先使**p**加**1**。

## 6.3.2 用指针变量作函数参数接收数组地址

在第**5**章**5.4**节中介绍过可以用数组名作函数的参数。前面已经多次强调：数组名代表数组首元素的地址。用数组名作函数的参数，传递的是数组首元素的地址。很容易推想：用指针变量作函数形参，同样可以接收从实参传递来的数组首元素的地址(此时，实参是数组名)。下面将第**5**章**5.4**节中的例**5.7**程序改写，用指针变量作函数形参。

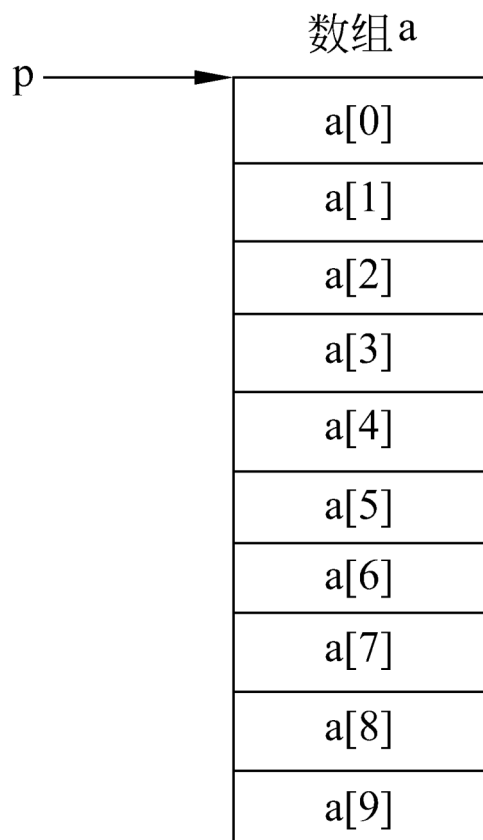
例**6.6** 将**10**个整数按由小到大的顺序排列。

在例**5.7**程序的基础上，将形参改为指针变量。



```
#include <iostream>
using namespace std;
int main( )
{void select_sort(int *p,int n);           //函数声明
int a[10],i;
cout<<"enter the originl array:"<<endl;
for(i=0;i<10;i++)                         //输入10个数
cin>>a[i];
cout<<endl;
select_sort(a,10);                        //函数调用，数组名作实参
cout<<"the sorted array:"<<endl;
for(i=0;i<10;i++)                         //输出10个已排好序的数
cout<<a[i]<<" ";
cout<<endl;
return 0;
}void select_sort(int *p,int n)            //用指针变量作形参
{int i,j,k,t;
for(i=0;i<n-1;i++)
{k=i;
for(j=i+1;j<n;j++)
if(*(p+j)<*(p+k)) k=j;                    //用指针法访问数组元素
t=*(p+k);*(p+k)=*(p+i);*(p+i)=t;
}
}
```

运行情况与例**5.7**相同。



图**6.13**

本例与例**5.7**在程序的表现形式上虽然有不同，但实际上，两个程序在编译以后是完全相同的。**C++**编译系统将形参数组名一律作为指针变量来处理。

实际上在函数调用时并不存在一个占有存储空间的形参数组，只有指针变量。

实参与形参的结合，有以下4种形式：

实 参	形 参
数组名	数组名 (如例5.7)
数组名	指针变量 (如例6.6)
指针变量	数组名
指针变量	指针变量

在此基础上，还要说明一个问题：实参数组名**a**代表一个固定的地址，或者说是指针型常量，因此要改变**a**的值是不可能的。如

**a++;**            //语法错误，**a**是常量，不能改变

而形参数组名是指针变量，并不是一个固定的地址值。它的值是可以改变的。在函数调用开始时，它接收了实参数组首元素的地址，但在函数执行期间，它可以再被赋值。如

```
f(array[],int n)
{ cout<<array;           //输出array[0]的值
  array=array+3;          //指针变量array的值改变了，指向array[3]
  cout<<*arr<<endl;      //输出array[3]的值
}
```

### 6.3.3 多维数组与指针

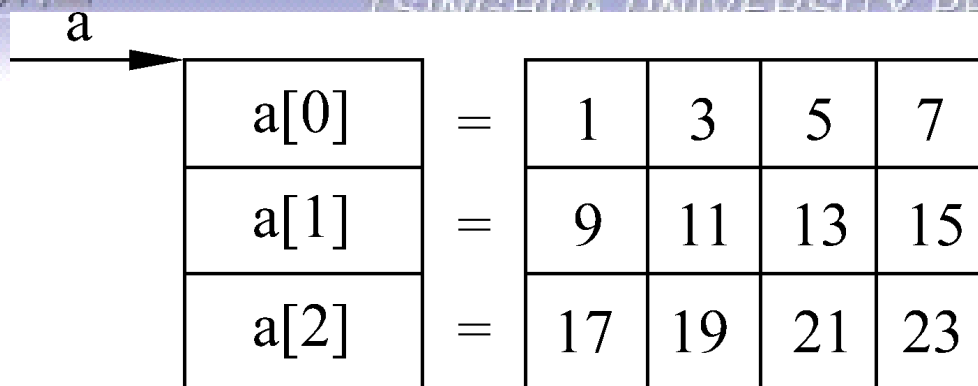
用指针变量可以指向一维数组中的元素，也可以指向多维数组中的元素。

#### 1. 多维数组元素的地址

设有一个二维数组  $a$ ，它有**3**行**4**列。它的定义为

```
int a[3][4]={1,3,5,7},{9,11,13,15},{17,18,21,23};
```

$a$ 是一个数组名。 $a$ 数组包含**3**行，即**3**个元素： $a[0]$ 、 $a[1]$ 、 $a[2]$ 。而每一元素又是一个一维数组，它包含**4**个元素(即**4**个列元素)，例如， $a[0]$ 所代表的一维数组又包含**4**个元素： $a[0][0]$ 、 $a[0][1]$ 、 $a[0][2]$ 、 $a[0][3]$ ，见图6.14。可以认为二维数组是“数组的数组”，即数组 $a$ 是由**3**个一维数组所组成的。



a[0]	=	1	3	5	7
a[1]	=	9	11	13	15
a[2]	=	17	19	21	23

图6.14

从二维数组的角度来看，**a**代表二维数组首元素的地址，现在的首元素不是一个整型变量，而是由4个整型元素所组成的一维数组，因此**a**代表的是首行的起始地址(即第**0**行的起始地址，**&a[0]**)，**a+1**代表**a[1]**行的首地址，即**&a[1]**。

