第7章 数组

华中科技大学计算机学院
甘早斌

主要内容

- 8.1 概述
- 8.2 一维数组
- 8.3 字符数组
- 8.4 字符串处理函数
- 8.5 多维数组
- 8.6 数组的应用程序设计*

补充知识 1/5

- PASCAL之父Niklaus Wirth
  - 尼古拉斯·沃斯(1934.2.15-)，生于瑞士温特图尔，计算机科学家，毕业于斯坦福大学，是好几种编程语言的主设计师
  - 1984年他因发展了这些语言而获图灵奖
  - 程序=算法+数据结构(Algorithm + Data Structures=Programs)
- 数据结构
  - 数据结构是指相互之间存在一种或多种特定关系的数据元素的集合，它是数据在计算机存储、组织数据的方式。
  - 通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。数据结构往往同高效的检索算法和索引技术有关

补充知识 2/5

- 数据结构的研究对象
  - 数据的逻辑结构
    - 指数据元素之间的逻辑关系,与他们在计算机中的存储位置无关。逻辑结构包括：集合、线性结构、树形结构、图形结构
  - 数据的存储结构
    - 指数据的逻辑结构在计算机存储空间的存放形式
  - 数据的运算结构
    - 算法的设计取决于数据（逻辑）结构，而算法的实现依赖于采用的存储结构。数据的存储结构实质上是它的逻辑结构在计算机存储器中的实现
    - 数据的运算是在数据的逻辑结构上定义的操作算法，如检索、插入、删除、更新和排序等

补充知识 3/5

- 常用结构
  - (1) 数组 (Array) （本章内容）
  - (2) 栈 (Stack)
    - 是只能在某一端插入和删除的特殊线性表。
    - 它按照先进后出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）
  - (3) 队列 (Queue)
    - 一种特殊的线性表，它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列是按照“先进先出”或“后进后出”的原则组织数据的。队列中没有元素时，称为空队列。

补充知识 4/5

- (4) 链表 (Linked List)
  - 是一种物理存储单元上非连续、非顺序的存储结构，它既可以表示线性结构，也可以用于表示非线性结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。
  - 链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。
- (5) 树 (Tree)
  - 是包含n（n>0）个结点的有穷集合K，且在K中定义了一个关系N，N满足以下条件：
    - (a) 有且仅有一个结点 K0，他对于关系N来说没有前驱，称K0为树的根结点。简称为根（root）。
    - (b) 除K0外，K中的每个结点，对于关系N来说有且仅有一个前驱。
    - (c) K中各结点，对关系N来说可以有m个后继（m>=0）。

补充知识 5/5

- (6) 图 (Graph)
  - 图是由结点的有穷集合V和边的集合E组成。其中，为了与树形结构加以区别，在图结构中常常将结点称为顶点，边是顶点的有序偶对，若两个顶点之间存在一条边，就表示这两个顶点具有相邻关系。
- (7) 堆 (Heap)
  - 在计算机科学中，堆是一种特殊的树形数据结构，每个结点都有一个值。通常我们所说的堆的数据结构，是指二叉堆。堆的特点是根结点的值最小(或最大)，且根结点的两个子树也是一个堆。
- (8) 散列表 (Hash)
  - 若结构中存在关键字和K相等的记录，则必定在f(K)的存储位置上。由此，不需比较便可直接取得所查记录。称这个对应关系f为散列函数(Hash function)，按这个思想建立的表为散列表。

7.1 数组概述 1/3

- 简单数据类型的变量
  - 仅能描述一个单独的数据
  - 对客观对象的描述能力十分有限
  - 如何描述一群有联系的数据集合？(全班C语言考试成绩)
- 数组
  - 属于构造类型
  - 是相同数据类型数据的集合(不同数据类型数据如何考虑?)
- 元素
  - 组成数组的这些数据
  - 任何类型(简单类型、构造类型)

7.1 数组概述 2/3

- 数组特点
  - 其所有元素数目固定
  - 其所有元素类型相同
  - 其所有元素顺序存放(在内存中也是连续存放的)
- 数组作用
  - 集中管理
    - 将相关的同类型数据集中用一个标识符（数组名）表示
  - 元素顺序存放，但可随机定位
    - 用若干个数字序号（下标）来区别各数组元素
  - 例如定义float score[30]，可表述30位学生成绩
  - 用数组具有什么好处?

7.1 数组概述 3/3

- 问题
  - 计算全班30位同学某门课程的平均成绩
- 解决方法
  - 设置30个float型变量来记录成绩
  - 设置一个有30个float型元素的数组来记录成绩
- 问题分析
  - 参与运算的平均成绩，其数据类型都相同(符合数组特点)
  - 30位同学属于一个班，用数组可把30个成绩表示成一个整体
- 用数组的优点
  - 便于循环处理
  - 提高效率，便于书写、检查、修改(对海量数据效果更明显)

7.2 一维数组

- 7.2.1 一维数组的声明
- 7.2.2 一维数组的使用
- 7.2.3 一维数组的初始化
- 7.2.4 一维数组的存储结构
- 7.2.5 一维数组的运算
- 7.2.6 一维数组作为函数参数

7.2.1 一维数组的声明

- 维数
  - 标识一个数组元素所需要使用的下标的个数
- 一维数组
  - 只有一个下标
  - 可用于表示一个线性的数据队列
- 使用数组的要求
  - 先声明数组
  - 对它进行初始化，然后才能使用

7.2.1 一维数组的声明

- 要解决三个问题
  - 确定数组的数据类型
  - 给数组定义一个名字，以便在程序中使用
  - 指明数组的大小，即数组中元素的个数
- 声明形式
  - [存储类型说明符] [类型修饰符] 类型说明符 数组名[常量表达式]={初值表};
  - 存储类型说明符：extern、static
  - 类型修饰符：const、volatile
  - 类型说明符：int 、 char 、 …
  - 数组名：是一个标识符，是一个地址常量，用以表示数组中打头元素的地址

const和volatile的用法 1/4

- **const**
  - const是一个C语言的关键字，它限定一个变量不允许被改变
  - **修饰的对象：**变量，函数的参数和返回值，函数本身
  - 作用:
    - (1) 可以定义const常量 例如： const int i=100;
      - 在另一连接文件中引用const常量
      1. extern const int i; //正确的引用
      2. extern const int i=10; //错误！常量不可以被再次赋值
    - (2)便于进行类型检查 例如：
      void f(const int i) { .....}
      - 编译器就会知道i是一个常量，不允许修改；

