

C++高级语言程序设计

- 王晨宇
- 北京邮电大学网络空间安全学院

第4章 指针和引用

- 指针的概念简单，使用广泛、灵活、复杂。
- 正确使用指针，能有效使用各种复杂数据结构，动态分配内存，高效使用数组和字符串，编写通用程序等；
- 使用指针不当，可导致程序运行出错或死机。
- **学习建议：理解指针本质，多用指针编程，多调试，不怕出错。**

4.0 数组

4.1 指针和指针变量

4.2 指针与数组

4.3 指针数组和指向指针的指针变量

4.4 指针与函数

4.5 new和delete运算符

4.6 引用和函数

4.7 单向链表及其应用

4.8 其他

4.0 数组

- 数组的概念及特点
 - 是同类型同性质的的一组元素顺序存放构成的数据集合。
 - 所有数据共用同一个名字，通过下标区分不同的数据
 - 处理时可通过循环控制变量控制下标的变化来批量处理数组中的数据。

4.0 数组

- 数组的定义

数据类型 数组名[整型常量表达式];

- 数组名代表数组在内存中的首地址，该地址由系统自动分配;
- 整型常量表达式代表数组的长度，此处不可使用变量说明长度;
- 数组中包含的分量（元素）用下标区分，下标范围0~长度-1;

int **workDay**[5];

数组类型

数组名

数组元素个数

4.0 数组

- 一维数组的定义

如，包含5个学生成绩数据的数组可定义为：

```
float s[5];
```

还可如下定义：

```
#define N 5  
float s[N];
```

```
const int n=5;  
float s[n];
```

符号常量做数
组长度说明

4.0 数组

- 一维数组的定义

要避免以下错误:

C99之前元素数量必须 是编译阶段确定的
常量

```
int n=5, s[n];
```

```
double d[ ];
```

```
float b[3.4]
```

长度不允许用变量

长度不允许为空

长度不允许非整型

4.0 数组

- 一维数组的存储

假设系统将地址为0012FF4C开始的一段连续内存分配给整型数组s，则数组在内存中存储如下：

:



4.0 数组

- 一维数组的访问

数组名[下标]

- 数组元素相当于同类型的普通变量，可参与该类型变量允许的一切操作。
- 对于数值型数组，程序只能操作数组元素，不能操作数组名。

4.0 数组

- 一维数组的初始化
- 在定义数组的同时对元素赋值。
 - (1) 全部元素赋初值，例如：
`int days[7]={0,1,2,3,4,5,6};`
 - (2) 部分元素赋初值，例如：
`int days[7]={0,1,2,3,4};`
 - (3) 全部数组元素赋初值时，可以不指定长度：
`int days[]={0,1,2,3,4,5,6};`
 - (4) 全局数组和静态数组的初值都自动赋为0；而局部自动数组的初值不确定。

4.0 数组

- 二维数组

问题：打印如下的杨辉三角型，数据如何存储？

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

这种有行有列的数据结构适合用二维数组来描述。

4.0 数组

- 二维数组定义

数据类型 数组名[常量表达式1][常量表达式2];

- 表达式1代表行，表达式2代表列；元素个数为行、列长度的乘积。
- 行、列下标皆从0开始；
- 二维数组“按行”存放，即一行元素存储完毕之后再存储下一行元素。

4.0 数组

- 二维数组的存储

如，假设有定义 `float a[2][3]`；且系统为其分配的首地址为 1000，则数组各元素在内存中存储如下：

a	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
	1000	1004	1008	100C	1010	1014

4.0 数组

- 二维数组的初始化

(1)按存放顺序对所有元素赋初值

```
int    a[2][3]={1,2,3,4,5,6};    或:
```

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

全部元素初始化可省略第一维长度

(2)按行给所有元素赋初值，每行数据组织在一对花括号内。

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

4.0 数组

- 二维数组的初始化

(3) 按行给部分元素赋初值, 未被赋值的元素自动为0

。 `int b[3][4]={ {1,2},{0,3,4},{0,0,5}}`

对应的数组**b**为:

$$b = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 0 \end{pmatrix}$$

(4) 按行赋初值也可省略第一维的长度。

`int c[][3]={ {1},{ },{2}};`

对应的数组**c**为:

$$c = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 0 \end{pmatrix}$$

4.0 数组

- 二维数组的访问

- 对于数值型数组，程序中只能使用其元素。

数组名[行下标][列下标]

- 输入、输出、批量处理时需使用双重循环，外循环控制行的变化，内循环控制列的变化。如：

```
for(i=0;i<2;i++)  
    for(j=0;j<3;j++)  
        cin>>a[i][j];
```

4.0 数组

- 二维数组的应用

【例】求一个 3×3 方阵的最大元素及下标。

```
max=a[0][0];
```

```
imax=jmax
```

```
=0;
```

记录最大元素行列下标

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

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

```
if
```

```
( a[i][j]>max)
```

```
{
```

```
max=a[i]
```

```
  [j] ;
```

与后面逐元素比较、替换最大值，并记录其行列下标

4.0 数组

- 字符数组

问题：如何将任意长度的英文句子中的所有小写字母转换成大写字母？

分析：

- 该问题处理的是**文本**型数据，可通过字符数组描述；
- 该问题的特点是每次处理的文本**长度**可能都在**变化**，该如何输入？
- 对每次要处理的句子事先数出长度再像一般数组一样处理显然不合适，实际操作时是将整个句子以字符串形式处理的；
- 字符数组处理字符串有何不同于一般数组的策略？

4.0 数组

- 字符串的概念

系统自动加入的串结束符

- 双引号引起的一串字符。如, “**ab123**” (常量)

a	b	1	2	3	\0
---	---	---	---	---	----

- 若有定义char s[20]; 通过特殊的初始化、输入、处理、输出可用来处理可变的字符串。

特点:

系统自动在有效字符末尾添加 '\0' (字符串结束符)

。

4.0 数组

- 字符数组初始化

- (1) 用字符串为字符数组初始化

```
char s[10] = {"I am fine"};  
char s[10] = " I am fine";
```

s是字符串， 系统自动在末尾添加' \0 '

- (2) 字符串数组初始化

```
char a[3][8]={"COBOL","FORTRAN", "PASCAL"};
```

4.0 数组

- 字符串输入和输出

假设有定义 `char s[100];`

- 字符串的整体输入

- ① `cin>>s` ; 不能提取s中空白符后面的内容

- ② `gets(s);`

- 字符串的整体输出

- ① `cout<<s;`

- ② `puts(s);`

函数`gets`和`puts`的原型说明在`stdio.h`中

4.0 数组

- 字符串输入和输出

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
```

```
{
    char
```

```
    s[100];
```

```
    cin>>s;
```

```
    out<<s<<endl;
```

```
    system("pause");
```

```
    return 0;
```

直接使用数组名

不可提取
空格符



```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
```

```
{
    char
```

```
    s[100];
```

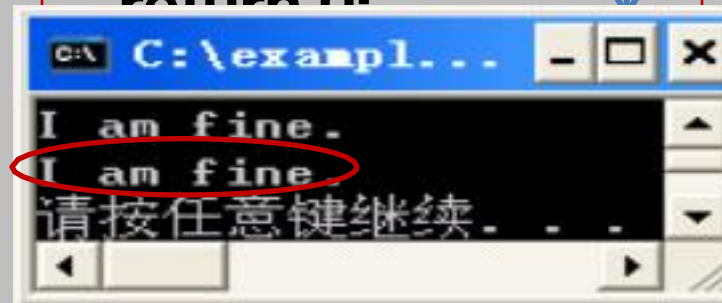
```
    gets(s);
```

```
    puts(s);
```

```
    system("pause");
```

```
    return 0;
```

可提
取空
格符



4.0 数组

- 字符串处理原则

整体输入、整体输出；

- 不用数组长度控制循环，而通过当前字符是否达到'\0'判断循环结束与否；
- 构造字符串时要保证结尾一定要有' \0 '；

4.0 数组

- **string**类

C++在处理字符串方面提供有两种方法:

- 按C风格的字符数组处理;
- 按**string**类型的对象处理;

4.0 数组

- string类

```
// strtype1.cpp -- using the C++ string class
#include <iostream>
#include <string>           // make string class available
int main()
{
    using namespace std;
    char charr1[20];        // create an empty array
    char charr2[20] = "jaguar"; // create an initialized array
    string str1;            // create an empty string object
    string str2 = "panther"; // create an initialized string

    cout << "Enter a kind of feline: ";
    cin >> charr1;
    cout << "Enter another kind of feline: ";
    cin >> str1;           // use cin for input
    cout << "Here are some felines:\n";
    cout << charr1 << " " << charr2 << " "
        << str1 << " " << str2 // use cout for output
        << endl;
    cout << "The third letter in " << charr2 << " is "
        << charr2[2] << endl;
    cout << "The third letter in " << str2 << " is "
        << str2[2] << endl;    // use array notation

    return 0;
}
```

包含头文件

4.0 数组

- **string**对象的初始化

```
char first_date[] = {"Le Chapon Dodu"};  
char second_date[] {"The Elegant Plate"};  
string third_date = {"The Bread Bowl"};  
string fourth_date {"Hank's Fine Eats"};
```

4.0 数组

- 赋值、拼接和附加

使用string类时，某些操作比使用数组时更简单。例如，不能将一个数组赋给另一个数组，但可以将一个string对象赋给另一个string对象：

```
char charr1[20];           // create an empty array
char charr2[20] = "jaguar"; // create an initialized array
string str1;               // create an empty string object
string str2 = "panther";   // create an initialized string
charr1 = charr2;           // INVALID, no array assignment
str1 = str2;               // VALID, object assignment ok
```

4.0 数组

- 赋值、拼接和附加

可以使用运算符+将两个string对象合并起来，还可以使用运算符+=将字符串附加到string对象的末尾：

```
string str3;  
str3 = str1 + str2;           // assign str3 the joined strings  
str1 += str2;                 // add str2 to the end of str1
```

4.0 数组

- 其他操作

确定字符串中字符数的方法:

```
int len1 = str1.size();    // obtain length of str1
int len2 = strlen(charr1); // obtain length of charr1
```

4.0 数组

- string类I/O

可以使用cin和运算符>>来将输入存储到string对象中。使用cout和运算符<<来显示string对象，其句法与处理char型数组相同。但每次读取一行而不是一个单词时，使用的句法不同。

```
// strtype4.cpp -- line input
#include <iostream>
#include <string>           // make string class available
#include <cstring>          // C-style string library
int main()
{
    using namespace std;
    char charr[20];
    string str;

    cout << "Length of string in charr before input: "
          << strlen(charr) << endl;
    cout << "Length of string in str before input: "
          << str.size() << endl;
    cout << "Enter a line of text:\n";
    cin.getline(charr, 20);    // indicate maximum length
    cout << "You entered: " << charr << endl;
    cout << "Enter another line of text:\n";
    getline(cin, str);        // cin now an argument; no length specifier
    cout << "You entered: " << str << endl;
    cout << "Length of string in charr after input: "
          << strlen(charr) << endl;
    cout << "Length of string in str after input: "
          << str.size() << endl;

    return 0;
}
```

4.0 数组

- string类I/O

cin.getline(charr,20);

函数getline()是istream类的一个方法（cin是一个istream对象）。第一个参数是目标数组；第二个参数是数组长度，getline()使用它来避免超越数组的边界。

getline (cin, str) ;

这里没有使用句点表示法，表明这个getline()不是类方法。它是定义在头文件<string>中的一个普通函数。它将cin作为参数，指出到哪里去查找输入，而无需指出字符串长度。

cin>>str

之所以可行，是使用了string类的一个友元函数。对>>进行了重载。

4.1 指针和指针变量

- 内存单元：可直接访问的计算机内存的最小单位。多数计算机以一个字节为一个最小的内存单元。
- **内存单元的地址**：为每一内存单元所指定的一个唯一编号，以区分不同内存单元。
- 存储单元：一个变量所分得的连续内存单元的总和。
如一个int型变量占用的存储单元为4字节，一个char型变量占用的存储单元为1字节，一个double型变量占用的存储单元为8字节。
- 当说明一个变量时，系统为该变量分配一个连续的内存单元，即一个存储单元。

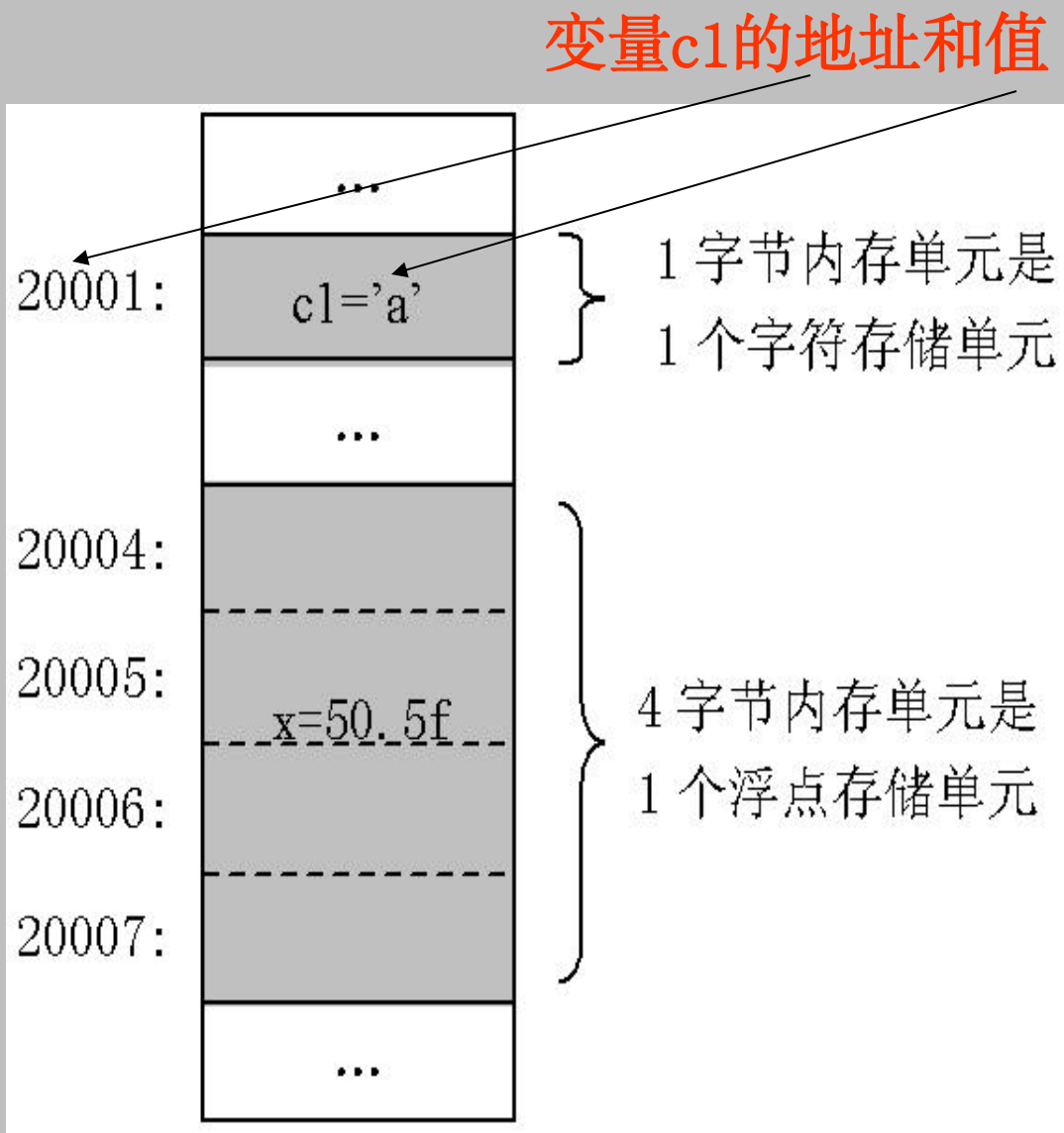
- 设有说明语句:

```
char c1='a';
```

```
float x=50.5f;
```

在程序执行时，系统为变量c1和x分配内存单元(设分配的内存单元分别为20001和20004)，如图所示。

- 注意区分变量的地址和变量的值。



指针的概念

- 指针的引入：变量的地址仅能指示和定位内存单元的开始位置，而无法确定多少个连续内存单元组成一个存储单元。
- 指针：新类型数据，其值代表地址，其型代表多少个连续内存单元组成一个存储单元。
- 指针分为：常量指针、变量指针和函数指针。
- 注意指针与地址的区别，不能混为一谈。

指针变量

- 指针变量：存放指针的变量。

- 说明指针变量的格式：

《存储类型》<类型>*<变量名1>《,*<变量名2>,...》；

其中，存储类型任选；变量名前的*指明所说明的变量为指针变量；而类型则指出指针变量所指向的数据类型，即指针所指向的内存单元中存放的数据的类型。

- 例如：

```
int *p1,*p2,i,j;
```

- 说明：

- 指针变量的值是一个地址，取值范围是内存地址范围，在PC机中用4个字节来存放地址值，即不同类型的指针变量所分配的存储单元大小相同。

- 定义指针变量时，其类型定义了指针变量所指数据的类型，即确定了指针所指数据占用的存储单元，如p1所指数据为整型。
- 与普通变量一样，指针变量也只有在具有确定的初值后才可使用。指针变量的初值通常是所指类型的某个变量的指针。
- 例如：

```
int j,*p;
```

```
p=&j; //使整型指针变量p指向整型变量j
```

此处，一元运算符“&”称为**取指针运算符**，其操作数是一个变量，其运算结果是取指定变量的指针。当然，在说明指针变量时也可对其初始化，

例如：

```
int j,*p=&j;
```

指针的运算

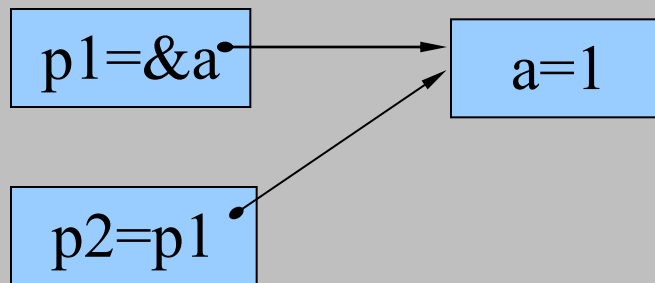
- 指针的运算：赋值、关系、算术和强制类型转换。
- 指针的赋值：将一个指针赋给一个指针变量。
 - 将同类型的任一变量的指针赋给指针变量。如：

