第 6 章　指针和引用

本章摘要

指针是 C++语言中的一个重要的概念，它提供了一种较为直观的地址操作的手段。正确而灵活地运用它，可以有效地表示复杂的数据结构，能动态分配内存，能方便地使用字符串，能直接处理内存地址等，这对设计系统软件是很必要的。而引用是 C++中的新概念，它可为变量起别名，主要用作函数参数以及函数的返回类型。

6．1　指针

6．1．1　地址和指针的概念

我们知道计算机的内存储器被划分为一个个的存储单元，每个存储单元存放 8 个二进制位（即一个字节）。存储单元按一定的规则编号，这个编号就是存储单元的地址。也就是说计算机中存储的每个字节是一个基本内存单元，有一个地址。计算机就是通过这种地址编号的方式来管理内存数据读写的准确定位的。

计算机是如何从内存单元中存取数据的呢？从程序设计的角度看，有两种办法：一是通过变量名，二是通过地址。程序中定义的变量是要占据一定的内存空间的，例如，C++语言中短整型变量占 2 字节，整型变量占 4 个字节，实型（float）变量占 4 字节。程序中定义的变量在编译时被分配内存空间。在变量分配内存空间的同时，变量名也就成为了相应内存空间的名称，在程序中可以用这个名字访问该内存空间，表现在程序语句中就是通过变量名存取变量内容（这就是程序中定义变量的用途，即程序中通过定义变量来实现数据在内存中的存取）。

假设程序已定义了 2 个整型变量 i、j，编译时系统分配 0x2000 至 0x2003 四个字节单元给变量 i（实际上内存单元的地址为 32 位，4 个字节，这里我们为了简单起见，采用两个字节表示内存单元地址），0x2004 至 0x2007 四个字节单元给 j，如图 6-1 所示。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该变量在内存的存放地址，对变量值的存取都是通过地址进行的。例如，语句“ i+=j；”执行是这样的；根据变量名与地址的对应关系（这个对应关系是在编译时确定的），找到变量 i 的地址 0x2000，然后从由 0x2000 开始的四个字节单元中取出数据（即变量 i 的值 3），再从 0x2004 开始的四个字节单元中取出数据（即变量 j 的值 6），将它们相加后再将其和（9）送回到 i 所占用的 0x2000、0x2001、0x2002、0x2003 字节单元中。这种按变量地址存取变量值的方式也称为“直接访问”方式。

但是，有时使用变量名不够方便或者根本没有变量名可用，这时就可以直接用地址来访问内存单元。举个例子来说，学生公寓中每个学生住一间房，每个学生就相当于一个变

量的内容，变量名指定为学生姓名，房间是存储单元，房号就是存储单元地址。如果知道了学生姓名，可以通过这个名字来访问该学生，这相当于使用变量名访问数据。如果知道了房号，同样也可以访问该学生，这相当于通过地址访问数据。

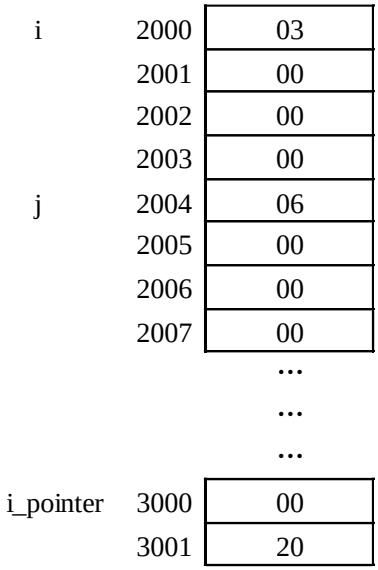

图 6-1　变量的存储分配

由于通过地址能访问指定的内存存储单元，我们可以说，地址“指向”该内存存储单元（如同说，房间号“指向”某一房间一样）。因此，将地址形象化地称为“指针”，意思是通过它能找到以它为地址的内存单元。一个变量的地址称为该变量的“指针”。如果有一个变量专门用来存放另一变量的地址（即指针），则它称为“指针变量”。在 C++语言中有专门用来存放内存单元地址的变量类型，这就是指针类型。指针变量就是具有指针类型的变量，它是用于存放内存单元地址的。

假设我们定义了一个指针变量 i_pointer 用来存放整型变量的地址，它被分配为 0x3000、0x3001 字节单元。可以通过下面语句将变量 i 的地址（0x2000）存放到变量 i_pointer 中。

i_pointer=&i；

这时，i_pointer 的值就是 0x2000，即变量 i 所占用单元的起始地址，如图 9-1 所示。要存取变量 i 的值，也可以来用间接方式：先找到存放“i 的地址”的变量 i_pointer，从中取出 i 的地址（0x2000），然后到 0x2000 至 0x2003 四个字节单元中取出 i 的值（3）。

通过变量名访问一个变量是直接的，而通过指针访问一个变量是间接的。就好像要在学生公寓中找一位学生，不知道他的姓名，也不知道他住哪一间房，但是知道 101 房间里有他的地址，走进 101 房间后看到一张字条：“找我请到 302”，这时按照字条上的地址到 302 去，便顺利地找到了他。这个 101 房间，就相当于一个指针变量，字条上的字便是指针变量中存放的内容（另一个内存单元的地址），而住在 302 房间的学生便是指针所指向的内容。

6．1．2　指针的定义和使用

如前所述，变量的指针就是变量的地址。存放变量地址的变量是指针变量，用来指向另一个变量。为了表示指针变量和它所指向的变量之间的联系，在程序中用“*”符号表示“指向”。符号“*”称为指针运算符。

1．指针变量的定义

C++语言规定所有变量在使用前必须定义，指定其类型，并按此分配内存单元。指针变量不同于整型变量和其他类型的变量，它是用来专门存放地址的，必须将它定义为“指针类型”。

定义指针变量的一般形式为：

基类型　　* 指针变量名

例如：

```
int      *i_pointer;
float    *f_pointer;
```

在定义指针变量时要注意两点：

（1）指针变量前面的“*”表示该变量的类型为指针型变量。注意：指针变量名是i_pointer1、 i_pointer2，而不是* i_pointer1、* i_pointer2。

（2）在定义指针变量时必须指定基类型。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。既然指针变量是存放地址的，那么只需要指定其为“指针型变量”即可，为什么还要指定基类型呢？我们知道整型数据和实型数据在内存中所占的字节数是不相同的（前者为 2 字节或 4 字节，后者为 4 字节或 8 字节），在后面将要介绍指针的移动和指针的运算（加、减），例如“使指针移动 1 个位置”或“使指针值加 1”，这个“1”代表什么呢？如果指针是指向一个字符型变量的，那么“使指针移动 1 个位置”意味着移动 1 个字节单元，“使指针加 1”意味着使地址值加 1。如果指针是指向一个整型变量的，则增加的不是 1 而是 4。因此必须规定指针变量所指向的变量的类型，即基类型。一个指针变量只能指向同一个类型的变量。不能一会儿指向一个整型变量，一会儿又指向一个实型变量。

例如，用 int　* i_pointer；定义了指针变量 i_pointer，它的基类型为整型，只有整型变量的地址才能放到这个指向整型变量的指针变量 i_pointer 中。即指针变量 i_pointer 可以用来指向整型变量，但不能指向实型变量。

在定义一个指针之后，一般要使指针能有明确的指向。与常规的变量未赋值相同，没有明确指向的指针不会引起编译器出错，但是对于指针可能导致无法预料的或者隐藏的灾难性后果。因此，在使用指针之前，一定要先使指针有明确的指向。那么，怎样使一个指针变量指向另一个变量呢?可以使用如下的赋值语句。例如：

```
int   * i_pointer1，* i_pointer2；
int   i，j；
i_pointer1=&i；
i_pointer2=&j；
```

其中，“&”是地址运算符，&i 就是取变量 i 在内存中的地址。

上面的语句将变量 i 的地址存放到指针变量 i_pointer1 中，因此* i_pointer1 就“指向”了变量 i。如图 6-2 所示。

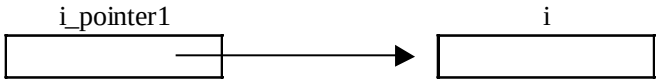

图 6-2　指针变量和“指向”的变量

同样，将变量 j 的地址存放到指针变量 i_pointer2 中，因此 i_pointer2 就“指向”了变量 j。

需要注意的是，指针变量中只能存放地址（指针），不能将一个整型量（或任何其他非地址类型的数据）赋给一个指针变量。例如下面的赋值是不合法的：

```
int *p;
p=200;（p 为指针变量，200 为整数）。
```

在定义指针变量时，可以给该指针变量赋初值。

例如， int　* i_pointer1=&i;

它等价于：

```
int　i;
int　* i_pointer1;
i_pointer1=&i;
```

2．指针变量的引用

在定义了指针变量并明确了它的指向后，就可以使用指针变量了。利用指针变量，提供了一种对变量的间接访问形式。对指针变量的引用形式为：

```
*指针变量
```

其含义是指针变量所指向的值。

例 6-1　通过指针变量访问整型变量。

```
#include <iostream>
using namespace std;
int main()
{
   int *i_pointer;
   int i;
   i_pointer=&i;
   *i_pointer=3;
   cout<<i<<endl;
   return 0;
}
```

程序的运行结果为：

3

在程序中，i_pointer 代表指针变量，它指向整型变量 i，而* i_pointer 是 i_pointer 所指向的变量。可以看到，*i_pointer 和变量 i 是同一回事。赋值语句“* i_pointer=3；”的含意是将 3 赋给指针变量 i_pointer 所指向的变量。它和语句“ i=3；”的作用相同。

在程序中有两个有关的运算符：

（1）&：取地址运算符。

（2）* ：指针运算符（或称“指向”运算符）。

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

下面对“&”和“*”运算符再做些说明：

假设执行了如下语句：

```
int   * pointer_1，* pointer_2；
int   i，j；
pointer_1=&i ；
pointer_2=&j ；
```

（1）若有& * pointer_1，它的含义是什么?

