第3章

构造数据类型

◇ **引言**

在实际应用中，除了使用基本数据类型描述所处理的问题外，人们经常要处理一些更复杂的数据对象，这些复杂的数据对象无法用单一的基本数据类型来描述，它们需要用一些简单的基本数据类型组合成比较复杂的数据类型才能予以描述和定义。为此，C++中提供了**构造数据类型**。

构造数据类型是用基本数据类型构造的用户自定义数据类型，用来对复杂的数据对象进行描述与处理。构造数据类型也称为自定义数据类型，包括枚举、数组、指针、字符串、引用类型、结构和联合。

◇ **学习目标**

(1) 掌握枚举类型的使用；

(2) 深入理解数组的概念，掌握数组应用的一般方法；

(3) 深入理解指针的概念，掌握指针的使用；

(4) 注意指针与数组的区别，会使用多重指针以及指针与数组的多种混合体，会分配动态数组；

(5) 理解字符串的概念，会使用字符串；

(6) 理解引用的概念，掌握引用型函数参数的用法；

(7) 掌握结构与联合类型的使用，并注意二者的区别。

3.1 枚举类型

在生活中人们都有这样的常识：一个星期有7天，分别是星期一、星期二、…、星期日；交通灯只有红、黄、绿3种颜色。类似这样的情况还有很多例子，在计算机中可以用int、char等类型来表示这些数据。但是，如果将星期一至星期日表示为1～7的整数，一方面在程序中容易将它们与其他不表示星期的整数混淆；另外，由于它们只能取有限的几种可能值，这样在程序中对数据的合法性检查成为一件比较麻烦的事。C++中的枚举类型就是专门用来解决这类问题的。

3.1.1 枚举类型的定义

如果一个变量只有几种可能的取值，可以使用枚举类型来定义。枚举类型属于用户自定义数据类型。所谓“枚举”，是指将变量所有可能的取值一一列举出来，变量的取值只限于列举出来的常量。

枚举类型的声明的一般形式如下：

```
enum  枚举类型名  {枚举常量 1,枚举常量 2,…,枚举常量 n};
```

其中：

- 枚举类型名以及枚举常量为标识符，遵循标识符的取名规则。
- 在定义一个枚举类型时定义了多个常量，供枚举类型变量取值，我们称此常量为枚举常量。当没给各枚举常量指定值时，其值依次默认为 0、1、2、…在定义枚举类型时，也可以使用赋值号另行指定枚举常量的值。

例如：

```
enum weekday { SUN,MON,TUE,WED,THU,FRI,SAT };
```

定义了 7 个枚举常量以及枚举类型 weekday。枚举常量具有默认的整数与之对应：SUN 的值为 0、MON 的值为 1、TUE 的值为 2、…、SAT 的值为 6。

```
enum city{ Beijing,Shanghai,Tianjin = 5,Chongqing};
```

枚举常量 Beijing 的值为 0，Shanghai 的值为 1，Tianjin 的值为 5。对于没有指定值的枚举常量，编译器会将前一个常量值加 1（下一个整数）赋给它，所以 Chongqing 的值为 6。

枚举类型定义了以后就可以使用枚举常量、枚举类型来定义变量了，定义枚举变量的方法与定义其他变量的方法一样。例如：

```
enum city city1,city2;
city city1,city2;
```

用两种方法定义了 city1、city2 两个枚举类型的变量名。

枚举类型变量也可以在定义枚举类型的同时定义，例如：

```
enum city{ Beijing,Shanghai,Tianjin = 5,Chongqing} city1,city2;
```

在定义枚举类型的同时定义枚举类型变量可以省略枚举类型名，例如：

```
enum { Beijing,Shanghai,Tianjin = 5,Chongqing} city1,city2;
```

在定义变量时可以顺便给出初值，若不给初值，默认初值为随机的无意义的数。

3.1.2 枚举类型的使用

定义一个枚举类型后，就可以直接使用各个枚举常量了。用枚举类型建立枚举变量后可以对枚举变量实施赋值以及进行其他运算，包括对枚举变量进行赋值，其值要求为同一枚举类型，否则在编译时会出错。

例如：

```
weekday d1,d2,d3,d4;
d1 = SUN;
d2 = 6;                    //错误
d3 = Shanghai;             //错误
```

其中，对 d2 所赋之值是整数 6，不是枚举常量；可以将一个整型值强制转换成同类型的枚举常量赋给枚举变量：

```
d2 = (weekday)6;
```

对 d3 的赋值不是同类型的枚举常量。

枚举常量、枚举类型的变量可进行算术运算、关系运算。对枚举类型实施算术、关系运算时，枚举值转换成整型值参加运算，结果为整型值。如果要将结果赋给枚举变量，还要将结果转换成枚举值。

例如：

```
d1 = d1 + 2;                //是错误的,因为结果为 int 型
```

需要将它强制转换成枚举型：

```
d1 = (weekday)(d1 + 2);
d1++;                       //也是错误的
```

枚举常量、枚举类型的变量可直接进行各种形式的关系运算。

例如：

```
if(city1 == 3);
if(city2 >= Beijing);
if(Shanghai == 1);
if(city1 > SUN);
```

另外，枚举类型变量不能直接进行输入，例如：

```
cin >> d1;                  //错误
```

☆**注意**：

(1) 枚举常量是常量，不是变量，所以不能对枚举常量进行赋值。在上例中不能进行“Shanghai＝Beijing；”的赋值。

(2) 枚举常量的值不是列举的字符串，其值为整数。

(3) 编译器对赋给枚举变量的对象(数)进行类型检查，如类型不相符则发出警告。当类型相同，而值超出此类枚举类型的枚举常量范围时也是正常的。

【例 3-1】 输入城市代号，输出城市名称。

```
1  /*************************************************************
2   *   程序名:p3_1.cpp                                         *
3   *   功   能:枚举类型的使用,输入城市代号,输出城市名称        *
4   *************************************************************/
5  #include <iostream>
6  using namespace std;
7  enum city{ Beijing,Shanghai,Tianjin = 6,Chongqing};
8  int main()
9  {
10   int n;
11   cout <<"Input a city number ("<< Beijing - 1 <<" to exit):"<< endl;
12   cin >> n;
13   while(n >= Beijing){
14       switch(n) {
15         case Beijing: cout <<"Beijing"<< endl;break;
16         case Shanghai:cout <<"Shanghai"<< endl;break;
```

```
17          case Tianjin:cout <<"Tianjin"<< endl;break;
18          case Chongqing:cout <<"Chongqing"<< endl;break;
19          default:cout <<"Invalid city number!"<< endl;break;
20        }
21        cin >> n;
22      }
23    return 0;
24 }
```

运行结果：

```
Input a city number ( - 1 to exit):
1↙
Shanghai
8↙
Invalid city number!
-1↙
```

3.2 数　　组

在程序中经常需要处理成批的数据，如一个班学生某门功课的成绩，这类数据有一个共同的特点：它们有若干个同类型的数据元素，并且各个数据元素之间存在某种次序关系。如果用单个变量表示数据元素，一方面要建立很多变量，另一方面无法体现数据元素之间的关系。C++以及其他高级语言提供了数组这种数据类型来表示上述数据。

数组是一组在内存中依次连续存放的、具有同一类型的数据变量所组成的集合体。其中的每个变量称为数组元素，它们属于同一种数据类型，数组元素用数组名与带方括号的数组下标一起标识。数组可以是一维的，也可以是多维的。

3.2.1 一维数组的定义与使用

数组属于构造数据类型，在使用之前必须先进行类型定义。

1. 一维数组的定义

定义一维数组的一般形式如下：

```
数据类型　数组名[常量表达式];
```

其中：

- 数组元素的类型可以是 void 型以外的任何一种基本数据类型，也可以是已经定义的构造数据类型。
- 数组名是用户自定义的标识符，用来表示数组的名称，代表数组元素在内存中的起始地址，是一个地址常量。
- 常量表达式必须是 unsigned int 类型的正整数，表示数组的大小或长度，也就是数组所包含数据元素的个数。

• []是数组下标运算符，在数组定义时用来限定数组元素的个数。

例如，下面定义了两个不同类型的数组：

```
int a[5];                    //定义了一个 5 个元素的整型数组 a
weekday b[10];               //定义了一个 10 个元素的枚举数组 b,weekday 为已定义的枚举类型
```

数据类型相同的多个数组可以在同一条语句中予以定义。例如：

```
int a1[10],a2[20];           //同时定义了两个整型数组
```

数据类型相同的简单变量和数组也可以在一个语句中定义。例如：

```
int x,a[20];                 //同时定义了一个整型变量和一个整型数组
```

数组在定义之后，系统将会从内存中为其分配一块连续的存储空间，从第 1 个数据元素开始依次存放各个数组元素。例如，定义的数组 a 的内存排列(分配)示意如图 3-1 所示。

一维数组所占内存大小(字节数)的计算公式如下：

```
    n * sizeof(元素类型);
或
    sizeof(数组名);
```

其中，n 为数组的长度。

若定义“int a[100]；sizeof(a)＝100 * sizeof(int)＝400”，数组 a 所占内存的大小为 400 个字节。

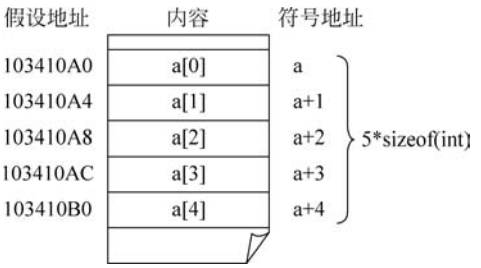

图 3-1　一维数组的内存排列示意图

在定义数组时可以使用类型定义 typedef 为数组类型取一个名字，格式如下：

```
    typedef   数据类型   数组名[常量表达式];
```

例如：

```
typedef int A[5];
```

定义了一个整型数组 A，同时 A 是一个类型名。因此，可以使用类型 A 定义变量：

```
A b;
```

定义了与 A 类型、长度相同的数组 b。

2. 一维数组的初始化

一维数组的初始化是指在定义数组的同时给数组中的元素赋值。其一般语法格式如下：

```
数据类型  数组名[常量表达式] = {初值 1,初值 2,…,初值 n};
                              └──初值表──┘
```

其中：

- {初值 1,初值 2,…,初值 n}称为初值表，初值之间用逗号分隔，所有初值用{ }括起来。
- 初值可以是一个变量表达式，初值与数组元素的对应关系是初值 i 为数组的第 i 个元素，所以，初值个数 n 不能超过数组的大小。
- 若初值表中的初值个数(项数)小于数组的大小，则未指定值的数组元素被赋值为 0，但初值表中的项数不能为 0。例如：

```
weekday b[10] = {MON,WED,FRI};
```

经过以上定义和初始化后，b 的前 3 个元素的值分别为 MON、WED、FRI，其余元素的值为默认值 0。

- 当对全部数组元素赋初值时，可以省略数组的大小，此时数组的实际大小就是初值列表中初值的个数。例如：

```
char str[] = {'a','b','c','d','e'};
```

则数组 str 的实际大小为 5。

- 在函数中定义数组时，如果没有给出初值表，数组不被初始化，其数组元素的值为随机值；在函数外定义数组，如果没有初始化，其数组元素的值为 0。
- 数组初值表可以用一个逗号结尾，其效果和没有逗号一样。例如：

```
int a[2] = {1,2};与 int a[2] = {1,2,};相同
int b[] = {1,2};与 int b[] = {1,2,};相同
```

☆**注意**：

在定义数组时，编译器必须知道数组的大小，并据此为整个数组分配适当大小的内存空间。因此，数组元素的个数一定是常量表达式，只有在定义数组时进行初始化才能省略数组的大小。

3. 一维数组的存取

对一维数组实施的存取操作有两类，即存取数组元素与读取数组元素的地址。数组元素是通过数组名及下标来标识的，这种带下标的数组元素也称为下标变量，下标变量可以像简单变量一样参与各种运算。存取一维数组元素的格式如下：

```
数组名[下标表达式];
```

其中：

- 下标表达式可以是变量表达式，用来标识数组元素，不同于数组定义时用来确定数组长度的常量表达式。
- 当定义了一个长度为 n 的一维数组 a 时，C++规定数组的下标从 0 开始，依次为 0、1、2、3、…、n－1，对应的数组元素分别是 a[0]、a[1]、…、a[n－1]，因此下标表达式的值要在 0～n－1 范围内。例如：

```
a[1 + 2] = 100;              //将数组 a 的第 4 个元素赋值 100
```

【例 3-2】 学生成绩排序。

分析：学生成绩由键盘输入，当输入一个负数时，输入完毕。

采用直观的**“选择排序法”**进行排序，基本步骤如下：

① 将 a[0]依次与 a[1]～a[n－1]比较，选出大者与 a[0]交换，最后 a[0]为 a[0]～a[n－1]中的最大者；

② 将 a[1]依次与 a[2]～a[n－1]比较，选出大者与 a[1]交换，最后 a[1]为 a[1]～a[n－1]中的最大者；

③ 同理，从 i＝2 到 i＝n－1，将 a[i]依次与 a[i＋1]～a[n－1]比较，选出较大者存于 a[i]中。

```
/*****************************************
*   程序名：p3_2.cpp                      *
*   功   能：数组应用——选择排序          *
*****************************************/
#include <iostream>
using namespace std;
int main()
{  const int MaxN = 5;
   int n,a[MaxN],i,j;
   for (n = 0;n < MaxN;n++)
   {
       cin >> a[n];                  //输入数组元素
       if(a[n]< 0)
           break;
   }

   //对数组元素逐趟进行选择排序
   for(i = 0;i < n - 1;i++)
       for(j = i + 1;j < n;j++)      //从待排序序列中选择一个最大的数组元素
           if(a[i]< a[j])
           {
               int t;
               t = a[i];             //交换数组元素
               a[i] = a[j];
               a[j] = t;
           }
   for(i = 0;i < n;i++)
       cout << a[i]<<"\t";           //显示排序结果
   return 0;
}
```

运行结果：

```
80  90  95  70  -1↙
95  90  80  70
```

4. 数组的地址

数组元素的地址通过数组名来读取，其格式如下：

```
数组名 + 整型表达式;
```

由于其地址不是实际的地址值，称这个地址表达式为符号地址表达式。例如一维数组元素 a[5]的符号地址表达式为 a+5。

若 a 是一个 int 型数组，数组的符号地址表达式 a+n 所表达的地址是第 n+1 个元素 a[n]的地址，代表的实际地址值为：

```
a + n * sizeof(int)
```

而不是 a+n。

(1) 在使用数组时最常犯的错误是数组元素越界，包括上标越界和下标越界。上标越界是指数组元素的访问地址值超过了数组的起始地址；下标越界是指数组元素的访问地址越过了数组中最后一个数组元素的地址。对于这种错误，编译器无法知道，往往在运行时出错，因此大家在进行程序设计时应格外注意。

(2) 数组名是一个地址常量，不能作为左值(赋值的目标)，因此，不能将一个数组整体复制给另外一个数组。

```
int a[5],c[5],i;
a = c;                          //错误
```

正确的方法是将对应的元素进行复制，见下列程序段：

```
for(i = 0;i < 5;i++)
    a[i] = c[i];                //将数组 c 中元素的值复制到数组 c 的对应元素中
```

还可以使用 memcpy 函数进行内存字节复制，memcpy 的使用格式如下：

```
Memcpy(目标地址 d,源地址 s,字节数 n);
```

其功能是将源地址 s 开始的 n 个字节复制到目标地址 d。

将数组 a 复制到 c 可以用下列语句实现：

```
memcpy(c,a,sizeof(a));
```

需要注意的是，使用 memcpy 函数要包含相应的头文件。

(3) 在函数中可以将一个一维数组作为函数的形式参数，用来接受一个一维数组传递过来的地址。

3.2.2 二维数组的定义与使用

一个班学生多门功课的成绩、在数学中经常用到的矩阵，这些都是二维表，用一维数组不便表示和存取，C++提供了二维数组来描述这样的二维表。

1. 二维数组的定义

二维数组在第一维的基础上增加了一维，其定义格式如下：

```
数据类型　数组名[常量表达式 2][常量表达式 1];
```

其中：

- 常量表达式 1 为第一维元素的个数，常量表达式 2 为第二维元素的个数。
- 二维数组 a[m][n]是以长度为 n 的一维数组为元素的数组，因此等价于以下定义方式：

```
typedef　数据类型　一维数组名[常量表达式 1];
一维数组名　二维数组名[常量表达式 2];
```