const和volatile的用法 2/4

- (3) 可以保护被修饰的东西，防止意外的修改，增强程序的健壮性。还是上面的例子，如果在函数体内修改了i，编译器就会报错；如:
  void f(const int i) { i=10;//error! }
- (4) 可以节省空间，避免不必要的内存分配。例如：
  1. #define PI 3.14159 file://常量宏
  2. const doulbe Pi=3.14159; file://此时并未将Pi放入ROM中
  3. .....
  4. double i=Pi; file://此时为Pi分配内存，以后不再分配!
- (5) 提高了效率。
  - 编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。

const和volatile的用法 3/4

- const用法
  - (1) 修饰变量 const int a = 1 或 int const a = 1 (两种一样)
  - (2) 修饰函数参数
    - 如果函数的输入参数采用指针，用const修饰，防止意外修改指针指向内存单元，起保护作用。
    - 如果参数采用值传递，当函数内多次使用其初值，加const，可防止代码无意中修改。如:
      void fun(const int value),
  - (3)修饰函数的返回值
    - const修饰符也可以修饰函数的返回值，是返回值不可被改变，格式如下:
      const int fun1();

const和volatile的用法 4/4

- volatile 的使用
  - 将变量定义为volatile表示告诉编译器该变量可能会被意想不到的改变，则优化器每次用到该值都重新从内存读取它。
  - volatile用在如下的几个地方：
    - (1) 中断服务程序中修改的供其它程序检测的变量需要加volatile；
    - (2) 多任务环境下各任务间共享的标志应该加volatile；
    - (3) 存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能由不同意义；
    - (4) 如果一个变量的值可能会被程序操作之外的其它操作所改变，那么你必需用volatile 声明。在嵌入式系统中其它操作是：中断服务程序的操作、硬件动作的操作。

7.2.1 一维数组的声明

- 例7.1 具有基本数据类型的一维数组的声明
  - #define SIZE 10
  - int array[5];
  - double d[5],e[SIZE];
  - char name[SIZE*5];
- 错误例子
  - unsigned int size;
  - char str[size],buffer[2*size];
- 错误原因
  - 数组的大小一经说明就不能改变 ？
  - 长度说明不是表达式，在编译之前就必须明确确定 ？
  - 在C99及C11标准中: unsigned int size=10;
  - 右边声明合法: char str[size],buffer[2*size];

7.2.1 一维数组的声明

- 例7.2 采用类型修饰符的一维数组的声明

  static int y[10];
  - 数组y中的每一个成员都是静态整型成员

  extern double s[2];
  - 作了一个外部双精度型数组的引用性声明
  - 应该在另外的源文件中通过double s[2];来定义s数组，这样第2个声明语句才有意义

7.2.2 一维数组的使用

- C提供的各种操作符
  - 针对基本数据类型的变量
- 数组
  - 是构造数据类型
  - 但其元素是基本数据类型的变量,或构造类型的变量.
- 访问数组
  - 不需设计专门的数组操作符
  - 方法：数组名[下标表达式]
- 例int a[5], j=2；
  - 5个元素依次是a[0],a[1],a[2],a[3],a[4]
  - 正确用法： a[j+2] 、a[++j] 、a[j--]、a[5*j-7]
  - 错误用法：a[j-3] 、a[2*j+1] ？

7.2.3 一维数组的初始化

- 显式初始化值的个数与说明长度相同

  int x[5]={0,1,2,3,4};

  int y[5]={0,1,2,3,4,5}; ？
- 有初始化值时，长度说明可缺省
  - 数组长度由初值个数确定

  int y[]={1,2,3,4,5,6,7,8};
- 初始化值的个数可以小于说明长度，但只能缺省最后连续元素的初值

  int z[10]={0,1,2,3,4}; /*前5个下标变量赋值*/

  int u[9]={ , 1, , ,2}; 错误!

例7.3 使用一维数组计算学生的平均成绩

```
1.  #include " stdio.h"
2.  void main(void)
3.  {
4.     int score[30],i,sum=0;
5.     double average;
6.     printf("input the scores please:\n");
7.     for(i=0;i<30;i++)
8.        scanf("%d",&score[i]);  /* 将键盘输入的成绩赋给各个数组元素 */
9.     for(i=0;i<30;i++)
10.       sum+=score[i];          /* 求学生成绩的累加和 */
11.    average=sum/30.0;          /* 计算平均成绩 */
12.    printf("sum=%d\n",sum);
13.    printf("average=%lf\n",average);
14. }
```

演示源程序\ex7_3.c

7.2.4 一维数组的存储结构

- 存放方法
  - 各个元素从数组名标明的起始地址开始在内存中连续存放
- 例如int a[5];
  - 对于32位机，1个int变量占4个字节空间

数组元素	a[0]	a[1]	a[2]	a[3]	a[4]
元素地址	a+0	a+1	a+2	a+3	a+4
	&a[0]	&a[1]	&a[2]	&a[3]	&a[4]

例7.8 观察一维数组的存储情况的程序

```
1.  #include "stdio.h"
2.  #define SIZE 3
3.  int main(void)
4.  {
5.     int x[SIZE]={1,3,5}, k;
6.     char s[SIZE+1]="ABC";
7.     float f[SIZE]={1.414,3.1415, 5.25};
8.     printf("the value of x is 0x%x\n",x);
9.     for(k=0;k<SIZE;k++)
10.       printf("x[%d]=%d\t addr=0x%x\n",k,x[k],&x[k]);
11.    return 0;
12. }
```

```
the value of x is 0x22ff30
k=0   x[0]=1  addr=0x22ff30
k=1   x[1]=3  addr=0x22ff34
k=2   x[2]=5  addr=0x22ff38
```

演示源程序\ex7_8.c

7.2.5 一维数组的运算

- C语言中，数组的运算最终都要归结到对具有基本数据类型的数组元素的操作
- C提供的各种操作符
  - 赋值运算、各种算术运算、++、--
  - 针对基本数据类型的变量
  - 可针对int、float、以及double类型数组中元素
- 合法操作

```
1. int x[3]={1,2,3},y[3]={4,5,6},z[3], k=1;
2. z[0] = x[0] + y[0];
3. z[1] = x[0] + y[3];
4. z[k] = ++x[0] + --y[k++];
5. z[1] = x[0] + y[x[1]];      /* 等价于z[1]= x[0]+y[2]; */
```

  - 不允许两个数组直接相加

z=x+y;　编译时给出提示“cannot add two pointers”

7.2.6 一维数组作为函数参数

- 参数的传递
  - 传值
  - 传地址
- 数组作为函数参数
  - 采用的是参数传址
    - 传送的是数组的地址
    - 或传送数组元素的地址
  - 还需传送数组元素的个数