**a[0]**,**a[1]**,**a[2]**既然是一维数组名，而**C++**又规定了数组名代表数组首元素地址，因此**a[0]**代表一维数组**a[0]**中**0**列元素的地址，即**&a[0][0]**。**a[1]**的值是**&a[1][0]**，**a[2]**的值是**&a[2][0]**。

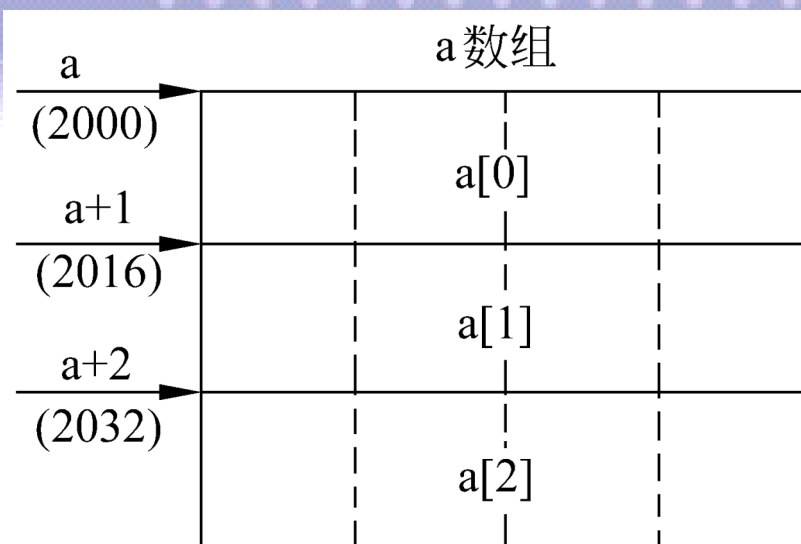


图6.15

0行1列元素的地址可以直接写为 **$\&a[0][1]$** ，也可以用指针法表示。 **$a[0]$** 为一维数组名，该一维数组中序号为1的元素显然可以用 **$a[0]+1$** 来表示，见图6.16。

欲得到 **$a[0][1]$** 的值，用地址法怎么表示呢？既然 **$a[0]+1$** 是 **$a[0][1]$** 元素的地址，那么， **$*(a[0]+1)$** 就是 **$a[0][1]$** 元素的值。而 **$a[0]$** 又是和 **$*(a+0)$** 无条件等价的，因此也可以用 **$*(*(a+0)+1)$** 表示 **$a[0][1]$** 元素的值。依此类推， **$*(a[i]+j)$** 或 **$*(*(a+i)+j)$** 是 **$a[i][j]$** 的值。

	$a[0]$	$a[0]+1$	$a[0]+2$	$a[0]+3$
$a$	2000	2004	2008	2012
$a+1$	1	3	5	7
	2016	2020	2024	2028
$a+2$	9	11	13	15
	2032	2036	2040	2044
	17	19	21	23

图6.16



## 2. 指向多维数组元素的指针变量

### (1) 指向数组元素的指针变量

例6.7 输出二维数组各元素的值。

这里采用的方法是用基类型为整型的指针变量先后指向各元素，逐个输出它们的值。

```
#include <iostream>
using namespace std;
int main( )
{ int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
  int *p;                //p是基类型为整型的指针变量
  for(p=a[0];p<a[0]+12;p++)
    cout<<*p<<" ";
  cout<<endl;
  return 0;
}
```

运行结果如下：

**1 3 5 7 9 11 13 15 17 19 21 23**

说明：

- ① **p**是指向整型数据的指针变量，在**for**语句中对**p**赋初值**a[0]**，也可以写成“**p=&a[0][0]**”。
- ② 循环结束的条件是“**p<a[0]+12**”，只要满足**p<a[0]+12**，就继续执行循环体。
- ③ 执行“**cout<<\*p;**”输出**p**当前所指的列元素的值，然后执行**p++**，使**p**指向下一个列元素。

(2) 指向由  $m$  个元素组成的一维数组的指针变量  
可以定义一个指针变量，它不是指向一个整型元素，而是指向一个包含  $m$  个元素的一维数组。这时，如果指针变量  $p$  先指向  $a[0]$ （即  $p = \&a[0]$ ），则  $p+1$  不是指向  $a[0][1]$ ，而是指向  $a[1]$ ， $p$  的增值以一维数组的长度为单位，见图 6.17。

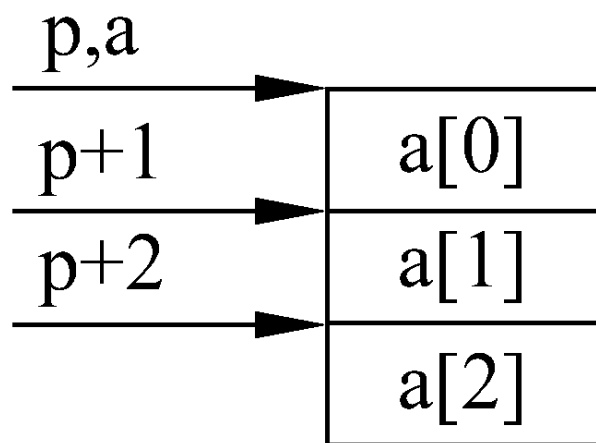


图 6.17

**例6.8** 输出二维数组任一行任一列元素的值。

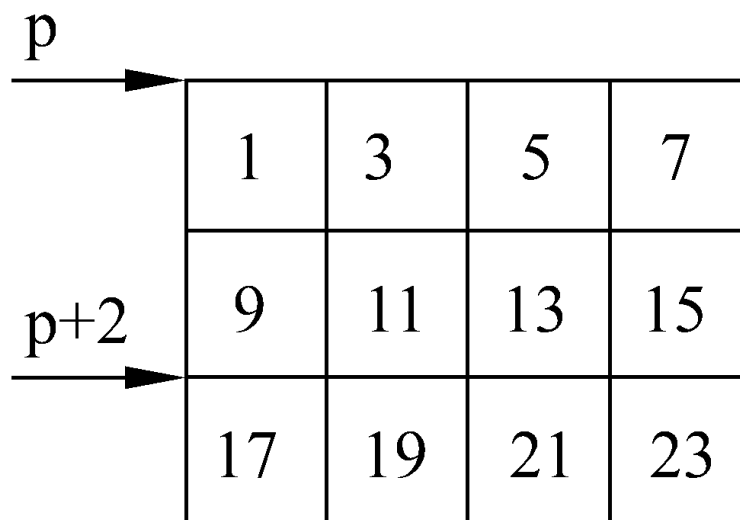
```
#include <iostream>
using namespace std;
int main( )
{ int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
  int (*p)[4],i,j;
  cin>>i>>j;
  p=a;
  cout<<*(*p+i)+j<<endl;
  return 0;
}
```