例如：

```
int M[2][3];
```

定义了一个整型二维数组 M，数组 M 也可以用下列方式定义：

```
typedef int M1[3];          //定义了一个一维整型数组 M1
M1  M[2];                   //以 M1 为类型定义数组 M
```

如果一维数组描述排列成一行的数据，那么二维数组则描述若干行这样的数据。因此，二维数组可以看作是数学上的一个矩阵。第一维元素个数为矩阵的列数，第二维元素个数为矩阵的行数。因此二维数组的定义格式可以写成：

```
数据类型　数组名[行数][列数];
```

在定义一个二维数组后，系统为它分配一块连续的内存空间，二维数组 a[m][n]占内存空间的计算公式如下：

```
    sizeof(数组名);
或
    m * sizeof(a[0]);
或
    m * n * sizeof(数据类型)
```

既然一个二维数组是由若干个一维数组排列构成的，二维数组在内存中的排列顺序为先顺序排列每个一维元素，构成一维数组；再将各个一维数组顺序排列，构成二维数组。

int M[2][3]的排列顺序如下：

(1) 先将 3 个 int 元素排列组成两个一维数组 M[0]、M[1]。

```
M[0]: M[0][0],M[0][1],M[0][2]
M[1]: M[1][0],M[1][1],M[1][2]
```

(2) 再将两个一维数组排成一个二维数组。

```
M: M[0],M[1]
```

(3) 数组 M 在内存中的排列如图 3-2 所示。

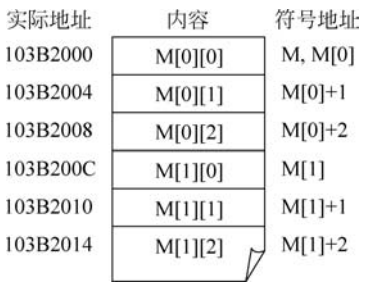

图 3-2　二维数组的内存排列示意图

2. 二维数组的初始化

二维数组的初始化形式与一维数组类似：

```
数据类型 数组名[常量表达式 2][常量表达式 1] = 初值表;
```

其中，初值表具有两种形式，即嵌套初值表和线性初值表。

1）嵌套初值表

以二维数组 M[m][n]为例，嵌套初值表的格式如下：

```
M 的初值表 = {M[0]初值表,M[1]初值表,…,M[m－1]初值表}
M[i]初值表 = {M[i][0]初值表,M[i][1]初值表,…,M[i][n－1]初值表};i 从 0 到 m－1;
```

嵌套初值表由一维初值表嵌套构成，各层的构成规则与一维数组的初值表相同。下面是对数组初始化的例子：

```
int M[3][4] = {{1,2,3,4},{3,4,5,6},{5,6,7,8}};    //M 数组元素被全部初始化
int a[2][3] = {{1},{0,0,1}};                      //初始化了部分数组元素
int b[][3] = {{1,2,3},};                          //初始化了全部数组元素
int d[][3] = {{1,3,5},{5,7,9}};                   //初始化了全部数组元素,省略了高维元素个数
```

2）线性初值表

线性初值表与一维数组的初值表相同，初值表的项目个数不超过各维元素个数的乘积（总元素个数）。

数组元素按内存排列顺序依次从初值表中取值，下列各数组使用了线性初值表，结果与使用嵌套初值表相同。

```
int M[3][4] = {1,2,3,4,3,4,5,6,5,6,7,8};          //M 数组元素被全部初始化
int a[2][3] = {1,0,0,0,1,1};                      //初始化了全部数组元素
int b[][3] = {1,0,0,0,0,0};                       //初始化了全部数组元素,省略了高维元素个数
```

当使用线性初值表而省略高维元素个数时，高维元素个数如下：

```
向上取整数(线性初值表项数/低维元素个数)
```

例如“int b[][3]={1,0,0,0};”，高维元素个数为 2。

初始化二维数组时要注意以下两点：

(1) 使用嵌套初值表时,每一维的初值数不能超过对应维元素的个数。例如:

```
int a[3][2] = {{1,2,3}};                    //错误,第一维元素个数为 2,却给了 3 个初值
```

(2) 即使使用嵌套初值表,只有最高维元素的个数能省略。例如:

```
int a[][] = {{1,2,3},{4,5,6}};              //错误,省略了多维元素个数
```

3. 二维数组的存取

存取二维数组元素的格式如下:

```
数组名[行下标表达式][列下标表达式]
```

其中:

- 行下标表达式与列下标表达式的值同样从 0 开始,a[i][j]表示数组的第 i+1 行、第 j+1 列的元素。由于数组元素是变量,可以对其进行各种操作。
- 数组元素如果定义数组 a[m][n],即数组第一维大小为 n,第二维大小为 m,a[i][j]的排列位置与在内存中的地址计算公式如下:

```
a[i][j]的排列位置 = 第一维大小 n * i + j + 1;
a[i][j]的地址 = a 的起始地址 + (第一维大小 n * i + j) * sizeof(数据类型)
```

数组的地址只能读取,二维数组的 a[m][n]地址表达式如下:

```
a,a[0]:          //为数组 a 的起始地址,即 a[0][0]的地址
a[i]:            //为数组的第 i + 1 行的地址,即 a[i][0]的地址
a[i] + j:        //为数组的第 i + 1 行的第 j + 1 元素的地址,即 a[i][j]的地址
a + k:           //为数组的第 k + 1 行的地址,即 a[k][0]的地址
```

【例 3-3】 计算一个班每个学生各门课程的总成绩,并计算每门课程的平均分。

```
1  /*********************************************************************
2   *   程序名: p3_3.cpp                                                *
3   *   功   能: 求学生多门功课的总分,并求所有学生各门功课的平均分      *
4   *********************************************************************/
5  #include<iostream>
6  using namespace std;
7  int main()
8  {
9      const int MaxN = 100,CourseN = 5;
10     int n,score[MaxN][CourseN + 1] = {0};
11     float aver[CourseN + 1] = {0};
12     for(n = 0;n < MaxN;n++)                          //输入学生成绩
13     {
14     for(int j = 0;j < CourseN;j++)
15         cin >> score[n][j];
16         if(score[n][0]< 0) break;                    //输入 - 1,结束输入
17     }
18     for(int i = 0;i < n;i++)                         //计算每个学生的总分
19         for(int j = 0;j < CourseN;j++)
```

```
            score[i][CourseN] = score[i][CourseN] + score[i][j];
    for(int j = 0;j < CourseN + 1;j++)                    //计算每门课程的平均分
    {
        for(int i = 0;i < n;i++)
            aver[j] = aver[j] + score[i][j];
        aver[j] = aver[j]/n;
    }
    for(i = 0;i < n;i++)                                  //输出每个人的成绩与总分
    {
        for(int j = 0;j < CourseN + 1;j++)
            cout << score[i][j]<<"\t";
        cout << endl;
    }
    cout <<" ---------------------------------------- "<< endl;
    for(i = 0;i < CourseN + 1;i++)                        //输出每门功课的平均分
        cout << aver[i]<<"\t";
    cout << endl;
    return 0;
}
```

运行结果：

```
70  71  72  73  74↙
82  83  84  85  86↙
92  93  94  95  96↙
-1  0    0   0   0↙
70    71    72    73    74    360
82    83    84    85    86    420
92    93    94    95    96    470
 ----------------------------------------------------------------
81.3333  82.3333  83.3333  84.3333  85.3333  416.667
```

程序解释：

(1) 定义 score[MaxN][CourseN+1]存储最多 MaxN 个学生 CourseN 门功课的成绩，定义列的大小为 CourseN+1 是为了在最后一列存储每个学生各门功课的总分。

(2) float aver[CourseN+1]用来存储 CourseN 门功课的平均分以及总分的平均分；单独定义一个浮点型一维数组来存储平均分是考虑到平均分要求更加精确，同时避免因学生人数太多，累加总分超出整数的表示范围。

3.2.3 多维数组

使用二维数组可以方便地存储一个班学生多门功课的成绩，而一个班学生的历年成绩则需要维数更高的三维数组，三维以及高于三维的数组称为多维数组。

三维数组是以二维数组为元素的数组，如果将一个二维数组看成是一张由行、列组成的表，三维数组则是由一张张表排成的一“本”表，第三维的下标为表的“页”码。同理，一个 n(n≥3)维数组是以一个 n−1 维数组为元素的数组。

1. 多维数组的定义

多维数组的定义与二维数组类似，可以一次定义，也可以逐步由低维数组定义。

例如：

```
int b[2][3][4];              //定义了一个三维数组
```

也可用下列方式分步定义：

```
typedef int B1[3][4];
B1 b[2];
```

多维数组在内存中的排列方式同样是先排低维数组，由低向高依次排列。例如 b[2][3][4]的排列顺序如下：

b
- b[0]
  - b[0][0]：b[0][0][0]，b[0][0][1]，b[0][0][2]，b[0][0][3]
  - b[0][1]：b[0][1][0]，b[0][1][1]，b[0][1][2]，b[0][1][3]
  - b[0][2]：b[0][2][0]，b[0][2][1]，b[0][2][2]，b[0][2][3]
- b[1]
  - b[1][0]：b[1][0][0]，b[1][0][1]，b[1][0][2]，b[1][0][3]
  - b[1][1]：b[1][1][0]，b[1][1][1]，b[1][1][2]，b[1][1][3]
  - b[1][2]：b[1][2][0]，b[1][2][1]，b[1][2][2]，b[1][2][3]

2. 多维数组的初始化与存取

多维数组的初始化形式与二维数组类似，有嵌套初值表、线性初值表两种形式。在使用线性初值表初始化时，数组元素按内存排列顺序依次从初值表中取值。

对多维数组进行存取包括对数组元素的存取和对数组元素的地址的读取，当一个多维数组的下标数与维数相同时，为对数组元素的存取。

例如 b[1][2][3]是对数组元素的存取。

对数组的元素进行存取时，各维的下标不能越界。

当下标个数小于维数时表示的是一个地址，当表示地址时，下标也不能越界。

例如，下列都是 b[2][3][4]的地址表达式：

```
b;                  //数组 b 的起始地址
b[1];               //b[1][0][0]的地址
b[2];               //错误,下标越界
b[0]+1;             //与 b[0][1]相同,b[0][1][0]的地址
b[1][2];            //b[1][2][0]的地址
b[1][2]+4;          //b[1][2][4]的地址,但数组 b 中没有 b[1][2][4]这个元素,故指向了其他地方
```

一个多维数组元素的地址有以下 n 种表示方法，以数组 b[K][M][N]的元素 b[k][m][n]为例(其中，0≤k＜K，0≤m＜M，0≤n＜N)：

(1) 使用取地址运算符 &。

```
b[k][m][n]的地址为 &b[k][m][n]
```

(2) 利用低维地址。

```
b[k][m]+n
```

(3) 利用起始地址。

```
&b[0][0][0]+k*M*N+m*N+n
```

只要内存够大，定义多少维数组没有什么限制。但一般来说，二维数组已经够用了。

3.2.4 数组与函数

数组名是一个地址，不能当作一个左值，但是可以作为函数的形参，接受实参传送来的地址。当形参接受实参传送来的地址后，形参数组与实参共享内存中的一块空间。函数体通过形参对数组内容的改变会直接作用到实参上。数组名作为形参是数组应用的一个重要方面。

【例 3-4】 编程将各班学生成绩分班按总分从高到低排序。

分析：首先要将各班学生成绩集中，然后对每个班的学生成绩排序。将各班学生成绩集中的过程是按班级排序的过程，因此要进行两次排序，这样设计了一个对数组各列排序的函数：

```
void sort(int a[][col],int n,int Cn,dir D)
```

形式参数 int a[][col]是一个列数为 col 的数组，n 为数组的行数，Cn 是要排序的列数，D 是枚举类型，指明排序是升序还是降序。升序的枚举值为 Asc，降序的枚举值为 Des。

学生成绩数组行的排列格式为学号、班号、成绩 1、成绩 2、总分。

```
/*****************************************************
*   程序名：p3_4.cpp                                  *
*   功   能：利用对一个多维数组的某列排序的函数      *
*           将学生某门功课的成绩分班级排序            *
*****************************************************/
#include<iostream>
using namespace std;
const col=5;
enum dir {Asc,Des};
void sort(int a[][col],int n,int Cn,dir D)     //排序
{
    int t[col];                                //用于暂存一行数据
    for(int i=0;i<n-1;i++)
        for(int j=i+1;j<n;j++)                 //从待排序序列中选择一个最大(小)的数组元素
            if(a[i][Cn]<a[j][Cn]&&D==Des||a[i][Cn]>a[j][Cn]&&D==Asc)
            {
                memcpy(t,a[i],sizeof(t));      //交换数组行
                memcpy(a[i],a[j],sizeof(t));
                memcpy(a[j],t,sizeof(t));
            }
}
int main(){
{
    const CourseN=5;
    int n,score[][CourseN]={{20140101,1,82,86,0},
                            {20140203,2,80,80,0},
                            {20140204,2,86,90,0},
                            {20140205,2,90,83,0},
                            {20140102,1,75,86,0}};
    n=sizeof(score)/sizeof(score[0]);
    for(int i=0;i<n;i++)                       //计算每个学生的总分
        for(int j=2;j<CourseN-1;j++)
            score[i][CourseN-1]=score[i][CourseN-1]+score[i][j];
```

```
34      sort(score,n,4,Des);                    //按总分降序排序
35      sort(score,n,1,Asc);                    //按班号的升序排序
36      for(i = 0;i < n;i++)                    //输出每个人的成绩与总分
37      {   for(int j = 0;j < CourseN;j++)
38              cout << score[i][j]<<"\t";
39          cout << endl;
40      }
41      return 0;
42  }
```

运行结果：

```
20140101    1  82  86  168
20140102    1  75  86  161
20140204    2  86  90  176
20140205    2  90  83  173
20140203    2  80  80  160
```

程序解释：

第 34、35 行是先按总分降序排序，后按班号的升序排序。这样，在按班级中（排序）时的成绩已经是从高到低有序排列，因此，这种顺序在各班中能保持。

（1）当数组作为形式参数，函数调用时传递的是地址。所以，形式参数中数组的大小（多维数组的最高维）没有意义，可以不带，也可以随便填一个正常数。例如第 10 行改为：

```
sort(int a[1][col],int n,int Cn,dir D)
```

（2）因为形式参数中数组的元素个数没有给定，在函数体中，不知道对哪个范围的元素进行操作，所以还需要添加一个参数 n 来指明存取的范围。显然，这个范围不能大于实在参数中数组的大小。

☆**注意：**

（1）使用数组名传递地址时，虽然传递的是地址，但形参与实参的地址（数组）类型应一致。例如：

```
fun(int a[2][3]);
int b[3],c[3][4],d[5][4][3];
fun(b[3])                   //错误,传递的是数据,不是地址
fun(b);fun(c);fun(d);       //均错误
fun(d[1]);                  //正确
```

（2）形式参数中数组元素的个数没有给定，因此，在函数体中，对数组存取的下标可以为任意值而不会出现编译错误。但是，当这个下标超过了实在参数数组的个数范围时，存取的就不是实在参数数组中的内容了。例如：

```
fun(int a[]) { a[10] = 9;}
int b[4],c[11];
fun(b);                     //在函数体中的 a[10],超越了 b[4]的范围
fun(c);                     //正确
```

3.2.5 字符数组与字符串

存放字符型数据的数组称为**字符数组**，字符数组也分为一维数组和多维数组。前述的数组的定义及初始化同样适用于字符数组，除此以外，C++对字符数组的初始化还可以使用字符串形式。

1. 用字符进行初始化

用字符进行初始化的语法格式与其他类型的数组一样。

例如：

```
char s1[] = {'C','h','i','n','a'};
char s2[][4] = {{'H','o','w'},{'a','r','e'},{'y','o','u'}};
```

2. 用字符串进行初始化

在 C++中，对于字符串的处理可以通过字符数组实现，因此可以用字符串初始化字符数组，其语法格式如下：

```
char 数组名[常量表达式] = {"字符串常量"};
```