- 例7.9 对n个整数采用冒泡法对其排序
- 分析:　输入n个数存放在一数组中，便于循环处理
- 排序
  - 是将一个数据元素任意的序列
  - 按照一定的规则排列成为一个有序的序列
  - 按照由小到大排列称为升序，按照由大到小排列称为降序，

```
1. void  bubble_sort(int a[],int n)
2. {  对a[n]表示的n个整数进行排序处理
3. }
4. main()
5. { 输入n个数;
6.    调用bubble_sortsort对n个数排序;
7.    输出n个数检验排序效果
8. }
```


例7.9 冒泡排序

将n个整数存放于一个一维数组中，则对n个整数的排序就转变成对n个元素的整型数组的排序。

- 第一遍
  - 从下标为0的元素开始，对两两相邻的元素进行比较
    - 如果前一个元素大于后一个元素，就交换这两个元素的值
    - 循环n-1次比较
  - 在第一遍循环后
    - 不仅把最大整数移到数组最末尾（其下标为n-1）（像冒泡）
    - 还尽量把较大值往后挪，例如“31”
    - 还剩前面n-1个数需排序

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
原始	31	25	12	86	42	6
1-1	25	31	12	86	42	6
1-2	25	12	31	86	42	6
1-3	25	12	31	86	42	6
1-4	25	12	31	42	86	6
1-5	25	12	31	42	6	86

例7.9 冒泡排序

- 第2遍
  - 对前n-1个数，与第1遍一样循环处理
  - 还剩前n-2个数未排序
- 第i遍
  - 对前n-i+1个数，与第1遍一样循环处理
  - 还剩前n-i个数未排序
- 大循环结束时机(对i)
  - i>=n-1
  - 一遍循环中未发生交换，即已排好序

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1遍	25	12	31	42	6	86
2遍	12	25	31	6	42	86
3遍	12	25	6	31	42	86
4遍	12	6	25	31	42	86
5遍	6	12	25	31	42	86

例7.9 冒泡排序

```
1.  void bubble_sort(int a[],int n)    /* 形参用形式数组a[] */
2.  {
3.      int i,j,t,k; int flag=1;
4.      for(i=0; (i<n-1)&&flag; i++) * 共进行n-1轮"冒泡" */
5.      {
6.          flag=0;
7.          for(j=0; j<n-i-1; j++)/*只有前n-1-i个元素需要排序*/
8.              if(a[j]>a[j+1]) /* 对两两相邻的元素进行比较 */
9.                  {  t=a[j], a[j]=a[j+1], a[j+1]=t;
10.                 flag=1;
11.                 }
12.     }
13. }
```

- 演示：源程序\ex7_9.c

7.3 字符数组

- 7.3.1 字符数组的声明和使用
- 7.3.2 字符数组的初始化

字符数组的概念

- 以字符为元素的数组称为字符数组。
- 通过字符数组可以构造字符串，从而为文本处理奠定基础。
- 各种计算机语言编制的源程序，各种文本文档最终都要以ASCII码或者汉字的国标码等形式体现，都会以字符串的形式供程序处理。
- 因此字符数组、字符串，以及相关函数非常重要。它们在计算机专业的后续课程中都会发挥重要作用，需要熟练掌握。

7.3.1 字符数组的声明和使用

- 字符数组
  - 元素的数据类型为char或wchar_t
  - 声明格式
    - 与前面讨论的一维数组相同
    - char s[81];
- 字符串
  - 用一对双引号界定的一个字符序列
  - C语言没有规定字符串类型
    - 用一个字符数组来存放字符序列，并且在末尾加一个空字符 ´\0´ 来构造字符串

7.3.1 字符数组的声明和使用

- 字符串的存储

´W´	´u´	´h´	´a´	´n´	´\0´

- 字符串的长度
  - 字符串的长度 = 字符串的存储长度–1
- 设计字符数组的最小长度
  - 应该等于该字符串的存储长度
  - 或字符数组的最小长度应该等于该字符串的长度加1
- 字符数组的使用
  - 通过下标来访问字符数组中的具体字符元素

例7.10 编程产生大写和小写的26个英文字母字符串

```
#include "stdio.h"
int main(void)
{       char Capital[27],Lowercase[27];
        int i,delt='a'-'A'; /* 'a'-'A'等于0x20 */
        Capital[0]= 'A';
        Lowercase[0]=Capital[0]+delt; /* 'a'送Lowercase[0] */
        for(i=1;i<26;i++){
                Capital[i]=Capital[i-1]+1; /* 'B'、'C'、… 送Capital[1],…*/
                Lowercase[i]=Lowercase[i-1]+1; /* 'b'、… 送Lowercase[1],…*/
                Capital[26]= '\0';
        }       /* 送字符串终结符到Capital[26] */
        Lowercase[26]=Capital[26];          /* 送字符串终结符到Lowercase[26] */
        printf("%s\n",Capital); /* 输出字符串ABCD…YZ */
        printf("%s\n",Lowercase);  /* 输出字符串abcd…yz */
        return 0;   }
```

演示：源程序\ex7_10.c

- 程序运行结果如下：

```
ABCDEFGHIJKLMN
OPQRSTUVWXYZ
abcdefghijklmnop qrstuvwxyz
```


7.3.2 字符数组的初始化

- 通过初始化列表
  - char s1[8]={´W´,´u´,´h´,´a´,´n´,´\0´};
  - ´\0´必须在初始化列表中显示给出
- 用字符串初值
  - char s2[28]="Computer Science"
  - 末尾将自动加上一个"\0"
- 第三种方法
  - 数组长度未明显给出,由初始化字符串长度+1决定
  - char s3[]="Computer Engineering"
  - 字符数组的容量恰好等于字符串的存储长度

7.4 字符串处理函数

- 7.4.1 串操作函数的设计及使用
- 7.4.2 数字串与数值之间转换的函数

7.4.1 串操作函数的设计及使用

- 编译环境中定义了常用字符串处理标准库函数，头文件为string.h
- 串操作函数
  - 求字符串长度
  - 字符串的拷贝
  - 字符串的比较
  - 字符串的连接
  - 求字符串的子串
  - 删除字符串首尾空白字符
  - 从字符串中删除所有与给定字符相同的字符
  - 将字符串反转等函数

例7.11 求字符串长度的函数

```
1.  int strlen(char s[])
2.  {
3.      int j=0;
4.      while(s[j]!= '\0')
5.       j++;
6.      return j;
7.  }
```

```
7.  void main(void)
8.  {
9.      char str[]="there is a boat on the lake";
10.     int length;
11.     length = strlen(str);
12.     printf("length of the string is %d\n",length);
13. }
```

(演示：源程序\ex7_11.c)

- 运行结果：

length of the string is 28

例7.12 编写字符串复制的strcpy函数

```
void strcpy(char t[], char s[])
{
    int j=0;
    while(t[j]=s[j++])
        ;
}
```

(演示：源程序\ex7_12.c)

```
void main(void)
{
    char str1[30],  str2[]= "there is a
    boat on the lake.";
    strcpy(str1,str2);
    puts(str1);
}
```

- 运行结果：

there is a boat on the lake.

