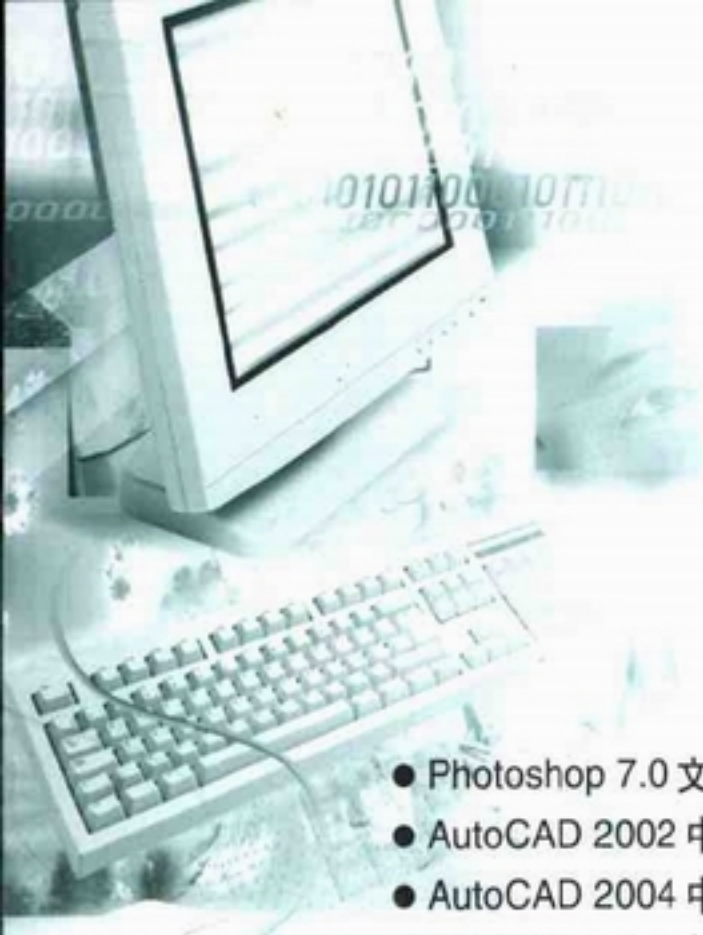
万水创作效果百例丛书

C语言精彩编程百例

温海 张友 童伟 等编著

中国水利水电出版社
www.waterpub.com.cn

责任编辑：马高山　封面设计：孙　平



- Photoshop 7.0 文字纹理滤镜 200 例
- AutoCAD 2002 中文版精彩设计百例
- AutoCAD 2004 中文版精彩设计百例
- Photoshop 7.0 中文版精彩设计百例
- Dreamweaver MX 中文版精彩设计百例
- Flash MX 中文版精彩设计百例
- Authorware 6.0 精彩设计百例
- Authorware 6.5 精彩设计百例
- 3DS MAX 5 精彩设计百例
- Visual Basic 精彩编程百例
- Visual C++ 精彩编程百例
- ASP.NET 精彩编程百例
- Delphi 7 精彩编程百例
- C 语言精彩编程百例

北京万水电子信息有限公司
Beijing Multi-Channel Electronic Information Co., Ltd.
地址：北京市海淀区长春桥路5号新起点嘉园4号楼1706室
邮编：100089
电话：(010)8256.2819（总机）
传真：(010)8256.4371
E-mail：mchannel@public3.bta.net.cn

ISBN 7-5084-1818-2/TP・775
定价：34.00 元

ISBN 7-5084-1818-2
9 787508 418186 >

万水创作效果百例丛书

C语言精彩编程百例

温海　张友　童伟　等编著

中国水利水电出版社

内 容 提 要

C 是一种通用的程序设计语言，它包含了紧凑的表达式、丰富的运算符集合、现代控制流以及数据结构等四个部分。C 语言功能丰富，表达能力强，使用起来灵活方便；它应用面广，可移植性强，同时具有高级语言和低级语言的优点，因此，在工程计算及应用程序开发中得到了广泛的应用。

众所周知，学习新的程序设计语言的最佳途径是编写程序，而本书正是通过了对 100 个典型实例的分析和讲解，来帮助读者掌握这门语言并积累大量经验，从而可以熟练地进行 C 程序设计。

全文共分为四篇，全面、系统地讲述了 C 语言各个方面的知识点和程序设计的基本方法，以及编写程序过程中值得注意的地方，内容深入浅出，通俗易懂。对于 C 语言的初学者来说，这是一本绝对好的入门教材，对于有经验的专业人员，也会发现本书很有价值。