例如：

```
char s3[] = "China";
char s4[][4] = {"how","are","you"};
```

前面讲过，字符串是以字符\0 结尾的依次排列的多个(一串)字符，因此用字符串初始化字符数组时，\0 附带在后面与前面的字符一起作为字符数组的元素。用字符串初始化数组后，数组名就是字符串的首地址，数组就是一个字符串。而在使用字符串初始化数组时，除非将\0 作为一个元素放在初值表中，否则\0 不会自动附在初值表中的字符后。因此，一个字符数组不一定是字符串。

在上例中，s1、s3 是一维字符数组，其中 s3 是一个字符串，s1 的大小为 5，s3 的大小为 6。s2、s4 是二维字符数组，其中 s4 是一个以字符串为元素的数组，即字符串数组，两者的长度均为 3。

☆**注意**：

(1) 用字符串初始化字符数组时，系统会在字符数组的末尾自动加上一个字符'\0'，因此数组的大小比字符串中实际字符的个数大 1；字符串长度用 strlen()函数求得。例如：

```
sizeof(s3) = strlen(s3) + 1;
```

(2) 当用字符初始化字符数组时，如果初值表中的初值个数(项数)小于数组的大小，则未指定值的数组元素被赋值为 0，这时字符数组就变成字符串了。

(3) 用字符串初始化一维字符数组时，可以省略花括号{}。

(4) 初始化时，若字符串中的字符个数大于数组长度，系统会提示语法错误。

(5) 使用下列方式初始化一个字符串是错误的：

```
s3[] = '\0';                //错误,试图用字符初始化数组
s3[] = "\0";                //将字符数组初始化成空串
```

3. 字符数组的使用

字符数组也是数组，人们同样可以通过数组名及下标引用数组中的元素。为方便对字符与字符串的处理，C++提供了许多专门处理字符与字符串的函数，这些函数原型在各自的头文件中定义。

表 3-1 列出了常用的字符与字符串处理函数的调用方法与功能简介，函数原型与详细的功能可以从 C++编译器的帮助文档中获得。

表 3-1　常用字符与字符串处理函数

函数的用法	函数的功能	头文件
strlen(字符串)	返回字符串的长度(不包括\0)	Cstring
strset(字符数组,字符 C)	将字符数组中的所有字符都设为指定字符 C,并以\0 结尾	
strlwr(字符串)	将字符串中的所有字符转换成小写字符	
strupr(字符串)	将字符串中的所有字符转换成大写字符	
strcmp(串 s1,串 s2)	比较两个字符串的大小,即按从左到右的顺序逐个比较对应字符的 ASCII 码值。若 s1 大于 s2,返回 1；若 s1 小于 s2,返回－1；若 s1 等于 s2,返回 0。串 s1、s2 可以是字符串常量	
strcpy(串 s1,串 s2)	将字符串 s2 复制到 s1 所指的存储空间中,然后返回 s1。其中,串 s2 可以是字符串常量	
strcat(串 s1,串 s2)	将字符串 s2 连接到 s1 所指的字符串之后的存储空间中,并返回 s1 的值	
toupper(字符)	将小写字符转换成大写字符	Ctype
tolower(字符)	将大写字符转换成小写字符	
atoi(字符串)	将数字字符串转换成整型数	Cstdlib
atol(字符串)	将数字字符串转换成长整型数	
atof(字符串)	将数字字符串转换成浮点数	
ultoa(无符号长整数,字符数组,进制)	将无符号长整型数转换成指定的进制数并以字符串的形式存放到字符数组中	

【例 3-5】 将若干个姓名按字母顺序重新排列，然后从中查找一个指定的名字，输出所在位置。

分析：将上述字符串存放在一个二维数组 NameTab[][]中，将排序程序设计成函数 order()，排序过程使用系统函数 strcmp()和 strcpy()。将查找程序设计成函数 find()，使用系统函数 strcmp()；由于是在排好序的字符串中查找，当进行到当前字符串比要找的大时，就不需要再向后进行了。

```
1  /*************************************
2   *   程序名: p3_5.cpp                  *
3   *   功   能: 字符串的排序与查找       *
4   *************************************/
5  #include<iostream>
6  using namespace std;
7  const NameLen=20;
8  void order(char name[][NameLen],int n)                    //字符串排序
9  {
10     char temp[NameLen];
```

```
    for(int i=0;i<n-1;i++)                                  //选择排序
        for(int j=i+1;j<n;j++)
            if(strcmp(name[i],name[j])>0)                   //比较两个字符串的大小
            {
              strcpy(temp,name[i]);                         //字符串交换
              strcpy(name[i],name[j]);
              strcpy(name[j],temp);
        }
}
int find(char name[][NameLen],int n,char searchname[NameLen])
{
    for(int i = 0;i < n;i++)
        if(strcmp(name[i], searchname) == 0)                //找到,返回位置
            return i + 1;
        else if(strcmp(name[i], searchname)> 0)             //未找完,但找不到,返回 0
            return 0;
    return 0;                                               //找完,找不到,返回 0
}
30  int main()
{
char NameTab[][NameLen] = {"GongJing","LiuNa","HuangPin","ZhouZijun",
    "LianXiaolei","ChenHailing","CuiPeng","LiuPing"};
    char searchname[NameLen];
    int n = sizeof(NameTab)/NameLen;
    order(NameTab,n);
    for(int i = 0;i < n;i++)                                //输出排序后的各姓名
        cout << i + 1 <<'\t'<< NameTab[i]<< endl;
    cout <<"Input the searching name:";
    cin >> searchname;
    if(n = find(NameTab,n, searchname))
        cout <<"Position:"<< n << endl;
    else
        cout <<"Not found!"<< endl;
    return 0;
}
```

运行结果：

```
1    ChenHailing
2    CuiPeng
3    GongJing
4    HuangPin
5    LianXiaolei
6    LiuNa
7    LiuPing
8    ZhouZijun
Input the searching name:LiuPing
Position:7
```

3.3　指　　针

指针是 C++语言最重要的特性之一，也是 C++的主要难点。指针提供了一种较为直观的地址操作的手段，正确地使用指针，可以方便、灵活而有效地组织和表示复杂的数据。指针在 C++程序中扮演着非常重要的角色，从某种程度而言，如果不能深刻地理解指针的概念，正确而有效地掌握指针，就不可能真正学好 C++，但是指针也是学习者最容易产生困惑并导致程序出错的原因之一。

3.3.1　指针的定义与使用

1. 地址与指针

当定义一个变量后，内存中将会划出一块由若干个存储单元组成的区域，用于保存该变量的数据。在内存里每个存储单元都有各自的编号，称为**地址**。如果将计算机的内存比作一条街，那么每个内存单元就是街上的房间，地址就是房间的门牌号码。

在定义一个变量后，变量获得了内存空间，变量名也就成为了相应内存空间的名称。由于人们并不关心变量所在内存单元的实际地址，在程序中往往利用变量名来存取该变量的内容（值），就好像变量名就是变量的值一样。至于变量究竟在内存的哪个单元，如果非要知道不可，可以使用取地址操作符 &。

那么地址又如何存放呢？在 C++中提供了**指针**类型，它是一种用于存放内存单元地址的变量类型，地址就存储在这种指针类型的变量中。正因为指针变量存储的是地址，用它来指明内存单元，所以形象地称这种地址变量为**指针**。指针变量就好像是保存门牌号码的房子。

有时，常把地址变量、地址、地址变量的值均统称为指针。

2. 指针变量的定义

每存储一个地址，就要定义一个指针变量。定义指针变量的格式如下：

```
数据类型　* 变量名;
```

其中：

- 数据类型是指针变量所指向对象的数据类型，它可以是基本数据类型，也可以是构造数据类型以及 void 类型。
- 变量名是用户自定义的标识符。
- * 表示声明的变量是一个指针变量，而不是普通变量。

在定义了指针变量后，无论指针指向何种类型的对象，由于指针变量存放的都是内存单元的地址，所以，其存储空间的大小均为 4 个字节。例如：

```
int * ip;                    //定义了一个 int 型的指针变量 ip
float * fp;                  //定义了一个 float 型指针变量 fp
typedef int A[10];
A * ap;                      //定义了一个 A 类型的指针变量 ap
sizeof(ip) = sizeof(fp) = sizeof(ap) = 4;
```

虽然所有指针变量都占 4 个字节，但不同类型的指针变量用于存放不同的类型变量的地址，上述定义的指针变量 ip 用于存放 int 型变量的地址，fp 用于存放 float 型变量的地址，正像

街道上的商店门牌、住宅门牌、银行门牌要放在不同的地方一样。

指针变量的运算规则与它所指的对象类型是密切相关的，在 C++ 中没有一种孤立的所谓的“地址”类型，因此声明指针时必须明确说明它是用于存放什么类型数据的地址。

3. 指针的初始化与赋值

定义了一个指针，只是得到了一个用于存储地址的指针变量。若指针变量既没有初始化，也没有赋值，其地址值是一个随机的数。也就是说，不能确定指针变量中存放的是哪个内存单元的地址，这时候指针所指的内存单元中有可能存放着重要数据或程序代码，如果盲目去访问，可能会对系统造成很大的危害，因此指针变量在使用之前必须有确定的指向，应先赋值，然后再引用。给指针变量赋值可以在定义变量时进行，语法形式如下：

```
数据类型 *指针变量名 = 初始地址表达式;
```

其中，初始地址表达式可以是地址常量、地址表达式、指针变量表达式。

在定义变量后，使用赋值语句给指针变量赋值的语法形式如下：

```
指针变量名 = 地址表达式;
```

在进行赋值时应注意：

(1) 不要将一个非地址常量、变量以及无意义的实际地址赋给指针变量。例如：

```
int *p = 100;              //错误,100 是一个 int 型常量,不是一个地址常量
int *p = (char *)100;      //危险!100 是一个无意义的实际地址,可能指向危险区域
int *p = NULL;             //一个空指针,也可以用 int *p = 0
```

(2) 可以使用一个已初始化的指针给另一个指针赋值，但类型必须一致，如果不一致，可进行强制类型转换。

```
Char *p = NULL;
int *ip = (int *)p + 100;  //将 char 型指针强制转换成 int 型指针
```

(3) 对于基本数据类型的变量、数组元素，可以使用取地址运算符 & 来获得它们的地址，但是只有类型一致才能赋值。

```
int a[10];                 //定义 int 型数组
int *i_pointer = a;        //定义并初始化 int 型指针
int *ip = &a;              //错误,地址类型不一致,a 的类型是数组,&a 是一个数组类型的地址
int *ip = &a[2];           //正确
char *cp = &a[2];          //错误,cp 的类型是 char 型指针,&a[2]是一个 int 型地址
```

(4) 有一种特殊的 void 类型指针，可以存储任何的类型地址；但将一个 void 类型的地址赋给非 void 类型的指针变量时要使用类型强制转换。

```
void v;                    //错误,不能定义 void 类型的变量
void *vp;                  //定义 void 类型的指针
int *ip,i;
vp = &i;                   //void 类型指针指向整型变量
vp = ip;                   //用 int 型地址给 void 型指针赋值
ip = (int *)vp;            //类型强制转换,void 类型地址赋值给整型指针
```

4. 指针运算

指针变量存放的是地址，因此指针的运算实际上就是地址的运算，但正是由于指针的这一特殊性，使指针所能进行的运算受到了一定的限制。指针通常进行下列几种运算：赋值运算、取值运算、算术运算、相减运算、比较运算。

1）* 和 & 运算

C++提供了两个与地址相关的运算符 * 和 &，对它们在不同情况下的具体含义必须注意区别。

* 称为指针运算符。当出现在数据定义语句中时，* 在数据类型与变量之间，是一个二元运算符，用来定义指针变量；当出现在指针变量表达式左边时，是一个一元运算符，表示访问指针所指对象的内容。作为一元运算符的格式如下：

```
* 指针变量表达式
```

其中，指针变量表达式不能是一个地址常量。例如：

```
int a[4] = {1,2,3};
int * ip = &a[2];
cout << * ip;        //输出 ip 指向单元的内容，内容为整型数 3
 * ip = 100;         //将 100 赋给 a[2]
```

上述程序段在内存中的情况如图 3-3 所示。

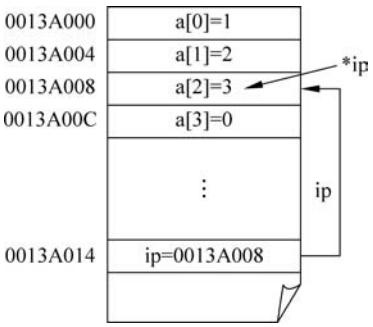

图 3-3　指针变量的使用

& 出现在变量左边时，是一个一元运算符，表示取变量的地址。但 & 操作的对象只能是左值，不能是变量表达式；并且，使用 & 操作得到的地址类型与它作用的变量类型相同。& 运算符常与 * 运算符搭配使用，例如：

```
int a,b;
int * pa, * pb = &b;          //pb 指向 b
pa = &a;                      //pa 指向 a
pa = &(a + 1);                //错误，a + 1 是一个变量表达式，不是一个左值
```

2）指针与整数的加减运算

指针的加减运算与普遍变量的加减运算不同，由于指针存储的是变量的内存地址，指针加上或减去一个整数 n，表示指针从当前位置向后或向前移动 n 个 sizeof(数据类型)长度的存储

单元。因此对于不同的数据类型，n 的实际大小就不同。例如程序段：

```
int b[2][3][4];
typedef char A[10];
int * p1 = b[1][0];
int * p2 = (int * )b[1];
int * p3 = (int * )(b + 1);
double * pd = (double * )p3;
A * pa = (A * )p3;
cout << p1 <<","<< p2 <<","<< p3 <<","<< pd <<","<< pa << endl;
cout << p1 + 1 <<","<< p2 + 1 <<","<< p3 + 1 <<","<< pd + 1 <<","<< pa + 1 << endl;
```

运行结果：

```
0013FF80,0013FF80,0013FF80,0013FF80,0013FF80
0013FF84,0013FF84,0013FF84,0013FF88,0013FF8A
```

p1、p2、p3、pd、pa 以不同的方法指向了 b[1][0][0]，但均加上 1 后，由于 p1、p2、p3 是 int 型指针，因此地址在原来的基础上加上了一个 sizeof(int)，pd 加上了 sizeof(double)，pa 加上了 sizeof(A)。

一般而言，指针的算术运算往往是和数组的使用相联系的，因为只有在使用数组时才可以得到连续分配的可操作的内存空间。对于一个独立变量的地址，如果进行算术运算，结果有可能得到一个异常的地址，因此在对指针进行算术运算时一定要确保运算结果所指向的地址是程序中正确分配的地址。

3）指针自增、自减运算

指针的自增、自减运算是指针加减运算的特例。指针的自增或自减表示指针从当前位置向后或向前移动 sizeof(数据类型)长度的存储单元。请注意下列表达式的计算顺序及含义：

```
int * p, * q, a = 5;
p = &a;
p++;                    //指针 p 后移 4 个字节
* p++;                  //先读取 p 指向的变量 a 的值 5,然后使指针 p 后移 4 个字节
( * p)++;               //读取 p 指向的变量 a 的值,然后使 p 指向的变量 a 自增 1
* ++p;                  //先使指针 p 后移 4 个字节,然后读取 p 指向的变量的值
++ * p;                 //将 p 指向的变量 a 自增 1
* q++ = * p++;          //这是一种常用的表达式,依次执行 * q = * p、q++、p++
```

4）两个指针相减

当两个指针指向同一个数组时，两个指针的相减才有意义。两个指针相减的结果为一个整数，表示两个指针之间数组元素的个数。

5）两个指针的比较运算

两个指针的比较一般用于下面两种情况，一是比较两个指针所指向的对象在内存中的位置关系；二是判断指针是否为空指针。

5. void 类型指针

指向 void 类型的指针是一种不确定类型的指针，它可以指向任何类型的变量。实际使用 void 型指针时，只有通过强制类型转换才能使 void 型指针得到具体变量的值。在没有转换前 void 型指针不能进行指针的算术运算。请看下面的程序段：

```
void * vp;                    //定义了一个 void 型指针 vp
int i = 6, * ip;
vp = &i;                      //vp 指向整型变量 i
cout <<"i = "<< * vp << endl;  //错误
cout <<"i = "<< * (int * )vp << endl;
ip = (int * )vp;              //ip 指向 vp 指向的变量 i
cout <<"i = "<< * ip << endl;
```