例7.13 编写比较两个字符串的函数并且加以应用

- 比较规则
  - 从两个字符串的第一个字符起开始
  - 按照字符ASCII码值的大小进行比较
- 返回规定
  - 当两个字符串相等时，返回0
  - 当第一个串大于第二个串时，返回一个大于零的值
  - 当第一个串小于第二个串时，返回一个小于零的值

strcmp("abc","abde")
返回 'c'-'d'的值
strcmp("abf","abde")
返回 'f'-'d'的值
strcmp("abc","abc")
返回 '\0'-'\0'的值
strcmp("abc","abcdef")
返回 '\0'-'d'的值


```
1.  int strcmp(char s[],char t[])
2.  {
3.      int j=0;
4.      while(s[j]==t[j] && s[j]!= '\0')  j++;
5.      return s[j]-t[j];
6.  }
7.  int main(void)
8.  {   char s1[]="car",s2[]="bus",s3[]="truck",s4[]="car";
9.      printf("%s is %s %s.\n", s1,strcmp(s1,s2)>0?"great \
10.             then":strcmp(s1,s2)<0?"less then" : "equal to",s2);
11.     printf("%s is %s %s.\n", s1, strcmp(s1,s3)>0?"great \
12.             then":strcmp(s1,s3)<0?"less then": "equal to",s3);
13.     printf("%s is %s %s.\n", s1,strcmp(s1,s4)>0?"great \
14.             then":strcmp(s1,s4)<0?"less then" : "equal to",s4);
15.     return 0;}
```

运行结果：

car is great then bus.
car is less then truck.
car is equal to car.

例7.14编写连接字符串的strcat函数

```
char * strcat(char t[],char s[])
{   int j=0,k=0;
    while(t[j++]!='\0') ;
    j--;
    while((t[j++]=s[k++]));
    return t;
}
```

```
void main(void)
{
    char s1[80]="I like ", s2[]="the C programming.";
    strcat(s1,s2);
    printf("%s\n",s1);
}
```

运行结果：

I like the C programming.

演示:源程序\ex7_14.c

例7.15 编写求字符串子串的strstr函数

```
int strstr(char cs[], char ct[])
{   int j=0,k;
    for(;cs[j]!= '\0';j++)   /*对cs串中每个字符*/
        if(cs[j]==ct[0] ){   /*如果cs串中某个字符等于字串的开始字符*/
            k=1;        /*记下已匹配一个字符，从下个字符继续*/
            while(cs[j+k]==ct[k]&&ct[k]!='\0')   /*相等且未到子串尾*/
                k++;  /*匹配次数加1*/
            if(k==strlen(ct))  /*若连续匹配次数与字串长度相同*/
                return j;       /*匹配成功，返回匹配位置下标*/
    }
    return -1;      /*没有找到字串，返回-1*/
}
```


例7.15 编写求字符串子串的strstr函数

```
void main(void)
{   char s1[80]="C is the most widely used programming language.",s2[]="use";
    int i,j=0;
    i = strstr(s1,s2);
    printf("the sub_string's beginning position is %d\n",i);
    while(j<i)
        putchar(s1[j++]);
    putchar('\n');
    while(putchar(s1[i++]))
        ;
    putchar('\n');
}
```

运行结果：

```
the sub_string's beginning position is 21
C is the most widely
used programming language.
```

(演示：源程序\ex7_15.c)

例7.16 编写删除字符串首尾空白字符的trim函数

```
int trim(char s[])
{   int i, num, j=0, k=0, len=strlen(s);
    while(s[j]== ' ' || s[j]== '\t' || s[j]== '\n' || s[j]== '\r')
        j++;       /* 计算首部空白字符的个数 */
    i=len-1;  /* i为字符串最后一个字符的下标，即'\0'前面一个字符的下标 */
    while(s[i-k]== ' ' || s[i-k]== '\t' || s[i-k]== '\n' || s[i-k]== '\r')
        k++;        /* 计算尾部空白字符的个数 */
    num=len-j-k;   /*计算非空白字符的个数 */
    for(i=0;i<num;i++)
        s[i]=s[i+j];     /*将第1个非空白字符s[0+j]复制到s[0],…*/
    s[num]='\0';          /*赋'\0',形成字符串*/
    return strlen(s);       /*返回去掉空白字符后的串长度*/
}
```

(演示EX8_19_21.C)

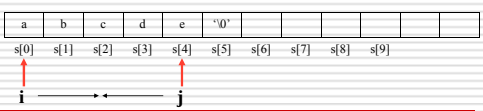
例7.17 编写从字符串s中删除所有与给定字符相同的字符的delete_c函数

```
void delete_c(char s[],char c)
{
    int j=0,  k=0;            /*j-读指示器，k-写指示器*/
    while(s[j]!='\0'){
        if(s[j]!=c)
            s[k++]=s[j];
        j++;
    }
    s[k]='\0';
}
```


例7.18 编写将字符串反转的reverse函数

将一个字符串首尾颠倒过来如：将"abcde"颠倒为"edcba"

```
void reverse(char s[])
{
    int  j, k;   /*j-前指示器  k-尾指示器*/
    char c;
    for(j=0, k=strlen(s)-1; j<k; j++,k--)
        c=s[j], s[j]=s[k], s[k]=c;
}
```



trim, delete_c, reverse函数的应用

```
void main(void)
{   char str[80]="    atbtctdtetft     ";
    printf("before trim,the string is \"%s\"\n",str);
    trim(str);
    printf("after trim,the string is \"%s\"\n",str);
    delete_c(str, ´ t ´ );
    printf("after delete 't',the string is \"%s\"\n",str);
    reverse(str);
    printf("after reverse,the string is \"%s\"\n",str);
}
```

- 运行结果：

before trim,the string is "   atbtctdtetft   " after delete ´ t ´ ,the string is "abcdef"
after trim,the string is "atbtctdtetft" after reverse,the string is "fedcba"