“&”和“*”两个运算符的优先级别相同，但按自右而左方向结合，因此先进行* pointer_1 的运算，它就是变量 i，再执行&运算。因此，&* pointer_1 与&i 相同，即变量 i 的地址。

如果有　　pointer_2=&* pointer_1；

它的作用是将&i（i 的地址）赋给 pointer_2，pointer_2 原来指向 j，经过重新赋值后，它已不再指向 j 了，而也指向了 i。

（2）*&i 的含义是什么?

先进行&i 运算，得 i 的地址，再进行*运算。即&i 所指向的变量，*&i 和* pointer_1 的作用是一样的，它们等价于变量 i。即*&a 与 i 等价。

（3）（* pointer_1）++相当于 i++。注意括号是必要的，如果没有括号，就成为了* pointer_1++，由于++和*为同一优先级别，而结合方向为自右而左，因此它相当于*（pointer_1++）。由于++在 pointer_1 的右侧，是“后加”，因此先对 pointer_1 的原值进行*运算，得到 i 的值，然后使 pointer_1 的值改变，这样 pointer_1 不再指向 i 了。

例 6-2 输入 a 和 b 两个整数，按大小顺序排列输出。

```
#include <iostream>
using namespace std;
int main()
{
    int *p1, *p2;
    int a,b,temp;
    p1=&a;
    p2=&b;
    cin>>a>>b;
    if (*p1>*p2)
    {
```

```
        temp=*p1;
        *p1=*p2;
        *p2=temp;
    }
    cout<< *p1 << "    "<< *p2 << endl;
    return 0;
}
```

程序运行情况如下：

```
8   4   ↙
4   8
```

在程序中，当执行赋值操作 p1 = &a 和 p2 = &b 后，指针变量 p1 和 p2 就指向了变量 a 与 b，这时引用指针 * p1 与* p2，就代表了变量 a 与 b。在程序运行过程中，指针变量与所指的变量之间的关系如图 6-3 所示。

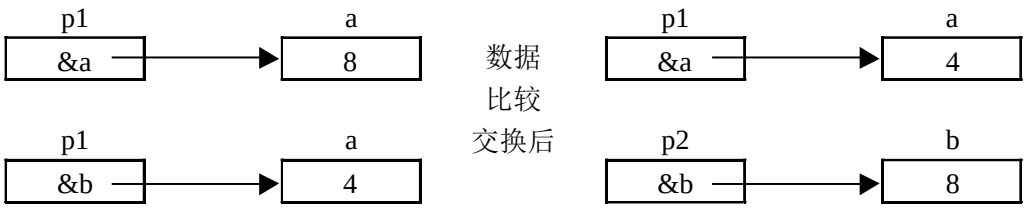

图 6-3　指针变量与所指的变量之间的关系

在程序的运行过程中，指针变量的值没有改变，但其所指向的变量的值发生了没变。下面将上述程序作如下修改：

```
#include <iostream >
using namespace std;
int main()
{
   int *p1, *p2,*p;
   int a,b,temp;
   p1=&a;
   p2=&b;
   cin>>a>>b;
   if (*p1>*p2)
   {
        p=p1;
        p1=p2;
        p2=p;
   }
   cout<< *p1 << "    "<< *p2 << endl;
```

```
    return 0;
}
```

程序运行情况如下：

```
8  4  ↙
4  8
```

程序的运行结果完全相同，但程序在运行过程中，a 和 b 并未交换、它们仍保持原值，但 p1 和 p2 的值改变了。p1 的值原为&a，后来变成&b，p2 原值为&b，后来变成&a。如图 6-4 所示。这样在输出*p1 和*p2 时，实际上是输出变量 b 和 a 的值，所以先输出 4，然后输出 8。

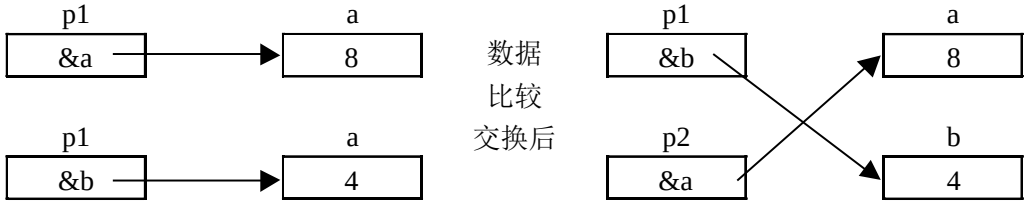

图 6-4　程序修改后，指针变量与所指的变量之间的关系

3．指针变量作函数的参数

函数的参数不仅可以是整型、实型、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到另一个函数中。由于被调用函数中获得了所传递变量的地址，因此在该地址空间的数据当函数调用结束后被物理地保留下来。

例 6-3　利用指针变量作为函数的参数，用函数调用的方法实现例 6-2 程序的功能。

```
#include <iostream >
using namespace std;
void change(int *pt1,int *pt2);
int main()
{
   int *p1, *p2;
   int a,b;
      p1=&a;
   p2=&b;
   cin>>a>>b;
   if (*p1>*p2) change(p1,p2);
   cout<< *p1 << "    "<< *p2 << endl;
   return 0;
}
void change(int *pt1,int *pt2)
{
   int temp;
```

```
    temp=*pt1;
    *pt1=*pt2;
    *pt2=temp;
}
```

由于在调用函数时，实际参数是指针变量 p1 和 p2，形式参数也是指针变量，实参与形参相结合，传值调用将指针变量 p1 和 p2 传递给形式参数 pt1 和 pt2。但此时传值传递的是变量地址，使得在被调用函数中 pt1 和 pt2 具有了 p1 和 p2 的值，指向了与调用程序相同的内存变量 a 和 b，并对其在内存存放的数据进行了交换，其执行结果与例 6-2 相同。

思考下面的程序，是否也能达到相同的效果呢？

```
#include <iostream >
using namespace std;
void change(int *pt1,int *pt2);
int main()
{
   int *p1, *p2;
   int a,b;
      p1=&a;
   p2=&b;
   cin>>a>>b;
   if (*p1>*p2) change(p1,p2);
   cout<< *p1 << "    "<< *p2 << endl;
   return 0;
}
void change(int *pt1,int *pt2)
{
   int *p;
   p=pt1;
   pt1=pt2;
   pt2=p;
}
```

程序运行情况如下：

```
8   4   ↙
8   4
```

以看到，程序运行结束后，并未达到预期的结果，输出与输入完全相同。其原因是对函数 change（）来说，函数内部进行指针相互交换指向（即形参 pt1 和 pt2 交换指向），而它们所指向的变量在内存存放的数据并未交换；由于 C++语言中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。因此函数调用结束后，main（）函数中实参 p1 和 p2 保持原指向，所以结果与输入相同。

而前面的程序可以成功实现是因为调用函数虽然不可能改变实参指针变量的值，但可

以改变实参指针变量所指变量的值。

因此，为了使在函数中改变了的变量值能被 main 函数使用，可以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（如 main 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：①在主调函数中设 n 个变量，用 n 个指针变量指向它们；②然后将指针变量作实参，将这 n 个变量的地址传给所调用的函数的形参；②通过形参指针变量，改变该 n 个变量的值；④主调函数中就可以使用这些改变了值的变量。

6．1．3　指针与数组

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和数组元素（把数组起始地址或某一元素的地址放到一个指针变量中）。所谓数组的指针是指数组的起始地址，数组元素的指针是数组元素的地址。

1．指向一维数组的指针

假设我们定义一个一维数组，该数组在内存会由系统分配一个存储空间，其数组的名字就是数组在内存的首地址。若再定义一个指针变量，并将数组的首地址赋给指针变量，则该指针变量就指向了这个一维数组。对一维数组的引用，既可以用传统的数组元素的下标法（如 a[i]），也可使用指针的表示方法，即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（占内存少，运行速度快）。

定义一个指向数组元素的指针变量的方法，与前面介绍的指向变量的指针变量相同。

例如：

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

应当注意，如果数组 a 为 int 型，则指针变量也应是指向 int 型的。下面是对该指针变量赋值：

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

把 a[0]元素的地址赋给指针变量 p。也就是说，p 指向 a 数组的第 0 个元素。

C++语言规定数组名代表数组的首地址，也就是第 0 个元素的地址。因此，下面两个语句等价：

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

“p＝a”的作用是“把 a 数组的首地址赋给指针变量 p”，而不是“把数组 a 各元素的值赋给 p”。

假设 p 是一个指针变量，并已给它赋了一个地址，使它指向某一个数组元素。*p 就表示 p 当前所指向的数组元素。

按 C++的规定：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（而不是将 p 值简单地加 1）。例如，数组元素是实型，每个元素占 4 个字节，

则 p+1 意味着使 p 的值（地址）加 4，以使它指向下一个元素。p+1 所代表的地址实际上是 p+1*d，d 是一个数组元素所占的字节数（对短整型，d=2；对整型，d=4；对 float 实型，d＝4；对字符型，d=1）。