图书在版编目（CIP）数据

C 语言精彩编程百例/温海等编著. —北京：中国水利水电出版社，2003
（万水创作效果百例丛书）

ISBN 7-5084-1818-2

Ⅰ.C… Ⅱ.温… Ⅲ.C 语言—程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 102473 号

书　　名	C 语言精彩编程百例
作　　者	温海　张友　董伟　等编著
出版、发行	中国水利水电出版社（北京市三里河路 6 号　100044）
	网址：www.waterpub.com.cn
	E-mail：mchannel@public3.bta.net.cn（万水）
	sale@waterpub.com.cn
	电话：（010）63202266（总机）、68331835（营销中心）、82562819（万水）
经　　售	全国各地新华书店和相关出版物销售网点
排　　版	北京万水电子信息有限公司
印　　刷	北京北医印刷厂
规　　格	787×1000 毫米　16 开本　　22.75 印张　　499 千字
版　　次	2004 年 1 月第一版　2004 年 1 月北京第一次印刷
印　　数	0001—5000 册
定　　价	34.00 元

凡购买我社图书，如有缺页、倒页、脱页的，本社营销中心负责调换

前　言

C是一种通用的程序设计语言，它包含了紧凑的表达式、丰富的运算符集合、现代控制流以及数据结构等四个部分。C语言功能丰富，表达能力强，使用起来灵活方便。它应用面广，可移植性强，同时具有高级语言和低级语言的优点，因此，在工程计算及应用程序开发中得到了广泛的应用。

经验告诉我们，学习新的程序语言的最佳途径就是亲手实践，而本书正是通过对100个典型实例的分析和讲解，来帮助读者掌握这门语言并积累大量经验，从而可以熟练地进行C程序设计。

全书分为四篇，按照由浅及深、循序渐进的原则，全面、系统地讲解了C语言各个方面的知识点和常用的程序设计基本技巧，以及编写程序过程中值得注意的地方，内容深入浅出，通俗易懂。

第一篇由44个实例组成，主要侧重于对C语言基础知识的介绍，集中大量实例来讲解指针、数组以及位操作等对于C语言初学者而言较难理解的知识点。

第二篇由26个实例组成，覆盖自定义结构类型、I/O操作及部分常用函数，至此读者已拥有独立完成一般程序设计的能力。

第三篇由23个实例组成，以C语言的常用算法为主，重点向读者演示了数据结构的应用以及常用的数值算法的C语言实现方法。

第四篇由7个实例组成，这一篇的作用是验证、提升读者的编程能力，作为前三篇的总结应用，涵盖了本书中所介绍的大部分知识点。

我们希望读者通过对这本书的学习，能够具有较强的C语言编程技能，具备一定的独立编程能力，本书所附带的程序源代码都已经过编译，读者可验证程序运行的结果。

在这里感谢中国水利水电出版社给我们这次机会，使我们能够把自己的经验传授给广大读者；同时感谢出版社的编辑们，是他们的认真监督和辛勤工作才能使该书顺利完成；还要感谢参与本书编写的李伯苓、王晓霞、刘逸飞、段广仁、谭友利、辛知庆、李冠、任宝山、吴仪委等人，是他们的辛勤劳动使本书顺利的与大家见面。

由于编者水平有限，书中缺点和错误在所难免，恳请广大读者批评指正，并提出宝贵的意见和建议。

编者

2003年10月

目　　录

第一篇　基础知识篇

第二篇　深入提高篇

第三篇　常用算法篇

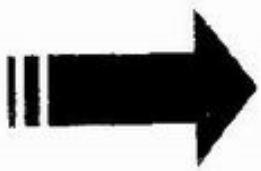
第四篇　综合应用篇

第一篇
基础知识篇

想成为一个优秀的C程序员，首先要做的就是熟练掌握C语言的各基本要素，并会正确使用它们，因为C语言的基本要素是所有C程序的根本和基石。基础篇作为全书的第一篇，其目的就是通过程序实例向读者介绍C语言的基本要素，让读者能够尽可能快地掌握它们。

对于成功的C程序设计而言，正确理解并熟练使用指针是至关重要的，所以指针这部分内容是本篇的重点。由于指针的概念比较抽象，对于初学者而言很难一下子理解，因此在这里列举了大量的实例，并作了全面、详尽的分析，希望读者通过对这些程序的学习，能够加深对指针的理解；希望读者通过本篇的学习，能够模仿书中提供的实例编写一些小程序，以便对C程序设计有初步的了解。

Let's GO!