运行情况如下：

2 3↙

23

由于执行了“**p=a**”，使**p**指向**a[0]**。因此**p+2**是二维数组**a**中序号为**2**的行的起始地址（由于**p**是指向一维数组的指针变量，因此**p**加**1**，就指向下一个一维数组），见图**6.18**。**\*(p+2)+3**是**a**数组**2**行**3**列元素地址。**\*(\*(p+2)+3)**是**a[2][3]**的值。



图**6.18**

### 3. 用指向数组的指针作函数参数

一维数组名可以作为函数参数传递，多维数组名也可作函数参数传递。

例**6.9** 输出二维数组各元素的值。

题目与例**6.7**相同，但本题用一个函数实现输出，用多维数组名作函数参数。

```
#include <iostream>
using namespace std;
int main( )
{ void output(int (*p)[4]);           //函数声明
  int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
  output(a);                          //多维数组名作函数参数
  return 0;
}
```

```
void output(int (*p)[4])
```

//形参是指向一维数组的指针变量

```
{ int i,j;
```

```
for(i=0;i<3;i++)
```

```
for(j=0;j<4;j++)
```

```
cout<<*(*p+i)+j<<" ";
```

```
cout<<endl;
```

```
}
```

运行情况如下:

**1 3 5 7 9 11 13 15 17 19 21 23**

## 6.4 字符串与指针

在C++中可以用3种方法访问一个字符串(在第5章介绍了前两种方法)。

### 1. 用字符数组存放一个字符串

例6.10 定义一个字符数组并初始化，然后输出其中的字符串。

```
#include <iostream>
using namespace std;
int main( )
{ char str[]="I love CHINA!";
  cout<<str<<endl;
  return 0;
}
```



运行时输出：

**I love CHINA!**

## 2. 用字符串变量存放字符串

例**6.11** 定义一个字符串变量并初始化，然后输出其中的字符串。

```
#include <string>
#include <iostream>
using namespace std;
int main( )
{ string str="I love CHINA!";
  cout<<str<<endl;
  return 0;
}
```

### 3. 用字符指针指向一个字符串

例**6.12** 定义一个字符指针变量并初始化，然后输出它指向的字符串。

```
#include <iostream>
using namespace std;
int main( )
{ char *str="I love CHINA!";
  cout<<str<<endl;
  return 0;
}
```

对字符串中字符的存取，可以用下标方法，也可以用指针方法。

例**6.13** 将字符串**str1**复制为字符串**str2**。

定义两个字符数组**str1**和**str2**，再设两个指针变量**p1**和**p2**，分别指向两个字符数组中的有关字符，通过改变指针变量的值使它们指向字符串中的不同的字符，以实现字符的复制。

```
#include <iostream>
```

```
using namespace std;
```

```
int main( )
```

```
{ char str1[]="I love CHINA!",str2[20],*p1,*p2;
```

```
p1=str1;p2=str2;
```

```
for(;*p1!='\\0';p1++,p2++)
```

```
    *p2=*p1;
```

```
    *p2='\\0';
```

```
p1=str1;p2=str2;
```

```
cout<<"str1 is: "<<p1<<endl;
```

```
cout<<"str2 is: "<<p2<<endl;
```

```
return 0;
```

```
}
```

运行结果为

**str1 is: I love CHINA!**

**str2 is: I love CHINA!**

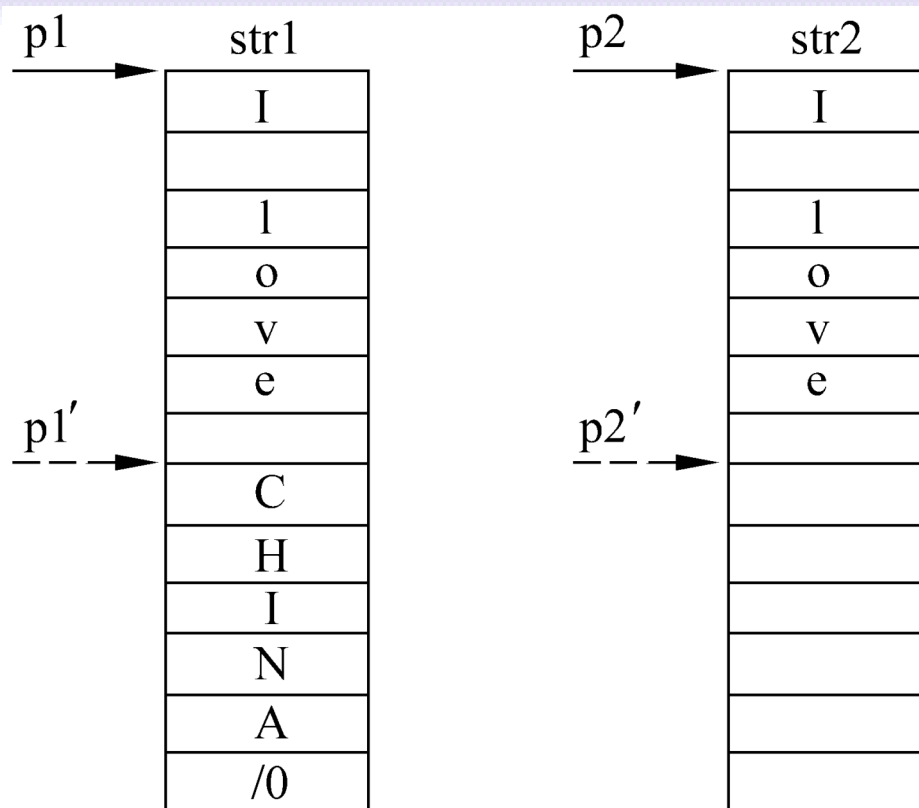


图6.19

这个例子用来说明怎样使用字符指针。其实，对例6.13来说，用**string**变量来处理是十分简单的：

```
string str1="I love CHINA!",str2;    //定义string变量
```

```
str2=str1;                          //将str1复制到str2
```