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。这里需要说明的是 a 代表数组首地址，a+i 也是地址，它的计算方法同 p+i。例如 p+5 和 a+5 的值是&a[5]，它指向 a[5]。

（2）*（P+i）或*（a+i）是 p+i 或 a+i 所指向的数组元素，即 a[i]。例如，*(P+5)或*(a+5)就是 a[5]。实际上，在编译时，对数组元素 a[i]就是处理成*（a+i），即按数组首地址加上相对位移量得到要找的元素的地址，然后找出该单元中的内容。

（3）指向数组的指针变量也可以带下标，如 p[i]与*（p+i）等价。

有了指向一维数组的指针后，对数组中元素的引用可以有两种方法：

（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 是指向数组的指针变量，其初值 p=a。

用下标法比较直观，能直接知道是第几个元素。例如，a[5]是数组中序号为 5 的元素（序号从 0 算起）。但 C++编译系统是将 a[i]转换为*(a+i)处理的，即先计算元素地址。因此采用下标法寻找数组元素费时较多（先要计算元素地址）。

用指针变量的方法不直观，难以很快地判断出当前处理的是哪一个元素。要仔细分析指针变量 p 的当前指向，才能判断当前引用的是第几个元素。但是，用指针变量直接指向元素，在循环引用数组中的元素值时，不必每次都重新计算地址（如下例所示），而且像 p++这样的自加操作是比较快的。这种有规律地改变地址值（p++）能大大提高执行效率。

例 6-4　输入 10 个整数后，输出。

```
#include <iostream >
using namespace std;
int main()
{
   int a[10],i;
   int *p;
   for (i=0;i<10;i++)
         cin>>a[i];
   for (p=a;p<a+10;p++)
         cout<<*p<<"    ";
   cout<<endl;
   return 0;
}
```

在使用指针变量时，有几个问题要注意：

（1）指针变量可以实现使本身的值改变。例如，指针变量 p 来指向元素，用 p++使 p 的值不断改变，这是合法的。如果不用 p 而使 a 变化（例如，用 a++）行不行呢?假如将上

述程序的最后两行改为：

```
for (p=a;a<p+10;a++)
    cout<<*a<<"    ";
```

是不行的。因为 a 是数组名，它是数组首地址常量。a++是什么意思呢？这是无法实现的。

（2）注意指针变量的运算。如果先使 p 指向数组 a（即 p=a），则：

① p++，使 p 指向下一元素，即 a[1]。若再执行*p，取出下一个元素 a[1]的值。

② *p++，由于++和*同优先级，结合方向为由右而左，因此它等价于*（p++），作用是先得到 p 指向的数组元素的值（即*p），然后再使 p 自加 1，指向下一个数组元素。

③ *（p++）与*（++p）作用不同。前者是先取*p 值，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++）时，得 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，即元素值加 1，而不是指针值加 1。

将++和--运算符用于指针变量十分有效，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。

2．指向多维数组的指针

用指针变量可以指向一维数组，也可以指向多维数组。但是指向多维数组的指针要复杂得多。下面我们以二维数组为例，说明一下多维数组的指针。

（1）二维数组的地址

设有一个二维数组的定义为：

```
int  a[3][4]={{1，2，3，4}，{5，6，7，8}，{9，10，11，12}}；
```

a 是一个数组名。可以将 a 数组看成是一个一维数组，它包含 3 个元素：a[0]，a[1]，a[2]，而每一元素又是一个一维数组，它又包含 4 个元素，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]，如图 6-5 所示。

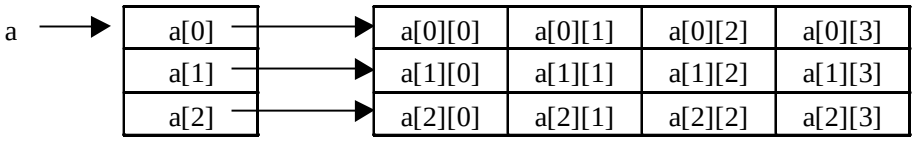

图 6-5　二维数组的表示

从二维数组的角度来看，a 代表整个二维数组的首地址，也就是第 0 行的首地址，那么 a+1 就代表第 1 行的首地址（即&a[1]）。

a 是二维数组，a[0]、a[1]、a[2]可以看成是一维数组名，而 C++语言规定了数组名代表数组的首地址，因此 a[0]代表第 0 行一维数组中第 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]。即 a[i]本身并不占实际的内存单元，它也不存放 a 数组中各个元素的值，它只是一个地址（如同一个一维数组名 x 并不占内存单元而只代表地址一样）。

正如有一个一维数组 x，x+l 是元素 x[1]的地址一样。对于二维数组 a，a[0]是一个一维

数组名，则 a[0]+1 是第 0 行第 1 列元素（即 a[0][1]）的地址（&a[0][1]）。

对于一维数组 x 来说，x[0]和*（x+0）等价，x[i]和*（x+i）等价。因此，对于二维数组 a 来说，a[0]和*（a+0）等价，它们的值都是＆a[0][0]，a[i]和*（a+i）等价，它们的值都是＆a[i][0]。a[0]+1 和 *（a+0）+1 等价，它们的值都是&a[0][1]，a[i]+j 和 *（a+i）+j 等价，它们的值都是&a[i][j]。

既然 a[0]和*（a+0）的值都＆a[0][0]，那么*（a[0]）和*（*（a+0））就是 a[0][0]。同理，*（a[i]+j）和*（*（a+i）+j）就是 a[i][j]。

我们将与二维数组相关的地址表示形式列举如下表 6-1 所示。

表 6-1　二维数组 a 相关的地址

表 示 形 式	含　义
a	二维数组名、数组首地址、第 0 行首地址
a[0]、*（a+0）、*a	第 0 行第 0 列元素地址
a+i、&a[i]	第 i 行首地址
a[i]、*（a+i）、&a[i][0]	第 i 行第 0 列元素地址
a[i]+j、*（a+i）+j、&a[i][j]	第 i 行第 j 列元素地址
（a[i]+j）、(*(a+i)+j)、a[i][j]	第 i 行第 j 列元素的值

（2）指向二维数组的指针变量

在了解上面有关二维数组的地址表示后，可以用指针变量指向二维数组及其元素。

- 指向数组元素的指针变量。

例 6-5　用指针变量输出二维数组的数组元素的值。

```
#include <iostream >
using namespace std;
int main()
{
   int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
   int i,*p;
   p=&a[0][0];
   for (i=0;i<12;i++)
   {
        cout.width(4);
        cout<<*p;
        p++;
        if (i%4==3) cout<<endl;
   }
   return 0;
}
```

程序中 p 是一个指向整型变量的指针变量，它指向整型二维数组的数组元素。每次使 p 值加 1，以移向下一个元素。

例 6-5 是顺序输出数组中各元素之值，比较简单。如果要输出某个指定的数组元素（例如 a[2][3]），则应事先计算该元素在数组中的相对位置（即相对于数组起始位置的相对位移量）。

设有二维数组 a[n][m]，在 a[i][j]元素之前有 i 行元素（每行有 m 个元素），在 a[i][j]所在行中，a[i][j]的前面还有 j 个元素，即 a[i][j]之前共有 i* m+j 个元素。因此，计算 a[i][j]在数组中的相对位置的计算公式为

i*m+j

例如，对例 6-5 中的二维数组，它的第 2 行第 3 列元素(a[2][3])对 a[0][0]的相对位置为 2*4+3＝11。如果开始时使指针变量 p 指向 a（即 a[0][0]），为了得到 a[2][3]的值，可以用*(p+2*4+3)表示，即(p+11)是 a[2][3]的地址。

- 指向由 m 个元素组成的一维数组的指针变量

上例的指针变量 p 是指向整型变量的，p+1 所指向的元素是 p 所指向的元素的下一元素。可以改用另一方法，使 p 不是指向整型变量，而是指向一个包含 m 个元素的一维数组。这时，如果 p 先指向 a[0]（即 p＝＆a[0]），则 p+l 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。

定义指向由 m 个元素组成的一维数组的指针变量的方法为：

类型标识 （*指针变量名）[数组长度]；

例如，“ int (*p)[4]；”表示 p 是一个指针变量，它指向包含 4 个元素的一维数组。需要注意的是，p 两侧的括号不可缺少，如果写成*p[4]，由于方括号[ ]运算级别高，因此 p 先与[4]结合，是数组，然后再与前面的*结合，*p[4]是指针数组（指针数组下面再介绍）。

定义了“int (*p)[4]；”后，p 所指的对象是有 4 个整型元素的数组，即 p 是行指针，如图 6-6 所示，此时 p 只能指向一个包含 4 个元素的一维数组，p 的值就是该一维数组的首地址。p 不能指向一维数组中的第 j 个元素。

设有一个二维数组 int a[3][4]，且定义一个指向 4 个元素组成的一维数组的指针变量 p，则执行 p=a（或 p=&a[0]）后，p 就指向了二维数组 a 的第 0 行，p+1 就指向了第 1 行，p+2 就指向了第 2 行，如图 6-6 所示。