实例1 数据类型转换

实例说明

本例旨在介绍数据的类型转换。C语言规定，不同类型的数据需要转换成同一类型后才可进行计算，在整型、实型和字符型数据之间通过类型转换便可以进行混合运算。除了上述内容，我们还要介绍一些常用算术运算符的优先级与结合性，希望读者能够熟记。

注意点：并非所有类型的数据之间都可以进行转换，例如，指针和上述三种类型数据之间不能够进行类型换算。

知识要点

当混合不同类型的变量进行计算时，便可能会发生类型转换。

相同类型的数据在转换时有规则可循，如字符必定先转换为整数（C语言规定字符类型数据和整数数据之间可以通用），short型转为int型（同属于整型），float型数据在运算时一律转换为双精度（double）型，以提高运算精度（同属于实型）。

不同类型的数据发生转换时，遵循低级类型向高级类型转换的原则，例如int型数据与double型数据进行运算时，是先将int型数据转换成double类型，然后再进行运算，结果为double类型。

此外，在一个赋值语句中，若发生类型转换，则是赋值语句右部（表达式一侧）的值转换成左部（目标一侧）的类型。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
    // 定义变量并赋初值
    int    a = 5;
    char   c = 'a';
    float  f = 5.3;
    double m = 12.65;
  double result;
```

```
    // 同类型数据间进行运算并输出结果
    printf("a + c = %d\n", a + c);
    printf("a + c = %c\n", a + c);
    printf("f + m = %f\n", f + m);

    // 不同类型数据间进行运算并输出结果
    printf("a + m = %f\n", a + m);
    printf("c + f = %f\n", c + f);

    // 将上述四个变量进行混合运算，并输出结果
    result = a + c * (f + m);
    printf("double = %f\n", result);
}
```

程序分析

程序中分别定义了一个整型数据 a，一个字符型数据 c，以及两个实型数据 f 和 m。

当整型数据和字符型数据进行运算时，结果会随输出格式说明的不同而不同，当结果以整型输出格式“%d”输出时，结果为整数，若以字符型输出格式“%c”输出时，结果为字符。

当整型数据和双精度型数据进行运算时，C 先将整型数据转换成双精度型数据，再进行运算，结果为双精度类型的数据。同样，当字符型数据和实型数据进行运算时，C 先将字符型数据转换成实型数据，然后进行计算，结果为实型数据。

在表达式求解时，按运算符的优先级别的高低次序执行，例如先乘除后加减。若在一个运算对象两侧的运算符的优先级别相同，那么按照“自左向右”的方向进行结合，但若在表达式中存在括号，则括号中运算的优先级别最高，最先被执行，所以程序中算式 a + c * (f + m)的运算次序为，先执行(f + m)中的运算，然后将其结果与 c 相乘，最后同 a 相加。

请注意，代码行中的“=”是赋值运算符，不属于算术运算符。赋值运算符的结合性是按照“自右向左”的规则执行的。因此，在代码行 result = a + c * (f + m)中，是先得出算式 a + c * (f + m)的结果，而后再将此结果赋给双精度变量 result。

实例2 转义字符

实例说明

C中的字符常量是用单引号括起来的一个字符。此外，C还允许一种特殊形式的字符常量，就是以一个"\"开头的字符序列，通常称它们为转义字符。

鉴于转义字符的特殊性，在此有必要作出介绍。在实例当中会看到一些常用的转义字符，如换行符、回车符等。通过对例题的学习，希望读者能够理解这些常用转义字符的含义，并能够在今后的编程中熟练使用它们。

知识要点

转义字符是C语言中表示字符的一种特殊形式。

通常使用转义字符表示 ASCII 码字符集中不可打印的控制字符和特定功能的字符，如用于表示字符常量的单撇号（'），用于表示字符串常量的双撇号（"）以及反斜杠（\）等。转义字符用反斜杠（\）后面跟一个字符或一个八进制或十六进制数表示。

字符常量中使用单引号和反斜杠以及字符常量中使用双引号和反斜杠时，都必须使用转义字符表示，即在这些字符前加上反斜杠。

使用转义字符时需要注意以下三点问题：

（1）转义字符中只能使用小写字母，每个转义字符只能看作一个字符。

（2）\v 垂直制表和\f 换页符对屏幕没有任何影响，但会影响打印机执行响应操作。

（3）在C程序中，使用不可打印字符时，通常用转义字符表示。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>3

void main()
{
    // 换行符'\n'，用于输出换行
    printf("How are you?\n");
    printf("I am fine.\n\n");

    // 横向跳格符'\t'，使跳到下一个输出区
```

```
    printf("How are you?\t");
    printf("I am fine.\n\n");

    // 退格符'\b'，使当前的输出位置退一格，即输出的起始位置左移一位
    printf(" How are you?\n");
    printf(" \bI am fine.\n\n");

    // 回车符'\r'，使当前输出位置回到本行开头
    printf("                I am fine.");  // I 前面共有 16 个空格
    printf("\rHow are you?\n\n");

    // 多个转义字符的混合运用
    printf("note:\n  a s\ti\b\bk\rp\n");
}
```