```
int a=1,*p1,*p2;
```

```
p1=&a; //将变量a的指针赋给p1
```

```
p2=p1; //同类型指针变量之间的赋值
```

使指针变量p1和p2都指向变量a，如图所示。



访问指针变量所指变量要用一元运算符“*”，该运算符称为**取变量运算符**，要求一个指针变量作为它的操作数，它的运算结果为取其操作数所指变量。如：

```
int *ip1,*ip2,i=100,j;
```

```
ip1=&i;ip2=&j;
```

```
*ip2=*ip1+200;
```

其中，***ip1**取的是ip1所指的变量，即变量i；***ip2**取的是ip2所指的变量，即变量j。

- 0赋给指针变量是使指针变量的值为“空”，即值为0的指针变量不指向任一变量。例如：

```
#include<iostream.h>
void main(void)
{  int* pt=0;
   *pt=100;           //A
   cout<<*pt<<'\n';
}
```

该程序能编译和链接，但当程序运行到A行时，系统提示“该程序执行了非法的操作”，并终止执行。原因是pt不指向任一内存单元，当然不允许向pt所指内存单元赋值。

- 静态存储类型的指针变量，其缺省的初值为0。

- C++允许将一个整型常数经强制类型转换后赋给一个指针变量。例如：

```
float *fp;
```

```
fp=(float*)5000;
```

- 仅在设计系统程序或对计算机内存分配非常清楚，且有明确目的时才有意义，否则可能产生严重后果。
- 初学者不应使用这种方法初始化指针变量。

– 向一个未初始化的指针变量所指存储单元赋值是极其危险的。例如：

```
#include<iostream.h>
void main(void)
{ int*pp;
  cout<<“输入一个整数:”;
  cin>>*pp;    //A
  cout<<*pp<<'\n';
}
```

程序看似正确，但存在潜在问题。原因：指针变量pp为局部变量，系统为它分配内存时，并未对它初始化，即pp的值是随机的，pp所指存储单元可能是未分配的内存，也可能是已分配的内存。若是后者，把数据写入该存储单元，轻者导致程序运行结果出错，重者导致系统出错或崩溃。

该程序可被编译和链接，但在编译时，程序行A有一个警告"warning C4700: local variable 'pp' used without having been initialized",执行时输入100，输出也是100。

- 不同类型的指针变量之间的赋值经强制类型转换后是允许的，但应有明确的目的和意义。例如：

```
#include<iostream.h>
void main(void)
{  int *p1;
   float x=2.5,*p3=&x;
   cout<<"*p3="<<*p3<<"\n";  //A
   p1=(int*)p3;                  //B
   cout<<"*p1="<<*p1<<"\n";  //C
}
```

执行程序后，输出：

*p3=2.5

*p1=1075838976

A行取p3所指变量时，按实数格式解释，正确输出2.5；经B行赋值后，p1和p3的值相同，但p1的指针类型是整型，故C行取p1所指变量时，应按整数格式解释，输出的值不是2.5，而是1075838976。

指针的算术运算

- 指针变量执行“++”或“--”操作：使指针变量指向下一个或上一个变量，而指针变量的值实际加或减 $\text{sizeof}(\text{<指针变量类型>})$ 。
- 指针变量加或减一个整型值 n ，即：
$$\text{<指针变量>} = \text{<指针变量>} \pm n$$

使指针变量指向其后或其前的第 n 个变量，指针变量的值实际增减为：

$$\text{<指针变量>} = \text{<指针变量>} \pm \text{sizeof}(\text{<该指针变量的类型>}) * n$$
- 两个同类型指针值相减，代表两个指针间的变量个数。主要用于数组应用中。

例4.1 指针变量的算术运算。

```
#include<iostream>
using namespace std;
void main(void)
{ int a[10]={10,20,30,40,50,60}, *p1=&a[0], *p2=&a[5];
```

```
    cout<<"\n*p1="<<*p1;
```

```
    cout<<"\n*p2="<<*p2;
```

```
    p1++; p2--;
```

```
    cout<<"\n*p1="<<*p1;
```

```
    cout<<"\n*p2="<<*p2;
```

```
    p1+=2;
```

```
    cout<<"\n*p1="<<*p1;
```

```
    cout<<"\n*(p2+1)="<<*(p2+1);
```

```
    cout<<"\np2与p1之间整型变量的个数="<<p2-p1;
```

```
}
```

程序运行结果:

*p1=10

*p2=60

*p1=20

*p2=50

*p1=40

*(p2+1)=60

p2与p1之间整型变量的个数=1

指针的关系运算

- 所有关系运算均可用于指针，同类型的指针比较才有意义。
- 按指针值的大小(作为无符号整数)进行比较。
 - 相等比较：判断两个指针是否指向相同的变量；
 - 不等比较：判断两个指针是否指向不同的变量；
 - 与0比较：判断指针的值是否为空。
- 主要用于数组方面的应用。

例4.2 指针变量的关系运算。

```
#include<iostream.h>
```

```
void main(void)
```

```
{ int a[5]={100,200,300,400,500},*p1,*p2,sum=0;
```

```
  for(p2=&a[0];p2<=&a[4];p2++)
```

```
    cout<<*p2<<'\\t'; //问: for循环执行后, p2=?
```

```
  p1=&a[0]+5;      //使p1指向数组a的最后元素的后面
```

```
  p2=&a[0];        //使p2再指向数组a的第0个元素
```

```
  while(p2!=p1)    //此处条件还可表示为: p2<p1
```

```
    sum+=*p2++;
```

```
  cout<<"元素之和为: "<<sum<<"\\n";
```

```
}
```

程序输出:

100 200 300 400 500

元素之和为: 1500

***p2++:** 因运算符++和*的优先级相同, 按从右到左的顺序计算, 先进行p2++的运算, 即 “*p2++;”等价于 “sum+=*p2,p2++;”。

- “*” 既是乘法运算符，又是取变量运算符。
- “&” 可表示“按位与”，又可表示取指针。

- 注意下列几种运算符的混合运算：

- 取变量运算符 “*” 和取指针运算符 “&” 的优先级相同，按自右向左的方向结合。设有说明语句：

```
int a[5]={100,200,300,400,500},*p=&a[0],b;
```

- **&*p**: 先做 “*” 运算，再做 “&” 运算，即先取 p 所指变量，再取该变量的指针。该表达式的值为变量 a[0] 的指针。

- ***&a[0]**: 先做 “&” 运算，得到变量 a[0] 的指针，再做 “*” 运算，取该指针所指变量，即 a[0]。

– “++”、“--”、取变量运算符 “*” 和取指针运算符 “&” 的优先级相同，按自右向左的方向结合。例如：

- **p=&a[0],b=*p++:** 对于*p++, 先做 “*”, 再做 “++”, 因 “++” 为后置运算, 故等同于先取 *p, 再使指针p加1。结果: b为100, p指向a[1]。
- **p=&a[0],b=*++p:** 对于*++p, 先做 “++”, 再做 “*”, 因 “++” 是前置运算, 故先使指针p加1, 再取*p。结果: b为200, p指向a[1]。
- **p=&a[0],b=(*p)++:** 对于(*p)++, 先做 “*”, 再做 “++”, 即先取*p, 再使*p加1。结果: b为100, p仍指向a[0], a[0]的值改为101。
- ***(p++):** *(p++)等同于*p++。
- **p=&a[1],b=++*p:** ++*p等同于++(*p)。结果: b为201, a[1]的值改为201, p仍指向a[1]。

指针值的输出

例4.3 指针变量值的输出。

```
#include<iostream.h>
```

```
void main(void)
```

```
{ int i=10,*pi=&i;
```

```
char c[]="abcd",*pc=&c[0];
```

```
cout<<"*pi="<<*pi<<"\t"<<"pi="<<pi<<"\n";
```

```
cout<<"*pc="<<*pc<<"\t"
```

```
<<"pc="<<(int*)pc<<"\n";//输出pc的值
```

```
cout<<"c[]="<<c<<"\n";
```

```
cout<<"c[]="<<pc<<"\n"; //输出pc所指的字符串
```

```
}
```

程序运行结果:

```
*pi=10   pi=0x0065FDF4
```

```
*pc=a    pc=0x0065FDE8
```

```
c[]=abcd
```

```
c[]=abcd
```

- 通常，用cout直接输出指针时输出的是指针的16进制值；
- 但因用cout输出字符型指针时，系统规定的输出是其所指的字符串，故若输出字符指针的值，则应转成其它类型的指针后再用cout输出。

4.2 指针与数组

- 指针与数组关系密切，可用指针访问数组中任一元素。
- 用指针访问数组，可使程序紧凑，运行速度提高。
- C/C++规定，在说明一个数组后，**该数组名便作为一个常量指针来使用**，其值为该数组在内存中的起始地址，即数组的第1个元素的起始地址；其型是该数组元素类型。

指针与一维数组

- 若说明了一个与一维数组的类型相同的指针变量，并使该指针变量指向数组的第1个元素，则就可用该指针变量存取所指数组的所有元素。
- 例4.4 用指针访问数组元素。

```
#include<iostream>
using namespace std;
int main(void)
{  int a[10],i,*p;
```

```
    //用指针变量初始化数组a
    for(p=a,i=0;i<10;i++,p++) *p=i;
```

//输出a数组的所有元素，有以下三类四种方法：

for(**p=a**,i=0;i<10;i++) //①指针变量

cout<<"a["<<i<<"]="<<***p++**<<'\\t';

for(cout<<'\\n',**p=a**,i=0;i<10;i++)//②指针常量

cout<<"a["<<i<<"]="<<***(p+i)**<<'\\t';

for(cout<<'\\n',i=0;i<10;i++) //③指针常量

cout<<"a["<<i<<"]="<<***(a+i)**<<'\\t';

for(cout<<'\\n',i=0;i<10;i++) //④下标

cout<<"a["<<i<<"]="<<**p[i]**<<'\\t';

cout<<'\\n';

return 0;

}

例4.4说明:

- 设 $p=a$, 则 $p+i$ 、 $a+i$ 、 $\&a[i]$ 均为 $a[i]$ 的指针; $*(p+i)$ 、 $*(a+i)$ 、 $p[i]$ 和 $\&a[i]$ 均为 $a[i]$ 。
 - 访问数组 a 中元素的方法有三类, 但效率不同:
 - 指针变量法: $p++$ 等价于 $p=p+4$
 - 指针常量法: $p+i$ 等价于 $p+i*4$, $a+i$ 等价于 $a+i*4$
 - 下标法: $p[i]$ 等价于 $*(p+i)$
- 指针变量法在更新指针值时比其它方法少做一次乘法。
- C/C++数组下标从0开始也是为了提高运行效率。设整型数组 a 的下标从指定整型值 m 开始、 n 结束, 这样数组 a 的元素依次为 $a[m]$ 、 $a[m+1]$ 、...、 $a[n]$, 而 $a[i]$ ($n \geq i \geq m > 0$) 的指针的算式为:
 $a+(i-m)*4$, 比从0开始还要多做一次减法
 - C/C++不对数组下标和指针越界做检查, 也是为了提高运行效率, 因此编程时应关注数组下标和指针是否越界。

指针与多维数组

- 以二维数组为例，讨论指针与多维数组的关系。
- 因二维数组元素在内存中按行逐列连续存放，故可将二维数组分配的连续内存作为一维数组使用。例如，下列程序段输出二维数组a的所有元素：

```
int a[3][3]={1,2,3},{4,5,6},{7,8,9}},i,j,*p;  
for(p=&a[0][0],i=0;i<3;i++,cout<<'\\n')  
    for(j=0;j<3;j++,cout<<'\\t')  
        cout<<*(p+3*i+j); //或: cout<<*p++;
```

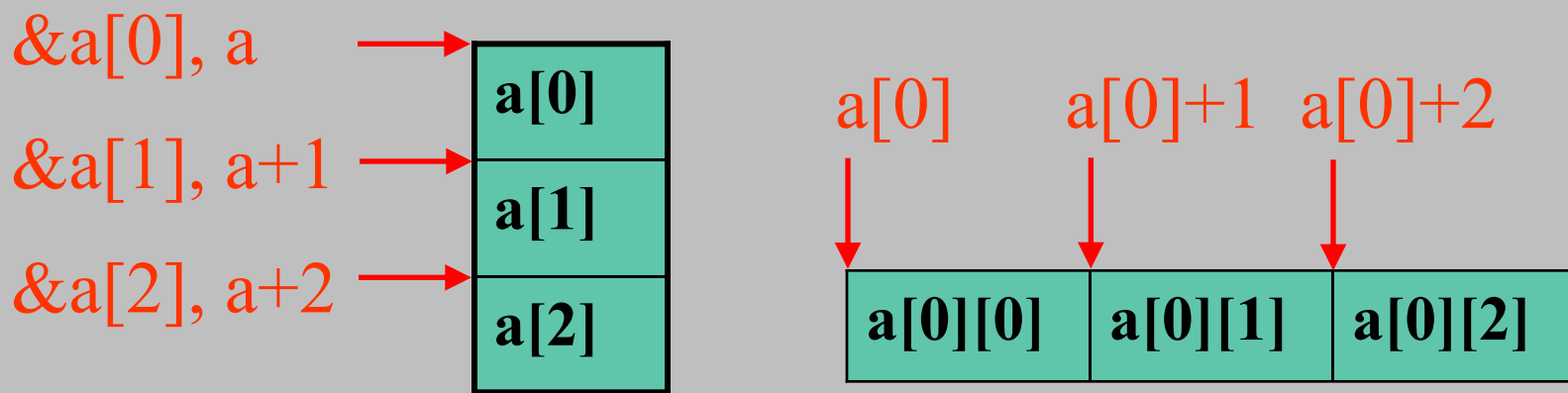
可见，若 $p = \&a[0][0]$ ，则 $a[i][j]$ 的指针还可表示为
 $(p + 3 * i + j)$

不足：算式复杂、运行效率低且未体现二维数组的特点。

设有如下数组说明:

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

在C/C++中, 可将二维数组的每一行看成一个元素, 即数组a包含了三个元素a[0]、a[1]、a[2], 数组名a是该一维数组a的指针, 即a[0]元素的指针, a+1是a[1]元素的指针, a+2是a[2]元素的指针。如图a所示。

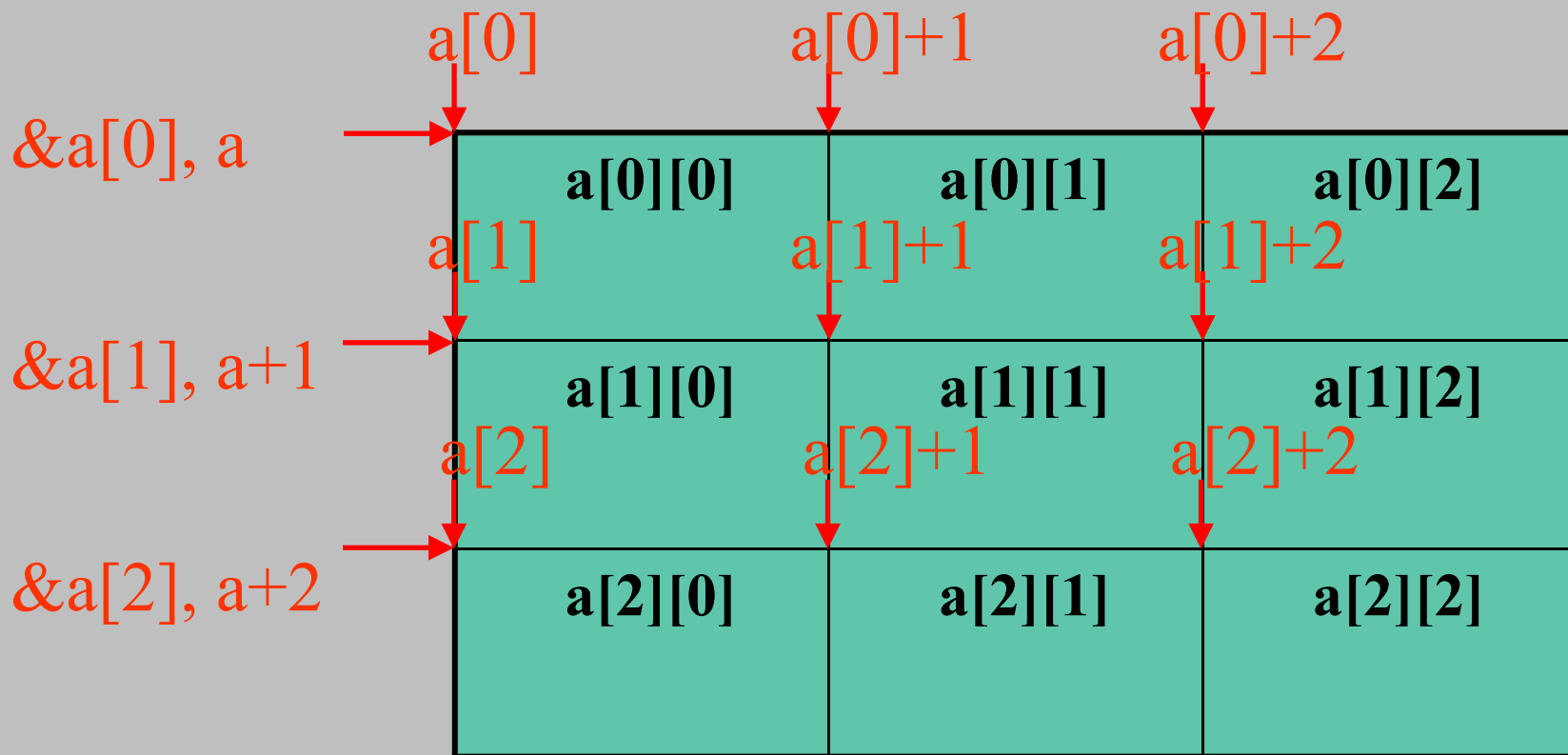


图a 一维数组a的指针

图b 一维数组a[0]的指针

因a[0]又是一个一维数组, 包含三个元素a[0][0]、a[0][1]、a[0][2], 因此a[0]就是一维数组a[0]的指针, 即数组元素a[0][0]的指针, a[0]+1是数组元素a[0][1]的指针, a[0]+2是数组元素a[0][2]的指针, 如图b所示。a[1]、a[2]类推。

- 指针与二维数组的关系可用下图概括。其中：
 - **行指针**：行方向的指针 a 、 $a+1$ 和 $a+2$ ，是指向一维数组的指针，其指针类型为含有三个整型元素的一维数组；
 - **列指针**：列方向的指针 $a[0]$ 、 $a[0]+1$ 、 $a[0]+2$ 、...、 $a[2]+2$ ，其指针类型为整型。
 - **注意**：尽管 a 与 $a[0]$ 、 $a+1$ 与 $a[1]$ 、 $a+2$ 与 $a[2]$ 的值相同，但类型不同，若有必要，可相互转换。



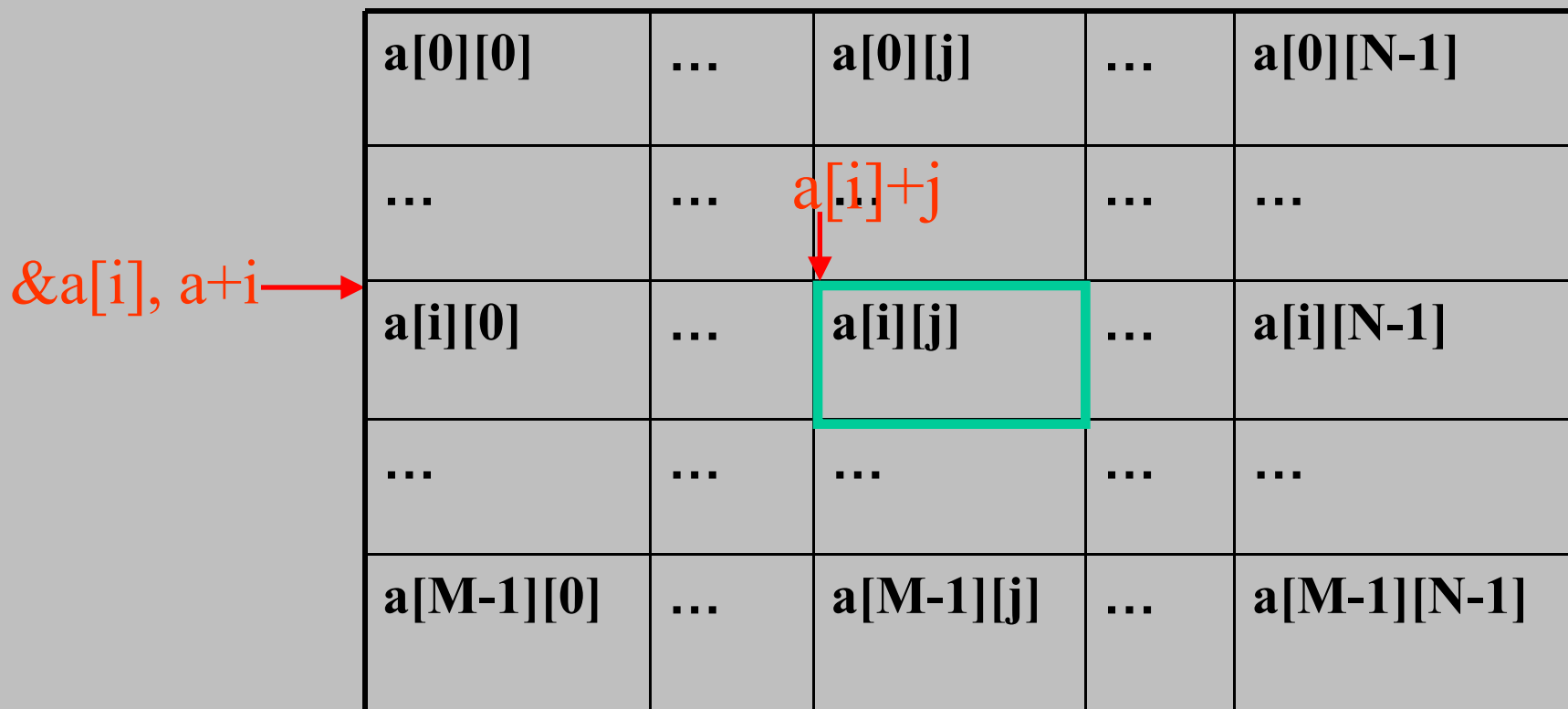
- 推广：设有说明 “`int a[M][N];`”(M、N均为符号常量)，则元素 `a[i][j]` ($0 \leq i < M$, $0 \leq j < N$) 的指针为：

① `a[i]+j`, 原因如图 ② `*(a+i)+j`, 因 `a[i]` 等价于 `*(a+i)`

③ `*(&a[i])+j` ④ `&a[i][j]`

元素 `a[i][j]` 也有如下等价表示：

① `*(a[i]+j)` ② `*(*(a+i)+j)` ③ `(*(&a[i]))[j]`



<code>a[0][0]</code>	...	<code>a[0][j]</code>	...	<code>a[0][N-1]</code>
...	...	<code>a[i]+j</code>
<code>&a[i], a+i</code> → <code>a[i][0]</code>	...	<code>a[i][j]</code>	...	<code>a[i][N-1]</code>
...
<code>a[M-1][0]</code>	...	<code>a[M-1][j]</code>	...	<code>a[M-1][N-1]</code>

例4.5 用指针输出二维数组的元素值。