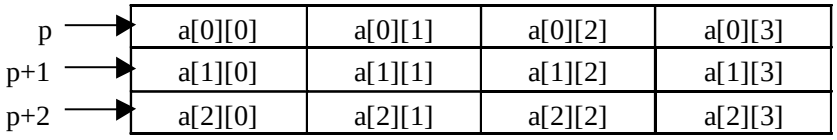

图 6-6 指向 4 个元素组成的一维数组的指针变量 p

此时，如果要访问 a[2][3]元素，用指针变量 p 可以表示为 *（*（p+2）+3）。

例 6-6 有 3 个学生，每人学 4 门功课，求每个学生的平均成绩。

```
#include <iostream >
using namespace std;
int main()
{
   int a[3][4]={{75,88,93,85},{58,76,65,80},{75,85,90,87}};
```

```
    int (*p)[4],i,j,sum;
    float aver;
    p=a;
    for (i=0;i<3;i++)
    {
        sum=0;
        for (j=0;j<4;j++)
        {
            sum+= *(*(p+i)+j);
        }
        aver=(float)sum/4.0;
        cout<<"第 "<<i+1<<" 个学生的平均成绩为 "<<aver<<endl;
    }
    return 0;
}
```

3．指向字符串的指针

在第 4 章中，我们学过用字符数组来存放一个字符串，通过数组名来表示字符串，数组名就是数组的首地址，是字符串内存存放的起始地址。

我们也可以不定义字符数组，而定义一个字符指针。用字符指针指向字符串中的字符。

例如：

```
char   *str=”I am a boy!”;
```

在这里没有定义字符数组，而是定义了一个字符指针变量 str。给定一个字符串常量“I am a boy!”，C++语言对字符串常量是按字符数组处理的，在内存开辟了一个字符数组用来存放字符串常量。在定义字符指针变量 str 时把字符串首地址（即存放字符串的字符数组的首地址）赋给 str。

因此，

```
char   *str=”I am a boy!”;
```

等价于下面两行：

```
char   *str;
str=”I am a boy!”;
```

可以看到 str 被定义为一个指针变量，指向字符型数据，请注意它只能指向一个字符变量或其他字符类型数据，不能同时指向多个字符数据，更不是把“I am a boy!”这些字符存放到 str 中（指针变量只能存放地址），也不是把字符串赋给*str，只是把“I am a boy!”的首地址赋给指针变量 str。即不要认为上述定义行等价于

```
char   *str;
*str = ”I am a boy!”;
```

我们也可以将字符数组的名赋予一个指向字符类型的指针变量，让字符类型指针指向字符串在内存的首地址，这样对字符串的表示就可以用指针实现。

例如：

```
char   str[20];
char   *p = str;
```

这样一来，字符串 str 就可以用指针变量 p 来表示了。

例 6-7　用指针变量实现字符串的复制。

```
#include <iostream >
using namespace std;
int main()
{
   char str1[12]="I am a boy!",str2[20];
      char *a,*b;
   for (a=str1,b=str2;*a!='\0';a++,b++)
   {
         *b=*a;
   }
   *b='\0';
   cout<<"String 1 is "<<str1<<endl;
   cout<<"String 2 is "<<str2<<endl;
   return 0;
}
```

虽然用字符数组和字符指针变量都能实现字符串的存储和运算，但它们二者之间是有区别的，不应混为一谈，主要有以下几点：

（1）字符数组由若干个元素组成，每个元素中存放一个字符，而字符指针变量中存放的是地址（字符串的首地址），不是将字符串放到字符指针变量中。

（2）赋值方式不同。对字符数组只能对各个元素赋值，不能用以下办法对字符数组赋值。

```
char   str[12];
str=“I am a boy！”;
```

而对字符指针变量，可以采用下面方法赋值：

```
char   *a;
a=“I am a boy！”;
```

这是因为“I am a boy！”是一个字符串常量，当编译器遇到一个字符串常量时，就将其存放在内存 data 区的 const 区中，以'\0'作为结束符，并记下其起始地址，在所生成的代码中使用该地址，这样，字符串常量就“变成”了地址。

因此，语句　a=“I am a boy！”；　实际上是将字符串的首地址符给一个指针变量 a。而 str 是数组名，虽然也是地址，但它是数组首地址常量，显然不能用另外的一个地址来修改它，因此语句　str=“I am a boy！”；　是错误的。

（3）如果定义了一个字符数组，在编译时为它分配内存单元，它有确定的地址。而定

义一个字符指针变量时，给指针变量分配内存单元，在其中可以放一个地址值，但如果未对它赋予一个地址值，则它并未具体指向一个确定的字符数据。

如果将例 6-7 改写为如下程序：

```
#include <stdio.h>
int main()
{
   char *str="I am a boy!";
   char   *a;
   for ( ;*str!='\0';str++,a++)
   {
        *a=*str;
   }
   *a='\0';
   printf("String 1 is %s\n",str);
   printf("String 2 is %s\n",a);
   return 0;
}
```

虽然编译时没有错误，但这种方法是危险的，不宜提倡。因为编译时虽然给指针变量 a 分配了内存单元，但 a 的值并未指定，在 a 单元中是一个不可预料的值。程序在执行时将一个字符串 str 复制到 a 所指向的一段内存单元中，即以 a 的值（地址）开始的一段内存单元中。而 a 的值由于没有赋值，它是不可预料的，它可能指向内存中空白的（未用的）用户存储区中（这是好的情况），也有可能指向已存放指令或数据的有用内存段，这就会破坏了程序，甚至破坏了系统，会造成严重的后果。在程序规模小时，由于空白地带多，往往可以正常运行，而程序规模大时，出现上述“冲突”的可能性就大多了。因此，还是采用例 6-7 的方法较好，即

```
char   *a，str[20]；
a=str；
```

先使 a 有确定值，也就是使 a 指向一个字符数组的开头。

4．指针数组

一个数组，如果其元素均为指针类型数据，称为指针数组，也就是说，指针数组中的每一个元素都相当于一个指针变量。一维指针数组的定义形式为：

```
类型标识   *数组名[数组长度]；
```

例如： char *str[4]；

由于[ ] 比*优先权高，所以首先是数组形式 str[4]，然后才是与“*”的结合。这样一来指针数组包含 4 个指针 str [0]、str[1]、str[2]、str[3]，各自指向字符类型的变量。

为什么要定义和使用指针数组呢?主要是由于指针数组可以用来指向若干个字符串，使字符串处理更加方便灵活。

例如，图书馆有若干本书，想把书名放在一个数组中，然后再对这些书目进行排序和

查询。按一般方法，字符串本身就是一个字符数组。因此要设计一个二维的字符数组才能存放多个字符串。但在定义二维数组时，需要指定列数，也就是说二维数组中每一行中包含的元素个数（即列数）相等。而实际上各字符串（书名）长度一般是不相等的。如果按最长的字符串来定义列数，则会浪费许多内存单元。

可以分别定义一些字符串，然后用指针数组中的元素分别指向各字符串。如果想对字符串排序，不必改动字符串的位置，只需改动指针数组中各元素的指向（即改变各元素的值，这些值是各字符串的首地址）。这样，各字符串的长度可以不同，而且移动指针变量的值（地址）要比移动字符串所花的时间少得多。

例 6-8　将若干本图书按书名从小到大顺序输出。

```
#include <string.h>
#include <iostream >
using namespace std;
void str_sort(char *name[],int n);
int main()
{
    char *book[5]={"飞狐外传","雪山飞狐","连城诀","天龙八部","射雕英雄传"};
    int i;
    str_sort(book,5);
    for (i=0;i<5;i++)
        cout<<book[i]<<endl;
    return 0;
}
void str_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;
        }
    }
}
```

在程序中定义指针数组 book。它有 5 个元素，其初值分别是“飞狐外传”、“雪山飞狐”、“连城诀”、“天龙八部”、“射雕英雄传”的首地址，如图 6-7（a）所示。这些字符串是不等长的（并不是按同一长度定义的）。sort 函数的作用是对字符串排序。sort 函数的形参 name 也是指针数组名，接受实参传过来的 book 数组的首地址，因此形参 name 数组和实参 book 数组指的是同一数组。用选择法对字符串排序。strcmp 是字符串比较函数，其函数原型为：

int strcmp（const char *string1，const char *string2）；

strcmp 函数的形参 string1 和 string2 是两个指向字符串常量的字符指针，之所以使用 const 修饰符，是因为在对两个字符串进行比较时，不允许对参加比较的字符串本身进行修改。

执行完 sort 函数后，指针数组的情况如图 6-7（b）所示。

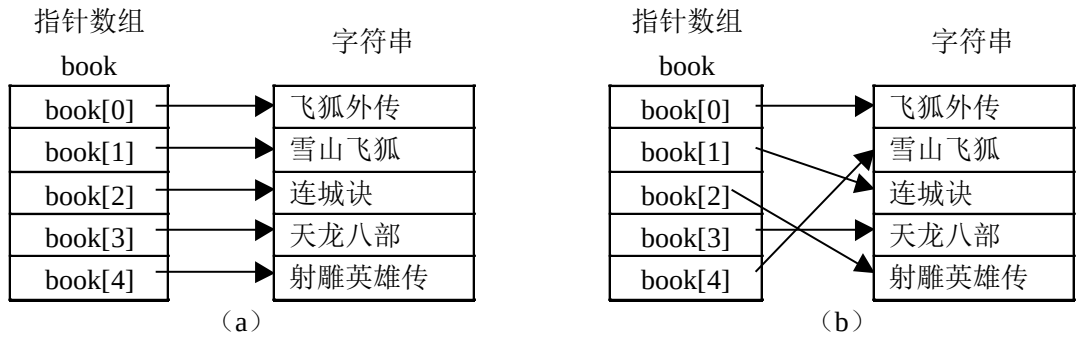

图 6-7　指向字符串的指针数组

5．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，简称维指向指针的指针。从图 6-7 可以看出，book 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 book 代表该指针数组首元素的地址，book+i 是 book[i]的地址。由于 book[i]的值是地址（即指针），因此 book+i 就是指向指针型数据的指针。可以设置一个指针变量 p，它指向指针数组的元素，p 就是指向指针型数据的指针变量。