程序分析

程序中共有五个输出模块，前四个输出模块都只用到了一个转义字符，不难得出结果。在第五个输出模块中，综合用到了前四个输出模块中的转义字符，在此，我们将重点分析第五个输出模块。

printf 函数先在当前行输出“note:”，然后换行。程序的输出位置跳到第二行后，首先从左端开始输出 “ a s”（注意，在字符 a 的左右两边各有一个空格），然后遇到“\t”,它的作用是跳格，即跳到下一个输出位置，在我们所用的输出系统中，一个输出区占 8 列（即 8 个空格）。则下一输出位置从第 9 列开始，所以在第 9 列上输出“i”。下面遇到两个“\b”，“\b”的作用是退一格，因此，“\b\b”的作用是使当前输出位置（第十列）退回到第 8 列输出“k”。最后，遇到“\r”，它代表回车（不换行），此时输出返回到本行的最左端（第一列），输出字符“p”。

所以第五个输出模块的最终输出是：

```
note:
p a s  ki
```

实例3 关系和逻辑运算

实例说明

这是一个介绍关系运算和逻辑运算的 C 程序实例。程序向读者介绍了所有的六种关系运算符和三种逻辑运算符，以及它们的优先级次序，旨在使读者通过此例能够熟练掌握它们，并能够对它们进行简单运用。

此外，本例还要让读者确立这样一种概念，那就是真（true）和假（false）的思维是关系和逻辑操作符概念的基础。在 C 中，true 代表非零值，false 代表零值。使用逻辑或关系操作符的表达式返回零作为假值，返回 1 作为真值。

知识要点

关系运算符中的“关系”二字指的是一个值与另一个值之间的关系，逻辑运算符中的“逻辑”二字指的是连接关系的方式。因为关系和逻辑运算符常在一起使用，所以在此将它们放在一起讨论。

下面列出的是关系和逻辑操作符的相对优先级：

最高　!

　　>　>=　<　<=

　　==　!=

　　&&

最低　||

需要注意的是，除运算符“!”之外，所有关系和逻辑操作符的优先级都低于算术操作符，也就是说，当算式中同时有算术操作符、关系和逻辑操作符（“!”除外）时，是在执行完所有的算术运算后，才开始执行关系和逻辑运算。

此外，同算术表达式一样，在关系或逻辑表达式中也可使用括号来修改原来的计算顺序。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
```

```
    // 定义一个整数类型的变量，用来存放后面算式的值
    int logic;

    int a = 1;
    int b = 2;
    int c = 3;

    logic = a+b>c&&b<=c;
    printf("logic = %d\n", logic);

    logic = a>=b+c||b==c;
    printf("logic = %d\n", logic);