即使 void 型指针 vp 指向了整型变量 i,但也不能用 * vp 访问 i 的内容,必须进行强制转换 * (int *)vp 后才能访问 i 的内容,所以 * (int *)vp 才是正确的写法。

3.3.2　指针与字符串

字符型指针用于存放字符型变量的地址,而字符串的本质是以'\0'结尾的字符数组,一个字符型指针存储了字符数组的第一个元素的地址也就存储了字符串的地址,这个指针就指向了字符串。在定义一个字符数组时,可以将一个字符串常量作为初值,但将字符串常量作为初值赋给字符数组和将字符串常量作为初值赋给字符指针变量,二者的含义是不同的。

例如:

```
char str[5] = "abcd";
char * p_str = "abcd";
```

上述字符串的定义有下列不同:

(1) 字符数组 str[5]被赋值为"abcd",因此,数组 str 的 5 个数组元素的值分别为字符'a'、'b'、'c'、'd'和'\0'。字符指针 p_str 被赋值为"abcd",则意味着指针 p_str 的值为字符串常量"abcd"的第一个字符'a'在内存中的地址。

(2) 字符指针 p_str 比 str 多分配了一个存储地址的空间,用于存储字符串的首地址。

当定义了一个指向字符串的指针后,即可以与字符数组相同的方式存取字符串了。具体表现在以下方面:

① 字符数组名与字符指针名等效。

例如"cout << str << endl; "与"cout << p_str << endl; "均输出字符串。

☆**注意**:

(1) 对于字符类型以外的数组与指针,使用 cout 输出的是地址,但对于字符数组与字符指针,用 cout 输出的是字符串。如果要输出字符数组的地址与字符指针的地址,要将它们强制转换成 void 型指针。例如,"cout <<(void *) str << endl;"与"cout <<(void *) p_str << endl;"均输出字符串地址。

(2) 空指针与指向空串的指针是不同的,空指针指针变量的内容为 0;指向空串的指针变量的内容不为 0,但指针指向的第一个单元的内容为 0。对一个空指针指向的内容存取是有危险的。例如:

```
char * p1 = NULL;             //空指针,NULL 为系统常量,其值为 0,常用来表示空指针
char * p2 = 0;                //空指针,0 用来给指针赋初值时,相当于 NULL
char * q1 = "\0";             //空串
char * q2 = '\0';             //空指针
```

② 可以将字符指针当字符数组使用,通过下标存取字符串中的字符:

```
* (p_str + 1);
```

p_str[1]为 p_str+1 指向的字符。

虽然指针变量可以相互赋值，但是对指针指向的字符串进行复制同样不能用简单的赋值语句实现，例如：

```
char *p="abcde";
char *q="12345";
p=q;                          //只是将 p 指向了 q,并没有将 q 复制给 p
```

要将 q 复制给 p，除了使用与复制字符数组相同的 3 种方法外，还可用以下程序段：

```
while(*q)                     //字符串的复制
    *p++=*q++;
    *q='\0';                  //加上字符串结束标志
```

由于字符指针变量本身不是字符数组，如果它不指向一个字符数组或其他有效内存，就不能将字符串复制给该字符指针。一个指针没有指向一个有效内存就被引用，则称为“野指针”操作。“野指针”操作容易引起程序异常，甚至导致系统崩溃，所以字符指针必须先初始化，然后再引用。

例如：

```
char str[12];
char *p_str;
strcpy(p_str,"Hello,");       //野指针操作,p_str 没有指向任何有效内存
p_str=str;                    //指向有效内存
strcpy(p_str,"Hello,");
```

3.3.3 指针与数组

1. 使用指针操作符 * 存取数组

在 C++中，指针与数组有着十分密切的关系。数组是一个在内存中顺序排列的同类型数据元素组成的数据集合，数组的各元素在内存中是按下标顺序连续存放的。指针的加减运算的特点使得指针操作符特别适合处理存储在一段连续内存空间中的同类型数据。这样，使用指针操作符对数组及其元素进行操作就非常方便。

1）一维数组的指针操作

当定义一维数组 T a[N](T 为类型)时，下式为存取数组元素 a[i]的等效方式：

```
*(a+i);
```

而 a+i 为 a[i]的地址。

2）多维数组的指针操作

对于多维数组，使用指针同样可以灵活地访问数组元素。若定义数组 T b[K][M][N]，下式为存取数组元素 a[i][j][k]的等效方式：

```
*(*(*(b+i)+j)+k);
```

其中，*(*(b+i)+j)+k 为 b[i][j][k]的地址，即“&b[i][j][k];”；*(*(b+i)+j)为 b[i][j][0]的地址，即“b[i][j];”；*(b+i)为 b[i][0][0]的地址，即“b[i];”。

2. 指针数组

指针数组是以指针变量为元素的数组，指针数组的每个元素都是同一类型的指针变量。

定义一维指针数组的语法形式如下：

```
类型名　* 数组名[下标表达式];
```

在指针数组的定义中有两个运算符，即 * 和[]，运算符[]的优先级高于 *，所以 * p[N]等价于 *(p[N])，p [N]表示一个数组，而 * 表示数组元素为指针变量。

例如：

```
int * p_a[5];
```

定义了一个指针数组，其中有 5 个数组元素，每个数组元素都是一个 int 类型的指针，也可以用下列方式定义。

(1) 定义指针：typedef int * I_P;

(2) 定义数组：I_P p_a[5]。

指针数组的一个常用用法是用它来存储若干行字符串，由于每行字符串的长度不一样，如果用二维字符数组存储将浪费空间。例如：

```
char * sentence[ ] = {"I'm a Chinese.",
                      "I come from Beijing.",
                      "I like computer.",
                      "Programming is fun!"
                     };
```

☆**注意**：

(1) 对指针数组的存取同样可以使用指针方式、数组方式以及指针与数组结合的方式。一维指针数组与二维指针数组对应，可采用存取二维数组的方式存取。

(2) 指针数组的每一个元素都是一个指针，因此必须先赋值，后引用。

3. 数组指针

数组指针是指向数组的指针。定义一维数组指针的语法形式如下：

```
类型名 ( * 指针名) [下标表达式];
```

虽然运算符[]的优先级高于 *，但是在数组指针的定义式中(* 指针名)改变了这种优先级，它表明定义的首先是一个指针，然后才是什么类型的指针。例如：

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

定义了一个指针，用于指向一个大小为 5 的 int 型数组，等效于下列定义方式。

(1) 定义数组类型：typedef int I_A[5];

(2) 定义指向数组的指针：I_A * a_p。

指针数组与数组指针的内存图如图 3-4 所示。

☆**注意**：

在给数组指针赋值时类型要匹配，例如：

```
int a[5];
int b[6];
a_p = a;                          //错，因为 a 是一个 int 型的地址
```

```
a_p = &b;                    //错,&b 是一个 int[6]的地址,而 a_p 指向的类型为 int[5]
a_p = &a;                    //正确
```

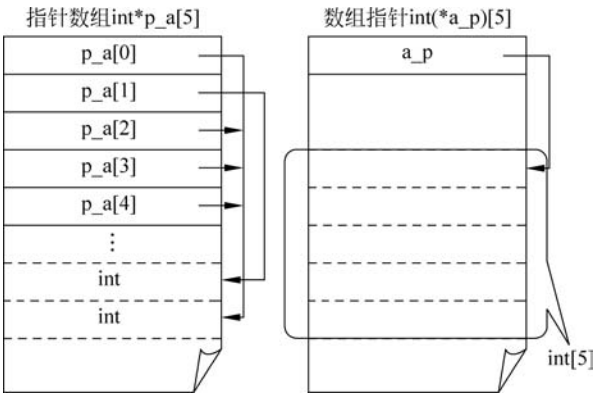

图 3-4　指针数组与数组指针

3.3.4　多重指针

如果已经定义了一个指针类型，我们再定义一个指针，用于指向已经定义的指针变量，后面定义的指针变量就是一个指向指针的指针变量，简称指向指针的指针，这样的指针也称二重(级)指针。其语法格式如下：

```
类型名　** 变量名;
```

其中，两个星号 ** 表示定义的是一个指向指针的指针变量。

例如：

```
int ** pp;
```

定义了一个二级指针变量，等效于下列定义方式：

```
typedef int * P;
P * p;
```

二级指针常用来指向一个指针数组。

例①：

```
int a[2][3] = {1,2,3,4,5,6};  //声明并初始化二维数组
int * p_a[3];                 //声明整型指针数组
p_a[0] = a[0];                //初始化指针数组元素
p_a[1] = a[1];
int ** pp;
pp = * p_a;
```

☆**注意：**

在使用二级指针时容易犯两类错误。

(1) 类型不匹配。例如：

```
pp = a;                       //错误,因为 pp 是一个 int ** 型变量,a 是一个 int[2][3]型的地址
```

```
pp = &a;                    //错误,pp 是一个 int ** 型变量,&a 是一个( * )int[2][3]型的地址
pp = &a[0];                 //错误,pp 是一个 int ** 型变量,&a[0]是一个( * )int[3]型的地址
```

(2) 指针没有逐级初始化。例如：

```
int i = 3;
int ** p2;
* p2 = &i;
** p2 = 5;
cout << ** p2;
```

虽然上述程序段编译、连接时均没有错误,但运行时出错。其原因在于"int ** p2;"只定义了一个指针变量,变量中的内容(地址)是一个无意义的地址,而"* p2＝&i;"是对无意义的内存单元赋值,这样是错误与危险的,正确的方法是从第一级指针开始逐级初始化。

例②：

```
int i = 3;
int ** p2;
int * p1;
* p1 = &i;                   //初始化一级指针
p2 = * p1;                  //初始化二级指针
** p2 = 5;                   //通过指针给变量 i 赋值
cout << ** p2;              //结果为 5
```

上述两个二级指针在内存中的分布如图 3-5 所示。

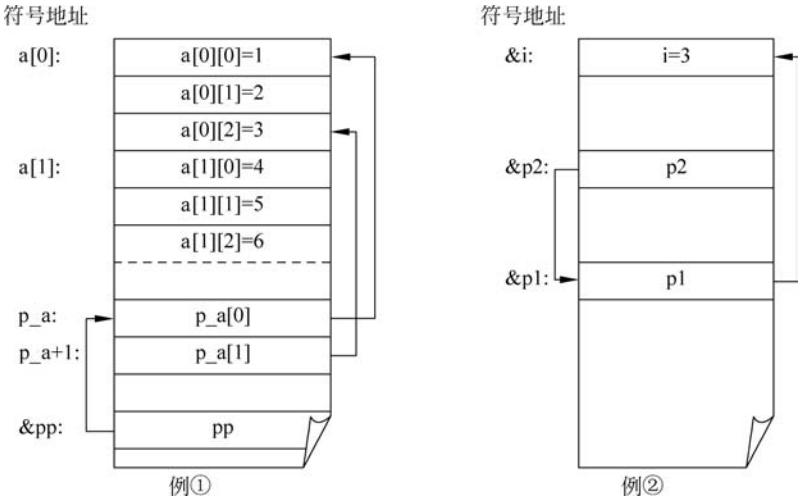

图 3-5　二级指针在内存中的分布

当一个二级指针指向一个指针数组后,对数组元素的存取可以使用指针方式、数组方式、混合方式：

```
p[i][j];                    //存取数组元素 a[i][j]的数组方式
*(*(p+i)+j);                //存取数组元素 a[i][j]的指针方式
*(p[i]+j);                  //存取数组元素 a[i][j]的混合方式
```

三重及以上的指针统称为多重指针,要定义一个 n 重指针(变量),需要 n 个 * 。例如：

```
int *** p;                  //定义了一个三重指针
```

一般而言，二重指针就足够使用了，三重指针的使用场合很少，使用多重指针要注意下面两点：

（1）不管是多少重指针，定义后只建立了一个指针变量，分配一个指针变量的空间。

（2）初始化多重指针要从第一层开始逐步向高层进行。

3.3.5 动态内存分配

对于类型相同的大量数据，我们可以用数组来存储。数组是一种静态内存分配方法，其长度是固定的，由数组定义语句确定，由编译系统予以分配。

在程序的运行过程中，有时我们很难事先确定需要多少数组元素，只能按预先估计的最大可能数量进行定义，这有可能造成内存的巨大浪费，而且一旦超过预先定义的最大长度，数组就会越界，造成程序异常乃至系统崩溃。

在 C++中，动态内存分配技术可以保证程序在运行过程中根据实际需要申请适量的内存，使用结束后还可以释放。C++通过 new 运算和 delete 运算实现动态内存分配。

1. new 运算

new 运算的作用是按指定类型和大小动态地分配内存，基本语法形式如下：

```
指针变量 = new 类型名(初值列表);
```

其中：

- 数据类型可以是基本数据类型，也可以是用户自定义的复杂数据类型。
- new 运算符在堆(内存)中创建一个由类型名指定类型的对象，如果创建成功，返回对象的地址，否则返回空指针 NULL。
- 初值列表给出被创建对象的初始值。
- 由于返回的是地址，所以要用事先定义一个类型相同的指针变量来存储这个地址。

例①：

```
int * ip;
ip = new int(5);                    //ip 指向一个初值为 5 的 int 型对象
```

也可以使用一条语句定义：

```
int * ip = new int(5);
```

首先定义了一个整型指针 ip，然后申请内存，创建一个 int 型数据对象，并将该数据对象初始化为 5，最后返回创建的数据对象的地址存入 ip。

使用 new 运算可以创建一个数据对象，也可以创建同类型的多个对象——数组。由于数组大小可以动态给定，所创建的对象称为动态数组。new 在创建动态数组时，需要给出数组的结构说明。其语法格式如下：

```
指针变量 = new 类型名 [下标表达式];
```

其中：

- 下标表达式与数组初始化时的常量表达式不同，可以是变量表达式。
- 用 new 申请失败时返回 NULL，申请一个动态数组往往需要较大的空间，因此在程序

中需要对 new 的返回值进行判断，看是否申请成功。

例②：

```
int * pa;
pa = new int [5];                    //pa 指向 5 个未初始化的 int 型数据对象的首地址
```

也可以使用一条语句定义：

```
int * pa = new int[5];
```

使用 new 也可以创建多维数组，其语法形式如下：

```
指针变量 = new 类型名 T [下标表达式 1][下标表达式 2][…]
```

其中：

- 当用 new 创建多维数组时，只有下标表达式 1 可以是任意正整数的表达式，而其他下标表达式必须是值为正整数的常量表达式。
- 如果内存申请成功，new 运算返回一个指向新分配内存首地址的指针，它是一个 T 类型数组的指针，而不是 T 类型指针。数组元素的个数为除最左边一维（最高维）外各维下标表达式的乘积。

例③：

```
int * pb;
pb = new int[3][4][5];
```

☆**注意**：

（1）用 new 运算符申请分配的内存空间必须用 delete 释放。

（2）delete 作用的指针必须是由 new 分配内存空间的首地址。

（3）对于一个已分配内存的指针，只能用 delete 释放一次。

（4）new 和 delete 运算实现了堆空间的动态分配，它在带来方便的同时也潜伏了可能的隐患：使用 new 进行内存分配之后，忘记了使用 delete 运算进行内存回收，即“内存泄露”，如果程序长时间运行，则有可能因内存耗尽而使系统崩溃，所以对 new 和 delete 要养成配对使用的良好习惯。

（5）如果使用 pb=new int[3][3][5]也是错误的，因为 new 返回的是一个 int[3][5]型地址，而 pb 是一个 int[4][5]型指针变量，因此指针变量类型应与 new 返回的地址类型一致。

上面的语句会产生错误，因为在这里 new 操作产生的是指向一个 int[4][5]的二维 int 型数组的指针，而 pb 是一个 int 型数据的指针。

正确的写法如下：

```
int ( * pb)[4][5];
pb = new int[3][4][5];
```

如此得到的指针 pb 既可以作为指针使用，也可以像一个三维数组名一样使用。

2. delete 运算

当程序不再需要由 new 分配的内存空间时，可以用 delete 释放这些空间。其语法格式如下：

```
delete 指针变量名;
```

如果删除的是动态数组，则语法格式如下：

```
delete [] 指针变量名;
```

其中：

- 括号[]表示用 delete 释放为多个对象分配的地址，[]中不需要说明对象的个数。
- 不管建立的动态数组是多少维，释放的格式都是一样的。

对于例①、②、③分配的空间，释放语句如例④：

例④：

```
delete ip;
delete [] pa;
delete [] pb;
```

3. 动态存储的应用

前面讲述的指针数组虽然每行元素的个数可以不同，但行数（指向每行的指针数）一定，无法满足计算过程中要求行数可变的应用需求。例如，计算过程中需要根据计算结果 m、n 建立 m×n 维动态二维矩阵。