7.4.2 数字串与数之间转换的函数

- 例7.19 编写将一个十进制数字串转换成为对应整数的函数atio
- atoi函数功能
  - 将s字符数组中存放的一个十进制数字串转换成为对应的整数，并返回该整数
- 算法：
  - ASCII码字符s[j]转换为对应数字
    - s[j]-'0'
  - 本位乘以10加下一位的算法
    - 54321 = （（（（5）*10+4）*10+3）*10+2*10）+1

- 例7.19 atoi函数

```
1.  #define BASE 10
2.  int atoi(char s[])
3.  {
4.      int j=0,num=0;
5.      for(;s[j]!='\0';j++)
6.          num=num*BASE +s[j]-'0';
7.      return num;
8.  }
```

- (演示: 源程序\ex7_19.c)

例7.20 编写将一个整数转换成基数为BASE的数字串的函数itoa

```
#define BASE 10
void itoa(int n,char s[])
{   int sign,j=0;
    if((sign=n)<0)
        n=-n;
    while(n>0){
        s[j++]=n%BASE+'0';
        n/=BASE;        }
    if(sign<0)
        s[j++]='-';
    s[j]='\0';              算法见p148
    reverse(s);
    }           (演示: 源程序\ex7_20.c)
```


例7.21 编写将一个十六进制数字串转换成对应整数的函数htoi

- 问题
  - 当基数BASE大于10，如：16，由于十六进制数的表示形式，如'a'和'A'，'b'和'B'
  - 以及0到f或F在ASCII码表的编码不连续性
  - 因此需要在转换中进行一定的调整
- htoi函数
  - 将一个存放在字符数组s中的十六进制数字串转换成为对应的整数
  - 并且返回转换后的整数

程序中使用的算法

- （1）本位乘以16加下一位的算法:
  num=num*16 + 下一位
- （2）ASCII码字符s[j]转换为对应数字的算法:
  - 当s[j]>='0' && s[j]<='9'，下一位为s[j]-'0';
  - 当s[j]>='a' && s[j]<='f'，下一位为s[j]-'a'+10;
  - 当s[j]>='A' && s[j]<='F' ，下一位为s[j]-'A'+10

htoi函数(演示: 源程序\ex7_21.c)

```
int htoi(char s[])
{ int j=0,num=0;
  if(s[j]=='0' && (s[j+1]=='x' || s[j+1]=='X'))
      j+=2;
  else
      return -1;
  for(;s[j]!='\0';j++){
     if(s[j]>='0' && s[j]<='9') num=num*16+s[j]-'0';
      if(s[j]>='a' && s[j]<='f') num=num*16+s[j]-'a'+10;
      if(s[j]>='A' && s[j]<='F') num=num*16+s[j]-'A'+10;
      }
      return num;
} /*思考：itoh: 如何将十进制整数转换为十六进制的数字串？ */
```


7.4.3 C11标准中新增的Unicode字符集和Unicode字符串**

- Unicode（单一码）是一种为每种语言中的每个字符设定统一且唯一的二进制编码，以实现跨语言、跨平台进行文本转换和处理要求的计算机字符编码，又称为统一码、万国码。
- C11标准中增强了对Unicode的支持。包括为UTF-16/UTF-32编码增加了char16_t和char32_t数据类型，提供了包含Unicode字符串转换函数的头文件<uchar.h>。

1．定义字符串的语法的BNF描述

- <串文字>::=[<编码前缀>] " [<s字符序列>]"
- <编码前缀>::= u8|u|U|L
- <s字符序列>::=<s字符>|< s字符序列><s字符>
- <s字符>::=<源字符集中除了双引号"，反斜杠\，换行符>|<转移序列>

2．各种字符串的定义及解释说明

- ASCII码字符串是"abc"，它表示按照ASCII码编码形成串，每个字符占1个字节；
- UTF-8字符串是u8"abc"，它表示按照UTF-8字符编码形成串，每个字符占1个字节；
- UTF-16字符串是u"abc"，它表示按照UTF-16字符编码形成串，每个字符占2个字节；
- UTF-32字符串是U"abc"，它表示按照UTF-32字符编码形成串，每个字符占4个字节；
- wchar_t类型字符串是L"abc"，它表示wchar_t类型的字符串，每个字符占2个字节。

例7.22 使用Pelles C编译器输出char16_t、char32_t，以及wchar_t类型字符串举例

```
1.  #include "stdio.h"
2.  #include <uchar.h>
3.  int main(int argc, char *argv[])
4.  {   char16_t s[200] = u"abc123";
5.      wchar_t s1[200] = L"Huazhong University of Technology";
6.      char32_t s2[200] = U"Huazhong University of Technology is in Wuhan";
7.      char *p=(char *)s2;
8.      printf("%ls\n",s);
9.      printf("%ls\n",s1);
10.     while(*p){  putchar(*p); p+=4; }
11.     printf("\n");
12.   return 0;
13. }
```


说明

- 例7.22程序中输出char32_t 类型字符串的方法是非标准的方法。它利用了char32_t 类型英文字母首位值与ASCII码相同，但后3位为0的特征。
- 例7.22程序都只能输出英文，不能输出汉字。用"%ls"能否输出汉字还需要汉字点阵字模库的支持。
- C11标准中已经定义了各种用于UTF-16、UTF-32，以及wchar_t类型字符串的处理函数；针对va_list类型的可变参数提供了与scanf 函数和printf函数用法类似的vscanf函数和vprintf函数。

7.5 多维数组

- 实际应用
  - 有时需要用多个下标来实现对数组元素的访问
  - 例如
    - 张三同学，学号为01，语文和数学成绩分别为85，91，李四同学，学号位02，语文和数学成绩分别为82，95
- 解决方法
  - 用二维数组可以描述学号-课程成绩表中的成绩数据
- 多维数组的用途
  - 二维数组可以描述数学中的矩阵或行列式
  - 三维数组可以描述空间中的点集
  - n维数组来描述n维线性空间中的n维向量

	语文	数学
01	85	91
02	82	95

7.5.1 多维数组的说明与使用

- 形式
  - 类型说明 数组名[常量表达式1] [常量表达式2]…[常量表达式n]={初值表};
- 类型说明：[存储类型说明符] [类型修饰符] 数据类型
  - 例如：int x[2][2];
- 声明了一个二维数组，每一维的大小都是2。x[0][0]，x[0][1]，x[1][0]，x[1][1]是这个二维数组中的4个元素。
  - 例如： double y[2][3][4];

- 声明了一个三维数组，它共有2×3×4=24个元素。第1个元素是y[0][0][0]，第2个元素是y[0][0][1]，…，最后一个元素是y[1][2][3]。
- 对其元素的引用
  - 数组名[下标1] [下标2]…[下标n]
  - 例如，x[1][0]=3;