    logic = !(a<c)+b!=1&&(a+c)/2;
    printf("logic = %d\n", logic);
}
```

程序分析

程序中的三个输出是 0、0 和 1，即分别为假、假、真。下面分析一下程序中三个算式的运算顺序。

算式一：a+b>c&&b<=c，实际上可表示成((a+b)>c)&&(b<=c)。C 首先进行算术运算 a+b，其值为 3（真），然后才根据关系和逻辑运算符的优先级进行运算，即分别运算 3>c 和 b<=c，它们的值非别为 0（假）和 1（真），最后将 0 和 1 相与（&&），得出最终结果为 0（假）。

算式二：a>=b+c&&b==c，在程序中亦可写成(a>=(b+c))&&(b==c)。首先得出 b+c 的值为 5（真），再分别计算出 a>=5 和 b==c，值分别为 0（假）和 0（假），将它们相或后，可得输出为 0（假）。

算式三：!(a<c)+b!=1&&(a+c)/2，可写成((!(a<c)+b)!=1)&&((a+c)/2)。由于嵌套括号的计算顺序是由里向外，所以算式的计算顺序可表示如下：

((!(a<c)+b)!=1)&&((a+c)/2) → ((!1+b)!=1)&&(4/2) → ((0+b)!=1)&&(4/2)

→ 1&&2 → 1

输出结果为 1。

实例4 自增自减

实例说明

C语言中有两个很有用的运算符，通常在其它计算机语言中是找不到它们的，那就是自增和自减运算符：++和--。自增运算符++对操作数增加一个单位，而自减运算符--对操作数减小一个单位。

本例正是要向读者介绍这两个运算符，使读者能够熟悉这两个运算符的一般用法。此外，自增和自减运算符是既能放在操作数之前，也能放在操作数之后的，但在具体的运算当中，这两种方法是有区别的，这也是本例所要讲述的重点，希望读者能够注意。

知识要点

在表达式当中，自增和自减运算符在操作数前或后是有区别的。增/减运算符位于操作数之前时，C先实施增/减操作，然后才使用操作数的值；若增/减运算符是在操作数的后边，那么，C是先使用操作数的值，而后再相应增/减操作数的内容。

注意点：

（1）自增运算符（++）和自减运算符（--），只能用于变量，而不能用于常量或表达式。

（2）++和--的结合方向是“自右向左”。

程序源码

该应用程序的源代码如下：

```
# include <stdic.h>

void main()
{
    int i, j, k;
    int m, n, p;

    i = 8;
    j = 10;
    k = 12;

    // 自增在操作数之前
    m = ++i;
```

```
    printf("i = %d\n", i);
    printf("m = %d\n", m);

    // 自减在操作数之后
    n = j--;
    printf("j = %d\n", j);
    printf("n = %d\n", n);

    // 自增、自减的混合运算
    p = (++m)*(n++)+(--k);
    printf("k = %d\n", k);
    printf("p = %d\n", p);
}
```

程序分析

在算式 m = ++i 中，对整形变量 i 进行了自增运算。由于自增运算符是置于 i 之前，所以是先对 i 进行加 1 操作，此时 i 的值已不再是 8，而是 9，然后再将自增后的 i 赋给变量 m，所以得到的输出为 9。

算式 n = j--是对变量 j 进行的自减操作，自减运算符位于操作数 j 之后，因此，赋给变量 n 的值就是 j 的原值 10，变量 n 的输出为 10，然后才进行自减操作，这时 j 的值减 1 变为 9。

最后进行的运算是同时包含自增和自减的混合运算。对于操作数 m 和 k 而言，自增和自减运算符位于它们之前，所以它们在算式中的值是经过自加和自减的，而对于变量 n 而言，自增运算符是位于它之后，因此它在算式中是以原值进行计算的。此时，我们再分析算式 p = (++m)*(n++)+(--k)，它实际上可以写成 p=10*10+11，所以变量 p 的输出结果为 111。

实例 5 普通位运算

实例说明

所谓位运算是指进行二进制位的运算。因为 C 语言的设计目的是取代汇编语言，所以它必须支持汇编语言所具有的运算能力，因此，C 语言提供了位运算的功能，这样与其他高级语言（如 PASCAL）相比，它便具有了很大的优越性。

C 语言共提供了六个位运算符，本例重点向读者介绍其中四个，它们分别是按位与（&）、按位或（|）、按位异或（^）以及取反（~）。其他两个位运算符会在下一个例子中介绍。

知识要点

下面将对程序中所涉及到的四个位运算符的具体使用规则作详细介绍。

按位与（&）运算符：参加运算的两个运算量，如果都为 1，则该位为 1，否则为 0。

按位或（|）运算符：两个相应位中只要有一个为 1，则该位的结果为 1。

按位异或（^）运算符：参加运算的两个相应位，同号则结果为 0（假），异号则结果为 1（真）。

取反（~）运算符：它是一个单目（元）运算符，用来对一个二进制数按位取反，即将 0 变为 1，1 变为 0。

注意点：

（1）位运算符中除了取反（~）运算符以外，其他的均为二目（元）运算符，即要求两侧各有一个运算量。

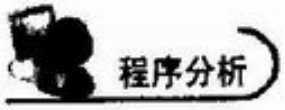
（2）位操作是对字节或字中的位（bit）进行测试、置位或移位处理，这里字节或字是针对 C 标准中的 char 和 int 数据类型而言。因此，位操作不能用于 float、double、long double、void 及其他复杂类型。

程序源码

该应用程序的源代码如下：