【例 3-6】 输入 m、n，建立 m×n 维动态二维矩阵。

分析：建立一个动态的二维数组存储矩阵要比建立一维数组复杂得多，步骤如下：

① 建立一个二重指针；

② 二重指针指向动态分配的 m 维指针数组；

③ 为指针数组的每个指针动态分配空间，并使指针指向它。

在使用 delete 释放时，其顺序与分配顺序相反：

① 释放指针数组的每个指针的空间；

② 释放指针数组的空间。

```
1  /************************************
2   *   程序名: p3_6.cpp                *
3   *   功   能: 动态二维矩阵           *
4   ************************************/
5  #include<iostream>
6  using namespace std;
7  int main()
8  {
9      int m,n;
10     int **dm;
11     cout<<"input matrix size m,n:";
12     cin>>m>>n;
13     dm=new int *[m];                    //建立 m 个指针,存储 m 行
14     for(int i=0;i<m;i++)                //为每行分配 n 个空间
15         if((dm[i]=new int [n])==NULL)
16         exit(0);
```

```
17      for(i = 0;i < m;i++)                        //输入矩阵元素
18      {
19          for(int j = 0;j < n;j++)
20              cin >> dm[i][j];
21      }
22      for(i = 0;i < m;i++)                        //输出矩阵元素
23      {
24          for(int j = 0;j < n;j++)
25              cout << dm[i][j]<<"\t";
26          cout << endl;
27      }
28      for(i = 0;i < m;i++)                        //释放 m 个指针指向的空间
29          delete[ ]dm[i];
30      delete[ ]dm;                                //释放 m 个指针
31    return 0;
32  }
```

运行结果：

```
input matrix size m,n:2 3 ↙
1 2 3↙
4 5 6↙
1    2    3
4    5    6
```

程序解释：

(1) 第 20、25 行输入、输出时对数组元素的存取可以写成指针形式：

```
*(*(dm + i) + j);
```

(2) 若要存储若干行文本，由于各行文本有长有短，为了节省存储空间，需要为每行分配不同的空间。将第 15 行修改成分配各行需要的空间数即可。

思考：

建立一个动态数组后，在程序的运行过程中若发现空间不够，应做以下步骤的处理：

(1) 重新分配一个容量足够大的动态数组；

(2) 将原来的动态数组中的内容复制到新数组中；

(3) 释放原来的数组的空间。

【例 3-7】 动态创建多维数组。

分析：当用 new 动态创建多维数组时，如果内存申请成功，new 运算返回一个指向新分配内存首地址的指针，它是一个 **T 类型数组**的指针，而不是 T 类型指针，数组元素的个数为除最左边一维(最高维)外各维下标表达式的乘积。

```
1  /*************************************
2   *   程序名: p3_7.cpp                  *
3   *   功   能: 动态创建多维数组        *
4   *************************************/
5  #include <iostream>
```

```
using namespace std;
int main()
{
    float ( * p)[3][4];
    int i,j,k;
    p = new float[2][3][4];
    for (i = 0; i < 2; i++)
      for (j = 0; j < 3; j++)
        for (k = 0; k < 4; k++)
          * ( * ( * (p + i) + j) + k) = i * 100 + j * 10 + k;        //通过指针访问数组元素
    for (i = 0; i < 2; i++)
    {
      for (j = 0; j < 3; j++)
      {
        for (k = 0; k < 4; k++)
          //将指针 cp 作为数组名使用,通过数组名和下标访问数组元素
          cout << p[i][j][k]<<" ";
        cout << endl;
      }
    cout << endl;
  }
  return 0;
}
```

运行结果：

```
0  1  2  3
10  11  12  13
20  21  22  23

100  101  102  103
110  111  112  113
120  121  122  123
```

思考：

建立一个动态数组后，如果在程序的运行过程中数组溢出，程序运行结果又将是什么？请读者尝试修改程序，使数组溢出并分析程序的运行结果。

3.3.6 指针与函数

1. 指针作为函数参数

在 C++语言中，函数之间的数据传递有多种方法，主要有值传递方式和地址传递方式两种，而地址传递方式又有指针方式和引用方式。

当需要在不同的函数之间传递大量数据时，程序执行时调用函数的开销就会比较大，这时，如果需要传递的数据是存放在一个连续的内存区域中，就可以只传递数据的起始地址，而不必传递数据的值，这样就会减小开销、提高效率。C++的语法对此提供了支持，函数的参数不仅可以是基本类型的变量、对象名、数组名或函数名，而且可以是指针。

指针作为函数参数是一种地址传递方式。指针可以作为函数的形参，也可以作为函数的

实参。如果以指针作为函数的形参，在调用时实参将值传递给形参，也就是使实参和形参指针变量指向同一内存地址，这样在子函数运行过程中，通过形参指针对数据值的改变也同样影响着实参所指向的数据值，从而实现参数双向传递的目的，即通过在被调用函数中直接处理主调函数中的数据而将函数的处理结果返回其调用者。

【例 3-8】 用传指针方式实现两个数据交换。

分析：用传值的方式无法实现两个数据交换。用指针作为形参，从实参传入要交换数据的地址，在函数体内将指针指向的两个数据交换存储位置，这样通过“釜底抽薪”的方式实现了数据交换。为了实现不同类型的数据交换，形式参数采用 void 指针。

```
/*************************************************
*   程序名: p3_8.cpp                             *
*   功   能: 实现两个数据交换的通用程序          *
*************************************************/
#include <iostream>
using namespace std;
void swap_i(int *num1,int *num2)                    //整型数交换
{   int t;
    t=*num1;
    *num1=*num2;
    *num2=t;
}
void swap(void *num1,void *num2,int size)           //所有类型数据交换
{   char *first=(char *)num1,*second=(char *)num2;
    for(int k=0;k<size;k++)
    {
       char temp;
       temp=first[k];
       first[k]=second[k];
       second[k]=temp;
    }
}
int main()
{
  int a=3,b=6;
  double x=2.3,y=4.5;
  char c1[8]="John",c2[8]="Antony";
  cout<<"before swap:a="<<a<<" b="<<b<<endl;
  swap_i(&a,&b);
  cout<<"after swap:a="<<a<<" b="<<b<<endl;
  cout<<"before swap:x="<<x<<" y="<<y<<endl;
  swap(&x,&y,sizeof(x));
  cout<<"after swap:x="<<x<<" y="<<y<<endl;
  cout<<"before swap:c1="<<c1<<" c2="<<c2<<endl;
  swap(&c1,&c2,sizeof(c1));
  cout<<"after swap:c1="<<c1<<" c2="<<c2<<endl;
  return 0;
}
```

运行结果：

```
before swap:a = 3 b = 6
after swap:a = 6 b = 3
before swap:x = 2.3 y = 4.5
after swap:x = 4.5 y = 2.3
before swap:c1 = John c2 = Antony
after swap:c1 = Antony c2 = John
```

另外，常用指针作为形参接受指针、数组作为实参传送过来的地址，在函数体内，可以使用数组、指针形式存取实参的值。表 3-2 是常用的传递实例，完整的程序请读者自己完成。

表 3-2　指针作为实参的实例

数据存储方式	函数原型	调用形式	功能说明
int a[]={1,2,3,4,5};	int sum(int * p,int n)	sum(a,5);	指针指向的 n 个数求和
char * p="abcde";	toupper(char * p)	toupper(p);	指针指向的字符串变成大写
int a[][3]={1,2,3,4,5,6};	ds(int (* p)[3],int m,int n)	ds(a,2,3)	显示二维静态数组的各元素
int ** dm; //动态数组	dd(int ** p,int m,int n)	dd(dm,m,n)	显示二维动态数组的各元素

2. 指针型函数

除 void 类型的函数以外，每个被调用函数在调用结束之后都要有返回值，指针也可以作为函数的返回值。当一个函数声明其返回值为指针类型时，这个函数就称为指针型函数。

指针型函数的定义形式如下：

```
数据类型 * 函数名(参数表);
```

其中，数据类型表明函数返回指针的类型；* 表示函数的返回值是指针型，可以使用多个 * 返回多重指针。

使用指针型函数的最主要目的就是要在函数调用结束时把大量的数据从被调用函数返回到主调函数中，而通常非指针型函数在调用结束后只能返回一个值。

在调用指针型函数时，需要一个同类型的指针变量接受返回的值。

【例 3-9】 用指针型函数返回两个超长整数的和。

分析：当一个整数大于 10^{10} 时，C++ 就无法用整型变量进行存储与运算了。如果采用字符串存储整数，则可以存储任意长的整数。数字串中的数字字符不是数，要将它们变成数，从低位到高位逐位相加、进位，最后将每位数变成数字字符。

数字字符与数的关系为数字字符＝数＋'0'。

```
1  /**********************************
2  *   程序名: p3_9.cpp               *
3  *   功  能: 超长整数加法           *
4  **********************************/
5  #include <iostream>
6  using namespace std;
7  char * ladd(char * s1,char * s2)
8  {
```

```
9       int n1,n2,n;
10      char * res,c = 0;
11      n1 = strlen(s1);                                    //n1 = 数字串 s1 的长度
12      n2 = strlen(s2);                                    //n2 = 数字串 s2 的长度
13      n = n1 > n2 ? n1 :n2;                               //数字串 s1、s2 的最大长度
14      res = new char [n + 2];                             //申请存结果串的内存
15      for(int i = n + 1;i > = 0;i -- )                    //将 s1 从低位开始搬到 res,没有数字的位
                                                            //以及最高位填'0'
16          res[i] = i > n - n1 ? s1[i - n - 1 + n1] :'0';
17      for(i = n;i > = 0;i -- )
18      {
19          char tchar;
20          tchar = i > n - n2 ? res[i] - '0' + s2[i - n + n2 - 1] - '0' + c :res[i] - '0' + c;
                                                            //将数字符变成数
21          c = tchar > 9 ? 1 :0;                           //设进位
22          res[i] = c > 0 ? tchar - 10 + '0' :tchar + '0'; //将数字变成数字字符
23      }
24      return res;
25  }
26  int main()
27  {
28      char num1[100],num2[100], * num;
29      cin >> num1 >> num2;
30      num = ladd(num1,num2);
31      cout << num1 <<" + "<< num2 <<" = "<< num << endl;
32      delete [] num;
33      return 0;
34  }
```

运行结果：

```
12345678901234 ↙
99999 ↙
12345678901234 + 99999 = 012345679001233
```

程序解释：

(1) 第 14 行为结果数字串分配内存空间。如果在函数体内使用一个临时数组存储数字串，函数调用结束后临时空间就不存在了，返回的是一个错误的地址。

(2) 第 31 行释放数字串所占空间。使用“cout << ladd(num1,num2)<< endl;”同样可以调用函数进行运算，并显示结果，但每次调用函数 ladd 时分配的内存空间都没有释放。如果调用次数多了，大量的空间不能释放，就会造成死机。因此要避免编写在函数体内申请空间、在函数体外释放空间的程序。

3. 指向函数的指针

在程序运行时，不仅数据要占用内存空间，程序的代码也被调入内存并占据一定的内存空间。每一个函数都有函数名，实际上，这个函数名就是该函数的代码在内存中的起始地址。当调用一个函数时，编译系统就是根据函数名找到函数代码的首地址，从而执行这段代码。由此看来，函数的调用形式为**函数名(参数表)**，其实质就是**函数代码首地址(参数表)**。

函数指针就是指向某个函数的指针，它是专门用于存放该函数代码首地址的指针变量。一旦定义了某个函数指针，那么它就与函数名有同样的作用，在程序中就可以像使用函数名一样通过指向该函数的指针来调用该函数。类似于指向某个数组的指针一样，它代表该数组的首地址，可以通过该指针访问数组元素。

函数指针的定义语法形式如下：

```
数据类型 (*函数指针名)(形参表);
```

其中：

- 数据类型为函数指针所指函数的返回值类型；形参表则列出了该指针所指函数的形参类型和个数。
- 函数指针名与 * 外面的圆括号( )是必需的，圆括号改变了默认的运算符的优先级，使得该指针变量被解释为指向函数的指针。如果去掉圆括号，将被解释为函数的返回值为指针。

函数指针在使用之前也要进行赋值，使指针指向一个已经存在的函数代码的起始地址。其语法形式如下：

```
函数指针名 = 函数名;
```

赋值符右边的函数名所指出的必须是一个已经声明过的和函数指针具有相同返回类型和相同形参表的函数。在赋值之后，就可以通过函数指针名直接引用该指针所指向的函数了，即该函数指针可以和函数名一样出现在函数名能出现的任何地方。

调用函数指针指向的函数有以下两种格式：

```
① 函数指针名(实参表);
② (* 函数指针名)(实参表);
```

例如：

```
int add(int a,int b);              //定义函数
int (*fptr)(int a,int b);          //定义函数指针
fptr = add;                        //函数指针赋值,最好是 fptr = &add
```

采用下列任何一种形式调用函数 add：

```
add(1,2);                          //用函数名调用函数 add
(*fptr)(1,2);                      //用指向函数的指针调用函数 add
fptr(1,2);                         //用指向函数的指针调用函数 add
```

虽然 3 种调用形式的结果完全相同，当用指向函数的指针调用函数 add()时，习惯上使用(*fptr)(1,2)，因为这种形式能更直观地说明是用指向函数的指针来调用函数的。

指向函数的指针常用来实现菜单程序，根据不同的选择执行菜单项中对应的功能，各功能由指向函数的指针实现。

【例 3-10】 用指向函数的指针实现各种算术运算的菜单程序。

分析：为了实现菜单功能，需要定义一个函数指针数组存储各函数名(地址)，调用时通过

各数组元素指向的函数名来调用对应的函数。

```
/**********************************
*   程序名：p3_10.cpp               *
*   功　能：模拟简单菜单程序        *
**********************************/
#include <iostream>
using namespace std;
int add(int a,int b) {
    return a + b;
}
int sub(int a,int b) {
    return a - b;
}
int mul(int a,int b) {
    return a * b;
}
int divi(int a,int b) {
    if (b == 0) return 0x7fffffff;
    else return a/b;
}
int ( * menu[ ])(int a,int b) = {add,sub,mul,divi};
int main()
{
    int num1,num2,choice;
    cout <<"Select operator:"<< endl;
    cout <<"     1:add"<< endl;
    cout <<"     2:sub"<< endl;
    cout <<"     3:multiply"<< endl;
    cout <<" 4:divide"<< endl;
    cin >> choice;
    cout <<"Input number(a,b):";
    cin >> num1 >> num2;
    cout <<"Result:"<< menu[choice - 1](num1,num2)<< endl;
    return 0;
}
```

运行结果：

```
Select operator:
    1:add
    2:sub
    3:multiply
    4:divide
3↙
Input number(a,b):4 5↙
Result:20
```

思考：

如果 p3_10 的运行结果中选择操作为 4↙，输入为 Input number(a,b):5 0↙输出结果将是什么？

3.3.7 指针常量与常量指针

一个指针可以操作两个对象，一个是指针值（即地址），一个是通过指针间接访问的变量的值，于是指针也可分为指针常量（constant pointer）和常量指针（pointer to constant）。

1. 指针常量

指针常量是相对于指针变量而言的，也就是指针值不能被修改的指针。

如果在定义指针时指针前用 const 修饰，被定义的指针就变成了一个指针类型的常量，指针类型的常量简称为**指针常量**。定义指针常量的格式如下：

```
数据类型 *const 指针名 = 变量名;
```

（1）修饰符 const 与指针变量紧邻，说明指针的值不允许修改，所以一定要在定义时给出初值：

```
int a = 2;
int b = 3;
int *const p = &a;              //定义了一个指针常量并初始化
p = &b;                         //错误，指针常量的值为常量，不能指向其他变量
p = NULL;                       //错误，指针常量的值为常量，不能被修改
```

（2）因为 const 修饰的是指针，而不是指针指向的对象的值，所以指针指向的对象的值可以被更改：

```
*p = 4;                         //正确，指针常量所指变量的值可以被修改
*p = b;                         //正确，指针常量所指变量的值可以被修改
```