定义一个指向指针数据的指针变量的一般形式为：

基类型　　* * 指针变量名

例如，有如下程序：

```
#include <iostream>
using namespace std;
int main()
{
    char *book[5]={"Mastering Visual C++ 6.0","The C++ Programming Language",
                   "C++ Primer","C++ Program Design","Object Oriented Methods"};
    char **p;       // 定义指向字符指针数据的指针变量 p
```

```
    p=book+1;
    cout<<*p<<endl;
    cout<<**p<<endl;
    return 0;
}
```

程序运行结果为：

```
The C++ Programming Language
T
```

在程序中，由于*p 代表 book[1]，它指向字符串“The C++ Programming Language”，因此“cout<<*p”就从第一个字符开始输出字符串“The C++ Programming Language”；而**p 是*p（值为 book[1]）指向的“The C++ Programming Language”第 1 个字符元素的内容，即“T”，因此“cout<<**p”输出字符“T”。

6．1．4　指针与结构体

可以设一个指针变量用来指向一个结构体变量，此时该指针变量的值是结构体变量在内存存放的起始地址。指针变量也可以用来指向结构体数组中的元素。

1．指向结构体变量的指针

与定义指向其它类型变量的指针一样，可以定义一个指向结构体变量的指针。

例如：

```
struct   Student
{
  int num；
  char   name[20]；
  char   sex；
  int age；
}
Student   stu_1；
Student   *p；
```

然后可以让指针变量 p 指向一个结构体变量：

```
    p=&stu_1；
```

这样 p 就指向了变量 stu_1。我们就可以通过指针 p 来引用变量 stu_1 各成员的值，使用的是（*p）.num 这样的形式。（*p）表示 p 指向的结构体变量，（*P）.num 是 p 指向的结构体变量中的成员 num。注意*p 两侧的括弧不可省，因为成员运算符“．”优先于“*”运算符，*p.num 就等价于*（p.num）了。

为了使用方便和直观，可以把（*p）．num 改用 p->num 来代替，它表示*p 所指向的结构体变量中的 num 成员。同样，(*p)．name 等价于 p->name。

也就是说，以下三种形式等价：

① 结构体变量．成员名

② (*p)．成员名

③ p->成员名

其中->称为指向运算符。

例 6-9　指向结构体变量的指针的使用。

```
#include <iostream >
using namespace std;
struct Student
{
   int num;
   char name[20];
   char sex;
   int age;
};
int main()
{
   Student stu1={1001,"ZhangSan",'M',21};
   Student *p;
   p=&stu1;
   cout<< p->num <<endl;
   cout<< p->name <<endl;
   if (p->sex=='M') cout<<"男" << endl ;
         else    cout << "女" <<endl ;
   cout << p->age<< endl;
   return 0;
}
```

2．链表

用数组存放数据时，必须事先定义固定的长度（即数组元素的个数）。比如，要保存一个班级的学生某一门课程的成绩，可以采用一维数组，但有的班级有可能有 100 人，而有的班可能只有 30 人，如果要用同一个数组先后存放不同班级的学生成绩，则必须定义长度为 100 的数组。如果事先难以确定一个班的最多人数，则必须把数组定得足够大，以便能存放任何班级的学生数据。显然这将会浪费内存。因此，在有些情况下，要根据需要开辟内存单元来保存数据，链表就是一种可以动态地进行存储分配的数据结构。

（1）链表概述

链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一为存储数据元素的数据域，二为存储下一个结点地址的指针域。可以将结点结构定义为：

```
struct Node {
   Elemtype data；
   Node *next；
```

}

其中，Elemtype 可以是基本数据类型（如整型、实型或字符型），也可以是我们自定义的数据类型（如自定义的一个结构体类型 Student）。

由此可见，链表这种数据结构必须利用指针变量才能实现，指针作为维系结点的纽带，可以通过它实现链式存储。假设有五个学生某一门功课的成绩分别为 A、B、C、D 和 E，这五个数据在内存中的存储单元地址分别为 1248、1488、1366、1022 和 1520，其链表结构如图 6-8 所示。

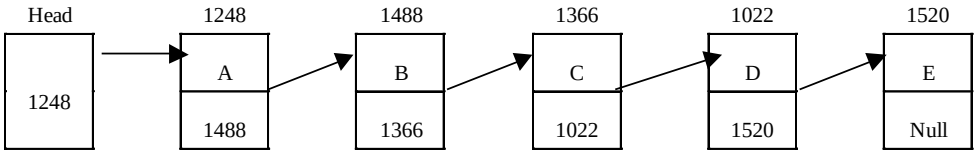

图 6-8　链表示意图

链表有一个“头指针”变量，图 6-8 中以 head 表示，它存放一个地址，该地址指向链表中第一个结点，第一个结点又指向第二个结点……直到最后一个结点，该结点不再指向其他结点，它称为“表尾”，它的地址部分存放一个“NULL”（表示“空地址”），链表到此结束。链表中每个结点都包括两个部分：用户需要用的实际数据和下一个结点的地址。

可以看到链表中各结点在内存中可以不是连续存放的。要找到某一结点 C，必须先找到它的上一个结点 B，根据结点 B 提供的下一结点地址才能找到 C。链表有一个“头指针”，因此通过“头指针”可以顺序往下找到链表中的任一结点，如果不提供“头指针”，则整个链表都无法访问。链表如同一条铁链一样，一环扣一环，中间是不能断开的。打个比方，幼儿园的老师带领孩子出来散步，老师（作为头指针）牵着第一个小孩的手，第一个小孩的另一只手牵着第二个孩子……这就是一个“链”，最后一个孩子有一只手空着，他是“链尾”。要找这个队伍，必须先找到老师，然后顺序找到每一个孩子。

图 6-8 的链表每个结点中只有一个指向后继结点的指针，该链表称为单链表。

（2）new 和 delete 运算符

链表结构是动态地分配存储的，即在需要时才分配一个结点的存储单元。怎样动态地分配和释放存储单元呢？C++使用运算符 new 和 delete 进行内存的分配和释放。

运算符 new 用于内存分配的使用形式为：

```
p=new  type;
```

其中，type 是一个数据类型名，p 是指向该数据类型的指针。New 从称为堆的一块自由存储区中为程序分配一块 sizeof（type）字节大小的内存，该块内存的首地址被存于指针 p 中。

使用 new 动态分配内存时，如果没有足够的内存满足分配要求，new 将返回空指针（NULL），可以在程序中对内存的动态分配是否成功进行检查。

运算符 delete 用于释放 new 分配的存储空间。它的使用形式一般为：

```
delete  p;
```

其中，p 必须是一个指针，保存着 new 分配的内存的首地址。

例 6-10　内存的动态分配

```
#include <iostream>
using namespace std;
int main()
{
   int *p;
   p=new int;
   *p=25;
   cout <<*p <<endl;
   delete p;
   return 0;
}
```

在程序中定义了一个整型指针变量 p，然后用 new 为其分配了一块内存，p 指向这个内存块，然后在这个内存块中赋值 25，并将其输出，最后撤消指针 p，释放 p 指向的存储空间。

（3）链表的建立和输出

建立链表的思想是：一个一个地动态生成结点并输入各结点数据，同时建立起各结点前后相链的关系。这种关系可以是正挂、倒挂或插入。

用正挂的方法建立单链表的过程有以下几步：

① 定义链表的数据结构。

② 创建一个空表。

③ 利用 new 运算符向系统申请分配一个节点。

④ 将新节点的指针成员赋值为空。若是空表，将新节点连接到表头；若是非空表，将新节点接到表尾。

⑤ 判断一下是否有后续节点要接入链表，若有转到③，否则结束。

例 6-11　创建一个存放正整数的单链表（输入数据以-1 结束）。

```
struct SLink
{
   int num;
   SLink *next;
};
SLink *create(SLink *head)
{
   SLink *p1,*p2;
   p1=p2=new SLink;
   cin>>p1->num;
   p1->next=NULL;
   while (p1->num!=-1)
   {
```

```
            if (head==NULL) head=p1;
            else    p2->next=p1;
            p2=p1;
            p1=new SLink;
            p1->next=NULL;
            cin>>p1->num;
        }
        return head;
    }
```

单链表的输出过程有以下几步：

① 找到表头。

② 若是非空表，输出节点的值成员，是空表则退出。

③ 找到下一个节点的地址。

④ 转到②。

例 6-12　将例 6-11 创建的单链表输出。

```
#include <iostream >
using namespace std;
struct SLink
{
    int num;
    SLink *next;
};
SLink *create(SLink *head);
void print(SLink *head);

int main()
{
     SLink *head;
     head=NULL;
     head=create(head);
     print(head);
     return 0;
}

void print(SLink *head)
{
    SLink *p;
    p=head;
    while (p!=NULL)
```

```
    {
        cout<<p->num<<endl;
        p=p->next;
    }
}
```