```
    # include <stdio.h>

void main()
{
        // 定义了一个无符号字符型变量，此变量只能用来存储无符号数
```

```
    unsigned char result;

    int a, b, c, d;
    a = 2;
    b = 4;
    c = 6;
    d = 8;

    // 对变量进行“按位与”操作
    result = a & c;
    printf("result = %d\n", result);

    // 对变量进行“按位或”操作
    result = b | d;
    printf("result = %d\n", result);

    // 对变量进行“按位异或”操作
    result = a ^ d;
    printf("result = %d\n", result);

    // 对变量进行“取反”操作
    result = ~a;
    printf("result = %d\n", result);
}
```



在实例中，对操作数进行的几个位运算都是极简单的位运算，目的仅是使读者能够了解它们的运算规则。首先，读者把程序中定义的变量按照二进制的格式写出，然后再根据本例“知识要点”中的阐述，不难得出结果，结果分别是2、12、10和253。

实例 6 位移运算

实例说明

通过上个实例的学习，相信读者对位运算已经有了初步了解。

在实例 5 当中，已经介绍了四个位运算符。本例将重点介绍另外两个位运算符，它们是左移位运算符（<<）和右移位运算符（>>）。

知识要点

左移位操作符（<<）：用来将一个数的各二进制位全部左移若干位。

标准的左移语句是：variable << 左移位数。在左移的过程中，高位左移后溢出，舍弃不起作用。左移一位相当于操作数乘以 2 的一次方；左移两位相当于操作数乘以 2 的平方。

右移位操作符（>>）：将一个数的各二进制位全部右移若干位。

标准的右移语句是：variable>>右移位数。在右移的过程中，移到右端的低位将被舍弃，对于无符号数，高位补零。右移一位表示操作数除以 2，右移 n 位相当于操作数除以 2 的 n 次方。

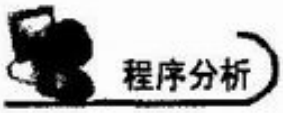
在右移时，需要注意符号位问题。对于无符号数，右移时左边高位移入零。对于有符号数，若原来符号位为零（即该数为正），则左移也是移入零，但如果符号位原来为 1（即负数），则左边移入 0 还是 1，要取决于所用的计算机系统。

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>

vcid main()
{
        unsigned a, b, c, d;
        int n;
        a = 64;
        n = 2;

        // 将操作数 a 右移(6-n)位
        b = a >> (6-n);
        printf("b = %d\n", b);
```

```
        // 将操作数 a 左移 n 位
        c = a << n;
        printf("c = %d\n", c);

        // 对操作数 a 进行混合位运算
        d = (a >> (n-1)) | (a << (n+1));
        printf("d = %d\n", d);
}
```



屏幕的输出结果如下所示：

```
b = 4;
c = 256;
d = 544;
```

程序首先对变量 a 进行了右移四位的操作，具体的操作可表示为：

```
0000 0000 0100 0000 → 0000 0000 0000 0100
```

所以，得出输出为 4。

然后，程序又对变量 a 进行了左移两位的操作，具体的操作可表示为：

```
0000 0000 0100 0000 → 0000 0001 0000 0000
```

所以，得出的输出结果为 256。

在最后的混合运算中，程序分别对 a 进行了左移和右移的操作，再将所得的数按位相或，对应的操作可表示如下：

```
0000 0000 0010 0000 | 0000 0010 0000 0000 → 0000 0010 0010 0000
```

因此，d 的最终输出为 544。

实例7 字符译码

实例说明

在 C 语言中，字符常量的存储形式和整数的存储形式类似，所以字符常量可以像整数一样在程序中参与相关的运算。本实例中进行的相关运算正是用到了上述的知识点。

这是一个将字符译码的简单程序，译码规律是：用原来字符后面的第六个字符代替原来的字符。例如，字符 a 后面的第六个字符是 g，则用 g 代替 a。本例题中的原字符串是“Chinese”。

知识要点

在这里向读者介绍的是字符数据在内存中的存储形式及其使用方法。

将一个字符常量放到一个字符变量中，实际上并非把该字符本身放入到内存单元中去，而是将该字符相应的 ASCII 代码放到存储单元中。例如，字符 c 的 ASCII 代码为 99，字符 h 的为 104。