2. 常量指针

如果在定义指针变量时数据类型前用 const 修饰，被定义的指针变量就是指向常量的指针变量，指向常量的指针变量简称为**常量指针**。定义常量指针的格式如下：

```
const 数据类型 *指针变量 = 变量名;
```

（1）定义一个常量指针后，指针指向的对象的值不能被更改，即不能通过指针来更改所指向的对象的值，但指针本身可以改变，指向另外的对象。

```
const int a = 2;
int b = 3;
const int *p = &a;              //定义了一个常量指针并初始化
p = &b;                         //正确，常量指针可以指向其他变量
p = NULL;                       //正确，指针常量的值可以被修改
```

（2）因为 const 修饰的是指针指向的值，而不是指针，所以指针指向的值不能被更改。

```
*p = 4;                          //错误, 常量指针所指变量的值不可以被修改
*p = b;                          //错误, 常量指针所指变量的值不可以被修改
```

(3) 为了防止通过一个非常量指针修改常量指针指向的值,将一个常量指针赋给一个非常量指针是错误的。

```
int *q;
q = p;                           //错误,将一个常量指针赋给非常量指针
```

(4) 可以将一个非常量指针赋给一个常量指针,这是因为 const 修饰的是指针指向的内容,而不是指针的值(地址值)。

```
p = NULL;                        //正确
p = q;                           //正确
```

(5) const 用在数组的类型前修饰数组元素,数组元素为常量的数组称为**常量数组**,常量数组的元素不可改变,也不可将地址赋给非常量指针。

```
const int a[3] = {1,2,3};        //定义常量数组
a[0] = 0;                        //错误,常量数组的元素不可修改
int *p = a;                      //错误,常量数组的地址不能赋给非常量指针
```

☆**注意**:

常量数组的所有元素必须全部赋初值。

```
const int a[] = {1,2,3};         //正确
const char a[] = {'1','2','3'};  //正确
const int a[10] = {1,2,3};       //错误,常量数组元素没有全部赋初值
```

3. 指向常量的指针常量

指针常量保护指针的值不被修改,常量指针保护指针指向的值不被修改,为了将两者同时保护,可以定义指向常量的指针常量,简称为**常指针常量**。常指针常量意为一个指向常量的指针,指针值本身也是一个常量。常指针常量的定义格式如下:

```
const 数据类型 *const 指针变量 = 变量名;
```

(1) 左边的 const 与数据类型相结合,表明数据的值是常量;右边的 const 用在变量前,表明变量的值是常量。

(2) 定义一个常指针常量后,修改指针的值与修改指针指向内容的值都是错误的:

```
const int a = 2;
int b = 3;
int *q = &b;
const int * const p = &a;        //定义了一个常量指针并初始化
p = &b;                          //错误,常指针常量的值为常量,不能指向其他变量
p = NULL;                        //错误,常指针常量的值为常量,不能被改变
*p = b;                          //错误,常指针常量所指的值为常量,不能被改变
p = q;                           //错误,常指针常量的值为常量,不能被改变
q = p;                           //错误,不能将一个常指针赋给非常指针
```

常指针类型通常用作函数的形参，以防止在函数体内通过形参修改实参指向的值，以保护实参。

3.4 引　　用

3.4.1 引用的定义

从逻辑上理解，**引用**(reference)是已存在变量的别名(alias)。通过引用，我们可以间接访问变量，指针也能间接访问变量，但引用在使用上比指针更安全。引用的主要用途是描述函数的参数和返回值，特别是传递较大的数据变量。

对引用型变量的操作实际上就是对被引用变量的操作。当定义一个引用型变量时，需要用已存在的变量对其初始化，于是引用就被绑定在那个变量上，对于引用的改动就是对它所绑定的变量的改动，反之亦然。

定义一个引用型变量的语法格式如下：

```
数据类型 & 引用变量名 = 变量名;
```

其中：

- 数据类型应与被引用变量的类型相同；
- & 是引用运算符，在这里是二元操作符；
- 变量名为已定义的变量。

例如：

```
int x;
int & refx = x;
```

refx 是一个引用型变量，它被初始化为对整型变量 x 的引用。即给整型变量 x 起了一个别名 refx，refx 称为对 x 的引用，x 称为 refx 的引用对象。

当定义一个引用变量后，系统并没有为它分配内存空间。refx 与被引用变量 x 具有相同的地址，即 refx 与 x 使用的是同一内存空间。对引用变量值的修改就是对被引用变量的修改，反之亦然。

```
x = 3;
cout << refx;                    //结果为 3
refx = 5;
cout << x;                       //结果为 5
```

引用变量的内存图见图 3-6。

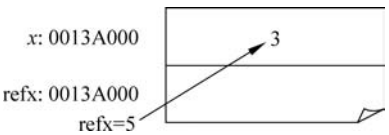

图 3-6　引用变量的内存图

从以上分析可知，从物理实现上看，引用是一个隐性的指针，即引用值是引用所指向的变量的值。引用与所引用的变量值的关系看似直接访问，实为间接访问，这幕后的转换工作是由编译做的。编译将引用转换为指针操作，因而，引用不能操作自身的地址，每次访问引用，实际上是在访问所指向的变量。

与指针相比，引用不占用新的地址，节省了内存开销，而且隐去了地址操作。一方面，引用封锁了对地址的可修改性，使得间接访问操作更安全了。指针是低级的直接操作内存地址的机制，其功能强大，但若使用不慎极易产生错误。在 C++语言中，指针可由整型数强制类型转换得到，处理不当可能对系统造成极大的破坏。另一方面，引用封装了指针，它不直接操作地址，不可以由强制类型转换得到，因而具有较高的安全性，避免了由于使用指针直接访问内存地址而产生的一些难以察觉的错误，因而，高级编程多用引用，低级编程多用指针，这主要是从安全因素着眼的。

3.4.2　引用与函数

1. 引用作为函数的参数

若引用作为函数的形参，在进行函数调用时，进行实参与形参的结合，其结合过程相当于定义了一个形参对实参的引用。因此，在函数体内，对形参进行运算相当于对实参进行运算。

【例 3-11】 使用引用参数实现两个数的交换。

```
1  /*********************************************
2   *   程序名: p3_11.cpp                       *
3   *   功   能: 使用引用参数实现两个数的交换    *
4   *********************************************/
5  #include <iostream>
6  using namespace std;
7  void swap(int &refx,int &refy)
8  {
9      int temp;
10     temp = refx;
11     refx = refy;
12     refy = temp;
13 }
14 int main()
15 {
16     int x = 3,y = 5;
17     cout <<"before swap:x = "<< x <<" y = "<< y << endl;
18     swap(x,y);
19     cout <<"after swap:x = "<< x <<" y = "<< y << endl;
20     return 0;
21 }
```

运行结果：

```
before swap:x = 3 y = 5
after swap:x = 5 y = 3
```

程序解释：

执行 swap(x, y)时，首先进行参数传递，其过程相当于执行：

```
int& refx = x;
int& refy = y;
```

这样定义了对 x、y 的引用 refx、refy。在函数体中，交换变量 refx 和 refy 的值就是交换变量 x 和 y 的值。

与指针相比，引用作为函数参数具有下面两个优点：

（1）函数体的实现比指针简单。用指针作为形参，在函数体内形参要带着 * 参加运算；而用引用作为形参，在函数体内参加运算的为形参变量。

（2）调用函数语法简单。用指针作为形参，实参需要取变量的地址；而用引用作为形参，与简单传值调用一样，实参为变量。

2. 引用作为函数的返回值

函数返回值类型为引用型，在函数调用时，若接受返回值的是一个引用变量，相当于定义了一个对返回变量的引用；若接受返回值的是一个非引用变量，函数返回变量的值赋给接受变量。

如果函数返回值类型为引用型，则要求返回值为左值。这样，函数调用式可以当作左值。

【例 3-12】 使用引用作为函数的返回值。

```
/*********************************************
*   程序名: p3_12.cpp                          *
*   功  能: 使用引用作为函数的返回值           *
*********************************************/
#include <iostream>
using namespace std;
int max1(int a[],int n)          //求数组 a[]中元素的最大值
{
  int t = 0;
  for(int i = 0;i < n;i++)
      if(a[i]> a[t]) t = i;
    return a[t] + 0;
}
int& max2(int a[],int n)         //求数组 a[]中元素的最大值
{
  int t = 0;
  for(int i = 0;i < n;i++)
      if(a[i]> a[t]) t = i;
return a[t];
}
int& sum(int a[],int n)          //求数组 a[]中元素的和
{
  int s = 0;
  for(int i = 0;i < n;i++)
      s += a[i];
  return s;
}
int main()
```

```
29 {
30      int a[10] = {1,2,3,4,5,6,7,8,9,10};
31      int m1 = max1(a,10);
32      int m2 = max2(a,10);
33      int &m3 = max2(a,10);
34      int &m4 = sum(a,10);
35      cout <<"m1 = "<< m1 << endl;
36      cout <<"m2 = "<< m2 << endl;
37      cout <<"m3 = "<< m3 << endl;
38      cout <<"m4 = "<< m4 << endl;
39      m3 += 10;
40      max2(a,10) -= 100;
41      cout << sum(a,10)<< endl;
42      return 0;
43 }
```

运行结果：

```
m1 = 10
m2 = 10
m3 = 10
m4 = -858993460
 -35
```

程序解释：

(1) 第 32 行使用 int 型变量接受函数返回的 int & 型的值。

(2) 第 33、34 行使用 int& 型变量接受函数返回的 int & 型的值，但是不能使用引用变量接受返回类型为非引用型的函数值，int &m4＝max1(a, 10)是错误的。

(3) 第 33 行 m3 引用的返回值变量 a[t]在函数调用结束时仍然有效，所以 m3 的值是有效的。

(4) 第 34 行 m4 引用的返回值变量 s 是函数体内的变量，在函数调用结束时失效，故 m4 的值－858993460 是一个无效的值。因此，当函数返回类型是引用型时，返回变量不能是临时变量。

(5) 第 12 行，a[t]＋0 作为函数 max1()的返回值，但在 max2()中不能使用 a[t]＋0 作为返回值。当函数返回类型是引用型时，返回值一定是一个左值。

(6) 第 39 行，可通过引用变量修改返回变量的值。

(7) 第 40 行，当函数返回类型是引用型时，函数调用形式可以作为左值接受右值对象的值。

(8) 由于第 39 行将返回变量 a[9]的值变成 20，第 40 行将 a[9]变成－80，最后数组 a[]中元素的和为－35。

3.4.3　常引用

如果在定义引用变量时用 const 修饰，被定义的引用就是**常引用**。定义常引用的格式如下：

```
const 数据类型 & 引用变量 = 变量名;
```

定义一个常引用后，就不能通过常引用更改引用的变量的值了。

例如：

```
int i (100);
const int & r = i;
```

若试图使用 r=200 更改 i 的值是错误的，但通过 i 本身可以改变 i 的值：

```
i = 200;
```

此时 r 的值变成 200。

常引用类型常用作函数的形参类型，在把形参定义为常引用类型时，在函数体内就不能通过形参改变实参的值了，保证了函数调用时实参是“安全”的，这样的形参称为只读形参。

```
void fun(const int& x,int& y) {
      x = y;              //错误,x 不可修改
      y = x;
}
```

☆**注意**：

形参为常引用类型时，实参可以是常量、变量表达式；如果形参为非常引用类型，实参必须为左值。对于 void fun(const int& x, int& y)，调用 fun(100,200)是错误的，调用 fun(100,a)是正确的(a 为变量)。

3.5 结构与联合

数组是由类型相同的数据元素构成的，然而，在程序中往往需要处理一些由不同类型数据元素所构成的数据。例如，一个学生的基本情况由学号、姓名、性别、出生日期、成绩等数据元素构成，这些数据元素具有不同的数据类型，如果使用独立的不同数据类型的变量来描述这些信息，那么变量之间的关系不能体现出来。C++提供了描述不同数据类型的组合体的方法，即结构(struct)与联合(uion)。

C++语言是从 C 语言发展而来的，为了与 C 语言兼容，C++中也保留了 C 语言中的结构与联合。C++语言并不是纯粹的面向对象的程序设计语言，它追求的是程序设计的高效性，因此，为了保证程序设计的效率和方便，C++保留了一些面向过程的程序设计特征。在 C++中“类”是封闭的，类的主要成员都是私有属性(private)，而“结构”是开放性的，其成员属性都是公有属性(public)，因此，在需要开放性的地方，适合采用结构可以提高程序设计的效率。

在 C++中，依然保留了联合。在某些 C++程序设计中，联合的作用很重要，没有其他构造数据类型可以替代它。例如，在一些非标准扩展的程序设计中，使用联合是必要的；对于很多内存受限的系统，如移动设备、嵌入式系统、游戏程序等对内存的使用和调度要求较高的程序环境中，需要使用同一块内存代表不同的含义，最适合的数据结构就是联合。

3.5.1 结构

1. 结构类型的定义

结构类型将不同数据类型组合成一个整体类型，定义结构类型的格式如下：

```
struct 结构类型名
{
    数据类型 1   成员名 1;
    数据类型 2   成员名 2;
    …
    数据类型 n   成员名 n;
};
```

其中：

- struct 是关键字，表示定义的是一个结构类型；
- 结构类型名必须是一个合法的标识符，结构类型名省略时表示定义的是一个无名的结构体；
- 结构类型的成员数量不限，各成员构成成员表；数据类型可以是基本数据类型，也可以是构造数据类型；
- 结构类型定义的结束符，不能省略。

例如，下面定义了一个学生信息的结构类型：

```
enum gender {man,ferman};
struct student
{
    long no,birthday;              //学号、生日
    char name[22];                 //姓名
    gender sex;                    //性别
    float score;                   //成绩
};
```

☆**注意**：

(1) 结构类型由多个成员类型组合而成，所以结构类型变量所占内存的大小理论上应该为各个成员所占内存大小之和。为了提高对内存的存取速度，C++分配各个结构成员的内存空间以字为单位，以保证其地址在字的整数倍处，所以结构成员内存空间存在间隙。

```
student 的理论值 = sizeof(long) * 2 + sizeof(char) * 22 + sizeof(gender) + sizeof
(float)
                 = 8 + 22 + 4 + 4 = 38;
student 实际值 = sizeof(student) = 8 + 24 + 4 + 4 = 40;
```

因此，在程序中要避免用结构成员大小计算结构变量所占的内存。

(2) 定义了一个结构类型，但并没有定义变量，结构类型中的成员名既不能当作类型名，也不能当作变量使用。

例如：

```
score = 95;                        //错误，成员名不能当作变量
cout << sizeof(name);              //成员名不能当作类型名
```

2. 结构变量的定义与使用

结构变量的定义与其他类型变量的定义格式相同，既可以和结构类型一起定义，也可以在使用前临时定义。

结构变量的初始化方法与数组的初始化方法相似，初始化格式如下：

```
结构类型名 结构变量名 = {成员名 1 的值,成员名 2 的值,…,成员名 n 的值};
```

例如：

```
student s001 = {200507038,19850708,"ZhangShan",man,95.5};
```

☆**注意**：

（1）初始化结构变量时成员值表中的顺序要与定义时的顺序相同。

```
student s001 = {200507038,19850708,man,"ZhangShan",95.5};//成员值顺序错误
```

（2）只有在定义结构变量时才能对结构变量进行整体初始化，在定义了结构变量后每个成员单只能单独初始化。

```
student s001;
s001 = {200507038,19850708 ,"ZhangShan" ,man,95.5};   //错误,在定义结构变量后进行整体初始化
```

（3）在定义结构类型与结构变量时不能对成员进行初始化。

```
struct student
{
    long no = 20030108,birthday;    //错误
    char name[20];
    gender sex;
    float score = 95;               //错误
} s001;
```

（4）定义无名结构类型时一定要同时定义变量。

在定义结构变量后，相当于定义了多个成员变量，通过结构变量名存取各个成员变量的格式如下：

```
结构变量名.成员名;
```

例如：

```
student s002;
strcpy(s002.name,"LiGuohua");
```

通过 s002.name＝"LiGuohua"的方法给 s002.name 这个字符数组整体赋初值同样是错误的，可以通过逐个给结构变量赋值的方法来初始化结构变量。