## 6.5 函数与指针

### 6.5.1 用函数指针变量调用函数

指针变量也可以指向一个函数。一个函数在编译时被分配给一个入口地址。这个函数入口地址就称为函数的指针。可以用一个指针变量指向函数，然后通过该指针变量调用此函数。

例**6.14** 求 a 和 b 中的大者。

先按一般方法写程序：

```
#include <iostream>
using namespace std;
int main( )
{int max(int x,int y);           //函数声明
```

```
int a,b,m;  
cin>>a>>b;  
m=max(a,b);           //调用函数max, 求出最大值, 赋给m  
cout<<"max="<<m<<endl;  
return 0;  
}
```

```
int max(int x,int y)  
{int z;  
if(x>y) z=x;  
else z=y;  
return(z);  
}
```

可以用一个指针变量指向**max**函数，然后通过该指针变量调用此函数。定义指向**max**函数的指针变量的方法是：

**int (\*p) (int,int);**

└──┬──┬──┐  
└──┬──┬──┐ **p**所指向的函数的形参类型  
└──┬──┬──┐ **p**是指向函数的指针变量  
└──┬──┬──┐ 指针变量**p**指向的函数的类型

请将它和函数**max**的原型作比较

**int max(int,int);**                      //**max**函数原型

可以看出： 只是用(**\*p**)取代了**max**，其他都一样。  
现在将上面程序的主函数修改如下：

```
#include <iostream>
using namespace std;
int main( )
{int max(int x,int y);           //函数声明
int (*p)(int,int);              //定义指向函数的指针变量p
int a,b,m;
p=max;                          //使p指向函数max
cin>>a>>b;
m=p(a,b);
cout<<"max="<<m<<endl;
return 0;
}
```

请注意第7行的赋值语句“**p=max;**”。此语句千万不要漏写，它的作用是将函数**max**的入口地址赋给指针变量**p**。这时，**p**才指向函数**max**。见图6.20。





图6.20

指向函数的指针变量的一般定义形式为  
函数类型 (\*指针变量名) (函数形参表) ;

## 6.5.2 用指向函数的指针作函数参数

在C语言中，函数指针变量常见的用途之一是作为函数的参数，将函数名传给其他函数的形参。这样就可以在调用一个函数的过程中根据给定的不同实参调用不同的函数。

例如，利用这种方法可以编写一个求定积分的通用函数，用它分别求5个函数的定积分：每次需要求定积分的函数是不一样的。可以编写一个求定积分的通用函数**integral**，它有3个形参：下限**a**、上限**b**，以及指向函数的指针变量**fun**。函数原型可写为

```
double integral (double a, double b, double (*fun)(double));
```

分别编写**5**个函数**f1**, **f2**, **f3**, **f4**, **f5**, 用来求上面**5**个函数的值。然后先后调用**integral**函数**5**次, 每次调用时把**a**, **b**以及**f1**, **f2**, **f3**, **f4**, **f5**之一作为实参, 即把上限、下限以及有关函数的入口地址传送给形参**fun**。在执行**integral**函数过程中求出各函数定积分的值。

在面向对象的**C++**程序设计中, 这种用法就比较少了。有兴趣的读者可参阅作者所著的《**C**程序设计(第二版)》一书中的有关章节。

## 6.6 返回指针值的函数

一个函数可以带回一个整型值、字符值、实型值等，也可以带回指针型的数据，即地址。其概念与以前类似，只是带回的值的类型是指针类型而已。返回指针值的函数简称为指针函数。

定义指针函数的一般形式为

类型名 \*函数名（参数表列）；

例如

```
int *a(int x,int y);
```

## 6.7 指针数组和指向指针的指针

### 6.7.1 指针数组的概念

如果一个数组，其元素均为指针类型数据，该数组称为指针数组，也就是说，指针数组中的每一个元素相当于一个指针变量，它的值都是地址。一维指针数组的定义形式为

类型名\*数组名 [数组长度]；

例如

```
int *p[4];
```

可以用指针数组中各个元素分别指向若干个字符串，使字符串处理更加方便灵活。

## 例6.15 若干字符串按字母顺序（由小到大）输出。

```
#include <iostream>
using namespace std;
int main( )
{ void sort(char *name[],int n);          //声明函数
  void print(char *name[],int n);         //声明函数
  char *name[]={"BASIC","FORTRAN","C++","Pascal","COBOL"}; //定义指针
  数组
  int n=5;
  sort(name,n);
  print(name,n);
  return 0;
}
void sort(char *name[],int n)
{ char *temp;
  int i,j,k;
  for(i=0;i<n-1;i++)
  {k=i;
```

```
for(j=i+1;j<n;j++)  
if(strcmp(name[k],name[j])>0) k=j;  
if(k!=i)  
{ temp=name[i];name[i]=name[k];name[k]=temp;}  
}  
}
```

```
void print(char *name[],int n)  
{ int i;  
for(i=0;i<n;i++)  
cout<<name[i]<<endl;  
}
```