（4）链表的插入和删除

链表的一个重要特点是插入、删除操作灵活方便，不需移动结点，只需改变结点中指针域的值即可。下面我们分别讨论单链表的插入、删除操作实现。

设有一个存放整数的单链表，已存放有元素（a_1，a_2，…，a_n ），head 为该单链表的头指针。现要求在第 i 个数据元素 a_i 结点之前插入一个数值为 x 的结点。

要完成这一操作，首先需要在单链表中计数寻找到第 i-1 个结点并由指针 p 指示，然后为待插入元素 x 申请一个新的由指针 s 指示的结点空间，并置 x 为其数据域值，最后通过修改指针，重新建立 a_{i-1} 与 x、x 与 a_i 两两数据元素之间的链，从而实现三个元素之间链接关系的变化。x 插入时指针变化如图 6-9 所示，图中虚线所示为插入前的指针。

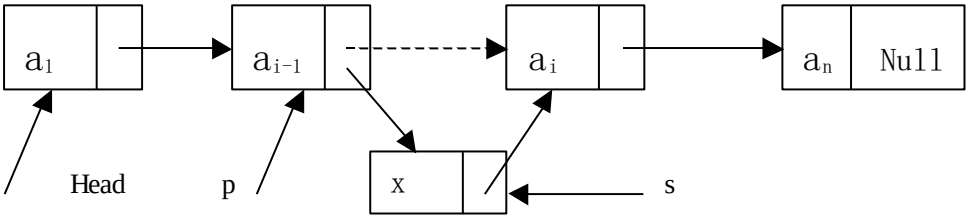

图 6-9　单链表的插入

例 6-13　编写一个函数，完成单链表的插入操作。

```
SLink *insert(SLink *head,int i,int x)
{
  SLink *p,*s;
  int k=1;
  p=head;
  while (p && k<i-1)
  {
      p=p->next;
      k++;
  }
  if (p==NULL||i<=0)
  {
      cout <<" 所给的插入位置 i 不合理，i<= 0 或 i>=n+1"<<endl;
      return NULL;
  }
  s=new SLink;
```

```
        s->num=x;
        if (i = =1)
        {
            s->next=head;
            head=s;
        }
        else
        {
            s->next=p->next;
            p->next=s;
        }
        return (head);
    }
```

若要在 head 为头指针的单链表（a_1，a_2，…，a_n ）中，删除第 i 个结点，首先搜索单链表以找到指定删除结点（即第 i 个结点，以指针 s 指示）的前驱结点（第 i-1 个结点，并以指针 p 指示），然后仅需修改结点 p 中的指针域，最后释放待删除结点 s 所占的存储空间。图 6-10 显示了删除时指针变化情况，虚线所示为删除前的指针。

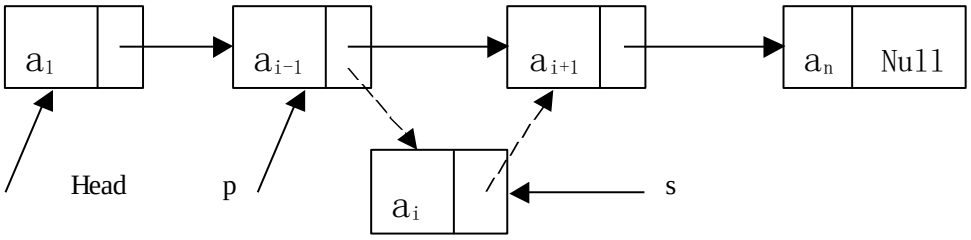

图 6-10　单链表的删除

例 6-13　编写一个函数，删除单链表中的一个结点。

```
SLink *delete_node(SLink *head,int i)
{
   SLink *p,*s;
   int k=1;
   p=head;
   while (p && k<i-1)
   {
       p=p->next;
       k++;
   }
   if (p==NULL || p->next==NULL || i<=0)
   {
```

```
            cout <<" 所给的删除位置 i 不合理，i<1 或 i>n"<<endl;
            return NULL;
        }
        s=p->next;
        if (i==1)
        {
            s=head;
            head=s->next;
        }
        else
        {
            s=p->next;
            p->next=s->next;
        }
        return (head);
    }
```

6．1．5　指针与函数

1．函数指针

每一个函数都占用一段内存单元，它们有一个起始地址。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。这个指向函数入口地址的指针称为函数指针。

指向函数的指针变量的一般定义形式为：

数据类型 （*指针变量名）（参数表）；

这里的“数据类型”是指函数返回值的类型。

例如：

int (*p)(int a，int b)；

定义 p 是一个指向函数的指针变量，它所指向的函数带回整型的返回值，并且带有两个整型参数。注意 *p 两侧的括弧不可省略，表示 p 先与 * 结合，是指针变量，然后再与后面的()结合，表示此指针变量指向函数，这个函数的返回值是整型的。

如果写成“int *p(int a，int b)；”，则由于（）优先级高于*，它就成了声明一个函数了（这个函数的返回值是指向整型变量的指针）。

上面定义的一个指向函数的指针变量 p，它不是固定指向哪一个函数的，而只是表示定义了这样一个类型的变量，它是专门用来存放函数的入口地址的。在程序中把哪一个函数的地址赋给它，它就指向哪一个函数。

和数组名代表数组起始地址一样，函数名代表该函数的入口地址。因此，在给函数指针变量赋值时，只需给出函数名而不必给出参数，如：

有一个函数 max 的原型为：

int max（int x，int y）；

则 p=max；的作用是将函数 max 的入口地址赋给指针变量 p。这时，p 就是指向函数

max 的指针变量，也就是 p 和 max 都指向函数的开头。

因为是将函数的入口地址赋给 p，而不牵涉到实参与形参的结合问题。因此不能写成“p=max（a，b）；”的形式。如果写成“p=max（a，b）；”，函数名 max 加上括号及参数就变成了函数调用，它的返回值是整型数，而 p 是一个函数指针，将一个整型数据赋给一个指针变量，会引起数据类型不匹配的编译错误。

在一个程序中，指针变量 p 可以先后指向不同的函数。但是需要注意的是，由于函数的差异（函数的差异反映在函数返回值类型的差异和函数参数的差异），导致了函数指针的差异。一个函数不能赋给一个不一致的函数指针。

例如，有如下的函数：

int fn1（int x，int y）；

int fn2（int x）；

定义如下的函数指针

int （*p1）（int a，int b）；

int （*p2）（int a）；

则语句“ p1=fn1； ”和 语句“ p2=fn2； ”都是正确的，因为函数指针和它指向的函数一致。

语句“ p1=fn2； ”会产生编译错误。因为 p1 是一个指向有两个整型参数和返回整型值的函数指针，而 fn2 是一个只有一个整型参数和返回整型值的函数，两者不一致，不能让一个函数指针指向与其类型不一致的函数。

定义了函数指针并让它指向了一个函数后，对函数的调用可以通过函数名调用，也可以通过函数指针调用（即用指向函数的指引变量调用）。

用函数指针变量调用函数时，只需将（*p）代表函数名即可，在（*p）之后的括弧中根据需要写上实参。

例如语句

c=（*p）（a，b）；

表示调用由 p 指向的函数（max），实参为 a、b，函数调用结束后得到的函数值赋给 c。

需要注意的是，p 是指向函数的指针变量，它只能指向函数的入口处而不可能指向函数中间的某一条指令处，因此不能用*（p+1）来表示函数的下一条指令。

函数指针变量常用的用途之一是把指针作为参数传递到其他函数。

设有一个函数（函数名为 sub），它有两个形参（x1 和 x2），定义 x1 和 x2 为指向函数的指针变量。在调用函数 sub 时，实参用两个函数名 f1 和 f2 给形参传递函数地址。这样在函数 sub 中就可以调用 f1 和 f2 函数了。

例 6-14　设计一个函数，在调用它的时候，每次实现不同的功能。

```
#include <iostream >
using namespace std;
int max(int x,int y);
int min(int x,int y);
int add(int x,int y);
void process(int i,int j,int (*p)(int a,int b));
```

```
int main()
{
      int x,y;
      cin>>x>>y;
   cout<<"Max is ";
   process(x,y,max);
   cout<<"Min is ";
   process(x,y,min);
   cout<<"Add is ";
   process(x,y,add);
   return 0;
}

int max(int x,int y)
{
   return (x>y?x:y);
}

int min(int x,int y)
{
   return (x>y?y:x);
}

int add(int x,int y)
{
   return x+y;
}

void process(int i,int j,int (*p)(int a,int b))
{
   int c;
   c=(*p)(i,j);
   cout<<c<<endl;
}
```

程序中 max、min 和 add 是已定义的 3 个函数，分别用来求最大值、求最小值和求和。在 main 函数中第一次调用 process 函数时，除了将 x 和 y 作为实参传给 process 的形参 i、j 外，还用函数名 max 作为实参将 max 函数的入口地址传送给 process 函数中的形参 p（p 是指向函数的指针变量）。这时，process 函数中的(*p)(i，j)相当于 max（i，j），执行 process 可以输出 i 和 j 中大者。在 main 函数中第二次调用 process 时，改以函数 min 作实参，此时

process 函数的形参 p 指向函数 min，在 process 函数中的函数调用(*p)(i，j)相当于 min(i，j)。同理，第三次调用 process 函数时，process 函数的形参 p 指向函数 add，在 process 函数中的函数调用(*p)(i，j)相当于 add(i，j)。

2．返回指针值的函数

一个函数可以返回一个整型值、字符值、实型值等，也可以返回指针型的数据，即地址。这种返回指针值的函数的一般定义形式为：

```
数据类型  *函数名（参数表）；
```