```
#include<iostream>
#include<iomanip>
using namespace std;
void main(void)
{ int a[3][3]={{1,2,3},{4,5,6},{7,8,9}},i,j,*p;

  cout<<"用指针输出数组的全部元素: \n";
  for(p=(int*)a,i=0;i<9;i++)
  { if(i&& i%3==0) cout<<"\n";
    cout<<*p++<<"\t";
  }
}
```

```
cout<<"\n用四种不同方法输出数组的元素: \n";
```

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

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

```
        cout<<*(a[i]+j)<<'t'
```

```
            <<*(*a+i)+j)<<'t'
```

```
            <<(*a+i)[j]<<'t'
```

```
            <<a[i][j]<<'n';
```

```
cout<<"用指针输出数组的各个元素: \n";
```

```
for(i=0,p=a[0];p<=a[2]+2;p++,i++)
```

```
{ if(i&& i%3==0) cout<<'n';
```

```
    cout<<setw(4)<<*p;
```

```
}
```

```
}
```

程序运行结果:

用指针输出数组的全部元素:

1	2	3
4	5	6
7	8	9

用四种不同方法输出数组的元素:

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	7	7
8	8	8	8
9	9	9	9

用指针输出数组的各个元素:

1	2	3
4	5	6
7	8	9

(11) 作业

- 在主函数中实现输入和输出，编写子函数，实现对一个50*50的二维数组的赋值和打印，要求使用指针作为参数和返回值

指针和字符串

- 字符串存于字符数组中，亦可用指针处理。
- 用指针处理字符串时，通常非常关心是否已处理到了字符串的结束字符，而并不太关心存放字符串的数组的大小。
- 用指针处理字符串，程序更加高效和简便。
- 例4.6 用指针实现字符串的复制。

```
#include<iostream>
#include<string.h>
using namespace std;
void main(void)
{ char s1[]="I am a student!";
  char*s2="You are a student!";
  char s3[30],s4[30],s5[30],*p1,*p2;

  for(p1=s3,p2=s1;*p1++=*p2++); //①用指针复制
```

将字符串常量"You are a student!"的首字符的指针赋值给指针变量s2

- 逐个字符复制，直到 '\0' 复制完才结束!
- 效率极高。

```
for(unsigned i=0;i<=strlen(s1);i++)//②用数组复制
```

```
    s4[i]=s1[i];
```

```
strcpy_s(s5,s2);
```

```
//③用库函数复制
```

```
cout<<"s3="<<s3<<"\ns4="<<s4
```

```
    <<"\ns5="<<s5<<"\ns2="<<s2<<'\\n';
```

```
}
```

程序运行结果:

```
s3=I am a student!
```

```
s4=I am a student!
```

```
s5=You are a student!
```

```
s2=You are a student!
```

- 逐个字符复制，直到 ‘\0’ 复制完才结束!
- 效率较低。

- 字符型指针变量与字符数组均可处理字符串，但有所不同：

- 定义形式不同。例如：

`char*p,s[100];`//内存:数组s占100字节，变量p占4字节

- 初始化形式相似，但含义不同。例如：

`char s[]="I am a student!";`

`char*p="You are a student!";`

对于字符数组s是把字符串初值送到其内存中去；而对于字符型指针p，是先把字符串常量存放到内存中，然后将该字符串的首字符的指针送到指针变量p中。

- 赋值方式不同。对于字符数组赋值，须逐个元素赋值；对于字符型指针，可将任一指针值赋给字符指针变量。例如：

`char s[]="I am a student!",t[100],*p;`

`t=s;` //错：因数组名为常量指针。改： `strcpy_s(t,s);`

`p="I love China!";` //正确

- 可以给字符数组直接输入字符串；而在给字符指针变量赋初值前，不允许将输入的字符串送到指针变量所指向的内存区域。例如：

```
char s1[50],s2[200],*p;
```

```
cin>>s1; //正确
```

```
cin>>p; //警告： p指向随机内存单元， 潜存危险
```

```
p=s2;
```

```
cin>>p; //正确， 此时p指向已分配的内存单元
```

- 字符数组名是常量指针， 不能改变； 而字符指针变量的值可变。 例如：

```
p=s1+5; //正确
```

```
p=p+5; //正确
```

```
s1=s1+2; //错误： 数组名是常量指针， 不能改变
```


4.3 指针数组和指向指针的指针变量

- 指针数组：每个元素均为同类型指针变量的数组。格式：
《存储类型》 <类型> *<数组名>[<整型常量表达式>];
因“[]”的优先级高于“*”，<数组名>与[<整型常量表达式>]构成一个数组，再与*结合，指明是一个指针数组，类型指明指针数组中每个元素所指变量的类型。例如：

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

说明p为一个整型指针数组，由4个整型指针变量元素组成。

- 例4.8 用指针数组输出数组中各元素的值。

```
#include<iostream.h>
```

```
void main(void)
```

```
{ float a[]={10,20,30,40},*p[]={a,a+1,a+2,a+3};
```

```
for(int i=0;i<4;i++) cout<<*p[i]<<"\t";
```

```
}
```

程序输出结果:

10 20 30 40

程序中说明了一个浮点指针数组p，它的每个元素依次指向数组a中的每个元素，如下图所示。

指针数组p

数组a

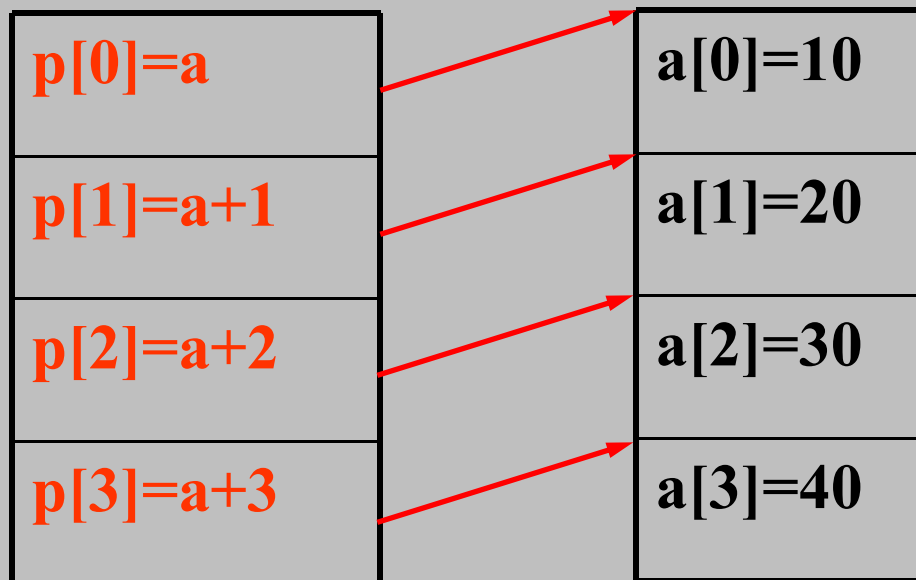
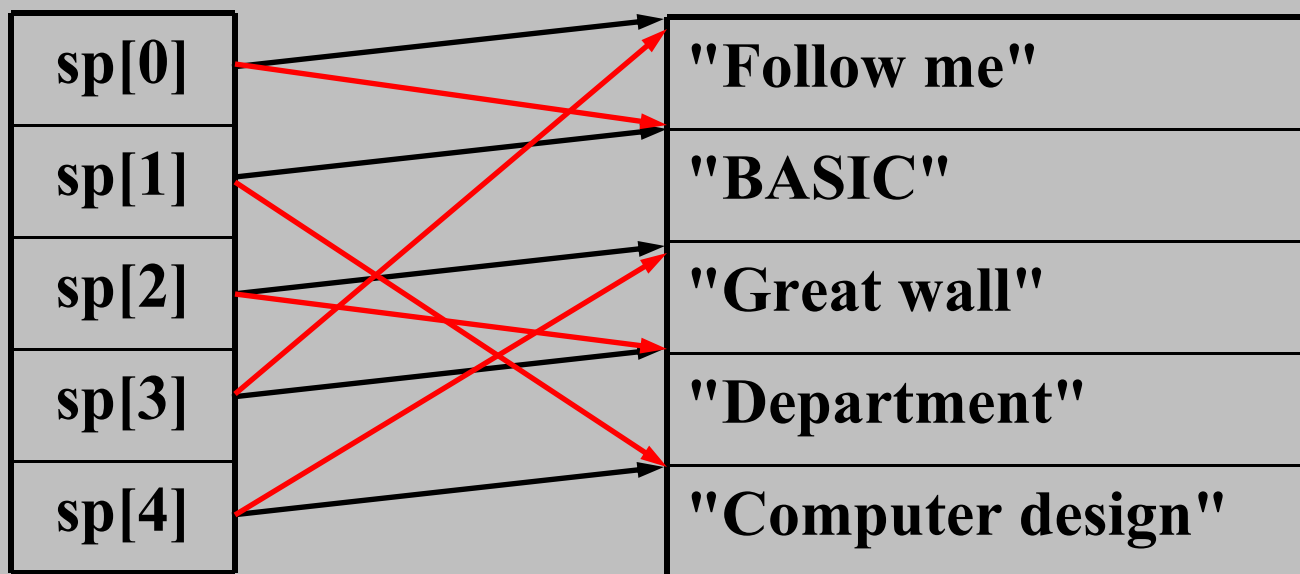


图. 指针数组p与数组a的关系

- 例4.8 使用指针数组，将若干字符串按升序排序后输出。
 - 指针数组sp的每个元素指向一个字符串，如图所示。
 - 用选择法排序。首先在sp[0] ~ sp[4]指针所指范围找到最小的字符串，并使该指针值与sp[0]中的指针交换，使sp[0]指向最小的字符串。接着在sp[1] ~ sp[4]指针所指范围找到最小的字符串，使sp[1]指向最小的字符串。依此类推，直到在sp[4]指针所指范围为止。
 - 用指针数组，排序时仅换指针，不换字符串，效率极高。



图例：

排序前 →

排序后 →

程序如下：

```
#include<iostream.h>
```

```
#include<string.h>
```

```
void main(void)
```

```
{ char *p,*sp[]={"Follow me","BASIC","Great wall",  
                "Department","Computer design"};
```

```
int i,j, //循环变量
```

```
    k,    //使sp[k]指向最小字符串
```

```
    n;    //字符指针数组大小，即字符串个数
```

```
n=sizeof(sp)/sizeof(sp[0]);
```

```
for(i=0;i<n-1;i++)    //选择法排序
```

```
{ for(k=i,j=i+1;j<n;j++)
```

```
    if(strcmp(sp[k],sp[j])>0) k=j;
```

```
        //若sp[i]所指不是最小字符串,  
if(k!=i)    //则使sp[i]指向最小字符串  
    p=sp[k],sp[k]=sp[i],sp[i]=p;  
}  
for(i=0;i<n;i++) //输出排序后的字符串  
    cout<<sp[i]<<'\n';  
}
```

程序运行结果:

BASIC

Computer design

Department

Follow me

Great wall

- 例4.9 使用指针数组访问二维数组。
 - 思路：先用二维数组的列指针初始化指针数组元素，再用指针数组访问二维数组。
 - 源程序：

```
#include<iostream.h>
void main(void)
{ int i,j,*p[3],a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
  for(i=0;i<3;i++)
    for(p[i]=a[i],j=0;j<3;j++)
      cout<<*(p[i]+j)<<','
        <<*(*(p+i)+j)<<','
        <<(*(*(p+i)))[j]<<','
        <<p[i][j]<<"\n";
}
```

程序运行结果:

1,1,1,1

2,2,2,2

3,3,3,3

4,4,4,4

5,5,5,5

6,6,6,6

7,7,7,7

8,8,8,8

9,9,9,9

- 指针与二维数组的关系

若有以下语句:

```
int a[3][3], *p[3], i, j;
```

```
for(i=0; i<3; i++) p[i]=a[i];
```

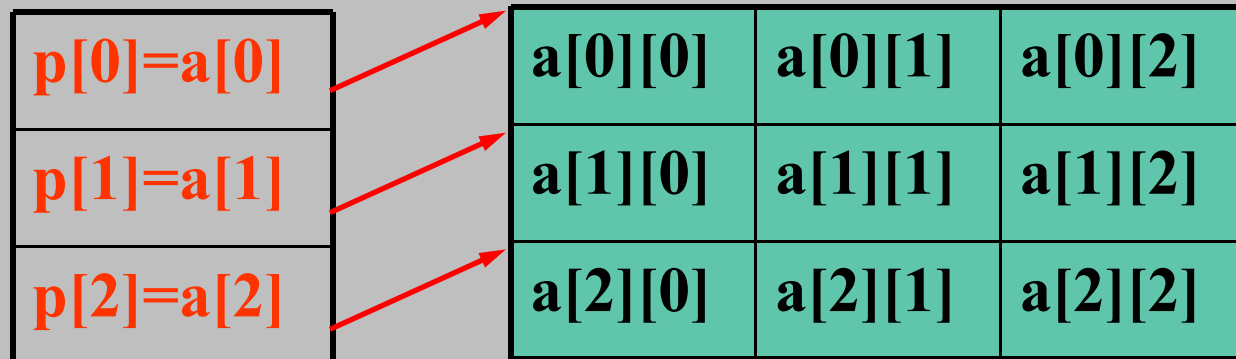
则二维数组元素 $a[i][j]$ ($0 \leq i < 3$, $0 \leq j < 3$)
的使用形式还可表示为:

① $*(p[i]+j)$

② $*(*(p+i)+j)$

③ $(*(p+i))[j]$

④ $p[i][j]$



指向一维数组的指针变量

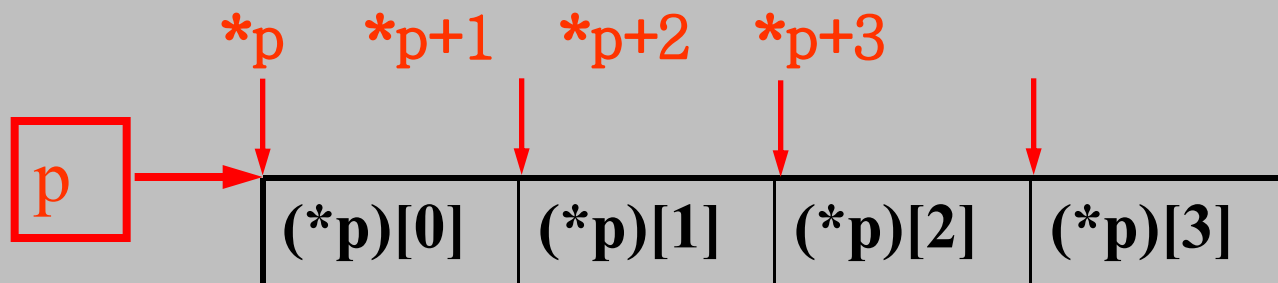
- 指向一维数组的指针常量，即二维数组的行指针常量。
- 指向一维数组的指针变量，即二维数组的行指针变量。

– 例如：

`int(*p)[4];`

其中(*p)指明p是一个指针变量，与[4]结合，表示该指针变量指向一个含有四个整型元素的一维数组。注意，圆括号不可少，否则p是一个指针数组。

– 因p是行指针，故*p、*p+1、*p+2、*p+3依次为p所指一维数组的4个元素的指针，对p所指一维数组的4个元素的访问形式为：*(p)、*(p+1)、*(p+2)、*(p+3)，再等价写成一维数组元素形式：(*p)[0]、(*p)[1]、(*p)[2]、(*p)[3]。



- 行指针与二维数组的关系

若有如下说明:

```
int a[3][4],(*p)[4]=a;
```

则元素 $a[i][j]$ ($0 \leq i < 3$, $0 \leq j < 4$) 的访问形式还可写成:

① $*(p[i]+j)$ ② $*(*(p+i)+j)$ ③ $(*(p+i))[j]$ ④ $p[i][j]$

- 小结: 访问二维数组的方法有

① 下标法

② 常量指针法(注意行指针与列指针的关系)

③ 变量指针法。即行指针变量与二维数组的行指针常量配合使用, 或指针数组与二维数组的列指针常量配合使用。

- 说明指向二维、三维数组的指针变量, 如:

```
int(*ptr)[20][50];
```

定义了一个指向一个20行50列的二维数组的指针变量ptr。

例4.10 用指向一维数组的指针变量输出二维数组中的元素。

```
#include<iostream.h>
```

```
void main(void)
```

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

```
cout<<"用行指针输出数组的各个元素: \n";
```

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

```
{ p=&a[i];
```

```
cout<<(*p)[0]<<','<<(*p)[1]<<','<<(*p)[2]<<'\n';
```

```
}
```

```
cout<<"用四种不同的方法输出数组的各个元素: \n";
```

```
for(p=a,i=0;i<3;i++)
```

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

```
cout<<*(*(p+i)+j)<<','<<*(p[i]+j)<<','
```

```
<<(*p[i])[j]<<','<<p[i][j]<<'\n';
```

```
}
```

for(p=a,i=0;i<3;i++,p++)//可否?

cout<<(*p)[0]<<'\t'<<(*p)[1]<<'\t'<<(*p)[2]<<'\n';

$(*p)[0] \iff (*\&a[i])[0] \iff (a[i])[0] \iff a[i][0]$

程序运行结果:

用行指针输出数组的各个元素:

1,2,3

4,5,6

7,8,9

用四种不同的方法输出数组的各个元素:

1,1,1,1

2,2,2,2

3,3,3,3

4,4,4,4

5,5,5,5

6,6,6,6

7,7,7,7

8,8,8,8

9,9,9,9

指向指针的指针变量

- 指向指针的指针变量，俗称二级指针。例如：

```
int x=10,*p=&x,**pp=&p;
```

其中，变量pp存放指针变量p的指针，为二级指针。

- 二级指针的定义格式：

《存储类型》<类型>**pp;

定义一个二级指针变量时，在其前面加两个“*”。类似地，定义一个三级指针变量，则在指针变量前加三个“*”。在C++中，对定义指针的级数并无限制。

- 一、二级指针变量较常用，三级以上的指针变量很少用。

- 例4.11 多级指针的简单使用。

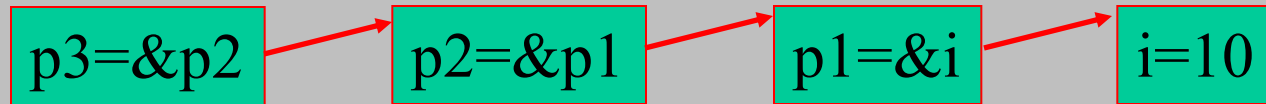
```
#include<iostream.h>
```

```
void main(void)
```

```
{  int i=10,*p1=&i,**p2=&p1,***p3=&p2;
```

```
    cout<<i<<','<<*p1<<','<<**p2<<','<<***p3<<'\n';
```

```
}
```



程序输出结果:
10,10,10,10

程序中指针变量的指向关系如上图所示。从图及输出结果可见，在二级指针变量前加一个“*”得到的是一个指针；加上两个“*”才能得到最终的普通变量。这种规则可以推广到任意多级的指针变量。

4.4 指针和函数

- 指针与函数的联系：
 - 做函数参数
 - 做函数返回值
 - 做函数的指针

指针做函数的参数

- 例4.12 实现两个数据的交换。

```
#include<iostream.h>
```

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

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

```
void main(void)
```

```
{ int a=2,b=3;
```

```
  cout<<"a="<<a<<",b="<<b<<"\n";
```

```
  swap(&a,&b);
```

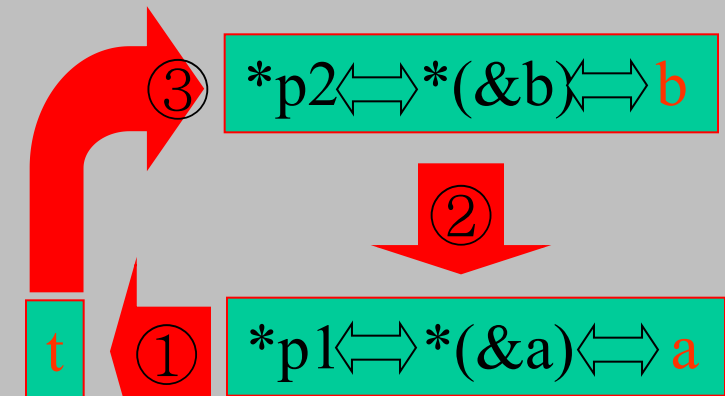
```
  cout<<"a="<<a<<",b="<<b<<"\n";
```

```
}
```

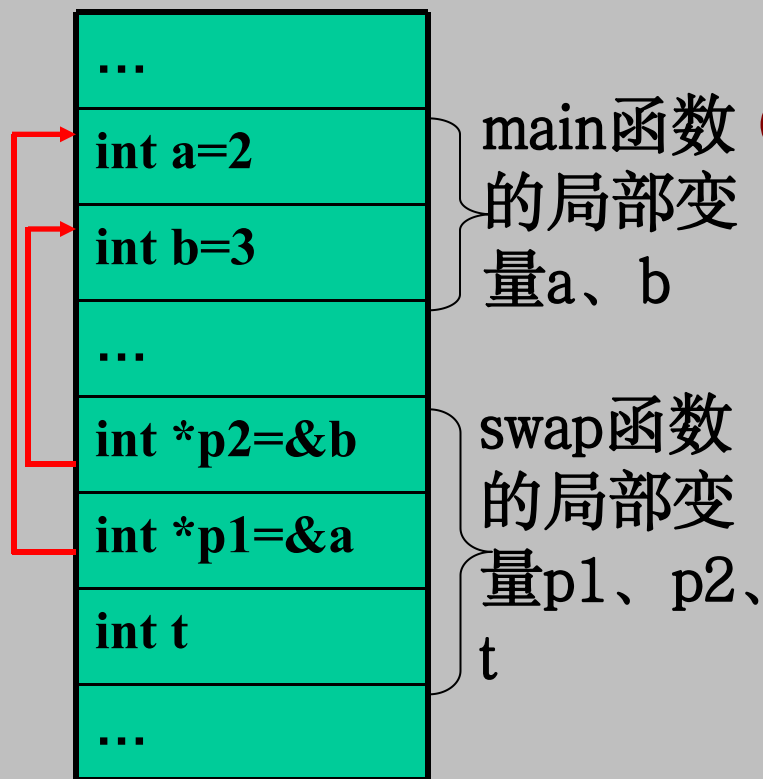
程序运行结果:

a=2,b=3

a=3,b=2



图a.swap函数对块作用域外的变量a、b的间接互换



图b.内存分配示意图

- 在main()函数中调用swap(&a,&b)函数时，系统为swap()函数分配内存并完成：

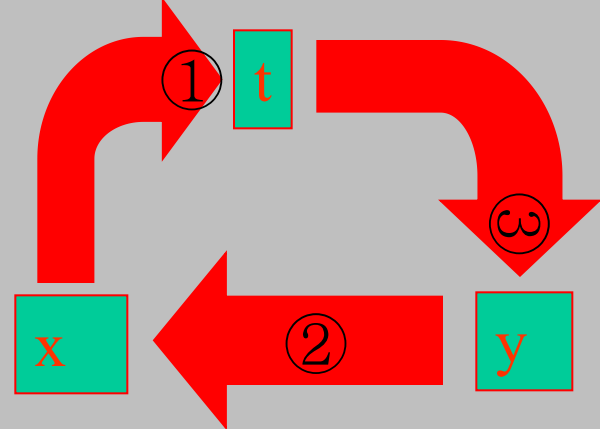
- ① **指针参数的值传递：** 实参&a、&b，即main()函数中局部变量a、b的指针，复制给形参指针变量p1、p2，如上图b所示；
- ② **对块作用域外的变量a、b的间接互换：** 虽然swap()函数在变量作用域范围内，无法直接访问main()函数中的局部变量a、b，但因其形参p1、p2分别获得a、b的指针，故可通过指针变量p1、p2，突破变量作用域的限制，间接访问a和b，通过交互*p1、*p2的值，实现main()函数中局部变量a、b的互换。如上图a所示。

- swap(&a,&b)函数调用结束后，指针变量p1、p2、t生存期结束，系统回收分配给swap()的内存，但swap()函数调用对main()函数中的局部变量a、b的互换是有效的。
- 结论：函数可通过指针变量参数的间接访问机制，突破变量作用域限制，访问指针变量所指的变量，即指针参数使函数能改变该函数作用域外的一个或一个以上变量的值。

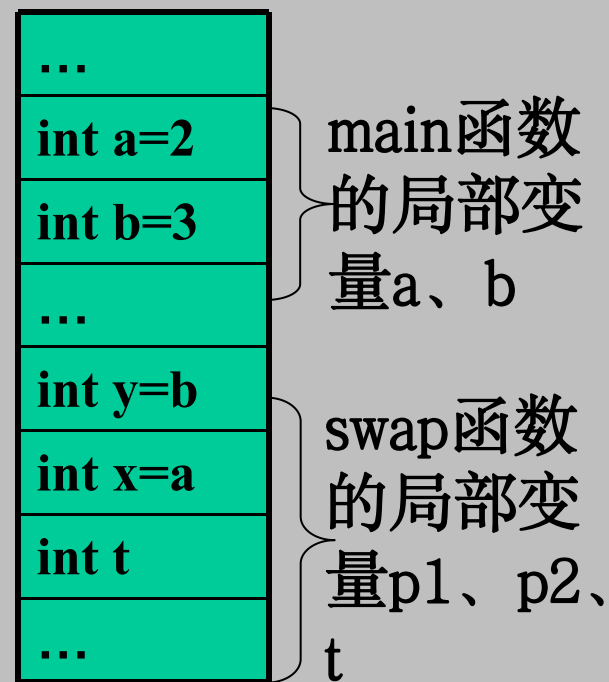
- 讨论：若将例4.12中的swap()函数的两个指针参数改为两个整型参数，并将程序改为以下形式，能否通过swap(a,b)调用，实现a、b的交换？

```
#include<iostream.h>
void swap(int x,int y)
{  int t; t=x; x=y; y=t; }
void main(void)
{  int a=2,b=3;
  cout<<"a="<<a<<",b="<<b<<'\n';
  swap(a,b);
  cout<<"a="<<a<<",b="<<b<<'\n';
}
```

- 答案：不能。因swap函数的形参x、y获得a、b实参的值后，仅做了形参x、y的互换，对a、b无任何影响。



图a.swap函数对自身局部变量x、y的直接互换



图b.内存分配

- 例4.13 使用数组或指针做函数的参数，实现数组的排序。
 - 数组名做函数的参数时，其作用与指针相同。
 - 数组或指针做函数的参数有四种可能，效果相同：①形参为数组，实参为数组名；②形参为指针，实参为数组名；③形参为数组名，实参为指针；④形参和实参都为指针。

```
#include<iostream.h>
```

```
void sort1(int*p,int n)
{ int i,j,t;
  for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
      if(*(p+i)>*(p+j))
        t=p[i],
        p[i]=p[j],p[j]=t;
}
```

```
void sort2(int b[ ],int n)
{ int i,j,t;
  for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
      if(b[i]>b[j])
        t=*(b+i),
        *(b+i)=b[j],b[j]=t;
}
```

```
void print(int *p,int n)
{ for(int i=0;i<n-1;i++,p++) cout<<*p<<',';
  cout<<*p<<'\n';
}
```

```
void main(void)
{  int a[6]={4,67,3,45,34,78};
    int b[6]={4,67,3,45,34,78},*pb=b;
    int c[6]={4,67,3,45,34,78},*pc=c;
    int d[6]={4,67,3,45,34,78};
    sort1(a,6); print(a,6);
    sort1(pb,6); print(b,6);
    sort2(d,6); print(d,6);
    sort2(pc,6); print(c,6);
}
```

程序运行结果:

3,4,34,45,67,78

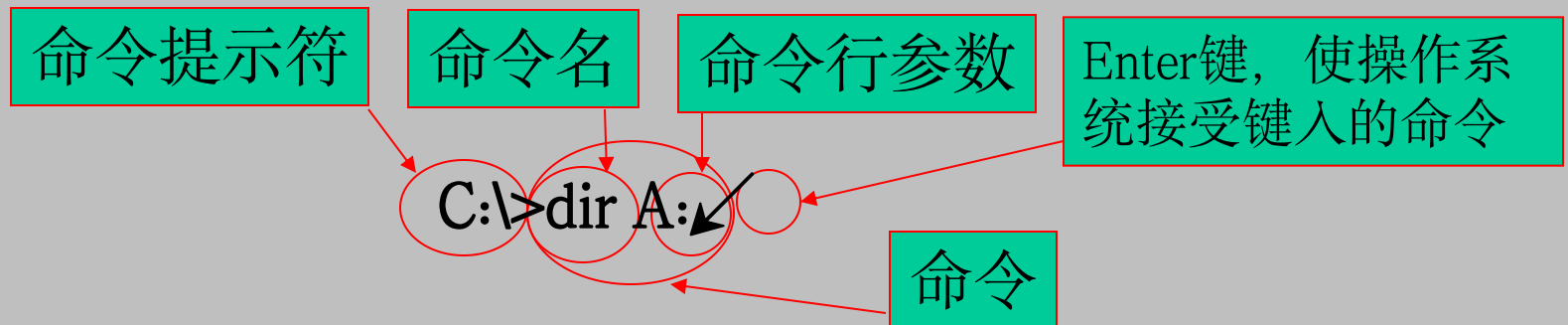
3,4,34,45,67,78

3,4,34,45,67,78

3,4,34,45,67,78

带参数的main()函数

- 命令行参数：
 - 命令：可执行程序。
 - 命令行参数：命令后所带的参数，由字符串组成，参数之间用空格分隔，参数的含义事先约定。
 - 命令行参数的作用：增加程序的灵活性、适应性。
 - 例如，在MS-DOS下：



其作用是列出并显示A盘文件目录。操作系统接受到该命令后，即调用名为“dir”的程序，并将整个命令行数据作为实参传给所调用的程序。

- **main()函数的参数**

C++语言允许main()函数带参数，其函数原型为：

```
int | void main(int argc 《,char *argv[] 《,char *envp[]》 》 );
```

或

```
int | void main(int argc 《,char**argv 《,char**env》 》 );
```

其中：

- arg是argument(参数)的缩写，c是count的缩写，v是vector的缩写；
- 第一个参数保存实际命令行所带的参数个数；
- 第二个参数是一个字符型指针数组或字符型二级指针，它的每个指针元素依次指向命令行所带的参数；
- 第三个参数也是一个字符型指针数组或字符型二级指针，它的每个元素指向当前运行系统的环境变量。该参数与操作系统有关，此处不做介绍。

- 例4.15 显示命令行参数，深入了解命令行参数的数据结构。

```
#include<iostream.h>
```

```
void main(int argc,char*argv[])
```

```
{ for(int i=0;i<argc;i++) cout<<argv[i]<<"\n"; }
```

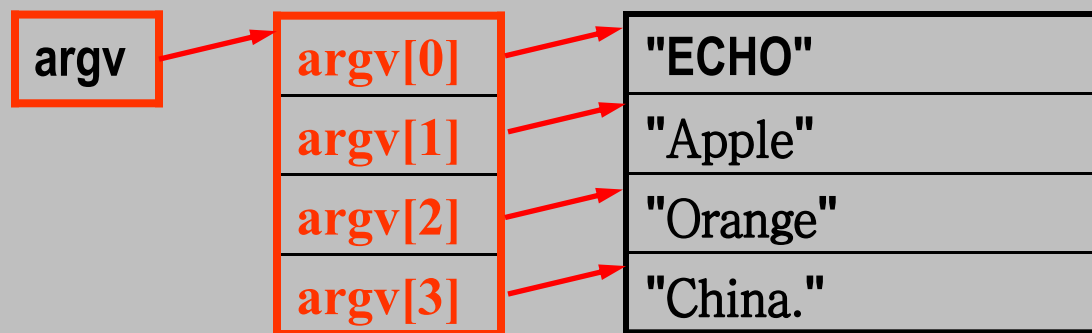
设程序经编译和连接后，生成的可执行文件名为ECHO.EXE。在DOS的环境下，键入命令：

C:\T>ECHO Apple Orange China. ✓

操作系统获得上述命令后，以空格为分隔标志，统计出字符串的个数传递给main()的形参argc，每个字符串的指针传给指针数组argv，如下图所示。

– 程序运行结果：

ECHO
Apple
Orange
China.



- 例4.16 用命令行参数计算任意个整数的和。

分析：按题意，可设计如下格式的命令：