运行结果为

**BASIC**

**COBOL**

**C++**

**FORTRAN**

**Pascal**

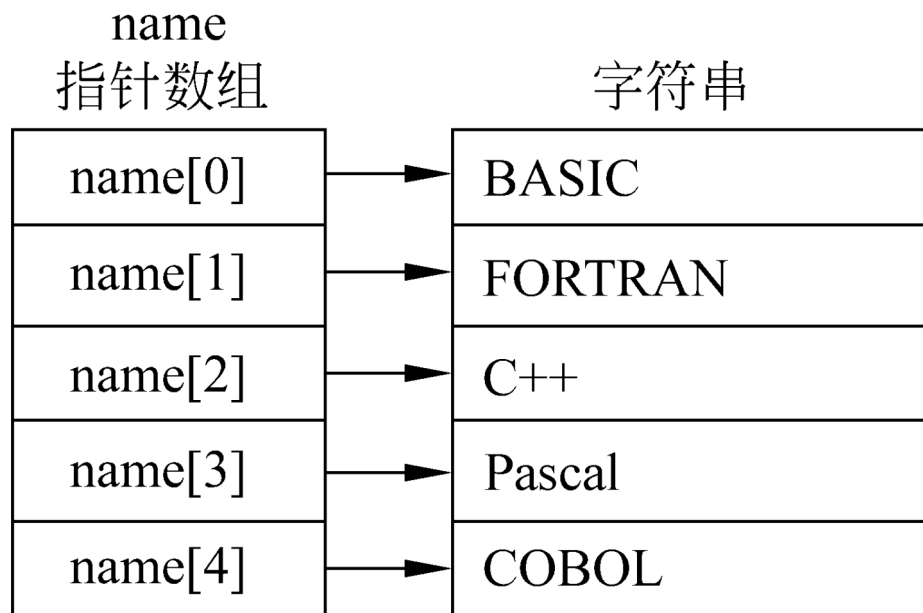


图6.21

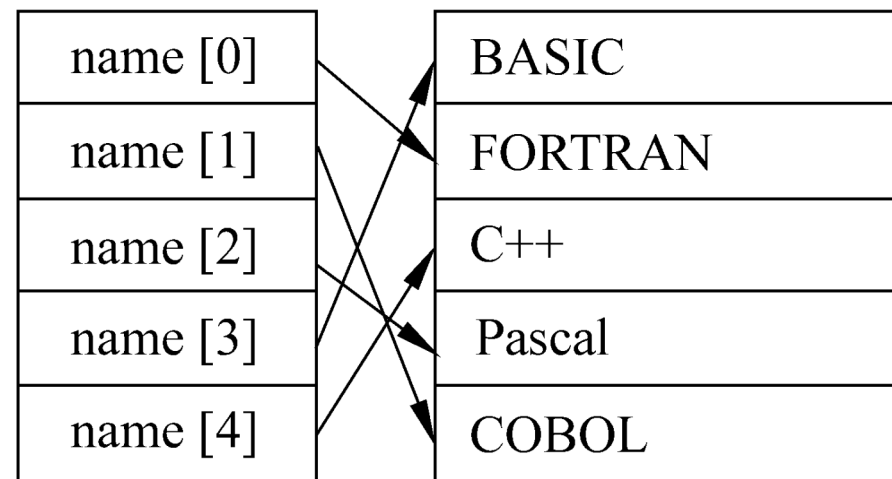


图6.22



**print**函数的作用是输出各字符串。

**name[0]~name[4]**分别是各字符串的首地址。**print**函数也可改写为以下形式：

```
void print(char *name[],int n)
```

```
{ int i=0
```

```
char *p;
```

```
p=name[0];
```

```
while(i<n)
```

```
{p=*(name+i++);
```

```
cout<<p<<endl;
```

```
}
```

```
}
```

其中“**\*(name+i++)**”表示先求**\*(name+i)**的值，即**name[i]**（它是一个地址）。将它赋给**p**，然后**i**加1。最后输出以**p**地址开始的字符串。

## 6.7.2 指向指针的指针

在掌握了指针数组的概念的基础上，下面介绍指向指针数据的指针，简称为指向指针的指针。从图6.22可以看到，**name**是一个指针数组，它的每一个元素是一个指针型数据(其值为地址)，分别指向不同的字符串。数组名**name**代表该指针数组首元素的地址。**name+i**是**name[i]**的地址。由于**name[i]**的值是地址(即指针)，因此**name+i**就是指向指针型数据的指针。还可以设置一个指针变量**p**，它指向指针数组的元素(见图6.23)。**p**就是指向指针型数据的指针变量。

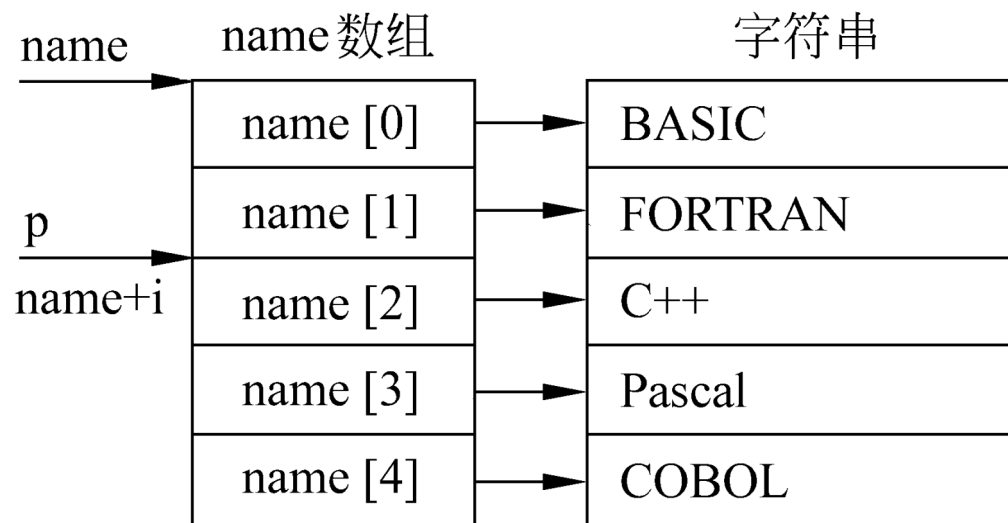


图6.23

怎样定义一个指向指针数据的指针变量呢？如下：

**char \*(\*p);**

从附录B可以知道，\*运算符的结合性是从右到左，因此“**char \*(\*p);**”可写成

**char \*\*p;**

## 例6.16 指向字符型数据的指针变量。

```
#include <iostream>
using namespace std;
int main( )
{ char **p;           //定义指向字符指针数据的指针变量p
  char *name[]={"BASIC","FORTRAN","C++","Pascal","COBOL"};
  p=name+2;           //见图6.23中p的指向
  cout<<*p<<endl;     //输出name[2]指向的字符串
  cout<<**p<<endl;     //输出name[2]指向的字符串中的第一个字符
}
```

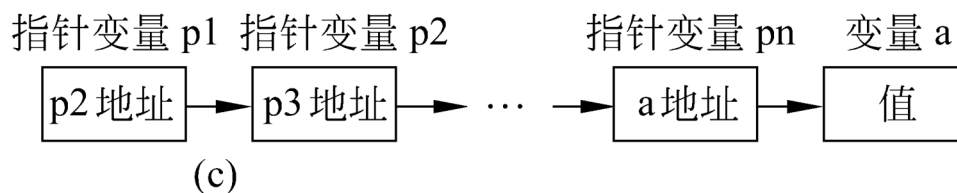
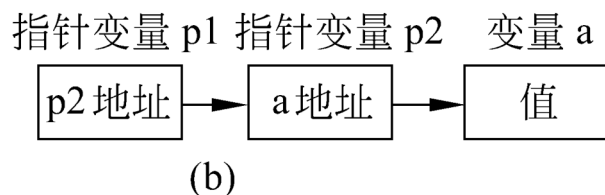
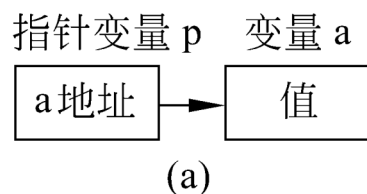
运行结果为