例7.23 对二维数组中元素的访问与操作

```
#include "stdio.h"
int main(void)
{   int x[2][3],a=2;  /* 声明2×3数组x */
    x[0][2]=8;     /* 将8赋给元素x[0][2] */
    scanf("%d",&x[1][2]);  /* 用&操作取元素x[1][2]的地址 */
    x[1][1]=x[0][2];  /* 将元素x[0][2]赋给元素x[1][1] */
    x[1][2]<<=a;  /* 将元素x[1][2]的内容左移2位 */
    printf("%d\n",x[1][2]);          /* 输出元素x[1][2]的值 */
    return 0;
}
```


例7.24 用二维数组表示学号-课程成绩表，计算每个同学的平均成绩并且输出数组中各元素的地址和内容

- 计算每个同学的平均成绩并且输出数组中各元素的地址和内容

```
1.  #include "studio"
2.  #define SIZE 2
3.  int main(void)
4.  {   int x[SIZE][SIZE+1];int i,j;
5.      for(i=0;i<SIZE;i++){
6.        for(j=0;j<SIZE;j++)
7.          scanf("%d",&x[i][j]);/*输入成绩*/
8.          x[i][SIZE]=(x[i][0]+x[i][1])/2; }/*计算学生的平均成绩*/
9.        printf("\n");
10.     for(i=0;i<SIZE;i++){
11.       for(j=0;j<SIZE+1;j++) /*输出成绩*/
12.           printf("%p\tx[%d][%d]=%d\n",&x[i][j],i,j,x[i][j]);
    }return 0;}
```

	语文	数学
01	85	91
02	82	95

- (演示：源程序\ex7_24.c)

- 例7.24 程序的运行结果
- 输入如下：

  85 91 82 95
- 程序的运行结果为：

```
0012FF68    x[0][0]=85
0012FF6C    x[0][1]=91
0012FF70    x[0][2]=88
0012FF74    x[1][0]=82
0012FF78    x[1][1]=95
0012FF7C    x[1][2]=88
```

7.5.2 多维数组的存储结构

二维数组x的逻辑存储结构

x[0][0] =85	x[0][1] =91	x[0][2] =88
x[1][0] =82	x[1][1] =95	x[1][2] =88

二维数组x的物理存储结构

地 址	元 素	值
0x0012FF68	x[0][0]	85
0x0012FF6C	x[0][1]	91
0x0012FF70	x[0][2]	88
0x0012FF74	x[1][0]	82
0x0012FF78	x[1][1]	95
0x0012FF7C	x[1][2]	88

7.5.3 多维数组的初始化

- 按照物理存储结构的顺序
  - int a[2][2]={85,91,82,95};
- 按照逻辑存储结构的顺序
  - 可读性好，但初值表的形式与数组的维数有关
  - int x[2][3]={{ 85,91,0}，{82,95，0}};
  - int d[2][2][2]={{{1,2},{3,4}},{{5,6},{7,8}}};
- 注意
  - 当数组的初值全部给出时，第1维大小的说明可以省略
  - int x[ ][3]={{ 85,91,0}，{82,95，0}};
  - int d[ ][2][2]={{{1,2},{3,4}},{{5,6},{7,8}}};
  - 其它维大小不能省略

7.5.4 二维字符数组

- 二维字符数组
  - 与其它二维数组类似
  - 用char说明的二维数组
    - char text[25][80];
- 初始化
  - 与其它二维数组类似
    - char s[2][4]={ ‘a’,’b’,’c’,’\0’,’d’,’e’,’f’,’\0’};
    - char s[2][4]={{‘a’,’b’,’c’,’\0’},{‘d’,’e’,’f’,’\0’}};
  - 用字符串对二维数组进行初始化
    - char devices[3][12]={“hard disk”,”CRT”,”keyboard”};
  - 省略第1维的方式
    - char devices[ ][12]={“hard disk”,”CRT”,”keyboard”};

- 二维字符数组的使用
  - 引用单个字符元素
    - weekend[i][j] =m;
  - 引用字符串
    - weekend[i]表示weekend数组中第i行字符串的首地址
    - 输出用：printf(“%s”,weekend[i]);

例7.25 字符串数组的输入输出操作举例

```
1.  #include "stdio.h"
2.  int main(void)
3.  {   int i;
4.      char devices[3][12]={"hard disk","CRT","keyboard"};
5.      devices[0][0]= 'H'; /* "hard disk"变为"Hard disk"*/
6.      devices[2][0]= 'K'; /* "keyboard "变为"Keyboard "*/
7.      for(i=0;i<3;i++) printf("%s\n", &devices[i][0]);
8.      scanf("%s",devices[1]);
9.      for(i=0;i<3;i++) printf("%s\n", devices[i]);
10.     return 0;
11. }
```

(演示：源程序\ex7_25.c)

7.6 数组的应用

7.6.1 矩阵乘法运算

$$A=\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{bmatrix},\quad B=\begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ b_{30} & b_{31} & b_{32} \end{bmatrix}$$

- 算法
  - 定义3个2维数组
  - 通过三重循环来实现 $C=A\times B$，其中 $c_{ij}=\sum_{k=0}^{3} a_{ik}b_{kj}$
    - 外层循环用于控制乘积矩阵C的行
    - 中间层循环用于控制乘积矩阵C的列
    - 内层循环用于计算乘积矩阵元素C_{ij}
- 注意：
  - 无论是一维数组，还是n维数组，当其作为实参时只需给出数组名，而对应形参则是第1维的大小不需给出，其余各维的大小需要给出。

例7.26 矩阵的乘法运算

演示：源程序\ex7_26.c

```
#include "stdio.h"
#define N 3
#define K 4
#define M 3
void mul_matrix(int a[][K],int b[][M],int C[][M],int n,int k,int m);
int main(void)
{   int A[N][K]={{1,2,3,4},{5,6,7,8},{9,0,1,2}};
    int B[K][M]={{1,2,3},{4,5,6},{7,8,9},{0,1,2}};
    int C[N][M]; int i,j;
    mul_matrix(A,B,C,N,K,M);
    for(i=0;i<N;i++){
        for(j=0;j<M;j++)
            printf("%8d  ",C[i][j]);
        printf("\n");}
    return 0;}
```


例7.26 矩阵的乘法运算

```
void mul_matrix(int a[][K],int b[][M],int c[][M],int n,int k,int m)
{
    int i,j,p,sum;
    for(i=0;i<n;i++)
        for(j=0;j<m;j++){
            sum=0;
            for(p=0;p<k;p++)
                sum+=a[i][p]*b[p][j];
            c[i][j]=sum;
        }
}
```