```
sum aa bb ... xx
```

其中，sum为可执行程序名，aa、bb、...、xx分别为整数。

```
#include<iostream.h>
```

```
#include<stdlib.h>
```

```
void main(int argc,char*argv[])
```

```
{ if(argc<2) //若用户输入参数不足，提示命令格式
```

```
{ cout<<"Usage:\nsum aa bb cc ... xx\n"; return; }
```


```
for(int i=1,sum=0;i<argc;i++) sum+=atoi(argv[i]);
```

```
for(cout<<argv[1],i=2;i<argc;i++)
```

```
    cout<<'+'<<argv[i];
```

```
    cout<<'='<<sum<<endl;
```

```
}
```



stdlib.h库中定义的函数，
将数值串转换成整数。

返回值为指针的函数

- 函数返回值可为指针。定义返回指针值的函数的格式：

<类型标识符>*<函数名>(<形式参数表>){ <函数体> }

其中，“*”说明函数返回一个指针，该指针所指数据类型由类型标识符指定。如：

```
float* f(...){ ... }
```

- 例4.17 将输入的一个字符串按逆序输出。

分析：若输入的字符串为“ABCDEFGH”。使字符型指针变量p1和p2分别指向字符串的第1个字符和最后一个字符。

- ①将p1和p2所指向的字符互换(即将A与H互换);
- ②使p1指向后一字符，p2指向前一字符，即p1++和p2--;
- ③若p1小于p2，则重复①；否则说明字符串已结束交换。

```

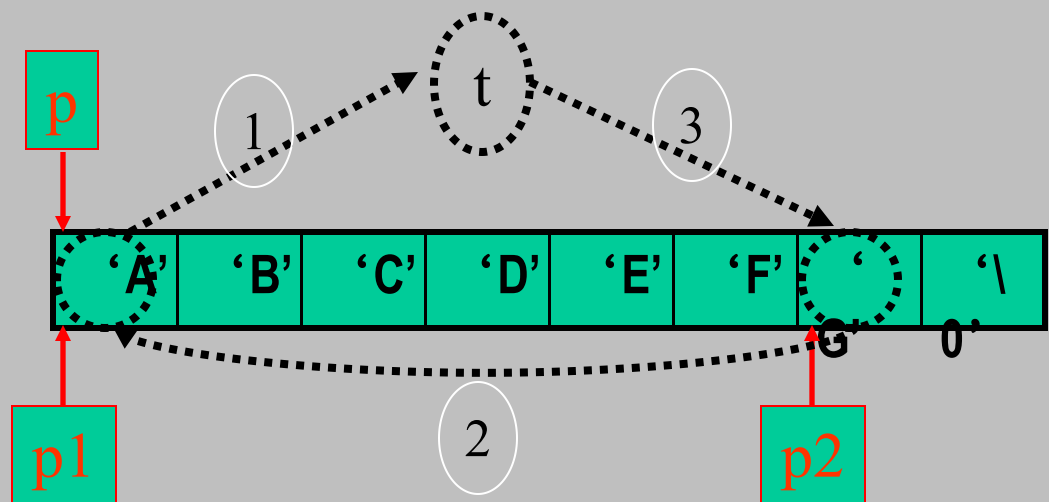
#include<iostream>
using namespace std;
char* invert(char*p)
{ char *p1=p,*p2=p,t;
  while(*p2) p2++;
  p2--;
  while(p1<p2)
  { t=*p1;*p1=*p2;*p2=t;p1++;p2--;}
  return p;
}

```

```

void main(void)
{ char s[]="ABCDEFGH";
  cout<<invert(s);//L
}

```



小技巧: invert()函数的返回值类型定义为“char*”更便于调用，但并非必须。若改为“void”类型亦可，则会使该函数的调用与输出分成两个语句，即L行语句需写成：

```

invert(s);
cout<<s<<'\n';

```

- 例4.18 自定义字符串复制函数和两个字符串连接的函数，并在main()函数中调用和测试。

```
#include<iostream.h>
```

```
char* strcpy(char* to,char* from)
{  for(char* p=to;*p++=*from++);
    return to;
}
```

```
char* strcat(char* to,char* from)
{  for(char*p=to;*p;p++);
    while(*p++=*from++);
    return to;
}
```

```
void main(void)
{  char s1[100]="大学",s2[200];
    strcpy(s2,"南通");
    cout<<"拼接后的字符串为: "<<strcat(s2,s1)<<"\n";
}
```

- 例4.19* 对函数返回的指针所指的单元赋值。

```
#include<iostream>
using namespace std;
int*f()
{ static int i;
  return &i;
}
```

```
void main(void)
{ *f()=200; //L
  cout<<*f()<<"\n";
  cout<<(*f()=400)<<"\n";
}
```

程序运行结果:
200 400

①L行表达式 “*f()=200”中，函数调用运算符的优先级最高，而赋值运算符的优先级最低。该表达式的计算顺序为：先调用函数f()，返回一个指向局部静态变量i的指针，然后将200赋给该指针所指的变量i。

有时，这种赋值方式可简化程序。

②函数只能返回全局变量或静态变量的指针，若返回局部变量的指针，则存在潜在危险。例如：

```
int* f()
{ int i; return &i; }
```

因执行return后，系统已收回变量i存储空间，所返回的指针所指内存单元已不能合法使用。

指向函数的指针

- 函数的指针：用该函数的名称表示。
- 函数指针变量：指向函数的指针变量。定义格式为：

<类型标识符>(*<变量名>)(《参数表》);

其中，类型标识符是所指函数返回值的类型，(《参数表》)是所指函数的参数表。例如：

```
float(*fp)(void);
```

即函数指针变量fp，所指函数无参且返回值类型为float。

- 说明：

①函数指针变量只能指向与该指针变量具有相同返回值类型和相同参数(个数及顺序一致)的任一函数。

②函数指针变量只能做赋值和关系运算，其他操作无意义。

③函数指针变量赋值后，可用该指针变量调用函数。

调用函数格式为:

(*<指针变量名>)(<实参表>)

或

<指针变量名>(<实参表>)

④指向函数的指针变量主要用作函数的参数。

• 例4.20 编程完成两个数的加、减、乘、除四则运算。

```
#include<iostream>
```

```
using namespace std;
```

```
float add(float x,float y){ return x+y; }
```

```
float sub(float x,float y){ return x-y; }
```

```
float mul(float x,float y){ return x*y; }
```

```
float div(float x,float y){ return x/y; }
```

```

void main(void)
{
    float a,b;
    float (*p)(float,float); char c;
    cout<<"输入数据格式: 操作数1 运算符 操作数2\n";
    cin>>a>>c>>b;
    switch(c)
    {
        case '+': p=add;break;
        case '-': p=sub;break;
        case '*': p=mul;break;
        case '/': p=div;break;
        default: p=NULL;
    }
    if(p) cout<<a<<c<<b<< '=' <<p(a,b)<<'\n';
    else cout<<"数据的格式不对!\n";
}

```

程序运行结果:

45.6+5.5 ✓

45.6+5.5=51.1

- 例4.21 已知一个一维数组的各个元素值，分别求出：数组的各个元素之和、最大元素值、下标为奇数的元素之和及各元素的平均值。

```
#include<iostream.h>
```

```
float sum(float*p,int n)
{  float sum=0;
   for(int i=0;i<n;i++) sum+=*p++;
   return sum;
}
```

```
float max(float*p,int n)
{  float m=*p++;
   for(int i=1;i<n;i++,p++)
       if(*p>m) m=*p;
   return m;
}
```



```
float oddsum(float*p,int n)
{   float sum=0;
    for(int i=1;i<n;i+=2,p+=2) sum+=*p;
    return sum;
}
```

```
float ave(float*p,int n){ return sum(p,n)/n;}
```

```
float process(//功能: 调用fp所指函数, 处理p所指数组
              float*p,           //指向数组的指针
              int n,             //p所指数组中元素的个数
              float(*fp)(float*,int)//函数指针变量
              )                  //返回值: 处理结果
{   return fp(p,n); }
```

```
void main(void)
{ float x[]={ 2.3f,4.5f,7.8f,34.6f,56.9f,77.f,
              3.34f,7.87f,20.0f};
  int n=sizeof(x)/sizeof(float);

  cout<<n<<"个元素之和="<<process(x,n,sum)
    <<"\n\t之最大值="<<process(x,n,max)
    <<"\n\t之平均值="<<process(x,n,ave)
    <<"\n\t之奇下标元素和="<<process(x+1,n,oddsum);
}
```

程序运行结果:

9个元素之和=214.31

之最大值=77

之平均值=23.8122

之奇下标元素和=123.97

- 函数指针数组：由同类型函数指针组成的数组。
- 一维函数指针数组的定义为：
《存储类型》类型(*数组名[数组大小])(《参数表》);
同样可定义多维函数指针数组，但一维的较常用。例如：
float(*funarr[20])(float*,int);

- 例4.22* 设计一个简单的计算器程序。

分析：可设计如下格式的命令：

```
calculate op1 op op2
```

其中，calculate为可执行程序名，op1、op、op2分别为第一操作数、运算符和第二操作数。程序如下。

```
#include<iostream.h>
```

```
#include<stdlib.h>
```

```
int add(int x,int y){ return x+y; }  
int sub(int x,int y){ return x-y; }  
int mul(int x,int y){ return x*y; }  
int divi(int x,int y){ return x/y; }
```

```
void OutResult(int a,int b,int(*f)(int,int))  
{ cout<<(*f)(a,b)<<endl; }
```

```
void main(int argc,char*argv[])  
{ int i,x,y,(*fun[4])(int,int)={add,sub,mul,divi};  
  if(argc!=4)  
  { cout<<"Usage: \ncalculate op1 op op2<CR>\n";  
    cout<<"example: \ncalculate 34*25<CR>\n";  
    return;  
  }
```

```
x=atoi(argv[1]); y=atoi(argv[3]);  
switch(argv[2][0])  
{ case '+':i=0; cout<<x<<'+'<<y<<'=';break;  
  case '-':i=1; cout<<x<<'-'<<y<<'=';break;  
  case '*':i=2; cout<<x<<'*'<<y<<'=';break;  
  case '/':i=3; cout<<x<<'/'<<y<<'=';break;  
  default:cout<<"Operation is error!\n"; return;  
}  
OutResult(x,y,fun[i]);  
}
```

若上述程序编译、链接后，生成的可执行程序名为calculate，则程序运行情况为：

calculate 1 + 2 ✓

1+2=3

(10) 作业

用指针的方法实现对二维数组的排序

4.5 new和delete运算符

- 如何保存一个班学生的数学成绩(班级人数事先无法确定)?
 - 预估班级人数的最大值(如100), 用符号常量MAX表示:
`#define MAX 100` //预估班级人数的最大值

```
...  
int n,a[MAX];  
cin>>n;      //输入班级实际人数  
if(n>MAX){ cout<<"班级人数太多! "; exit(2); }  
for(int i=0;i<n;i++) cin>>a[i];
```

不足: ①若 $n < MAX$, 则a数组的内存有闲置; ②若 $n > MAX$, 则程序无法运行, 尽管并非没有足够内存可用。

- 如何保存一个班学生的数学成绩(班级人数事先无法确定)?

– 根据程序运行时实际输入的班级人数，分配所需数组：

```
int n;
```

```
...
```

```
cin>>n;    //输入班级实际人数
```

```
int a[n];   //L
```

```
for(int i=0;i<n;i++) cin>>a[i];
```

编译时L行出错，原因是定义数组时下标表达式的值必须是编译时有确定值的常量。不过，用new运算符申请动态内存可以解决L行的问题，即L行改为：

```
int* a=new int[n];
```

```
if(a==0){ cout<<"没有足够内存! "; exit(2); }
```

- 用申请分配动态内存的方法，较好解决了班级人数事先无法确定时的内存分配问题。在C++中，new和delete运算符分别用于为指针变量申请分配动态内存和收回指针所指动态内存。

new运算符

- 格式:

`type* p=new type; //格式一`

或

`type* p=new type(value); //格式二`

或

`type* p=new type[size]; //格式三`

其中:

- type是一个已定义的类型名。
- 格式一的功能: 申请分配一个type类型的存储单元, 并把所分存储单元的指针赋给p。申请不成功时, p的值为0。
- 格式二除了完成格式一的功能外, 还将value的值作为p所指存储单元的初值。
- 格式三是分配指定类型的一维数组, size为元素个数。

- 举例:

```
float *fp1,*fp2; char *cp;  
fp1=new float(2.5f);    //L1格式二  
fp2=new float[10];      //L2格式三  
cp=new char;            //L3格式一  
*cp='A';                //L4  
if(fp2==0)              //L5  
{ cout<<"申请动态内存失败!\n"; exit(3); }  
for(int i=0;i<10;i++) fp2[i]=*cp+i; //L6
```

- 注意:

- 指针变量指向的用new申请分配的动态内存所代表的变量的初值是随机的, 因此**在使用前必须初始化**。
- 实际编程时, 用new申请分配动态内存后, 必须判断申请是否成功, 若new运算符申请的结果为0, 表示申请动态内存失败, 此时应终止程序执行或出错处理。如程序行L5。

- 为数组或结构体申请分配动态内存时，不能在分配内存的同时初始化。例如，下列语句是错误的。

```
int *pi=new int[10](1,2,3,4,5,6,7,8,9);
```

- 申请分配动态多维数组的一般格式为：

```
type(*p)[c2][c3]...[cn]
```

```
=new type[size][c2][c3]...[cn];
```

其中，type是一个已定义的类型名,size为整型表达式，c₂、...、c_n必须为整型常量表达式。例如：

```
int n;
```

```
cin>>n;
```

```
float(*p)[20]=new float[n][20];
```

使得行指针变量p指向n × 20的动态二维数组。此后即可像访问普通二维数组一样，用p访问该动态二维数组。例如：

```
p[0][0]=10.0f; // 或*(*(p+0)+0)=10.0f;
```

- 有时会出现用new运算符申请分配所得的指针类型与所赋值指针变量类型不一致的情况，例如：

```
float(*pt)[20];
```

```
pt = new float[20];
```

类型为： **float***

因指针类型不一致而编译出错。有两种处理方法。一是做强制类型转换：

```
pt = (float(*)[20])new float[20];
```

二是调整new运算：

```
pt = new float[1][20];
```

方法一会在编译时产生额外代码，运行时占用额外时间，且降低了程序的可读性；方法二则无上述不足。再如：

```
int *p3 = new int[10][20];
```

是错误的。正确的表示为：

```
int *p3 = (int*)new int[10][20];
```

或 `int *p3 = new int[10*20];`

delete运算符

- delete运算符用来将动态分配到的内存还给系统。格式：

`delete pointer;`

或

`delete []pointer;`

或

`delete [size]pointer;`

其中：

- pointer的值应由new分配的内存空间的指针。
- 第一种格式是把pointer所指向的内存空间还给系统；
- 第二、三种格式是把pointer所指向的动态数组的内存空间还给系统，可用size指明动态数组的元素个数，也可不用。

- delete运算符使用注意:

- 用new运算符分配的内存的指针值必须保存, 以使用delete运算符归还给系统, 否则会出错。例如:

```
float *fp,i;
```

```
fp=new float(24.5f);
```

```
fp=&i;          //L1
```

```
delete fp;
```

因L1行改变了fp的值, 在使用delete时, fp已不再指向动态分配的存储空间, 导致程序在运行过程中出错。

- 用delete运算符及时释放用new运算符申请分配但已不再使用的动态内存是程序员的职责。及时释放已不再使用的动态内存可提高内存利用率; 否则, 这部分内存将不会自动归还系统, 严重时系统会因内存短缺导致崩溃。

- 释放动态分配的多维数组，可用delete的第二或第三种格式，推荐使用第二种格式。例如：

```
int(*p1)[100]=new int[30][100];
```

```
int(*p2)[100][200]=new int[40][100][200];
```

...

```
delete []p1; //L或delete [30]p1;
```

```
delete []p2; // 或delete [40]p2;
```

但若把L行写为：

```
delete p1;
```

这时仅释放二维数组的第0行所占用的存储空间。

- 动态内存一旦释放，就不能再对其赋值。例如：

```
float *p=new float;
```

...