C++

C

指针数组的元素也可以不指向字符串，而指向整型数据或单精度型数据等。

在本章开头已经提到了“间接访问”一个变量的方式。利用指针变量访问另一个变量就是“间接访问”。如果在一个指针变量中存放一个目标变量的地址，这就是“单级间址”，见图6.24(a)。指向指针的指针用的是“二级间址”方法。见图6.24(b)。从理论上说，间址方法可以延伸到更多的级，见图6.24(c)。但实际上在程序中很少有超过二级间址的。

**图6.24**

## 6.8 有关指针的数据类型和指针运算的小结

### 6.8.1 有关指针的数据类型的小结

表6.1 有关指针的数据类型

定义	含义
<b>int i;</b>	定义整型变量
<b>int *p;</b>	<b>p</b> 为指向整型数据的指针变量
<b>int a[n];</b>	定义整型数组 <b>a</b> ，它有 <b>n</b> 个元素
<b>int *p[n];</b>	定义指针数组 <b>p</b> ，它由 <b>n</b> 个指向整型数据的指针元素组成
<b>int (*p)[n];</b>	<b>p</b> 为指向含 <b>n</b> 个元素的一维数组的指针变量
<b>int f( );</b>	<b>f</b> 为带回整型函数值的函数
<b>int *p( );</b>	<b>p</b> 为带回一个指针的函数，该指针指向整型数据
<b>int (*p)( );</b>	<b>p</b> 为指向函数的指针，该函数返回一个整型值
<b>int **p;</b>	<b>p</b> 是一个指向指针的指针变量，它指向一个指向整型数据的指针变量

## 6.8.2 指针运算小结

前面已用过一些指针运算（如 **$p++$** , **$p+i$** 等），现在把全部的指针运算列出如下。

### (1) 指针变量加/减 一个整数

例如： **$p++$** , **$p--$** , **$p+i$** , **$p-i$** , **$p+-i$** , **$p-=i$** 等。

**C++**规定，一个指针变量加/减一个整数是将该指针变量的原值(是一个地址)和它指向的变量所占用的内存单元字节数相加或相减。如 **$p+i$** 代表这样的地址计算： **$p+i*d$** ， **$d$** 为 **$p$** 所指向的变量单元所占用的字节数。这样才能保证 **$p+i$** 指向 **$p$** 下面的第 **$i$** 个元素。

### (2) 指针变量赋值

将一个变量地址赋给一个指针变量。如

**p=&a;**               //将变量 **a** 的地址赋给**p**  
**p=array;**           //将数组**array**首元素的地址赋给**p**  
**p=&array[i];**       //将数组**array**第**i**个元素的地址赋给**p**  
**p=max;**             //**max**为已定义的函数，将**max**的入口地址赋给**p**  
**p1=p2;**            //**p1**和**p2**都是同类型的指针变量，将**p2**的值赋给**p1**

**(3)** 指针变量可以有空值，即该指针变量不指向任何变量，可以这样表示：

**p=NULL;**

实际上**NULL**代表整数**0**，也就是使**p**指向地址为**0**的单元。这样可以使指针不指向任何有效的单元。实际上系统已先定义了

**NULL: #define NULL 0**

在**iostream**头文件中就包括了以上的**NULL**定义，**NULL**是一个符号常量。应注意，**p**的值等于**NULL**和**p**未被赋值是两个不同的概念。



任何指针变量或地址都可以与**NULL**作相等或不相等的比较，如

**if(p==NULL) p=p1;**

**(4) 两个指针变量可以相减**

如果两个指针变量指向同一个数组的元素，则两个指针变量值之差是两个指针之间的元素个数，见图**6.25**。

假如**p1**指向**a[1]**，**p2**指向**a[4]**，则**p2-p1=(a+4)-(a+1)=4-1=3**。

但**p1+p2**并无实际意义。

## (5) 两个指针变量比较

若两个指针指向同一个数组的元素，则可以进行比较。指向前面的元素的指针变量小于指向后面元素的指针变量。如图6.25中， $p1 < p2$ ，或者说，表达式“ $p1 < p2$ ”的值为真，而“ $p2 < p1$ ”的值为假。注意，如果 $p1$ 和 $p2$ 不指向同一数组则比较无意义。

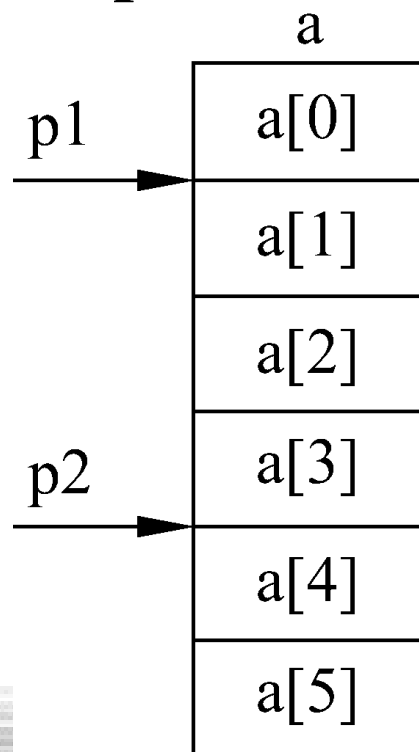


图6.25

## (6) 对指针变量的赋值应注意类型问题。

在本章前几节中介绍了指针的基本概念和初步应用。应该说明，指针是**C**和**C++**中重要的概念，是**C**和**C++**的一个特色。使用指针的优点是：①提高程序效率；②在调用函数时，如果改变被调用函数中某些变量的值，这些值能为主调函数使用，即可以通过函数的调用，得到多个可改变的值；③可以实现动态存储分配。

但是同时应该看到，指针使用实在太灵活，对熟练的程序人员来说，可以利用它编写出颇有特色的、质量优良的程序，实现许多用其他高级语言难以实现的功能，但也十分容易出错，而且这种错误往往难以发现。

## \*6.9 引用

### 6.9.1 什么是变量的引用

对一个数据可以使用“引用”(reference)，这是C++对C的一个重要扩充，引用是一种新的变量类型，它的作用是为一个变量起一个别名。假如有一个变量**a**，想给它起一个别名**b**，可以这样写：

```
int a;           //定义a是整型变量  
int &b=a;        //声明b是a的引用
```

以上语句声明了**b**是**a**的引用，即**b**是**a**的别名。经过这样的声明后，**a**或**b**的作用相同，都代表同一变量。

注意：在上述声明中，**&**是引用声明符，并不代表地址。不要理解为“把**a**的值赋给**b**的地址”。声明变量**b**为引用类型，并不需要另外开辟内存单元来存放**b**的值。**b**和**a**占内存中的同一个存储单元，它们具有同一地址。声明**b**是**a**的引用，可以理解为：使变量**b**具有变量**a**的地址。见图6.26，如果**a**的值是**20**，则**b**的值也是**20**。