7.6.2 基于分治策略的二分查找函数

- 分治策略的基本思想：将一个规模较大的问题分解为若干个规模较小的问题。
- 例7.27 运用分治策略的二分查找函数
  - 二分查找算法的思路
    - 将已排好序的n个元素数组a分成两半，取a[n/2]与x比较
      - 如果x=a[n/2]，则找到x，算法结束
      - 如果x<a[n/2]，则在数组a的前半部分继续查找x
      - 如果x>a[n/2]，则在数组a的后半部分继续查找x
  - 返回值
    - 如果找到x，返回该数所在单元的下标
    - 如果没有找到，返回-1


```
1.  #include "stdio.h"
2.  int BinarySearch(int a[],int x,int n)
3.  {
4.     int  front=0,  back=n-1,  middle;
5.     while(front<=back) {
6.           middle=(front+back)/2;  /* 计算中间单元的下标 */
7.           if(x<a[middle])
8.                 back=middle-1;     /* 查找单元变成原来的前半部*/
9.           else if(x>a[middle])
10.                front=middle+1;    /*查找单元变成原来的后半部*/
11.          else return (middle);     /* 找到，返回下标 */
12.    }
13.    return -1; /* 没有找到，返回-1 */
14. }
15.
16. void main(void)
17. {
18.       int x[]={1,3,5,7,9,11,13,15,17,19},index;
19.       index=BinarySearch(x,11,10);
20.       if(index!=-1)  printf("find %d!\n",x[index]);
21.       else  printf("not find!\n");
22. }
```

- 程序的运行结果是：
- find 11!

7.6.3 逆波兰表达式的生成**

- 中缀表达式：对于整型变量a,b,c,d,e,f的四则运算表达式a+b*(c-d)-e/f，它的特点是运算符位于左右操作数的中间，因此称为中缀表达式 。
- 后缀表达式（逆波兰表达式）：对于a+b，如果将运算符放在左右操作数的后面，形成a b +，则称它为a+b的逆波兰表达式，又称为后缀表达式。
  - 例子：对于中缀表达式：a+b*(c-d)-e/f
  - 其逆波兰表达式为： a b c d - * + e f / -
- 好处：生成逆波兰表达式是指将中缀表达式转换为后缀表达式。将一个中缀表达式转换为后缀表达式的好处是，可以利用值栈对其进行求值。

数据结构--堆栈

- 在生成逆波兰表达式的过程中要用到一种称为堆栈的数据结构。堆栈又称栈,它是一种一端固定(称为栈底),在另外一端(称为栈顶)进行入栈和出栈操作的数据结构。
- 栈内元素按照后进先出原则顺序存放，栈指针通常指向栈顶元素。入栈时栈指针sp先自增，指向新的待压单元，然后将数据data压入堆栈，形成新的栈顶元素；
  - 即： ++sp=data;
- 出栈操作一般是先弹出栈顶元素，然后栈指针sp自减指向新的栈顶元素;
  - 即： data=*sp-- ; 它将弹出栈顶元素*sp赋给data,然后做sp--。

将中缀表达式转换为逆波兰表达式的算法的思想

- 声明存放逆波兰表达式的字符数组out和存放运算符的堆栈stack；
- 从左到右扫描这个中缀表达式，扫描中对于中缀式中的数字字符直接依次将其存放到数组out中。
- 扫描中如果遇到运算符(称为当前运算符)，则将其与栈顶运算符进行优先级比较；
- 如果栈顶运算符得优先级高于或等于当前运算符，则进行出栈操作；出栈操作要考虑栈内是否存在左圆括号‘(’,如果存在“(”，则出栈操作从栈顶开始，到“(”结束；如果不存在“(”，则整个堆栈内运算符从栈顶开始依次全部出栈；
- 反之，如果栈顶运算符得优先级比当前运算符低，则将当前运算符压栈。
- 扫描中如果遇到左圆括号“(”，则直接将其压栈；
- 扫描中如果遇到右圆括号“)”，则将栈顶到左圆括号“(”之间的运算符依次出栈，并抛弃左圆括号“(”。
- 同时要保证操作数与运算符，以及运算符与运算符之间都有空格分隔。

例7.28 将输入的中缀四则运算表达式转换为后缀四则运算表达式。

演示：源程序\ex7_28.c

```
#include “stdio.h“
#include "string.h"
#define SIZE 100
void push(char c);
void pop(void);
char stack[SIZE];
char out[SIZE];
int top=-1,j=0;
int main(void)
{   char in[]="15+21*(41-12)-1128/12";
    int i=0;
    memset(out,' ', SIZE); /*全部初始化为空格*/
    while(in[i]!='\0')
    {
```


例7.28-continue

```
switch(in[i]) {
    case '0': case '1': case '2': case '3': case '4': case '5':
    case '6': case '7': case '8': case '9':
        out[j]=in[i];          /*数字字符直接进入逆波兰序列*/
        j++; break;
    case '+': case '-':        /*加减运算处理*/
        j++;/*在数字序列和操作符之间插入空格*/
        pop();/*出栈*/
        push(in[i]); /*入栈*/ break;
    case '*': case '/':/*乘除运算处理*/
        j++;/*在数字序列和操作符之间插入空格*/
        if(stack[top]=='+'||stack[top]=='-')/*如果栈顶运算符的优先级*/
          push(in[i]);/*比当前运算符低,则当前运算符压栈*/
        else
          pop();/*如果是乘或除，即优先级不低于当前运算符，出栈*/   break;
```


例7.28-continue

```
case '(':/*左圆括号处理*/
    push(in[i]);/*左圆括号'('直接入栈*/
    break;
case ')':/*右圆括号处理*/
    pop();/*栈中到左圆括号'('为止的所有运算符依次弹出*/
    top--;/*跳过栈中的左圆括号'('*/
    break;
default:
    printf("illegal input!\n");/*遇到非法字符，返回*/
    return -1;  }
    i++;  } //移动到下一个字符
    pop();     /*栈中剩余运算符全部依次出栈*/
    out[j]='\0';          /*形成逆波兰字符串*/
    printf("%s\n",out);/*输出逆波兰字符串*/
    return 0;  }
```


例7.28-continue

```
void push(char c) {/*运算符c入栈操作*/
    top++;
    stack[top]=c;
}
void  pop(void)  {  /*出栈算法：如果栈内有左圆括号，则运算符依次出栈直到遇到左圆括号为止; 否则，栈内所有操作符全部出栈*/
    while(top>=0&&stack[top]!='('){
                j++;
                out[j]=stack[top];
                top--;j++;
    }
    if(top==-1)/*如果到栈底，插入空格，与下一操作数分开*/
                j++;
}
```