通过一个指向结构变量的指针存取结构成员的，格式如下：

```
指向结构变量的指针名 -> 结构变量名的成员名;
```

（5）结构变量之间可以相互赋值，结构变量之间的赋值等价于各个成员之间逐一相互赋值。如果成员是数组，则将数组元素一一赋值。

例如：

```
s001 = s002;
```

相当于进行以下赋值：

```
s001.no = s002.no;
s001.birthday = s002.birthday;
...
s001.score = s002.score;
```

对于 s001.name 相当于进行了：

```
memcpy(s001.name,s002.name,sizeof(s002.name));
```

但对于结构变量中的指针，C++只是将地址进行了赋值。结构变量间的赋值称为**结构式拷贝**，属于浅拷贝（后面章节将详细介绍）。

【例 3-13】 使用结构数组存储学生信息，按学生成绩从高到低排序。

分析：按成绩排序时需要进行数组元素交换，利用结构式拷贝对结构体中存储学生姓名的字符数组进行复制，从而将成绩与姓名一并交换。

```
/*******************************************
*   程序名：p3_13.cpp                      *
*   功　能：学生成绩的排序，结构数组应用    *
*******************************************/
#include <iostream>
using namespace std;
struct student
{
  char name[20];
  float score;
};
int input(student s[],int n)                    //返回实际输入人数
{
    for(int i=0;i<n;i++)
    {
        cin>>s[i].name>>s[i].score;
        if (s[i].score<0) break;
    }
    return i;
}
void output(student s[],int n)
{   for(int i=0;i<n;i++)
        cout<<s[i].name<<"\t"<<s[i].score<<endl;
}
void sort(student a[],int n)
{
  for(int i=0;i<n-1;i++)                        //排序
      for(int j=i+1;j<n;j++)
          if(a[i].score<a[j].score)
          {
              student t;
              t=a[i];                           //交换数组元素
              a[i]=a[j];
              a[j]=t;
```

```
35            }
36 }
37 int main()
38 {
39      const int MaxNum = 100;                    //学生人数
40      int num;                                   //实际人数
41      student s[MaxNum];
42      num = input(s,MaxNum);
43      sort(s,num);
44      output(s,num);
45      return 0;
46 }
```

运行结果：

```
GongJing        80 ↙
LiuNa           90 ↙
zhouZijun       86 ↙
LianXiaolei     92 ↙
ChenHailing     98 ↙
CuiPeng         84 ↙
exit            -1 ↙
ChenHailing     98
LianXiaolei     92
LiuNa           90
zhouZijun       86
CuiPeng         84
GongJing        80
```

程序解释：

程序第 33～35 行对数组元素进行交换，$t=a[i]$等价于：

```
t. score = a[i].score;
memcpy(t.name,a[i].name,sizeof(a[i].name));
```

3. 结构体的应用

结构数组与链表体现了结构的主要应用。结构数组是以结构类型数据为元素的数组。例如，可以使用结构数组存储若干个学生的信息，但数组各个元素在内存中是连续存放的，需要系统提供连续的内存块，另一方面，当从数组中插入或删除数据元素时需要移动大量的元素。链表是通过指针将一个个数据元素链接起来，就像是一条“链子”，形象地称这种结构为链表。与数组相反，链表不需要整块内存，只需要多块零碎的内存，这样系统容易满足；当在链表中插入或删除一个元素时，不需要移动其他数据元素。由于链表具有这样的优点，因而有广泛的用途。

构成链表的数据元素称为结点，每个结点只有一个指针指向下一个结点的链表称为单向链表；若一个结点有两个指针，一个指向前一个结点（前驱结点），另一个指向后一个结点（后继结点），这样的链表称为双向链表。若链表的最后一个结点指向第一个结点，称这样的链表为循环链表或环形链表。

定义一个单向链表结点的格式如下：

```
struct 结点的类型名
  {
    数据成员定义;
    struct 结点类型名 * 指针名;
  }
```

例如，下面定义了一个学生信息的链表结点结构：

```
struct student
{
  char name[20];
  float score;
  struct student * next;
}
```

对链表的操作有建立、检索、插入、删除。

以学生信息链表为例，图 3-7 演示链表的建立过程。

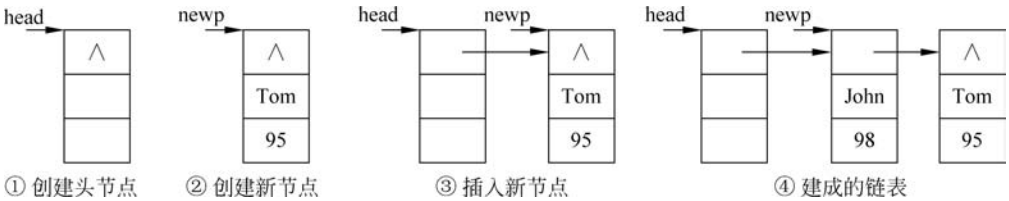

图 3-7　链表的建立

（1）创建头结点。

头结点也称哨兵结点，是链表的第一个结点，不存储数据，是为了方便链表操作而设的一个结点。若头结点的指针域值为空，表示链表为空。

创建头结点的语句如下：

```
newp = new student;
newp -> next = NULL
head = newp;
```

（2）创建新结点。

```
newp = new student;
```

（3）插入新结点。

如果要插入一个结点，首先要找到插入位置。假定要建立一个按成绩从高到低排列的链表，查找插入位置需要从头结点开始遍历链表，直到找到这样的一个结点，其后继结点的成绩比要插入结点的成绩低，插入点就在此结点之后。假定 p 指向当前结点，插入新结点 newp 的步骤如图 3-8 所示。

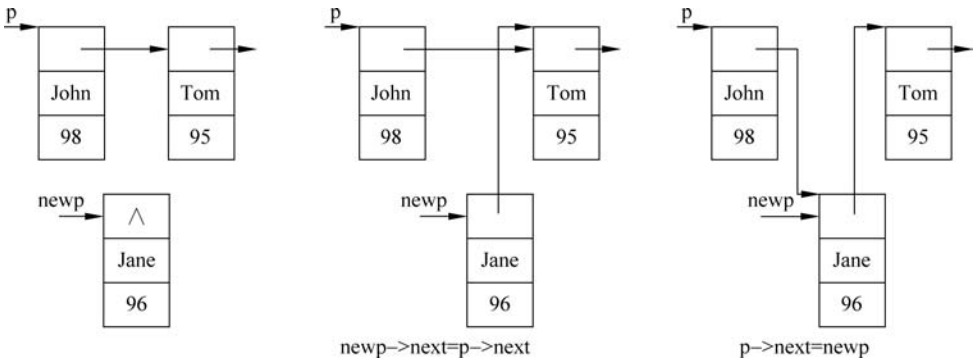

图 3-8　在链表中插入结点

（4）删除结点。

如果要删除一个结点，一定要有一个指向此结点前驱结点的指针，除非是头结点。删除结点的操作步骤如图 3-9 所示。

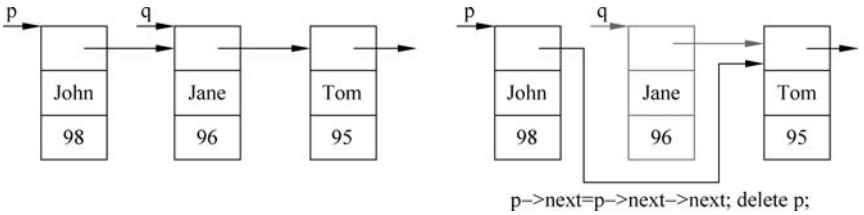

图 3-9　删除链表中的结点

【例 3-14】 使用单向链表按学生成绩从高到低的顺序存储学生信息，从中删除不及格学生的信息。

```
/*************************************************
*   程序名：p3_14.cpp                             *
*   功   能：单向链表的排序、查找、插入、删除     *
*************************************************/
#include <iostream>
using namespace std;
struct student
{
  char name[20];
  float score;
  struct student *next;
};
typedef student NODE;
NODE *Search(NODE *head,int key) {                    //查找关键字小于 key 的结点的前驱
    NODE *p;
    p=head;
    while(p->next!=NULL) {
        if(p->next->score<key)
            return p;
        p=p->next;
    }
    return p;
}
void InsertNode(NODE *p,NODE *newp) {                 //在 p 之后插入结点 newp
    newp->next=p->next;
    p->next=newp;
}
void DelNode(NODE *p) {                               //删除 p 结点的一个后继结点
    NODE *q;
    if(p->next!=NULL) {
        q=p->next;
        p->next=q->next;
        delete q;
```

```
34      }
35 }
36 void DelList(NODE * head) {                          //销毁整个链表
37      NODE * p;
38      p = head;
39      while(head -> next!= NULL) {
40            head = head -> next;
41            delete p;
42            p = head;
43          }
44      delete head;
45      }
46 void DispList(NODE * head) {                          //显示链表各元素
47      NODE * p;
48      p = head;
49      while(p -> next!= NULL) {
50          cout << p -> next -> name <<"\t"<< p -> next -> score << endl;
51          p = p -> next;
52      }
53 }
54 int main()
55 {
56   NODE * newp, * head, * p;
57   char name[20];
58   float score,low = 60;
59   if((newp = new NODE) == NULL) {
60     cout <<"new NODE fail!"<< endl;
61     exit(0);
62 }
63 head = newp;
64 head -> next = NULL;
65 cout <<"Input name and score( - 1 to exit):"<< endl;
66 cin >> name >> score;
67 while(score > 0)
68 {
69      if((newp = new NODE) == NULL)
70      {
71          cout <<"new NODE fail!"<< endl;
72          exit(0);
73      }
74      strcpy(newp -> name,name);
75      newp -> score = score;
76      newp -> next = NULL;
77      p = Search(head,score);
78      InsertNode(p,newp);
79      cin >> name >> score;
80 }
81 cout <<"Before delete:"<< endl;
82 DispList(head);
83 for(p = Search(head,low);p -> next!= NULL;p = Search(head,low))
```

```
84      DelNode(p);
85 cout <<"After delete:"<< endl;
86 DispList(head);
87 DelList(head);
88 return 0;
89 }
```

运行结果：

Input name and score (-1 to exit):		Before delete:		After delete:	
GongJing	80 ↙	ChenHailing	98	ChenHailing	98
LiuNa	90 ↙	LianXiaolei	92	LianXiaolei	92
Zhouzijun	86 ↙	LiuNa	90	LiuNa	90
LianXiaolei	92 ↙	Zhouzijun	86	Zhouzijun	86
ChenHailing	98 ↙	CuiPeng	84	CuiPeng	84
CuiPeng	84 ↙	GongJing	80	GongJing	80
exit	-1 ↙				

3.5.2 联合

1. 联合类型的定义

联合(union)类型也称为共用体类型，它是一种与结构类型类似的数据类型，提供了一种可以将几种不同类型数据存放于同一段内存，对其中各个成员可以按名存取的机制。定义联合类型的语法形式如下：

```
union 联合类型名
{
    数据类型 1  成员名 1;
    数据类型 2  成员名 2;
    …
    数据类型 n  成员名 n;
};
```

例如，下面定义了联合类型：

```
union UData
{
        char      Ch;
        short     Sint;
        long      Lint;
        unsigned  Uint;
        float     f;
        double    d;
        char      str[10]
};
```

与结构类型不同，联合类型虽然由多个成员类型组合而成，但联合类型变量的各个成员拥

有共同的内存空间，联合类型变量所占内存的大小应为各个成员所占内存大小的最大值。如果其中有构造数据类型，其大小为其中最长的基本数据类型的整数倍。

```
sizeof(UData)的理论值 = sizeof(str) = 10;
sizeof(UData)的实际值 = sizeof(double) * 2 = 16;
```

2. 联合变量的定义与使用

联合变量的定义与其他类型变量的定义格式相同，既可以和联合类型一起定义，也可以在使用前临时定义。

联合变量只能初始化第一个成员，初始化格式如下：

```
联合类型名 联合变量名 = {成员名 1 的值};
```

例如：

```
UData u = {65};
UData u2 = {"123456789"};        //错误，除非将 str[10]当作第 1 个成员
```

在定义无名联合类型时，其中的成员类型可以当作变量使用，例如：

```
union {
  char c;
  int i;
};
c = 'a';
i = 65;
cout << c;                       //显示 A
```

由于变量 c 与 i 有相同的内存单元，所以最终 c 的值为 65；由于无名联合既具有其成员共有内存单元的性质，又具有不同联合名直接存取成员的性质，所以无名联合常用在结构体中。

在定义联合变量后，相当于定义了多个成员变量，通过联合变量名存取各个成员变量的格式如下：

```
联合变量名.成员名;
```

通过一个指向联合变量的指针存取结构成员的格式如下：

```
指向联合变量的指针名 -> 联合变量的成员名;
```

3. 联合变量在内存中的排列

在定义联合变量后，联合变量在内存中获得了共同的内存，由于各个成员的数据类型不同，因此长度也不同。各成员在内存中的排列均从低地址开始，遵循“低地址低字节，高地址高字节”的原则。

例如：

```
UData u;
strcpy(u.str,"123456789");       //给成员 u.str 赋初值
```

其内存图如图 3-10 所示。

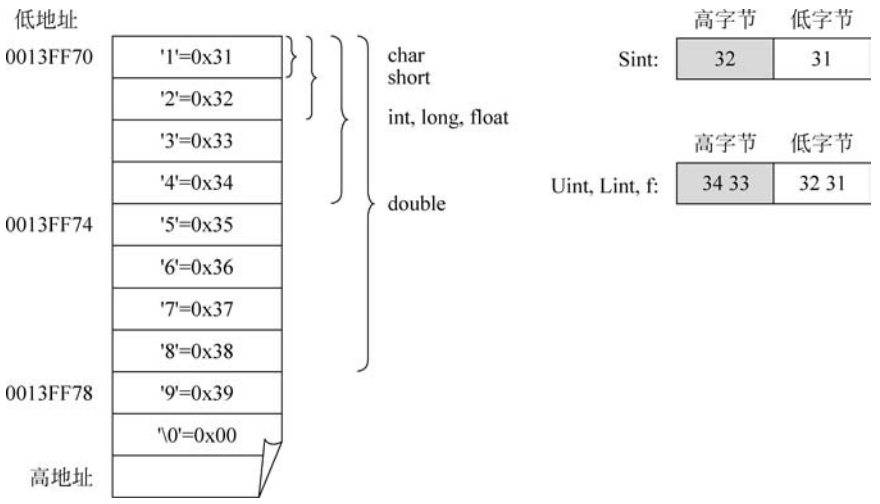

图 3-10　联合体在内存中的排列

【例 3-15】 显示联合体中各成员的内存排列。

```
/*******************************************************
*   程序名：p3_15.cpp                                  *
*   功　能：显示联合体中各类型成员的内存排列          *
*******************************************************/
#include <iostream>
using namespace std;
int main() {
    union UData
    {
        char       Ch;
        short      Sint;
        long       Lint;
        unsigned   Uint;
        float      f;
        double     d;
        char str[10];
    };
    UData u;
    strcpy(u.str,"123456789");
    cout <<"char:"<<'\t'<< u.Ch << endl;
    cout <<"short:"<<'\t'<< hex << u.Sint << endl;
    cout <<"long:"<<'\t'<< u.Lint << endl;
    cout <<"unsigned:"<<'\t'<< u.Uint << endl;
    cout <<"float:"<<'\t'<< u.f << endl;
    cout <<"double:"<<'\t'<< u.d << endl;
    cout <<"string:"<<'\t'<< u.str << endl;
    return 0;
}
```

运行结果：

```
char:       1
short:      3231
long:       34333231
unsigned:   34333231
float:      1.66889e-007
double:     6.82132e-038
string:     123456789
```

思考：

请对照联合体变量的各成员内存分布图，思考程序 p3_15 的运行结果，并给出理由。

本 章 小 结

(1) 枚举类型实际上是一组整型符号常量，其中的每个常量都可以进行各种运算，在对一个枚举变量赋值时一定要注意类型一致。

(2) 数组是一组同类型变量，可通过数组名加下标存取其中的单个变量。各个数组元素在内存中顺序排列，数组名表示的是数组的起始地址，可以使用指针运算符用指针方式存取数组元素。