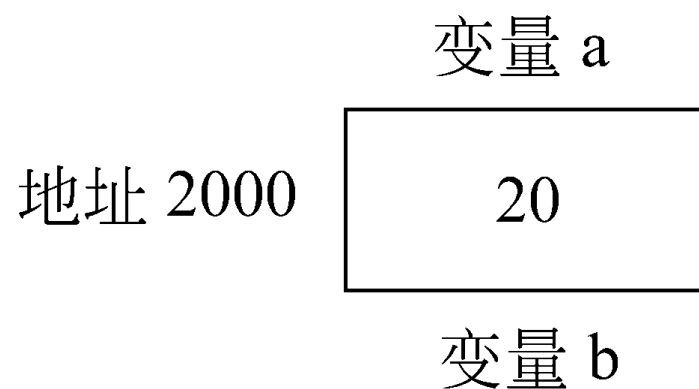


图6.26

在声明一个引用类型变量时，必须同时使之初始化，即声明它代表哪一个变量。在声明变量**b**是变量**a**的引用后，在它们所在函数执行期间，该引用类型变量**b**始终与其代表的变量**a**相联系，不能再作为其他变量的引用(别名)。下面的用法不对：

```
int a1, a2;
```

```
int &b=a1;
```

```
int &b=a2;           //企图使b又变成a2的引用（别名）是不行的
```

## 6.9.2 引用的简单使用

例6.17 引用和变量的关系。

```
#include <iostream>
#include <iomanip>
using namespace std;
int main( )
{ int a=10;
  int &b=a;           //声明b是a的引用
  a=a*a;             //a的值变化了， b的值也应一起变化
  cout<<a<<setw(6)<<b<<endl;
  b=b/5;             //b的值变化了， a的值也应一起变化
  cout<<b<<setw(6)<<a<<endl;
  return 0;
}
```

**a**的值开始为**10**，**b**是**a**的引用，它的值当然也应该是**10**，当**a**的值变为**100**（**a\*a**的值）时，**b**的值也随之变为**100**。在输出**a**和**b**的值后，**b**的值变为**20**，显然**a**的值也应为**20**。

运行记录如下：

<b>100</b>	<b>100</b>	( <b>a</b> 和 <b>b</b> 的值都是 <b>100</b> )
<b>20</b>	<b>20</b>	( <b>a</b> 和 <b>b</b> 的值都是 <b>20</b> )



### 6.9.3 引用作为函数参数

有了变量名，为什么还需要一个别名呢？C++之所以增加引用类型，主要是把它作为函数参数，以扩充函数传递数据的功能。

到目前为止，本书介绍过函数参数传递的两种情况。

**(1)** 将变量名作为实参和形参。这时传给形参的是变量的值，传递是单向的。如果在执行函数期间形参的值发生变化，并不传回给实参。因为在调用函数时，形参和实参不是同一个存储单元。

**例6.18** 要求将变量*i*和*j*的值互换。下面的程序无法实现此要求。

```
#include <iostream>
using namespace std;
int main( )
{ void swap(int,int);           //函数声明
  int i=3,j=5;
  swap(i,j);                     //调用函数swap
  cout<<i<<" "<<j<<endl;       //i和j的值未互换
  return 0;
}
```

```
void swap(int a,int b)  //企图通过形参a和b的值互换，实现实参i和j的
                        值互换
{ int temp;
  temp=a;               //以下3行用来实现a和b的值互换
  a=b;
  b=temp;
}
```

运行时输出**3 5**i和j的值并未互换。见图**6.27**示意。

为了解决这个问题，采用传递变量地址的方法。

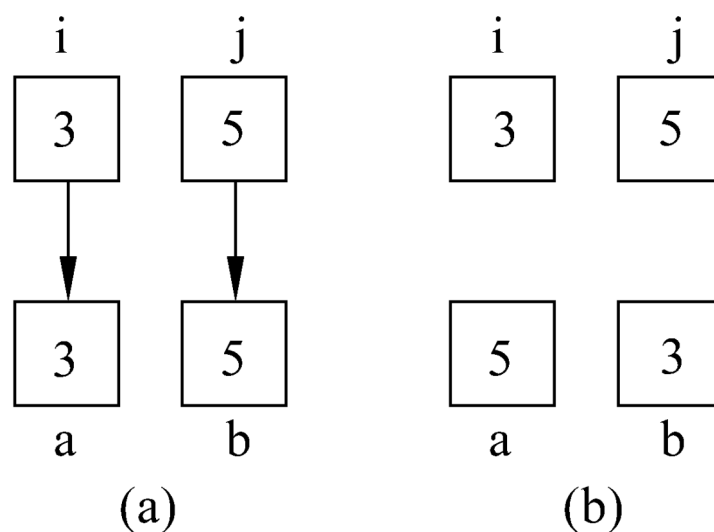


图6.27

**(2) 传递变量的指针。**形参是指针变量，实参是一个变量的地址，调用函数时，形参(指针变量)指向实参变量单元。程序见例6.19。

**例6.19** 使用指针变量作形参，实现两个变量的值互换。

```
#include <iostream>
using namespace std;
int main( )
{ void swap(int *,int *);
  int i=3,j=5;
  swap(&i,&j);
  cout<<i<<" "<<j<<endl;
  return 0;
}
```

//实参是变量的地址  
//i和j的值已互换

```
void swap(int *p1,int *p2)
{ int temp;
  temp=*p1;
  *p1=*p2;
  *p2=temp;
}
```