例如：

```
int   *a(int x，int y)；
```

a 是函数名，调用它以后能得到一个指向整型数据的指针(地址)。x、y 是函数 a 的形参，为整型。注意在 *a 两侧没有括弧，在 a 的两侧分别为 * 运算符和 () 运算符。而（）优先级高于*，因此 a 先与()结合，这显然是函数形式。这个函数前面有一个*，表示此函数是指针型函数（函数值是指针）。最前面的 int 表示返回的指针指向整型变量。

我们在前面有关链表操作中给出的几个函数，就是返回值为指针的函数。如：

```
SLink *create(SLink *head)；
SLink *insert(SLink *head,int i,int x)
SLink *delete_node(SLink *head,int i)
```

6．1．6　const 修饰符和指针

C++中使用 const 修饰符来定义常量。const 可以与指针一起使用，它们的组合情况较复杂，可归纳为三种：指向常量的指针、常指针和指向常量的常指针。

1．指向常量的指针

指向常量的指针是指一个指向常量的指针变量。在指针定义语句的类型前加 const 修饰符，表示指向的对象是常量。例如：

```
const char *name＝"chen"；        // 声明指向常量的指针
```

这个语句的含义为：声明一个名为 name 的指针变量，它指向一个字符型常量，初始化 name 为指向字符串"chen"。

由于使用了 const，不允许改变指针所指的常量，因此以下语句是错误的：

```
name[3]＝'a'；
```

但是，由于 name 是一个指向常量的普通指针变量，不是常指针，因此可以改变 name 的值。例如下面的语句是允许的：

```
name＝"zhang"；
```

该语句赋给了指针变量 name 另一个常量，即改变了 name 的值。

2．常指针

常指针是指把指针本身是声明为常量，而不是将它指向的对象声明为常量。在指针定义语句的指针名前加 const 修饰符，表示指针本身是常量。例如：

```
char * const   name="chen"；        // 常指针
```

这个语句的含义为：声明一个名为 name 的指针常量，该指针是指向字符型数据的常指

针，用字符串常量"chen"的地址初始化该常指针。

在定义指针常量时必须初始化。创建一个常指针，就是创建一个不能移动的固定指针，但是它所指的数据可以改变，例如：

```
name[3]＝'n';          // 合法
name＝"zhang";         // 出错
```

第一个语句改变了常指针所指的数据，这是允许的；但第二个语句要改变指针本身，这是不允许的。

3．指向常量的常指针

指向常量的常指针是指这个指针本身不能改变，它所指向的值也不能改变。要声明一个指向常量的常指针，二者都要声明为 const，并且必须在定义时进行初始化。例如：

```
const char * const name＝"chen";       //指向常量的常指针
```

这个语句的含义是：声明了一个名为 name 的指针常量，它是一个指向字符型常量的常指针，用字符串常量"chen"的地址初始化该指针。不难理解以下二个语句都是错误的：

```
name[3]＝'a';          // 出错，不能改变指针所指的值
name="zhang";          // 出错，不能改变指针本身
```

6．2　引用

6．2．1　引用的概念

引用是 C++语言中对一个变量或常量标识符起的别名。例如，已经定义了一个变量 val，然后再建立一个对这个变量的引用 rval，就为变量 val 起了一个别名，val 和 rval 就是指的同一个变量，它们的使用方式也完全一样。说明一个引用的方式如下：

```
int val;  // 定义 vaL 是整型变量
int   &rval= val;    //声明 rvaL 是 val 的引用
```

说明一个引用时，在引用标识符的前面加上“&”，后面跟上它所引用的标识符的名字。

当编译程序看到‘＆’时，就不为其后面的标识符分配内存空间，而只是简单地将它所引用的那个标识符所具有的内存空间赋给它。比如上面定义的变量 val 已经有了一个内存空间来存放它的值，则对它的引用 rval 也同样使用这个内存空间来存放值。即对 val 和 rval 的使用完全一样。

例如，语句　val=1；　将导致 rval 的值也变成 1，而语句　rval=2；　也也导致 val 的值变成 2。对 val 的操作和对 rval 的操作是完全等价的。

在声明一个引用类型变量时，必须同时对它进行初始化，即必须在声明引用时说明它所引用的对象。所引用的对象必须是已经有对应的内存空间的。

引用一旦初始化，它就被维系在它所引用的目标上，再不能改变，不能将这个别名再用在其他目标上。而且也没有办法对引用再赋其他值，因为对这个引用所做的所有操作将直接对应到它所引用的对象上。例如，已经做了如下定义：

```
int val1=1，val2=2；
```

int & rval=val1；

那么对 rval 的改变

rval=val2；

并不会导致 rvaL 成为对 val2 的引用，而是将 val1 的值从原来的 1 改成了 val2 的值 2。

例 6-15　一个验证引用和它所引用的对象同一性的程序示例

```
#include <iostream >
using namespace std;
int main()
{
    int val;
    int &rval=val;
    cout<<"& val="<<&val<<endl;
    cout<<"& rval="<<&rval<<endl;
    val=15;
    cout<<"val="<<val<<"    rval="<<rval<<endl;
    rval=30;
    cout<<"val="<<val<<"    rval="<<rval<<endl;
    return 0;
}
```

在 VC++中，程序的执行结果如下：

```
& val=0x0012FF7C
&rval=0x0012FF7C
val=15    rval=15
val=30    rval=30
```

这个程序首先输出了引用 rval 的地址和它所引用的变量的地址，可以看出 val 和 rval 的地址是一样的，都是 0x0012FF7C。程序然后又分别通过 val 和 rval 来改变变量的值，它们产生的结果也是一样的。从程序的运行结果可以看到，val 和对它的引用 rval 在使用方式、产生的效果上是完全一样的。

需要注意的是，并不是所有类型的数据都是可以引用的，可以声明对简单数据类型（如 int、float、double、char）变量的引用，也可以声明对一个结构体变量的引用，还可以声明对一个指针变量的引用。例如：

```
int    * p = new int  ；
int *    & rp =p；     //   声明 rp 是对指针变量 p 的引用
*rp=10；            // 对指针的引用可以和指针一样使用
```

但是，不能声明：

（l）对 void 的引用。因为 void 本身就表示没有数据类型，对它的引用也就没有意义。这一点和指针不一样，可以定义指向 void 的指针。

（2）对数组名的引用。因为数组是某个数据类型元素的集合，数组名表示该集合元素在内存存放的起始地址，它本身不是一个变量，所以，对数组名的引用没有意义。但可以

说明对数组的某一个元素的引用，如：

```
int   a[10];
int   & ra=a;   //错误！ 不能声明数组名的引用
int & ra = a[1];    //正确
```

（3）指向引用类型的指针。因为引用本身只是一个符号，它没有任何内存空间，所以不能定义指向引用类型的指针。如：

```
int   a;
int & ra=a;
int & * pra =&ra;    //错误！不能定义指向引用的指针
                     //“int & * ”表示指向 int 型引用的指针，这是不允许的；
                     // 而“int * &”表示对 int 型指针的引用，这是允许的。
```

6．2．2　用 const 限定引用

当用 const 来限定引用时表示不能通过引用改变被引用的空间的值。例如

```
int   i;
const   int   &ri=i;    //   ri 是 i 的一个 const 引用
ri=10;                  //   错误! 不能对 ri 赋值
cout<<ri ;              //   正确! 可以访问 ri 的值
```

但是，const 对 ri 的限制并不影响它所引用的变量 i，例如：

```
i=20;                   //   正确! i 不是常变量
```

另外，对一个常变量进行引用时，必须将这个引用定义为 const 的，例如

```
const   int   ci=10;
int   & rci=ci ;            // 错误! 必须定义成 const 引用
const   int   & rci=ci ;    // 正确!
```

6．2．3　引用作函数参数

引用在一般场合是没有什么用途的，单纯为一个变量取个别名也没有什么意义。引用最大的用途是作为函数的参数或返回值类型，从而扩充函数传递数据的功能。

我们知道，引用是它所引用的变量的一个别名，它们实际上是同一回事。因此，当函数的形参是引用类型时，它实际上是对实参所代表的变量的引用，它自己不具有独立的内存空间。这种实参传递给形参的方式称为按引用传递。

例 6-16　利用“引用形参”实现两个变量的值的互换

```
#include <iostream >
using namespace std;
void change(int &,int &);
int main()
{
    int a=3,b=8;
```

```
    change(a,b);
    cout<<"a="<< a <<"    b="<< b << endl;
    return 0;
}
void change(int &i,int &j)
{
    int temp;
    temp=i;
    i=j;
    j=temp;
}
```

程序运行后，输出结果为

a=8　b=3

在 change 函数的形参表列中声明 i 和 j 是整型变量的引用。当 main 函数调用 change 函数时，由实参把变量名传给形参（a 的名字传给引用变量 i），这样 i 就成了 a 的别名。同理，j 成为 b 的别名。i 和 a 代表同一个变量，j 和 b 代表同一个变量。在 change 函数中使 i 和 j 的值互换，显然，a 和 b 的值同时改变了。因此，在 main 函数中输出 a 和 b 已互换了的值。

比较这个程序和前面曾介绍过的用指针变量作为函数参数的程序，就会发现，用引用作为参数显得更加自然。用引用做参数时，函数调用时实参可以直接用变量名，函数的实现也非常自然，而用指针作为参数时，需要将变量的地址传递过去（函数调用时实参变量名前加&），在函数体中也需要通过指针对地址间接操作，显得非常麻烦。