```
delete p;
```

```
*p=5;
```

有潜在危险： p的值虽未改变，但因其所指内存已被回收，此后再对p所指内存赋值将是非法的和危险的。

应用举例

- 例4.23 定义全班学生学习成绩的结构体数组，每个元素包括：姓名、学号、C++成绩、英语成绩和这两门课程的平均成绩(通过计算得到)。设计四个函数分别完成：
 - (1)全班同学数据的输入
 - (2)求每位同学的平均成绩
 - (3)按平均成绩的升序排序
 - (4)输出全班成绩表

```
#include <iostream.h>
struct Student //定义描述学生数据的全局结构体类型
{ char name[9]; //姓名
  unsigned No; //学号
  float cpp, //C++成绩
        eng, //英语成绩
        ave; //平均成绩
};
```



```
void Input(    //功能: 输入全班学生数据
    Student *p, //全班学生数据的动态结构体数组的指针
    int n)    //班级人数
{ for(int i=0;i<n;i++,p++)
    { cout<<"输入第"<<i+1
        <<"个学生数据(姓名 学号 C++ 英语成绩):";
        cin>>(*p).name>>(*p).No>>(*p).cpp>>(*p).eng;
    }
}

void Average(Student *p,int n)
{ for(int i=0;i<n;i++,p++)
    p->ave=(p->cpp+p->eng)/2;
}
```

指针与结构体:

设有说明:

Student s,*p=&s;

则用指针p访问s中成员的方法有两种:

①(*p).No=1001;

②p->No=1001;

后者是前者的简写形式。

```
void Output(Student *p,int n)
{ cout<<"\t  全班成绩表\n"
  <<"姓名\t学号\tC++\t英语\t平均\n";
  for(int i=0;i<n;i++,p++)
    cout<<p->name<<"\t"<<p->No<<"\t"<<p->cpp<<"\t"
      <<p->eng<<"\t"<<p->ave<<"\n";
}
```

```
void Sort(Student*p,int n)
{ for(int i=0;i<n-1;i++)
  {   for(int k=i,j=i+1;j<n;j++)
      if(p[j].ave<p[k].ave) k=j;
      if(k!=i){ Student t=p[i];p[i]=p[k];p[k]=t; }
  }
}
```

```
void main(void)
{ int num; //班级人数
  Student *p; //存放学生数据的动态结构体数组的指针

  cout<<"输入班级人数:";
  cin>>num;
  p=new Student[num];
  if(!p){ cout<<"申请动态内存失败!\n"; return;}

  Input(p,num); //输入全班学生数据
  Average(p,num); //计算全班学生平均分
  Sort(p,num); //按平均分升序排序全班学生数据
  Output(p,num); //输出全班学生数据
  delete []p; //释放动态内存
}
```

- 例4.24 设某班学生人数为 n ，课程门数为 m (m 、 n 在程序运行时由用户输入)，试输入该班全体学生的所有课程的成绩，计算每门课程的平均成绩，然后输出该班全体学生的所有课程的成绩和每门课程的平均成绩。

[分析]

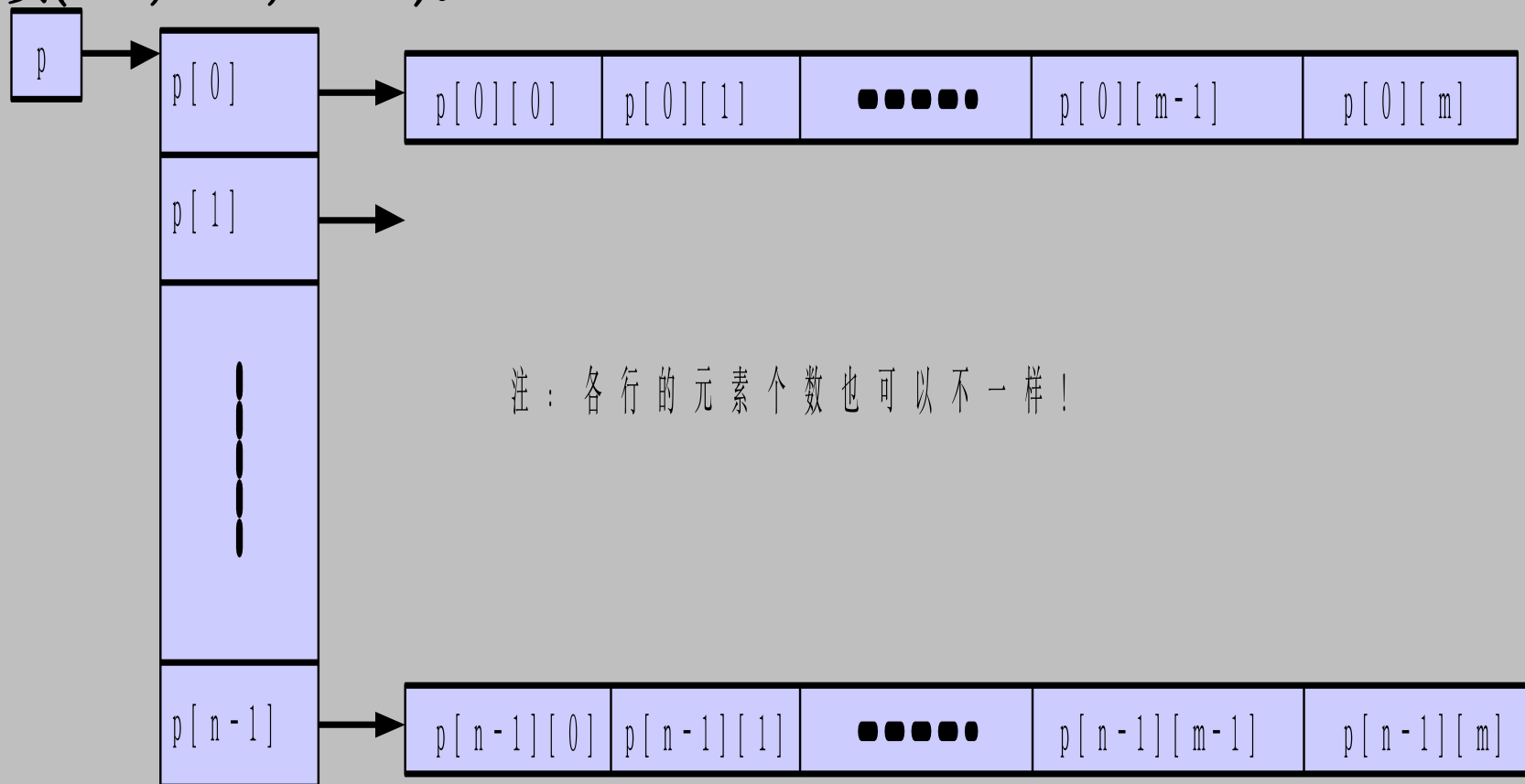
- 因 m 、 n 均在程序运行时由用户输入，故用：

```
float (*p)[m]=new float[n][m];
```

申请动态二维数组则会出现编译错误。原因是 m 不为常量。

- 现用二级指针、动态一维指针数组和动态一维数组建立行、列可变的动态二维数组。

- 方法1: 如图所示。p为二级指针，指向动态一维指针数组(元素为 $p[0] \sim p[n-1]$)；指针 $p[i]$ 再指向动态一维数组(元素为 $p[i][0] \sim p[i][m]$)。约定： $p[i][0] \sim p[i][m-1]$ 用于存放第i个学生的m门课程成绩， $p[i][m]$ 用于存放第i个学生的m门课程的平均成绩($i=0, \dots, n-1$)。



```
#include <iostream.h>
```

```
void main(void)
```

```
{ int m,n,i,j;
```

```
float **p,*q,sum;
```

```
cout<<"输入n,m:";
```

```
cin>>n>>m;
```

```
p=new float*[n];
```

```
if(!p){ cout<<"申请动态内存失败!\n"; return;}
```

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

```
{ cout<<"Enter student #"<<i+1<<"Score.\n";
```

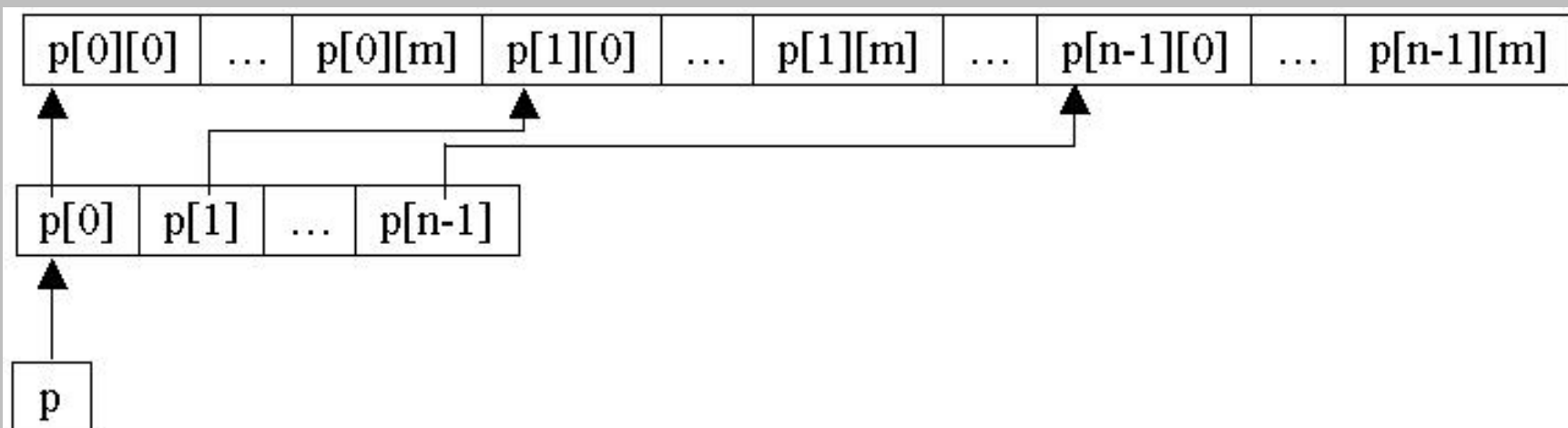
```
p[i]=new float[m+1];
```

```
if(!p[i]){ cout<<"申请动态内存失败!\n";return; }  
for(sum=0,j=0;j<m;j++)  
{ cin>>p[i][j]; sum+=p[i][j]; }  
p[i][m]=sum/m;  
}
```

```
for(i=0;i<n;i++)  
{ cout<<i+1;  
  for(j=0,q=p[i];j<m+1;j++) cout<<'\\t'<<q[j];  
  cout<<endl;  
}
```

```
for(i=0;i<n;i++) delete p[i];  
delete p;  
}
```

- 方法2: 用动态指针数组将动态一维数组转换成动态二维数组, 如图所示。先为二级指针p申请动态一维指针数组(元素为 $p[0] \sim p[n-1]$); 后使 $p[0]$ 指向含有 $n \times (m+1)$ 个元素的动态一维数组, 再用指针 $p[1] \sim p[n-1]$ 将 $p[0]$ 所指的动态一维数组分成 n 个等长的一维数组。这样建立的行、列可变的动态二维数组的特点是, 行与行之间的存储空间连续, 每行的元素个数相同。约定, $p[i][0] \sim p[i][m-1]$ 用于存放第 i 个学生的 m 门成绩, $p[i][m]$ 用于存放第 i 个学生的 m 门课程的平均成绩($i=0, \dots, n-1$)。




```
#include <iostream.h>
```

```
void main(void)
```

```
{ int m,n,i,j;
```

```
float **p,*q,sum;
```

```
cout<<"输入n,m:";
```

```
cin>>n>>m;
```

```
p=new float*[n];      //申请动态一维指针数组
```

```
if(!p){ cout<<"未申请到动态内存!\n"; return; }
```

```
p[0]=new float[n*(m+1)]; //申请动态一维数组
```

```
if(!p[0]){ cout<<"未申请到动态内存!\n"; return; }
```

```
//用指针数组将动态一维数组转换成动态二维数组
```

```
for(i=1;i<n;i++) p[i]=p[i-1]+m+1;
```

```
for(i=0;i<n;i++)
{ cout<<"Enter student #"<<i+1<<"Score.\n";
  for(sum=0,j=0;j<m;j++)
  { cin>>p[i][j]; sum+=p[i][j]; }
  p[i][m]=sum/m;
}
for(i=0;i<n;i++)
{ cout<<i+1;
  for(j=0,q=p[i];j<m+1;j++) cout<<"\t"<<q[j];
  cout<<endl;
}
delete p[0];
delete p;
}
```

4.6 引用和函数

引用的概念

- 下面的写法定义了一个引用，并将其初始化为引用某个变量。

类型名 & 引用名 = 某变量名;

```
int n = 4;
```

```
int & r = n;    // r引用了 n, r的类型是 int &
```

- 某个变量的引用，等价于这个变量，相当于该变量的一个别名。

4.6 引用和函数

引用的概念

```
int n = 4;  
int & r = n;  
r = 4;  
cout << r;    //输出4  
cout << n;    //输出 4  
n = 5;  
cout << r;    //输出5
```

4.6 引用和函数

引用的概念

- 定义引用时一定要将其初始化成引用某个变量。
- 初始化后，它就一直引用该变量，不会再引用别的变量了。
- 引用只能引用变量，不能引用常量和表达式。

4.6 引用和函数

- 关于引用类型变量的几点说明：
 - 定义引用类型变量时，必须用同类型的变量初始化。如：

float x;

int &px=x; //错：因px和x的类型不同

- 引用类型变量的初始化值不能是常数。如：

int &rf1=5; //错：因5不是变量

但如下的说明是正确的：

const int &rf2=5; //告诉系统，rf2是常数5的引用

- 也可定义指针变量的引用，如：

int a,*p=&a;

int* &rp=p;

- 可以声明引用类型变量的引用。例如：

int i,&ri1=i;

int &ri2=ri1;

- 可用动态内存的指针初始化指针变量的引用。例如：

```
float* &rp=new float[10]; //L1
```

```
for(int i=0;i<10;i++)cout<<(rp[i]=i+20.0f)<<'t';
```

```
delete rp;
```

- 可用动态分配的内存初始化一个引用变量。例如：

```
float &rf=*new float(20.0f); //L2
```

```
cout<<rf;
```

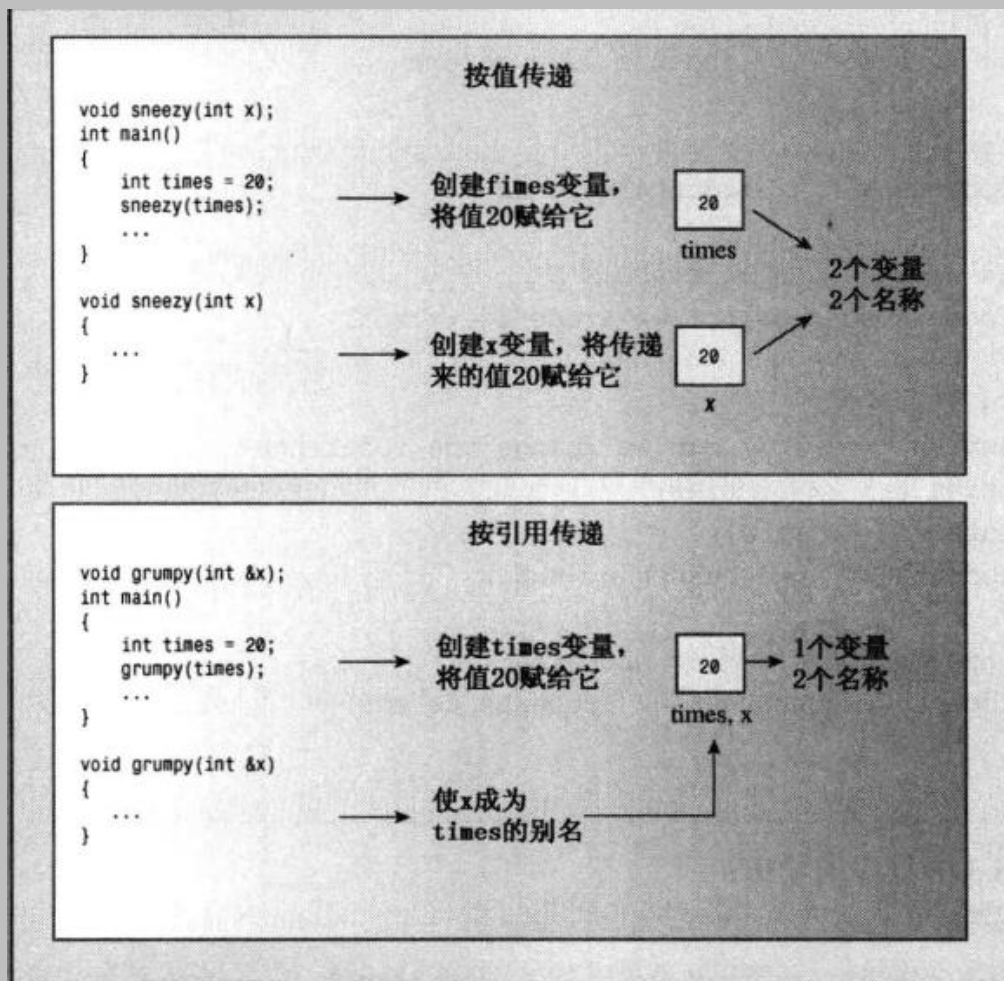
```
delete &rf;
```

因new运算的结果是指针，故在其前加“*”，取其所指的变量。delete运算的操作数是指针，故用delete释放动态内存时，要在引用类型变量rf前加取指针运算符“&”。

- 尽管别名与它所关联的变量的值始终保持一致，但应强调，别名也是变量，与它所关联的变量并不是同一个变量。
- “&”的含义：按位与运算符“&”（二元）；取指针运算符“&”（一元）；引用运算符，用于定义引用类型变量。

引用和函数

- 引用主要用作函数参数或函数的返回值。
- 引用类型变量做函数的参数



引用和函数

- 例4.25 编写一个程序，交换两个整数并输出。

```
#include<iostream>
```

```
using namespace std;
```

```
void swap1(int*p1,int*p2){ int t=*p1;*p1=*p2;*p2=t;}
```

```
void swap2(int&p1,int&p2)
```

```
{ int t=p1;p1=p2;p2=t;}
```

```
void main(void)
```

```
{ int x=300,y=400,a=100,b=200;
```

```
swap1(&x,&y); cout<<"x="<<x<<",y="<<y<<"\n";
```

```
swap2(a,b); cout<<"a="<<a<<",a="<<b<<"\n";
```

```
}
```

引用类型做函数参数时，形参是实参的别名。此处p1是a的别名，p2是b的别名。

– 程序执行时，程序输出为：

x=400 y=300

a=200 b=100

– 函数swap1()与swap2()都能交换两个数，但有区别：

①函数的定义和调用形式不同。对于引用类型的形参，函数调用时其实参使用变量名，形参是实参的别名，这种实参向形参传递数据的方式称为引用传递。而对于指针类型的形参，函数调用时其实参必须是变量的指针。

②访问被引用的变量的方式不同。在swap1()中要用取指针运算符“*”来取指针所指变量；而在swap2()中，使用别名(形参名)来引用所关联的变量(实参)。由此可见，引用也具有突破变量作用域限制的能力。

• 小结：用引用型的参数比用指针型的参数更能增加程序的可读性和编程的方便性。准确理解和正确使用引用比指针要容易。因此，实际编程时，引用做函数参数非常普遍。

- 例4.26 用指针变量的引用实现字符串指针的交换。

```
#include <iostream>
using namespace std;
void swap(char*&x,char*&y)
{ char *t=x;x=y;y=t; }

void main(void)
{ char *a="张珊",*b="李是";
  swap(a,b);
  cout<<a<<','<<b<<endl;
}
```

运行结果:
李是,张珊

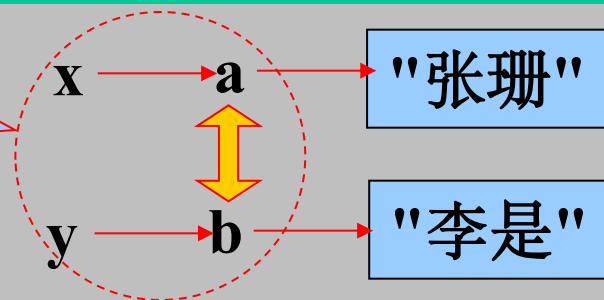
swap()函数的功能是交换main()函数中的指针变量a和b的值。

若使用二级指针实现两个字符串交换，程序如下：