(3) 一个多维数组是以低维数组为元素的数组，多维数组在内存中的排列与一维数组相同，即从低地址到高地址顺序排列。数组名表示的是数组的地址。

(4) 数组名表示的地址不能是左值，但可以当作函数的形式参数，接受实参传送的地址。

(5) 以 0 作为结尾符的字符数组为字符串，数组大小与字符串长度的关系为 sizeof(s) = strlen(s)+1。

(6) 地址变量称为指针，所有地址变量的长度都是 4 个字节。

(7) 用地址变量存储数组的地址时(指针指向了数组)，可通过指针名以指针方式存取数组元素，也可将指针名当作数组名以数组方式存取元素。但指针比数组多了一个存储地址的内存空间，因此指针名可以作为左值。

(8) 指针指向的动态申请的数组称为动态数组，动态数组所占的内存空间需要在程序中进行释放。

(9) 指针可作为函数的形参，接受实参传送的地址。

(10) 引用是一个已存在变量的别名，它与被引用的对象共有内存单元，因此在定义一个引用型变量时一定要以一个已存在的变量作为初值。当引用作为函数的形参时，可以不赋初值，在函数体内引用变量代替实参进行运算，对形参的改变反映到实参上。

(11) 引用型函数返回的对象被一个引用型变量接受，引用型变量成为返回对象的别名；若被一个非引用变量接受，则将返回变量的值赋给接受变量。引用型函数还可以是一个左值，接受右值对象的值。

(12) 常指针、常引用类型通常用作函数的形参，以防止在函数体内通过形参修改实参指向的值，以保护实参。

(13) 结构类型是各种已存在与已定义类型的组合体，同类型结构变量之间赋值等同于每一个成员之间的赋值，其中数组成员的赋值等同于数组的复制。

（14）联合类型是各种已存在与已定义类型的共同体，联合变量各成员拥有共同的内存空间。

习 题 3

1. 选择题

（1）执行以下语句后的输出结果是（　　）。

```
enum weekday {sun,mon,tue,wed = 4,thu,fri,sat};
weekday workday = mon;
cout << workday + wed << endl;
```

A. 6　　B. 5　　C. thu　　D. 编译错

（2）在 C++中引用数组元素时，其数组下标的数据类型允许是（　　）。

A. 整型常量　　B. 整型表达式

C. 非浮点型表达式　　D. 任何类型的表达式

（3）设有数组定义“char array [] = "China";”，则数组 array 所占的空间为（　　）。

A. 4 个字节　　B. 5 个字节　　C. 6 个字节　　D. 7 个字节

（4）若有说明“int a[][3] = {1,2,3,4,5,6,7};”，则 a 数组高维的大小是（　　）。

A. 2　　B. 3　　C. 4　　D. 无确定值

（5）以下定义语句不正确的是（　　）。

A. double x[5] = {2.0,4.0,6.0,8.0,10.0};

B. int y[5] = {0,1,3,5,7,9};

C. char c1[] = {'1','2','3','4','5'};

D. char c2[] = {1,2,3};

（6）若二维数组 a 有 m 列，则在 a[i][j]前的元素个数为（　　）。

A. j * m + i　　B. i * m + j　　C. i * m + j −1　　D. i * m + j + 1

（7）以下能对二维数组 a 正确初始化的语句是（　　）。

A. int a[2][] = {{1,0,1},{5,2,3}};

B. int a[][3] = {{1,2,3},{4,5,6}};

C. int a[2][4] = {{1,2,3},{4,5},{6}};

D. int a[][3] = {{1,0,1},{ },{1,1}};

（8）以下不能对二维数组 a 正确初始化的语句是（　　）。

A. int a[2][3] = {0};

B. int a[][3] = {{1,2},{0}};

C. int a[2][3] = {{1,2},{3,4},{5,6}};

D. int a[][3] = {1,2,3,4,5,6};

（9）定义以下变量和数组：

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

则下面语句的输出结果是（　　）。

```
for(k = 0;k < 3;k++)
   cout << a[k][2 - k]<<"\t";
```

A. 3 5 7　　B. 3 6 9　　C. 1 5 9　　D. 1 4 7

(10) 下面定义不正确的是(　　)。

A. char a[10] = "china";　　B. char a[10], * p = a; p="china";

C. char * a=0;　　D. int * p=10;

(11) 以下不能正确对字符串赋初值的语句是(　　)。

A. char str[5] = "good!";　　B. char str[] = "good!";

C. char str[8] = "good!";　　D. char str[5] = {'g','o','o','d'};

(12) 给出以下定义,则正确的叙述为(　　)。

```
char x[] = "abcdefg";
char y[] = {'a','b','c','d','e','f','g'};
```

A. 数组 x 和数组 y 等价

B. 数组 x 和数组 y 的长度相同

C. 数组 x 的长度大于数组 y 的长度

D. 数组 x 的长度小于数组 y 的长度

(13) 以下程序的输出结果是(　　)。

```
int main()
  {
    char st[20] = "hello\0\t\\";
    cout << strlen(st)<<"\t"<< sizeof(st);
    return 0;
  }
```

A. 9 9　　B. 5 20　　C. 13 20　　D. 20 20

(14) 下列程序的输出结构是(　　)。

```
#include <iostream>
using namespace std;
int main()
{
    char a[] = "Hello,World";
    char * ptr = a;
    while ( * ptr)
    {
        if( * ptr >= 'a'&& * ptr <= 'z')
            cout << char( * ptr + 'A' - 'a');
        else cout << * ptr;
        ptr++;
    }
    return 0;
}
```

A. HELLO,WORLD　　B. Hello,World

C. Hello,world　　D. hello,world

（15）要禁止修改指针 p 本身，又要禁止修改 p 所指向的数据，这样的指针应定义为（　　）。

A. const char * p="ABCD";

B. char const * p="ABCD";

C. char * const p="ABCD";

D. const char * const p="ABCD";

（16）有以下程序段：

```
int i = 0, j = 1;
int &r = i ;          //①
r = j;                //②
int * p = &i ;        //③
* p = &r ;            //④
```

会产生编译错误的语句是（　　）。

A. ④　　B. ③　　C. ②　　D. ①

（17）以下程序的输出结果是（　　）。

```
int main()
{
  char * str = "12345";
  cout << strlen(str)<<"\t"<< sizeof(str);
  return 0;
}
```

A. 6　5　　B. 5　6　　C. 5　4　　D. 5　5

（18）以下程序的输出结果是（　　）。

```
int main()
{
  char w[][10] = {"ABCD","EFGH","IJKL","MNOP"},k;
  for(k = 1;k < 3;k++) cout << w[k]<< endl;
}
```

A. ABCD　　B. ABCD　　C. EFG　　D. EFGH
FGH　　EFG　　JK　　IJKL
KL　　IJ　　O
M

（19）已知“char str1[8],str2[8]= {"good"}; ”，则在程序中不能将字符数组 str2 赋给 str1 的语句是（　　）。

A. str1 = str2;　　B. strcpy(str1,str2);

C. strncpy(str1,str2,6)　　D. memcpy(str1,str2,5);

（20）变量的指针是指该变量的（　　）。

A. 值　　B. 地址　　C. 名　　D. 一个标志

（21）对于变量 p，sizeof(p)的值不为 4 的是（　　）。

A. short int p;　　B. char *** p;

C. double * p　　D. char * p="12345";

（22）下面能正确进行字符串赋值操作的是（　　）。

A. char s[5] = {"ABCDE"};　　B. char s[5] = {'A','B','C','D','E'};

C. char *s; s = "ABCDE";　　D. char *s; cin >> s;

(23) 对于指向同一块连续内存的两个指针变量不能进行的运算是(　　)。

A. <　　B. =　　C. +　　D. -

(24) 若有语句"int *point,a = 4;"和"point = &a;",下面均代表地址的一组选项是(　　)。

A. a,point,*&a　　B. &*a,&a,*point

C. *&point,*point,&a　　D. &a,&*point,point

(25) 已有定义"int k = 2; int *ptr1,*ptr2;",且 ptr1 和 ptr2 均指向变量 k,下面不能正确执行的赋值语句是(　　)。

A. k = *ptr1 + *ptr2;　　B. ptr2 = k;

C. ptr1 = ptr2;　　D. k = *ptr1 *(*ptr2);

(26) 若有说明"int i,j = 2,*p = &i;",则能完成 i = j 赋值功能的语句是(　　)。

A. i = *p;　　B. *p = *&j;

C. i = &j;　　D. i = **p;

(27) 若有定义"int a[8];",则以下表达式中不能代表数组元素 a[1]的地址的是(　　)。

A. &a[0] + 1　　B. &a[1]　　C. &a[0]++　　D. a + 1

(28) 若有以下语句且 0≤k<6,则正确表示数组元素地址的表达式是(　　)。

```
int x[] = {1,3,5,7,9,11},*ptr = x,k;
```

A. x++　　B. &ptr　　C. &ptr[k]　　D. &(x+1)

(29) 下面程序段的运行结果是(　　)。

```
char *p = "abcdefgh";
p+=3;
cout<<strlen(strcpy(p,"ABCD"));
```

A. 8　　B. 12　　C. 4　　D. 出错

(30) 设有语句"int array[3][4];",则在下面几种引用下标为 i 和 j 的数组元素的方法中,不正确的引用方式是(　　)。

A. array[i][j]　　B. *(*(array + i)+ j)

C. *(array[i] + j)　　D. *(array + i *4 + j)

(31) 若有以下定义和语句,则对 s 数组元素的正确引用形式是(　　)。

```
int s[4][5],(*ps)[5];
ps = s;
```

A. ps+1　　B. *(ps+3)

C. ps[0][2]　　D. *(ps+1)+3

(32) 在说明语句"int *f();"中,标识符 f 代表的是(　　)。

A. 一个用于指向整型数据的指针变量

B. 一个用于指向一维数组的行指针

C. 一个用于指向函数的指针变量

D. 一个返回值为指针型的函数名

(33) 函数原型为 fun(int(* p)[3],int)，调用形式为 fun(a,2)，则 a 的定义应该为(　　)。

A. int ** a　　B. int(* a)[]

C. int a[][3]　　D. int a[3]

(34) 若"int i=100；"，在下列引用方法中，正确的是(　　)。

A. int &r=i;　　B. int &r=100;

C. int &r;　　D. int &r=&i;

(35) 在下列引用方法中，错误的是(　　)。

A. int i;
int &r=i;

B. int i;
int &r;r=i;

C. float f;
float &r=f;

D. char c;
char &r=c;

(36) 以下程序的执行结果为(　　)。

```
int f(int i){return ++i;}
int g(int &i){return ++i;}
int main() {
    int a(0),b(0);
    a += f(g(a));
    b += f(f(b));
    cout << a <<"\t"<< b;
    return 0;
}
```

A. 3　2　　B. 2　3　　C. 3　3　　D. 2　2

(37) 以下程序的执行结果为(　　)。

```
int& max(int& x,int& y){ return(x > y?x:y);}
int main() {
    int m(3),n(4);
    max(m,n) -- ;
    cout << m <<"\t"<< n;
    return 0;
}
```

A. 3　2　　B. 2　3　　C. 3　4　　D. 3　3

(38) 若定义结构体：

```
struct st {
    int no;
    char name[15];
    float score;} s1;
```

则结构体变量 s1 所占的内存空间为(　　)。

A. 15

B. sizeof(int)+sizeof(char[15])+sizeof(float)

C. sizeof(s1)

D. max(sizeof(int),sizeof(char[15]),sizeof(float))

(39) 若定义联合体：

```
union { int no;
        char name[15];
```

```
    float score;} u1;
```

则联合体变量 u1 所占的内存空间为(　　)。

A. 15

B. sizeof(int)+sizeof(char[15])+sizeof(float)

C. sizeof(u1)

D. max(sizeof(int),sizeof(char[15]),sizeof(float))

(40) 当定义"const char *p="ABC";"时,下列语句正确的是(　　)。

A. char *q=p;　　B. p[0]='B';　　C. *p='\0';　　D. p=NULL;

(41) 下列语句错误的是(　　)。

A. const int a[4]={1,2,3};　　B. const int a[]={1,2,3};

C. const char a[3]={'1','2','3'};　　D. const char a[]="123";

(42) 下列语句错误的是(　　)。

A. const int buffer=256;

B. const int temp;

C. const double *point;

D. const double *p=new double(3.1);

2. 程序填空题

(1) 以下是一个评分统计程序,共有 8 个评委打分,统计时,去掉一个最高分和一个最低分,其余 6 个分数的平均分即是最后得分,最后显示得分,显示精度为一位整数、两位小数。程序如下,请将程序补充完整。

```
#include <iostream>
using namespace std;
int main()
{
  float x[8] = ____①____;
  float aver(0),max____②____,min____③____;
  for(int i=0;i<8;i++){
      cin>>x[i];
      if(x[i]>max)
      ____④____;
      if(____⑤____)
          min=x[i];
      aver += x[i];
      cout<<x[i]<<endl;
  }
  aver = ____⑥____
  cout<<aver<<endl;
  return 0;
}
```

(2) 以下程序在 M 行 N 列的二维数组中找出每一行上的最大值,显示最大值的行号、列号、值。请将程序补充完整。

```
#include <iostream>
using namespace std;
```

```
int main()
{
    ___①___;
    int x[M][N] = {1,5,6,4,2,7,4,3,8,2,3,1};
    for(___②___;i < M;i++)
    {
        int t = 0;
        for(___③___;j < N;j++)
            if(___④___)
                ___⑤___;
        cout << i + 1 <<","<< t + 1 <<" = "<< x[i][t]<< endl;
    }
  return 0;
}
```

（3）函数 expand(char * s,char * t)在将字符串 s 复制到字符串 t 时，将其中的换行符和制表符转换为可见的转义字符，即用“\n”表示换行符，用“\t”表示制表符。请填空。

```
void expand(char * s,char * t)
{
  for(int i = 0,int j = 0;s[i] != '\0';i++)
    switch(s[i])
    {
      case '\n':t[ ___①___ ] = ___②___;
                t[j++] = 'n';
                ___③___;
      case '\t':t[ ___④___ ] = ___⑤___;
                t[j++] = 't';
                break;
    default: t[ ___⑥___ ] = s[i];
               break;
    }
    t[j] = ___⑦___;
}
```

3. 程序改错题

下列程序如有错请改正，并写出改正后的运行结果。

```
#include <iostream>
using namespace std;
int &add(int x,int y)
{
    return x + y;
}
int main()
{
    int n(2),m(10);
    cout <<(add(n,m) += 10)<< endl;
    return 0;
}
```

4. 编程题

（1）输入 10 个整数，将这 10 个整数按升序排列输出，并且奇数在前、偶数在后。比如，如

果输入的10个数是10 9 8 7 6 5 4 3 2 1,则输出1 3 5 7 9 2 4 6 8 10。

(2) 编程打印以下形式的杨辉三角形。

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
```

(3) 编写一个程序,实现将用户输入的一个字符串以反向形式输出。比如,输入的字符串是ancdefg,输出为gfedcba。

(4) 编写一个程序,实现将用户输入的一个字符串中的所有字符'c'删除,并输出结果。

(5) 编写一个程序,将字符数组s2中的全部字符复制到字符数组s1中,不用strcpy函数。在复制时,'\0'也要复制过去,'\0'后面的字符不复制。

(6) 不用strcat函数编程实现字符串连接函数strcat的功能,将字符串DStr连接到字符串SStr的尾部。

(7) 编程求两个矩阵的乘积,要求两个矩阵的维数由键盘临时输入。

(8) 编写一个函数,判断输入的一串字符是否为“回文”。所谓“回文”是指顺读和倒读都一样的字符串。例如,"level"、"ABCCBA"都是回文。

(9) 编写一个函数int SubStrNum(char *str,char *substr),它的功能是统计子字符串substr在字符串str中出现的次数。

(10) 编写一个函数,返回任意大的两整数之差。(提示:大整数用字符串来表示)

(11) 编写一个程序,求解约瑟夫(Josephus)问题。约瑟夫问题描述为:n个小孩围成一圈做游戏,给定一个数m,现从第s个小孩开始,顺时针计数,每数到m,该小孩出列,然后从下一个小孩开始,当数到m时,该小孩出列,如此反复,直到最后一个小孩。

(12) 现有一个电话号码簿,号码簿中有姓名、电话号码,当输入电话号码时,查找出姓名与电话号码;当输入姓名时,同样查找出姓名与电话号码;还允许不完全输入查找,例如输入010时查找出所有以010开头的号码,输入“杨”时列出所有姓名以“杨”开头的号码。