既然在内存中，字符数据是以 ASCII 代码存储，它的存储形式与整数的存储形式类似，因此，我们说 C 语言使得字符型数据和整数数据之间可以通用。

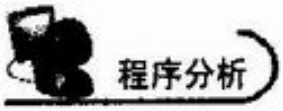
一个字符数据既可以以字符形式输出，也可以以整数形式输出。以字符形式输出时，需要先将存储单元中的 ASCII 码转换成相应的字符，然后输出。以整数形式输出时，直接将 ASCII 码作为整数输出。程序中对字符数据所进行的算术运算，实际上是对它们的 ASCII 码进行的算术运算。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
    // 定义字符型变量，并给它们付初值
    char c1, c2, c3, c4, c5, c6, c7;
    c1 = 'C';
    c2 = 'h';
    c3 = 'i';
    c4 = 'n';
```

```
    c5 = 'e';
    c6 = 's';
    c7 = 'e';

    // 输出源码
    printf("源码是: %c%c%c%c%c%c%c\n", c1, c2, c3, c4, c5, c6, c7);

    // 对字符进行译码运算
    c1 = c1 + 6;
    c2 = c2 + 6;
    c3 = c3 + 6;
    c4 = c4 + 6;
    c5 = c5 + 6;
    c6 = c6 + 6;
    c7 = c7 + 6;

    // 输出译码结果
    printf("密码是: %c%c%c%c%c%c%c\n", c1, c2, c3, c4, c5, c6, c7);
}
```



由于在本例中，只是实现一些简单的译码工作，应该不难分析出结果。

程序在屏幕的输出结果为：

源码是：Chinese

译码是：Inotkyk

例题中所涉及到字符的 ASCII 码，读者可到相关的 C 语言书籍上去查找。

实例8 指针操作符

实例说明

指针是C语言中的一个重要概念，也是C语言的一个重要特色。在后面的实例中，我们将全面、详细地介绍有关指针的内容。本例介绍的只是一些有关指针的基本知识，包括指针相关操作符&和*的基本概念和一些基本用法，目的是使读者对指针有个初步的了解，这样，在后面学习指针的过程中，可以很快地进入角色。

知识要点

指针（pointer）是变量的内存地址。指针变量（pointer variable）专门存放指向相应类型的指针。

指针有三个主要功能：帮助快速引用数组的元素；允许C函数修改调用变元的内容；支持链表和其他动态数据结构。

第一个指针操作符是&。这种一元操作符返回其操作数的内存地址（一元操作符只取一个操作数）。第二个指针操作符是星号（*）。它也是一元操作符，*是&的补，返回其操作数所示地址处的值。

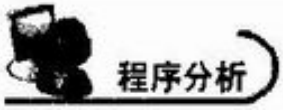
放置指针的变量必须恰当声明。声明中，存放内存地址的变量（即指针）必须在名字前冠以星号（*），由此指明该变量中存放指向变量类型的指针，例如，声明存放指向int类型的指针变量时书写成：int *p。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
        // 定义一个整形指针 p
        int *p;
        int begin, end;

        begin = 10;
        // 给指针 p 赋初值
```

```
    p = &begin;
    // 将指针指向的值传给变量 end
    end = *p;

    printf("begin = %d\n", begin);
    printf("end = %d\n", end);

    // 输出指针中的地址值
    printf("p = %d\n", p);
    printf("*p = %d\n", *p);
}
```



程序分析

程序首先声明一个整型指针变量，

```
int *p;
```

语句中 p 不是整数，而是指向一个整数的指针，两者之间的差别很大，希望读者能够理解这一点。指针指向的数据类型称为指针的基类型（base type）。指针变量本身也是变量，用于存放指向基类型对象的地址。

在给变量 begin 赋初值 10 后，执行语句，

```
p =&begin;
```

即是把变量 begin 的内存地址放入 p。这种地址是变量的物理地址，和变量的值无关。读者可以把&看作取变量的地址，所以上句可看作取变量 begin 的地址。

语句，

```
end = *p
```

是把变量 begin 中存放的值放入到 end 中，所以，变量 end 的值为 10。由于数值 10 是放在内存 1245048（所用机器不同，地址值也不尽相同）中，所以指针 p 中放的是地址值 1245048。读者可以把*看作“在某地址处”，所以上式可看作成 end 接受 p 中所放地址处的值。