```
#include <iostream.h>

void swap(char**x,char**y)
{ char *t=*x;*x=*y;*y=t; }

void main(void)
{ char *a="张珊",*b="李是";
  swap(&a,&b);
  cout<<a<<','<<b<<endl;
}
```



函数的返回值为引用类型*

- 若函数的返回值类型定义为引用类型，则该函数返回的一定是某个变量的别名，即相当于返回了一个变量，故可对其返回值做赋值操作。
- 例4.27 函数返回值为引用类型。

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

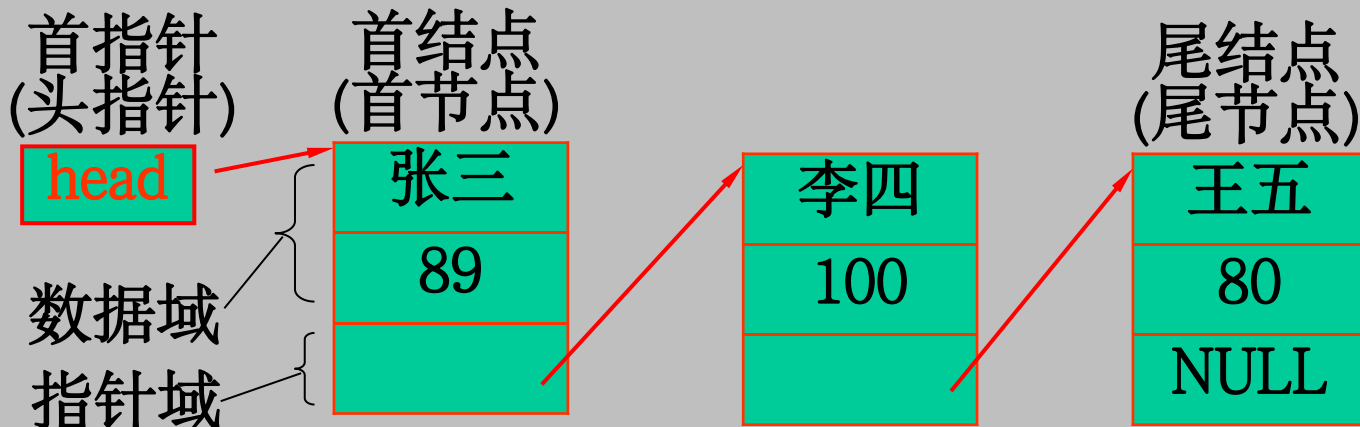
```
int& f(void)
{ static int count;
  return ++count;
}

void main(void)
{ f()=100;
  cout<<f()<<"\n";
}
```

- 程序运行结果：
101
- 函数f返回静态局部变量count的引用。
- **f()=100**：因赋值运算的优先级低于函数调用，故先调用函数f()，返回count的引用，再给count赋值100。
- 注意：函数不能返回自动存储类型或寄存器类型的局部变量的引用。

4.7 单向链表及其应用

- 若程序所处理的数据的增减是逐个进行的，则数据存储空间的静态分配和一次性动态分配难以满足要求。
- 单向链表是满足上述要求的一种数据结构，其结构如下图。
- 单向链表由若干同类型结点串接而成。每个结点包含两个域，即数据域和指针域next。数据域描述某一问题所需的数据，如描述学生成绩的数据包括姓名和成绩。指针域指向下一个结点。
- 单向链表结点串接原则：当前结点通过指针域指向下一结点；尾结点的指针域为空，表示链表结束。
- 环形链表：单向链表的尾结点的指针域指向首结点。



- 使用单向链表编程的一般步骤：
 - 定义结点的数据类型，如图中结点数据类型定义为：

```
struct node{  
    char name[20]; //数据域  
    int score;  
    node *next;  //指针域  
};
```

其中，next为指向这种结构体类型的指针，存放指向下一个结点的指针。

- 创建单向链表
- 根据解题需要对链表做有关操作，如插入或删除一个结点，链表的排序、查找、输出、释放等。

单向链表的建立和基本操作

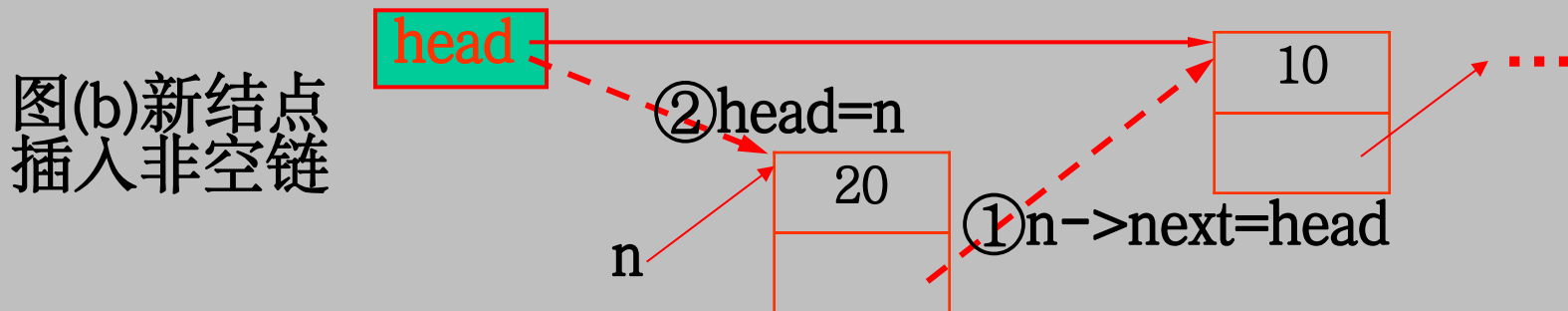
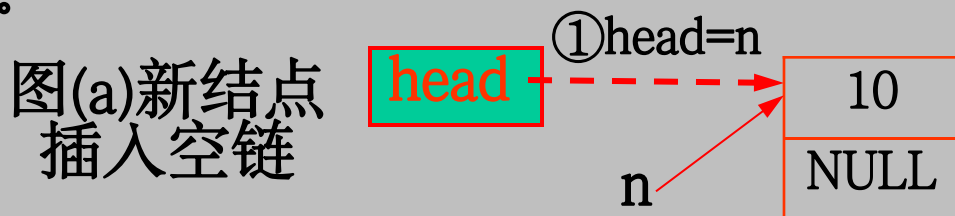
- 单向链表的建立和基本操作：
 - 建立一条无序链表
 - 建立一条有序链表
 - 遍历链表/输出链表上各结点的数据
 - 插入/删除链表上的某个结点
 - 释放链表各结点占用的内存空间
- 为突出重点且不失一般性，设每个结点的数据域只含一个整数。
结点类型为：

```
struct node{  
    int data;  
    node *next;  
};
```

1. 建立链表

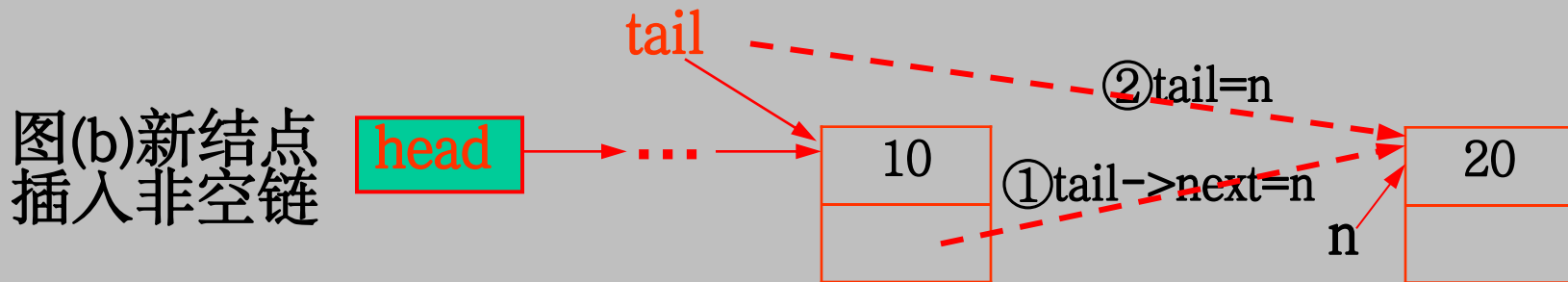
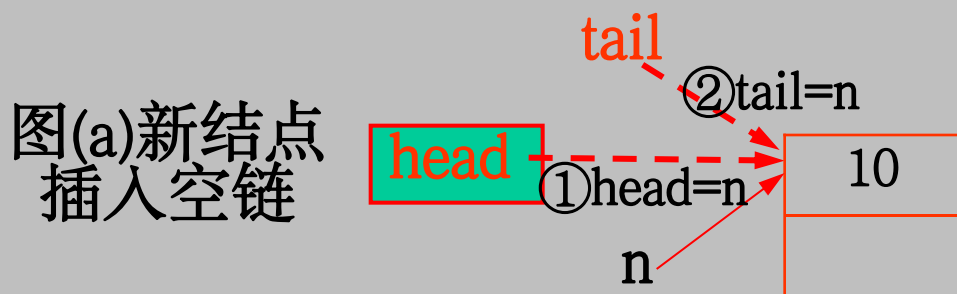
单向链表的建立是通过向空链表不断插入新结点实现的。按新结点插入单向链表的位置分，有首结点插入法、尾结点插入法和有序插入法，其中前两种方法产生的链表是无序，最后一种方法产生的链表是有序的。

(1)首结点插入法：设指针n指向新结点， head指向单向链表首结点。新结点插入链表分两种情况：一是空链表，如下图(a)所示；二是非空链表，如下图(b)所示。图中虚线部分是在原有基础上所作的操作。




```
node* Head_Create()
{
    node *head,*n;
    int a;
    head=NULL;           //空链
    cout<<"首结点插入法产生链表， 输入数据(-1结束): ";
    for(cin>>a; a!=-1; cin>>a)
    {
        n=new node;
        n->data=a;
        if(!head) head=n,n->next=NULL; //如图(a)
        else n->next=head,head=n;      //如图(b)
    }
    return head;
}
```

(2)尾结点插入法: 设指针 n 指向新结点, $head$ 指向单向链表首结点, $tail$ 指向单向链表尾结点。新结点插入链表分两种情况: 一是空链表, 如下图(a)所示; 二是非空链表, 如下图(b)所示。图中虚线部分是在原有基础上所作的操作。



```
node* Tail_Create()
{
    node *head,*n,*tail;
    int a;
    head=NULL;                //空链
    cout<<"尾结点插入法产生链表，输入数据(-1结束): ";
    for(cin>>a; a!=-1; cin>>a)
    {
        n=new node;
        n->data=a;
        if(!head) tail=head=n;    //如图(a)
        else tail->next=n,tail=n;  //如图(b)
    }
    if(head) tail->next=NULL;
    return head;
}
```

(3)结点的有序插入法： 设指针n指向新结点， head指向单向链表首结点。

```
node* Sort_Create(void)
```

```
{  node *head,*n;
```

```
    int a;
```

```
    head=NULL;           //空链
```

```
    cout<<"有序插入法产生链表， 输入数据(-1结束): ";
```

```
    for(cin>>a; a!=-1; cin>>a)
```

```
    {  n=new node;
```

```
        n->data=a;
```

```
        head=Insert(head,n);//将新结点n插入有序链表后，
```

```
    }           //使链表仍保持有序。
```

```
    return head;
```

```
}
```

2.将新结点插入有序链表，并保持链表有序

node* Insert(node*head,node*n) //升序插入

```
{ node *p1,*p2;
```

```
    if(head==NULL)           //若为空链
```

```
    { head=n; n->next=NULL;
```

```
      return head;
```

```
    }
```

```
    if(head->data>=n->data)    //新结点插入首结点前
```

```
    { n->next=head; head=n;
```

```
      return head;
```

```
    }
```

```
    p2=p1=head;
```

```
    while(p2->next&& p2->data<n->data)
```

```
    { p1=p2;p2=p2->next; }
```

```

if(p2->data<n->data)    //新结点插入尾结点后
{ p2->next=n; n->next=NULL; }
else//新结点插入p1和p2所指结点之间，如下图所示
{ n->next=p2; p1->next=n; }
return head;
}

```

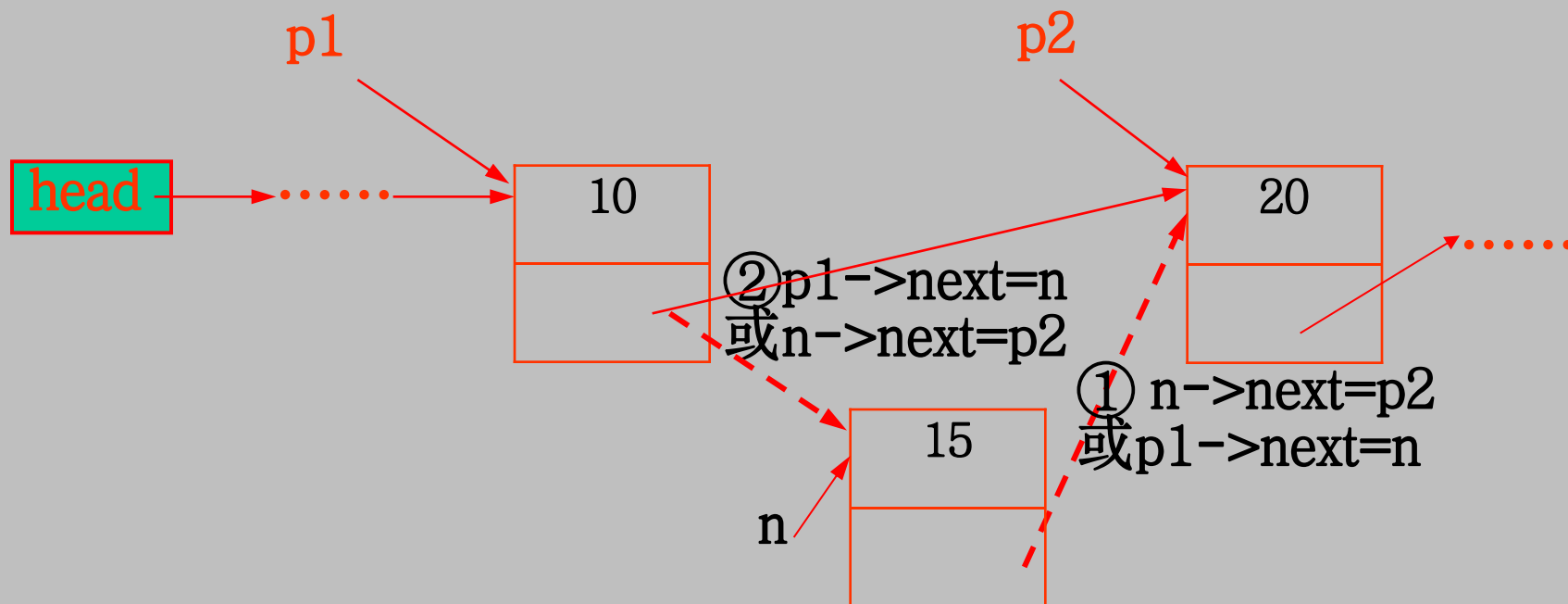


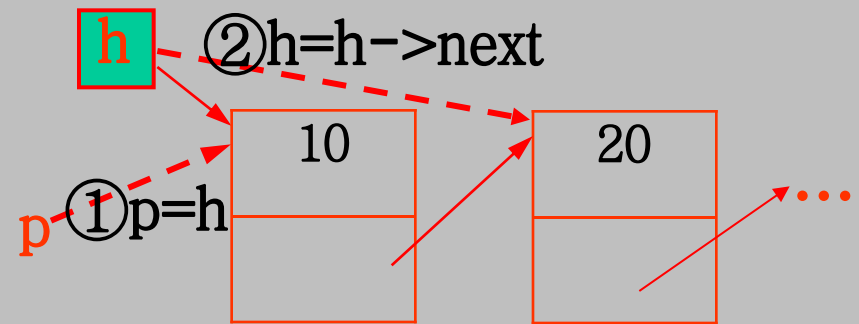
图 新结点插入p1所指结点后， p2所指结点前

3. 输出链表上各个结点的值

```
void Print(const node *head)
{ cout<<"链表上各结点的数据为: \n";
  while(head)
  { cout<<head->data<<"\t";
    head=head->next;
  }
  cout<<"\n";
}
```

4. 释放链表的结点空间

```
void deletechain(node *h)
{ node *p;
  while(h)
  { p=h; h=h->next; delete p; }
}
```



5. 删除链表上具有指定值的一个结点

删除链表上某个结点的步骤：先找到要删除的结点，然后删除已找到的结点。

```
node* Delete_one_node(node*head,int num)
{
    node*p1,*p2;
    if(head==NULL)
    {
        cout<<"链表为空，无结点可删!\n";
        return NULL;
    }
    if(head->data==num) //正好是首结点
    {
        p1=head;
        head=head->next;
        delete p1;
        cout<<"删除了一个结点!\n";
    }
}
```



```

else
{
    p1=head;p2=p1->next;
    while(p2->data!=num&& p2->next!=NULL)
    {
        p1=p2; p2=p2->next;
    }
    if(p2->data==num)
    {
        p1->next=p2->next;
        delete p2;
        cout<<"删除了一个结点!\n";
    }
    else cout<<"链表上没有找到要删除的结点!\n";
}
return head;
}

```

在链表上查找结点。
注意循环结束条件：
①p2所指结点是要查找的结点②p2指向链表的尾结点且不是要查找的结点。

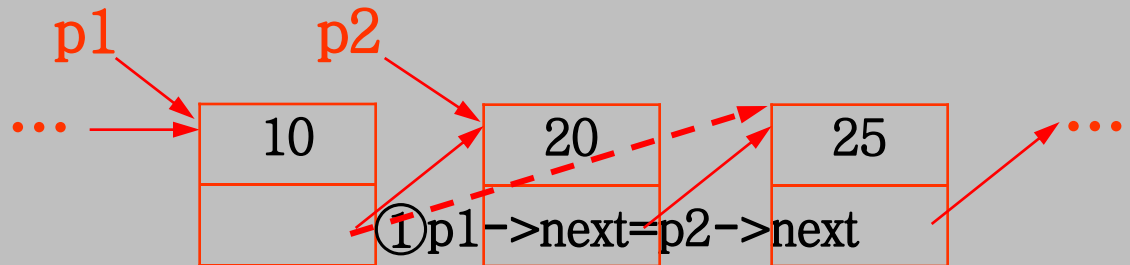


图 删除p2所指结点

6. 主函数