在使用引用参数时，需要注意：引用参数对应的实参必须具有对应的内存空间，即实参必须有一个合法的内存空间，以便能够对这个空间进行引用。下面对 change 进行调用的语句都是不对的：

change（10，20）；

change（x+5，y）；

引用作函数参数通常用在以下场合：

（1）函数需要返回多个值的场合。因为函数通过 return 只能最多返回一个数据，而多于一个值时，只能借助于引用或指针参数来返回。而引用比指针使用起来更加自然，所以建议在能够用引用参数的地方不要用指针参数。

（2）函数的参数是结构或类的对象。当参数是结构类型或对象时，一般会占用较多的内存空间，如果按值传递将需要分配较多的栈空间来存放形参的值，需要进行大量的数据复制操作，会消耗较多的空间和时间。如果设计为按引用传递，就只要生成一个实参的别名，所有的操作直接在实参的空间上进行，程序运行的效率比较高。

例 6-17　用引用返回多个值

```
#include <iostream >
using namespace std;
int factor(int,int &,int &);
```

```
int main()
{
    int number,squared,cubed,error;
    cin>>number;
    error=factor(number,squared,cubed);
    if (error!=0)
        cout<<"Error number!"<<endl;
    else
    {
        cout<<"Number="<<number<<endl;
        cout<<"Squared="<<squared<<endl;
        cout<<"Cubed="<<cubed<<endl;
    }
    return 0;
}
int factor(int n,int &rSquared,int &rCubed)
{
    if (n>20 || n<0)
        return 1;
    rSquared=n*n;
    rCubed=n*n*n;
    return 0;
}
```

程序运行结果为：

```
12
Number=12
Squared=144
Cubed=1728
```

6．2．4　引用返回值

返回引用值的函数在一般情形下并不多见，但是如果需要让函数调用作为左值时，就需要将函数返回值设计为是引用类型。函数的返回值为引用类型表示该函数的返回值是一个内存变量的别名。可以将函数调用作为一个变量来使用，可以为其赋值。

例 6-18　定义了一个求两个数中最小值的函数，函数的参数是两个整型变量．返回值是对这两个数中值较小的那个变量的引用。

```
#include <iostream >
using namespace std;
int &min(int &,int &);
int main()
```

```
{
    int a=3,b=5;
    cout<<"a="<<a<<"    b="<<b<<endl;
    cout<<"The minimum of a and b is "<<min(a,b)<<endl;
    min(a,b)=8;          // 将 a，b 中的较小值修改为 8
    cout<<"After running min(a,b)=8, a="<<a<<" b="<<b<<endl;
    min(a,b)=2*min(a,b);   // 将 a，b 中的较小值增加一倍
    cout<<"After running min(a,b)=2*min(a,b), a="<<a<<" b="<<b<<endl;
    return 0;
}
int &min(int &i,int &j)
{
    if (i<=j)
        return i;
    else
        return j;
}
```

程序执行结果为：

```
a=3    b=5
The minimum of a and b is 3
After running min(a,b)=8,    a=8    b=5
After running min(a,b)=2*min(a,b),    a=8    b=10
```

在程序中，因为函数的返回值定义为引用，所以 min(a，b)实际上表示 a 或 b 中的一个变量，从而可以为其赋值。当然，也可以将返回引用的函数作为一般的数来使用，正如程序中的

```
cout<<"The minimum of a and b is "<<min(a,b)<<endl;
```

在使用引用返回值时，要保证函数返回的引用在函数执行结束后还能有效，不能返回对一个函数体中的局部变量的引用，因为函数的局部变量在函数执行结束后不再有意义，因此对这个变量的引用也就没有意义。函数返回的引用应该是对某一个函数参数的引用，而且这个参数本身也是引用类型，因为这样才能保证函数返回的引用有意义。上面的例子就是这样，函数返回的 i 或 j 都是函数的参数，而且这两个参数都是引用类型的参数。

本章小结

指针是一种存放地址的变量，像其它变量一样，必须在使用前先定义。定义指针变量的一般形式为：

基类型　　* 指针变量名

有两个与指针有关的运算符：

（1）&：取地址运算符。

（2）* ：指针运算符（或称“指向”运算符）。

&运算符只能作用于变量，包括基本类型、数组的元素和结构体的成员，不能作用于数组名和常量。*运算是&的逆运算，它的操作对象是地址。

指针是 C++的一个特色。使用指针的优点是：

下表 6-2 是有关指针的数据类型的小结，为了便于比较，把其它一些类型的定义也列在一起。

表 6-2　有关指针的数据类型

定义	含义
int i;	定义整型变量 i
int *p;	定义指向整型数据的指针变量 p
int a[10];	定义整型数组 a，它有 10 个元素
int *p[10];	定义指针数组 p，它有 10 个指向整型数据的指针元素
int (*p)[10];	定义指向含 10 个元素的一维数组的指针变量 p
int f( );	f 为返回整型函数值的函数
int *p( );	p 为返回一个指针的函数，该指针指向整型数据
int (*p)( );	p 为指向函数的指针，该函数返回一个整数值
int **p;	p 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

引用是 C++对 C 的一个重要扩充，它的作用是为一个变量起一个别名。

在声明一个引用类型变量时，必须同时对它进行初始化，即必须在声明引用时说明它所引用的对象。所引用的对象必须是已经有对应的内存空间的。

引用一旦初始化，它就被维系在它所引用的目标上，再不能改变，不能将这个别名再用在其他目标上。而且也没有办法对引用再赋其他值，因为对这个引用所做的所有操作将直接对应到它所引用的对象上。

引用在一般场合是没有什么用途的，单纯为一个变量取个别名也没有什么意义。引用最大的用途是作为函数的参数或返回值类型，从而扩充函数传递数据的功能。

习　题

1．写出下列程序的运行结果

（1）
```
#include <iostream>
using namespace std;
int main()
{
    int a[3][3],*p,i;
    p=&a[0][0];
    for(i=0;i<9;i++) p[i]=i+1;
```

```
        cout<<a[1][2]<<endl;
        return 0;
    }
```
（2）
```
#include <iostream>
    using namespace std;
    int main()
    {
        int a[][3]={{1,2,3},{4,5,0}},(*pa)[3],i;
        pa=a;
        for(i=0;i<3;i++)
            if (i<2) pa[1][i]=pa[1][i]-1;
            else    pa[1][i]=1;
        cout<<a[1][0]+a[1][1]+a[1][2]<<endl;
        return 0;
    }
```
（3）
```
#include <iostream>
    using namespace std;
    void ss(char *s,char t);
    int main()
    {
        char str1[20]="abcddfefdbd",c='d';
        ss(str1,c);
        cout<<str1<<endl;
        return 0;
    }
    void ss(char *s,char t)
     {
        while(*s)
        {
            if(*s==t) *s=t-'a'+'A';
            s++;
        }
    }
```
（4）
```
#include <iostream>
    using namespace std;
    int main()
    {
        int x[6]={1,3,5,7,9,11},*k,**s;
        k=x;
```

```
        s=&k;
        cout<<*(k++)<<endl;
        cout<<**s<<endl;
        return 0;
    }
```

（5）

```
#include <iostream>
    using namespace std;
    int main()
    {
        int m[5],n[5],*px,*py,i;
        px=m;
        py=n;
        for(i=1;i<4;i++,px++,py++)
        {
            *px=i;
            *py=2*i;
        }
        px=&m[1];
        py=&n[1];
        for(i=1;i<3;i++)
        {
            *px+=i;
            *py*=i;
            cout<<*px++<<" "<<*py++<<endl;
        }
        return 0;
    }
```

2．编写一个函数 char *strfind(char *s,char *t)，用于查找字符串 t 在字符串 s 中第一次出现的位置，如果没有找到则返回 NULL，并编写一个主程序来测试该函数。在主程序中按行输入两个字符串，并输出查找结果。

3．编写一个函数 char *substr(char *s,int begin,int end)，用于取得字符串 s 中从 begin 位置开始到 end 位置（不包含 end 处的字符）结束的子字符串，并编写一个主程序来测试函数。

4．10 个小孩围坐一圈，并给他们依次编号，指定从第 s 个小孩开始报数，（从 1～n 报数），报到 n 的小孩出列，然后依次重复下去，直到所有的小孩出列，求出小孩的出列顺序。

5．编号为 1、2、…、n 的 n 个小孩按顺时针围坐一圈，每个小孩持有一个密码（正整数）。一开始任选一个正整数作为报数上限值 m，从第 1 个小孩开始按顺时针方向自 1 开始顺序报数，报到 m 时停止报数。报 m 的人出列，将他的密码作为新的 m 值，从他在顺时针方向上的下一个人开始重新从 1 报数，如此下去，直到所有的小孩全部出列。编写一个

程序输入小孩人数 n、每个小孩的初始密码及起始报数上限值 m，输出小孩的出列顺序。

6．建立一个有 10 个结点的单向链表，每个结点包括：学号、姓名、性别、联系电话。对该链表按学号从小到大顺序排列后输出。

7．编写一个程序，输入月份号，输出该月的英文月名。例如，输入“4”，输出“April”，要求用指针数组处理。

8．用指向指针的指针的方法对 n 个整数排序并输出。要求将排序单独写成一个函数。整数和 n 在主函数中输入，排序结果也在主函数中输出。