//形参是指针变量  
//以下3行用来实现i和j的值互换

形参与实参的结合见图6.28示意。

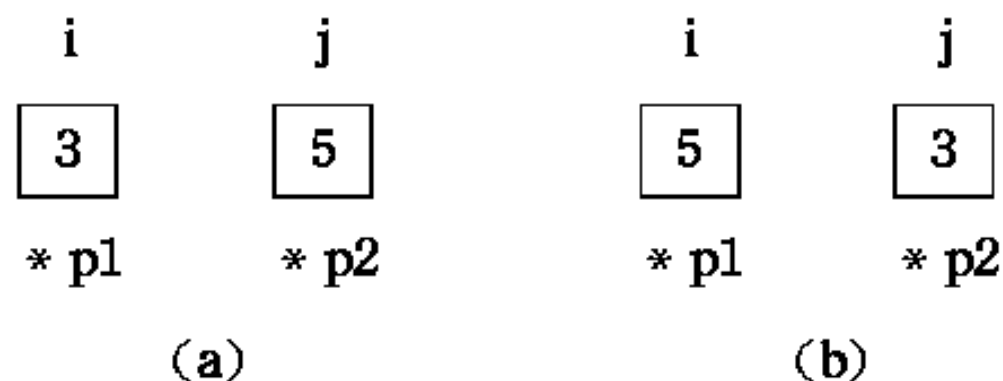


图6.28

这种虚实结合的方法仍然是“值传递”方式，只是实参的值是变量的地址而已。通过形参指针变量访问主函数中的变量(**i**和**j**)，并改变它们的值。这样就能得到正确结果，但是在概念上却是兜了一个圈子，不那么直截了当。

在**Pascal**语言中有“值形参”和“变量形参”（即**var**形参），对应两种不同的传递方式，前者采用值传递方式，后者采用地址传递方式。在**C**语言中，只有“值形参”而无“变量形参”，全部采用值传递方式。**C++**把引用型变量作为函数形参，就弥补了这个不足。

**C++**提供了向函数传递数据的第**(3)**种方法，即传送变量的别名。

**例6.20** 利用“引用形参”实现两个变量的值互换。

```
#include <iostream>
using namespace std;
int main( )
{ void swap(int &,int &);
  int i=3,j=5;
```

```
swap(i,j);  
cout<<"i="<<i<<" "<<"j="<<j<<endl;  
return 0;  
}
```

```
void swap(int &a,int &b)           //形参是引用类型  
{ int temp;  
temp=a;  
a=b;  
b=temp;  
}
```

输出结果为

**i=5 j=3**

在**swap**函数的形参表列中声明**a**和**b** 是整型变量的引用。

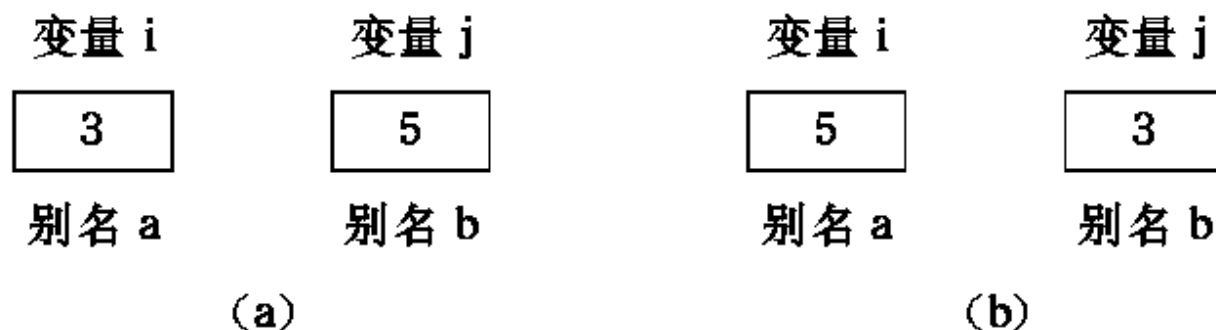


图6.29

实际上，在虚实结合时是把实参*i*的地址传到形参**a**，使形参**a**的地址取实参*i*的地址，从而使**a**和*i*共享同一单元。同样，将实参*j*的地址传到形参**b**，使形参**b**的地址取实参*j*的地址，从而使**b**和*j*共享同一单元。这就是地址传递方式。为便于理解，可以通俗地说：把变量*i*的名字传给引用变量**a**，使**a**成为*i*的别名。

请思考：这种传递方式和使用指针变量作形参时有何不同？  
对比例6.20（对比例6.10）可以发现



- ① 使用引用类型就不必在**swap**函数中声明形参是指针变量。指针变量要另外开辟内存单元，其内容是地址。而引用变量不是一个独立的变量，不单独占内存单元，在例**6.20**中引用变量**a**和**b**的值的数据类型与实参相同，都是整型。
- ② 在**main**函数中调用**swap**函数时，实参不必用变量的地址(在变量名的前面加**&**)，而直接用变量名。系统向形参传送的是实参的地址而不是实参的值。这种传递方式相当于**Pascal**语言中的“变量形参”，显然，这种用法比使用指针变量简单、直观、方便。使用变量的引用，可以部分代替指针的操作。有些过去只能用指针来处理的问题，现在可以用引用来代替，从而降低了程序设计的难度。

## 例6.21 对3个变量按由小到大的顺序排序。

```
#include <iostream>
using namespace std;
int main( )
{ void sort(int &,int &,int &); //函数声明，形参是引用类型
  int a,b,c;                    //a,b,c是需排序的变量
  int a1,b1,c1;                 //a1,b1,c1最终的值是已排好序的数列
  cout<<"Please enter 3 integers:";
  cin>>a>>b>>c;                //输入a,b,c
  a1=a;b1=b;c1=c;
  sort(a1,b1,c1);               //调用sort函数，以a1,b1,c1为实参
  cout<<"sorted order is "<<a1<<" "<<b1<<" "<<c1<<endl; //此时a1,b1,c1已排好序
  return 0;
}

void sort(int &i,int &j,int &k)    //对i,j,k 3个数排序
{ void change(int &,int &);      //函数声明，形参是引用类型
  if (i>j) change (i,j);          //使i<=j
```

```
if (i>k) change (i,k);           //使i<=k
if (j>k) change (j,k);           //使j<=k
}
void change (int &x,int &y)       //使x和y互换
{ int temp;
temp=x;
x=y;
y=temp;
}
```

运行情况如下:

**Please enter 3 integers: 23 12 -345✓**

**sorted order is -345 12 23**

可以看到：这个程序很容易理解，不易出错。由于在调用**sort**函数时虚实结合使形参**i,j,k**成为实参**a1,b1,c1**的引用，因此通过调用函数**sort(a1,b1,c1)**既实现了对**i,j,k**排序，也就同时实现了对**a1,b1,c1**排序。同样，执行**change (i,j)**函数，可以实现对实参**i**和**j**的互换。

引用不仅可以用于变量，也可以用于对象。例如实参可以是一个对象名，在虚实结合时传递对象的起始地址。这会在以后介绍。

当看到**&a**这样的形式时，怎样区别是声明引用变量还是取地址的操作呢？当**&a**的前面有类型符时（如**int &a**），它必然是对引用的声明；如果前面无类型符（如**cout<<&a**），则是取变量的地址。