```
void main(void)
{
    node *head; int num;
    head=Head_Create(); //头结点插入法产生链表
    Print(head);        //输出链表上的各结点值
    cout<<"输入要删除结点上的整数: \n";
    cin>>num;
    head=Delete_one_node(head,num); //删除指定结点
    Print(head);        //输出链表上的各结点值
    deletechain(head);
    head=Tail_Create(); //尾结点插入法产生链表
    Print(head);        //输出链表上的各结点值
    deletechain(head); //释放链表各结点占用内存
    head=Sort_Create(); //产生一条有序链表
    Print(head);        //输出链表上的各结点值
    deletechain(head);
}
```

4.8 其他

const型变量

- const型变量：定义时用const修饰的变量。
- 定义const型的常值变量。例如：

```
const int MaxLine=1000;
```

```
const float Pi=3.1415926;
```

用const修饰的常值变量必须在定义时初始化，定义后不得改变。

从使用效果看，用const型常值变量与用编译预处理命令define定义的符号常量是相同的。但两者有区别：

- ①处理方式不一样。符号常量是在编译之前，由编译预处理程序处理；而const型常值变量由编译程序处理，因此在调试程序时，可用调试工具查看其值。
- ②作用域不一样。const型常值变量的作用域，与一般变量的作用域相同；符号常量的作用域始于定义，止于文件结束之前。

- 定义const型指针:

- 将const放在指针变量的类型之前。例如:

```
float x,y;
```

```
const float* Pf=&x;
```

表示不允许通过指针变量Pf改变其所指变量的值，但可改变指针变量Pf的指向。例如:

```
*Pf=25;      //错误
```

```
Pf=&y;        //正确
```

- 将const放在指针变量的“*”之后。例如:

```
int n,i;
```

```
int *const pn=&n;
```

表示指针变量pn的值是常量，定义时必须初始化。即不能改变pn的值，但可以改变pn所指变量的值。例如:

```
*pn=25;      //正确
```

```
pn=&i;        //错误
```

- 将一个const放在指针变量的类型之前，再将另一个const放在指针变量的“*”之后。例如：

```
int j,k;
```

```
const int *const pp=&j;
```

表明指针变量pp的值是常量，所指变量的值也不允许通过pp指针改变。例如，以下用法是错误的：

```
*pp=25; pp=&k;
```

用这种形式定义的指针变量，在定义时必须赋初值。

注：①引用与指针相似，故以上用法也适用于引用型变量；

②const修饰指针后，编译程序可做相应语法检查，提高了使用指针的安全性。

③const型指针主要用做函数的参数，以限制在函数体内对指针变量的值的修改，或对指针所指变量的修改。

例4.28 `const` 型指针形参用于防止函数修改它的参数。

```
#include<iostream.h>
```

```
/*将参数s所指字符串中的空格转换成连字符并输出 */
```

```
void sp_to_dash(const char *s)
```

```
{ while(*s)
```

```
{ if(*s==' ') *s='_'; //L
```

```
cout<<*s++;
```

```
}
```

```
}
```

```
void main(void)
```

```
{ sp_to_dash("Hello !"); }
```

- 编译时，L行出错(error C2166: l-value specifies const object)。因函数 `sp_to_dash()` 的形参是 `const` 修饰的指针，在函数体内不能改变它所指的数据，而程序L行的语句 `*s='_';` 违约。

- 根据 `sp_to_dash()` 函数的功能和 `const` 型指针的限定，可将 `sp_to_dash()` 函数改写成：

```
void sp_to_dash(const char *s)
```

```
{ while(*s)
```

```
{ if(*s==' ') cout<< '_';
```

```
else cout<<*s;
```

```
s++;
```

```
}
```

```
}
```

类型名重定义语句 typedef

- 类型名重定义可增加程序的可读性和可移植性。格式为：

`typedef <类型> <标识符1> 《,<标识符2>...》;`

其中，类型为基本类型名(如int)，或自定义类型名(如结构体名等)，或是已重定义的类型名。例如：

```
typedef int LENGTH;          //A
```

```
typedef char* STRING;        //B
```

```
typedef int VEC[50];         //C
```

```
typedef struct node{          //D
```

```
    char*word;
```

```
    int count;
```

```
    struct node *left,*right;
```

```
}TREENODE,*TREEPTR;
```

```
typedef int(*FP)(void);      //E
```

- 经类型名重定义后的标识符可做类型说明符或强制类型转换的类型标识符。例如：

- A行指定用LENGTH代替int，可用LENGTH来说明变量：

LENGTH i,j; //等价于： int i,j;

- B行指定用STRING代替char*，可用STRING说明变量：

STRING s1,s2; //等价于： char *s1,*s2;

- C行将VEC指定为包含50个元素的一维整型数组，可用VEC来说明变量：

VEC x,y; //等价于： int x[50],y[50];

- D行将TREENODE定义为node类型，将TREEPTR定义为node类型的指针。可用这两个类型名来定义变量：

TREENODE pp,p1[10];//等价于： node pp,p1[10];

TREEPTR p; //等价于： node*p;

- E行把FP定义为指向函数的指针，可用FP说明变量：

FP f; //等价于： int(*f)(void);

- 重定义类型名的步骤:

- ①用定义变量的方法写出变量说明。例如, `int(*f)(void)`。

- ②将变量名换为新的类型标识符。例如, `int(*FP)(void)`。

- ③在前面加上typedef。例如, `typedef int(*FP)(void)`。

- ④用新类型标识符定义变量。例如, `FP f1,f2;.`

- 关于类型名重定义的几点说明:

- ①typedef只能重定义类型标识符, 不能定义变量。

- ②用typedef重定义的新类型名说明变量可增加程序的可读性和可理解性。例如:

- `typedef int LENGTH,WIDTH,SIZE;`

- `LENGTH l1,l2,l3; //变量l1、 l2、 l3表示长度`

- `WIDTH w1,w2,w3; //变量w1、 w2、 w3表示宽度`

- `SIZE s1,s2,s3; //变量s1、 s2、 s3表示大小`

- ③typedef只能重定义已有类型, 并不能定义新类型。

void型指针*

- **void型指针(无类型指针)**, 可接收任意类型的指针值; 但若将它赋给其它类型的指针变量时, 应做强制类型转换。例如:

```
int *ip=new int,*ip1;
```

```
float *fp=new float,*fp1;
```

```
void *p1,*p2;
```

```
p1=ip;      //正确
```

```
p2=p1;      //正确
```

```
ip1=p1;      //错误, 应写成: ip1=(int*)p1;
```

```
fp1=(float*)p2; //正确
```

- void型指针主要用于编写通用函数。

例4.30** 下列程序是用void型指针编写的通用排序程序，可分别对整数、实数、字符串等进行排序。

```
#include<iostream.h>
#include<string.h>
```

为便于函数指针调用不同类型数据的比较函数，此处将所有比较函数的两个形参都说明为void型指针，外加const修饰是防止对指针所指数据的修改。

```
int CmpI(const void*a, const void*b) //比较两个整数
{ return *(int*)a-*(int*)b; }
```

```
int CmpF(const void*a, const void*b) //比较两个实数
{ return *(float*)a-*(float*)b>0?1:-1; }
```

因两实数相减的结果转换成整数时有误差，如25.6-25.4的值为0.2，转换成整型时，其值为0，不能表示25.6大于25.4，故只能用条件表达式。

```
int CmpS(const void*a, const void*b) //比较两个字符串
{ return strcmp((char*)a,(char*)b); }
```

```
void Sort( //功能: 选择法通用类型数据排序
void* v, //一组待排序的数据的指针
int n, //待排序的数据的个数
int size, //每个排序的数据占用内存的字节数
int(*Cmp)(const void*, const void*))
    //函数指针: 比较两个同类数据
```

```
{ char*p,*q,t;
  for(int i=0;i<n-1;i++)
  { p=(char*)v+i*size; //求第i个元素的指针值
    for(int j=i+1;j<n;j++)
    { q=(char*)v+j*size; //求第j个元素的指针值
      if(Cmp(p,q)>0)
        for(int k=0;k<size;k++)//逐个字节交换数据
          { t=p[k];p[k]=q[k];q[k]=t; }
    }
  }
}
```

因Sort()函数事先无法知道数组元素类型, 只好把一个数组元素分成size个字节, 这样数组中第i个元素的指针为:

$p=(char*)v+i*size;$

其中, v是数组的指针, size为每个元素占用的字节数。

```
void main(void)
{ int i,vi[]={23,44,32,66,15,25};
  float vf[]={15.4f,34.7f,55.4f,5.6f,18.3f};
  char s[5][4]={"cat","car","cab","cap","can"};
```

```
Sort(vi,sizeof(vi)/sizeof(int),sizeof(int),CmpI);
```

```
cout<<"\n排序后的整数为: \n";
```

```
for(i=0;i<sizeof(vi)/sizeof(int);i++)
```

```
    cout<<vi[i]<<'t';
```

```
Sort(vf,sizeof(vf)/sizeof(float),sizeof(float),CmpF);
```

```
cout<<"\n排序后的实数为: \n";
```

```
for(i=0;i<sizeof(vf)/sizeof(float);i++)
```

```
    cout<<vf[i]<<'t';
```

```
Sort(s,sizeof(s)/sizeof(s[0]),sizeof(s[0]),CmpS);  
cout<<"\n排序后的字符串为: \n";  
for(i=0;i<sizeof(s)/sizeof(s[0]);i++)  
    cout<<s[i]<<"\t";  
cout<<"\n";  
}
```

编程题：用void型指针编写一个通用类型数据查找函数，并分别调用该函数查找指定的整数和字符串。

执行程序后，输出：
排序后的整数为：
15 23 25 32 44 66
排序后的实数为：
5.6 15.4 18.3 34.7 55.4
排序后的字符串为：
cab can cap car cat

- stdlib.h库中的qsort()函数与Sort()函数相似，其原型为：

```
void qsort(void *b,size_t num,size_t width,  
           int (*cmp)(const void *e1, const void *e2 ));
```

其中，size_t 类型名代表unsigned int。

- 例4.30 用stdlib.h库中的qsort()函数实现字符串排序。

```
#include<iostream.h>
```

```
#include<string.h>
```

```
#include<stdlib.h>
```

```
int CmpS(const void*a, const void*b) //比较两个字符串
```

```
{ return strcmp((char*)a,(char*)b); }
```

```
void main(void)
```

```
{ char s[5][4]={"cat","car","cab","cap","can"};
```

```
    qsort(s,5,sizeof(s[0]),CmpS);
```

```
    for(int i=0;i<5;i++) cout<<s[i]<<'\\n';
```

```
}
```

参数个数可变的函数*

- 参数个数固定的函数：在定义函数时明确规定函数的参数个数及类型，在调用函数时，实参的个数必须与形参相同。
- 参数个数可变的函数(简称变参函数)：在定义函数时不能确定函数的参数个数，参数的个数在调用时才能确定。
- 例4.31 求若干个整数的最大值。

```
#include<iostream.h>
#include<stdarg.h>
```

固定参数 可变参数

```
int max(int num,int s...)
{ va_list ap;
  int m=s;
  va_start(ap,s);
```

编写变参函数通常要使用头文件 `stdarg.h` 中说明的类型 `va_list` 和宏 `va_start`、`va_arg` 及 `va_end`。

- 变参函数：参数表有“...”。
- 参数包括：固定参数和可变参数，且固定参数至少有一个。
- 变参函数应有确定实参个数的方法。如参数 `num` 用来表示参数的个数。

`va_list`: `char*` 类型。

`va_start` 宏：使 `ap` 指向最后一个固定参数 `s` 后的第一个可变参数。


```
for(int i=1;i<num;i++)  
{    int t=va_arg(ap,int);  
    if(t>m) m=t;  
}
```

va_arg宏：从ap所指参数取一个int型数据赋给t，并使ap指向下一个可变实参。

```
va_end(ap);  
return m;  
}
```

va_end宏：结束可变参数的读取，以使变参函数能正常返回。

```
void main( )  
{ int a,b,c,d,e;  
  cout<<"输入五个整数: ";  
  cin>>a>>b>>c>>d>>e;  
  cout<<"a和b的大者为"<<max(2,a,b)<<endl;  
  cout<<"五个整数中的最大者为"  
    <<max(5,a,b,c,d,e)<<endl;  
}
```

- 变参函数的参数传递协议约定:

因可变参数的类型在定义时无法直接给出, 故只能在调用时, 由实参的类型来确定。例如 `max(5,a,b,c,d,e)` 函数调用时的形参和实参的分配情况如图 10-25 所示, 即 `max()` 函数的可变形参的类型 `int` 由可变实参 `b,c,d,e` 的类型确定。

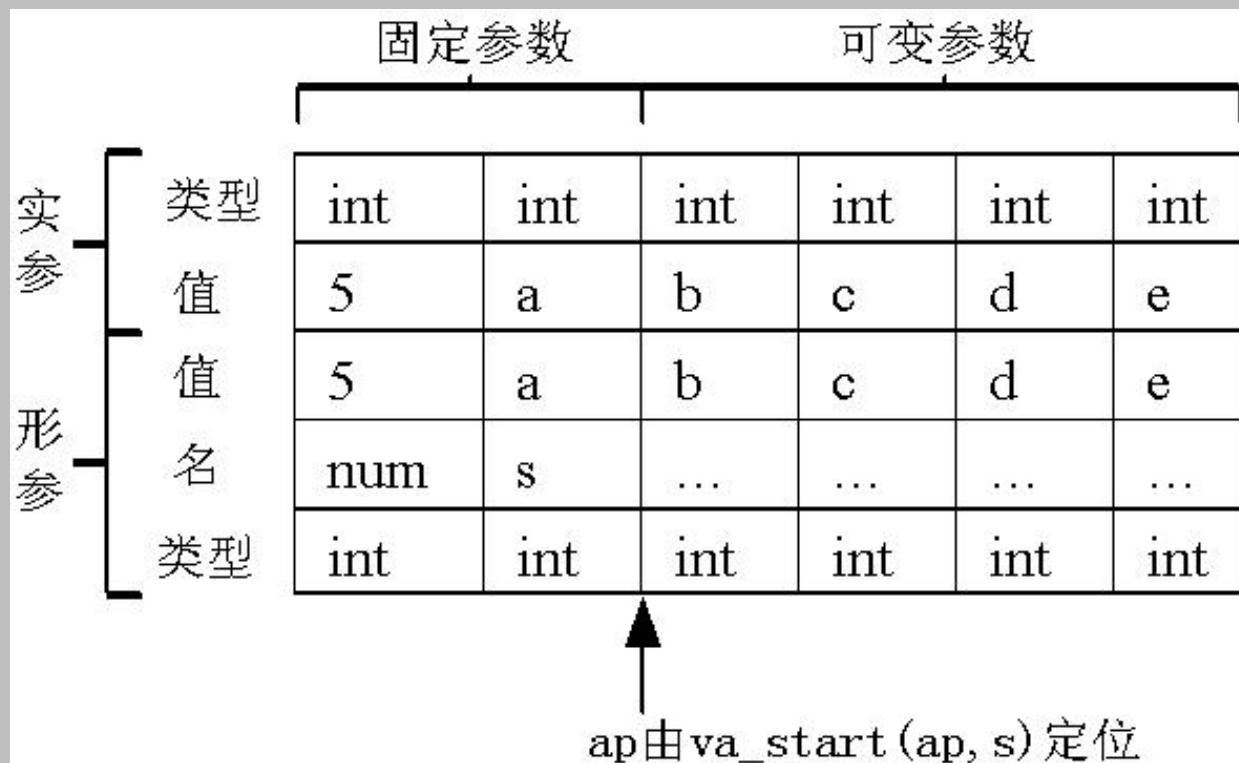


图 10-25 `max(5, a, b, c, d, e)` 函数调用时的形参和实参的参数传递情况。

- 变参函数的参数传递协议约定：当可变实参的类型为char时，可变形参的类型取int；当可变实参的类型为float时，可变形参的类型取double。
- 例4.32 例4.31的float版本。

```
#include<iostream.h>
```

```
#include<stdarg.h>
```

```
float max(int num,float f...)  
{ va_list ap;  
  float m=f;  
  va_start(ap,f);  
  for(int i=1;i<num;i++)  
  { float t=va_arg(ap,double);  
    if(t>m) m=t;  
  }
```

• 注意：若将语句
t=va_arg(ap,double);
改为
t=va_arg(ap,float);
则该程序将无法获得正确结果。

```

va_end(ap);
return m;
}

```

```

void main( )
{ float a,b,c,d,e;
  cout<<"输入五个实数: ";
  cin>>a>>b>>c>>d>>e;
  cout<<"a和b中的大者为"<<max(2,a,b)<<endl;
  cout<<"五个实数中的最大者为"<<max(5,a,b,c,d,e)<<endl;
}

```

		固定参数		可变参数			
实参	类型	int	float	float	float	float	float
	值	5	a	b	c	d	e
	值	5	a	b	c	d	e
	名	num	f
	类型	int	float	double	double	double	double

ap由va_start(ap, f)定位

图 10-26 max(5, a, b, c, d, e)函数调用时的形参和实参的参数传递情况

- 掌握变参函数的参数传递机制后，不用stdarg.h中说明的类型va_list和宏va_start、va_arg及va_end也可自编变参函数。
- 例4.33 例4.31的自编版本。

```
#include<iostream.h>
```

```
int max(int num,int s,...)
{  char *ap;           //定义char型指针ap
  int m=s;
  ap=(char*)&s+sizeof(int);//使ap指向固定参数s的末尾
  for(int i=1;i<num;i++)
  {  int t=*((int*)ap);  //从ap所指内存取一个整数
    if(t>m) m=t;
    ap=ap+sizeof(int);  //将ap指向下一个整数
  }
  return m;
}
```

```
void main( )  
{ int a,b,c,d,e;  
  
    cout<<"输入五个整数: ";  
    cin>>a>>b>>c>>d>>e;  
  
    cout<<"a和b中的大者为"  
        <<max(2,a,b)<<endl;  
    cout<<"五个整数中的最大者为"  
        <<max(5,a,b,c,d,e)<<endl;  
}
```