注意

- 值得注意的是pop函数已不是传统意义下的出栈函数。
- 同时，为了方便，已预先将中缀表达式字符串“15+21*(41-12)-1128/12”存放在字符数组in中，读者容易修改为从键盘输入。
- 本例的输出可以直接作为例7.29的输入。
- 两个例子结合起来就可以完全解决带括号的加、减、乘、除四则运算表达式的求值问题。

7.6.4 利用值栈对逆波兰表达式进行求值**

- 值栈：值栈是堆栈的一种，值栈中只存储操作数而不存储操作符。
- 值栈操作规律是：
  - 1.如果遇到的是操作数，将其入栈（也称为压栈）；
  - 2.如果遇到的是操作符则将先弹出（出栈）的元素作为右操作数，接着弹出的操作数作为左操作数，两者按照操作符规定的语义进行规定运算；
  - 3. 然后将运算结果压栈。
  - 4. 当整个逆波兰表达式处理完毕时，值栈中将只有一个元素，它就是表达式的运算结果。

- 值栈的入栈和出栈操作
  - 从程序中的入栈函数push可以看出：
    - 入栈操作是栈指针（实际是下标）top先自增，使top指向新的待压单元，然后将n值放入堆栈，形成新的栈顶元素。即：++top=n;
    - 同样，从程序中的出栈函数pop可以看出：
    - 出栈时先将栈顶元素赋给临时变量x，然后栈指针top自减，指向新的栈顶元素；
    - 即：x=top--;
  - 出栈操作的结果是通过将x的值返回给调用函数完成。

例7.29 设允许采用加、减、乘、除运算符和括号的四则运算表达式的逆波兰表达式已经得到；输入该逆波兰表达式，且每个变量或操作符之间用空格分隔,以ctrl+z结束；利用值栈对逆波兰表达式进行求值。

```
#include "stdio.h“
#define SIZE 1000
void push(int n);  /*压栈操作*/
int pop(void);                  /*出栈操作*/
int stack[SIZE];   /*将数组作为值栈使用*/
int top=-1;                     /*下标作为栈顶"指针"*/
int main(void)
{    char ch,ch1;
     int i=0,n=0,right_oprand;/* right_oprand为右操作数 */
     ch=ch1='\0';
     while((ch1=ch,ch=getchar())!=EOF)
     {
```

演示：源程序\ex7_29.c

例7.29-continue

```
switch(ch){
     case '0': case '1': case '2': case '3': case '4': case '5': \
     case '6': case '7': case '8': case '9':   /*ch为数字字符*/
          n=n*10+(ch-'0'); /*将数字字符串转换成为对应整数*/
          break;
     case ' ':/*ch等于空格*/
     case '\n':
          switch(ch1) {
          case '+':
            right_oprand=pop();/*右操作数出栈*/
            push(pop()+right_oprand);/*被加数出栈加加数，和再压栈*/
            break;
          case '-':
            right_oprand=pop();/*右操作数出栈*/
            push(pop()-right_oprand);/*被减数出栈减减数，差再压栈*/
            break;
```


例7.29-continue

```
case '*':
     right_oprand=pop();/*右操作数出栈*/
     push(pop()*right_oprand);          /*被乘数出栈乘以乘数，积再压栈*/
     break;
case '/':
     right_oprand=pop();/*右操作数出栈*/
     if(right_oprand)              /*除数为零吗*/
          push(pop()/right_oprand);/*除数非零，做除法，商再压栈*/
     else{
          printf("divide by zero!");/*输出除数为零*/
          return -1;                              /*返回异常*/
     }    break;
     case ' ':   break;
     default:
          push(n);  /*操作数压栈*/
          n=0;
}
```


例7.29-continue

```
case '+': case '-': case '*': case '/':
                    break;
               default:
                         printf("illegal input!\n");
                                             return -1;
               }
     }
     printf("result is %d\n",pop());/*输出运算结果*/
     return 0;
}
```

例7.29-continue

```
void push(int n)
{    top++;          /*栈指针指向待压单元*/
     stack[top]=n;  /*数据入栈*/
}
int pop(void)
{    int x;
     x=stack[top];  /*栈顶元素值赋给x*/
     top--;                  /*栈指针减1，指向下一单元*/
     return x;               /*返回栈顶元素值*/
}
```


程序分析

- 运行： 输入： 15 21 41 12 - * + 1128 12 /
  输出结果是： result is 530。
- main函数分三大部分：
  - 1. while循环控制一个一个字符地读取整个逆波兰表达式；
  - 2. 外层swich功能包括将读入的数字字符串转换成为对应整数，遇到空格或换行则根据前面一个字符是否是运算符进行两方面的操作：
    - （1）如果前一个字符不是运算符，说明操作数的转换完成，将其入栈；
    - （2）如果前一个字符是运算符，则通过出栈取操作数，然后完成规定运算，并且将运算结果压入值栈。
  - while循环结束时，说明整个逆波兰表达式已经处理完毕，所以从值栈中弹出最终结果并输出。

本章小结

- 本章首先介绍了数组产生的背景和相关基本概念。
- 在一维数组的讨论中，主要涉及一维数组的声明、使用、初始化及一维数组的存储结构。
- 同时也介绍了一维数组元素之间的运算，以及一维数组作为函数参数的使用方法。
- 字符数组是构造字符串的基础，也是构成文本的基础。
- 本章介绍了字符数组的声明和使用，以及字符数组的初始化。
- 在此基础上，介绍了串操作函数的设计、使用、数字串与数值之间转换的函数，以及相关函数涉及的算法、转换方法。

- 在多维数组方面，介绍了多维数组的声明与使用，多维数组的存储结构，多维数组的初始化，以及二维字符数组。
- 在数组的应用程序设计方面主要介绍了矩阵乘法运算，基于分治策略的二分查找函数，由中缀表达式生成逆波兰后缀表达式，以及利用值栈对逆波兰表达式求值。
- 这些程序所涉及的数组、堆栈、值栈等数据结构，以及相应的算法思想和编程技术都需要很好掌握。

Assignments:

- 必做题：
  - 7.1，7.2，7.3，7.4，7.5，7.7，7.8，7.9，7.10，
  - 7.11，7.12，7.13，7.14，7.15，7.16，7.17，7.18
- 我建议：
  - 后面习题，每题都做，并且搞懂！
  - （除涉及到内容**的题目外）