在最后输出模块中，先输出的是指针变量 p 的值（地址值），然后输出的是指针指向的值（变量 begin 的值）10。

实例9 if判断语句

实例说明

C支持两种类型选择语句，分别是if语句和switch语句。本例将介绍的是if语句。

例题所要实现的功能是任意输入三个整数，程序可按照这三个整数的大小顺序，分别输出它们。

知识要点

if语句的一般形式是：

```
if(expression) statement;
else statement;
```

其中statement可由单条语句构成，也可以是语句块，还可以什么内容也没有（空语句）。else子句（clause）是可选的。

如果表达式（expression）取真值（零之外的任何值），则执行if的目标语句或语句块。否则，执行else的目标语句或语句块。需要注意的是，if相关的代码和else相关的代码只能执行其一，两者不会同时执行。

C语言共提供了三种形式的if语句，在本例中用到了其中的两种，它们分别是：

```
(1) if (expression)  statement
```

例如：`if(a > b) printf("%d", a);`

```
(2) if (expressicn)  statement 1
    else  statement 2
```

例如：

```
if(a > b) printf("%d", a);
  else printf("%d", b);
```

第三种形式的if语句，将在下一个实例中介绍。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
        int x, y, z, mid, dec;
```

```
    printf("请任意输入三个整数：\n");
    scanf("%d %d %d", &x, &y, &z);

    if(x < y)
    {
        mid = x; x = y; y = mid;
    }
    if(x < z)
    {
        mid = x; x = z; z = mid;
    }
    if(y < z)
    {
        mid = y; y = z; z = mid;
    }

    printf("请输入一个整数，程序根据其正负判断输出：\n");
    scanf("%d", &dec);
    if(dec >= 0)    printf("最大整数为：%d\n", x);
    else    printf("最小整数为：%d\n", z);
}
```

程序分析

首先需要输入三个整数用以给变量x、y和z赋初值，然后程序将变量x和y相比较，将它们中较大的一个存入到变量x中，而另外一个存在变量y中。同样，对变量x和z也做同样的操作。这样，在变量x中存放的将是三个数中最大的一个。

做完上述工作后，接着比较变量y和z，找出两个之间的大者，存入到变量y当中，剩下的一个则被放入变量z之中。那么，存放在变量z中的就是三个数中最小的一个。

经过上述的比较工作，已完成对三个数按大小顺序排列的工作。最后，程序根据输入变量dec值的正负，判断输出三个数中的最大值还是最小值。

实例 10 else-if 语句

实例说明

在上一个实例中介绍了 if 语句的前两种形式，在这将要介绍的是 if 语句的第三种形式。

本例原题是：有一分段函数，y = f(x)。当 x 小于 6 时，y = x - 12；当 x 大于等于 6 且小于 15 时，y = 3x -1；当 x 大于等于 15 时，y = 5x + 9。本例根据原题，任意输入一整数 x，从而输出相应的 y 值。

知识要点

C 语言提供的第三种 if 语句的形式如下所示：

```
if (表达式 1)   语句 1
    else if (表达式 2)   语句 2
    else if (表达式 3)   语句 3
        ...                 ...
    else if (表达式 m)   语句 m
else  语句 n
```

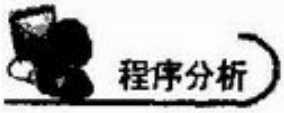
各条件自顶向下求值，当发现真值时，立即执行有关语句，跳过其后所有语句。如未发现真值，即所有测试都失败时，实施最末的 else 语句。

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>

void main()
{
        int x, y;
        printf("请输入自变量 x: ");
        scanf("%d", &x);

        if(x < 6)
        {
            y = x - 12;
            printf("x = %d, y = %d\n ", x, y);
        }
```

```
    else if(x < 15)
    {
        y = 3*x - 1;
        printf("x = %d, y = %d\n ", x, y);
    }
    else
    {
        y = 5*x + 9;
        printf("x = %d, y = %d\n ", x, y);
    }
}
```



首先给出自变量 x 的值，根据自变量 x 的值，程序将做出判断。

首先，判断 x 是否为小于 6 的整数，若是，则按照语句，

```
y = x - 12;
```

计算出因变量 y 的值，并将其输出，不再继续执行下面的判断语句，而是直接跳出 if 判断结构。在本例中则是跳出整个程序。

若 x 不是小于 6 的整数，接着会判断其是否小于 15，如果是，则会执行语句，

```
y = 3*x - 1;
```

计算出因变量 y 的值并输出，接着跳出程序。

如果判断还不满足条件，说明 x 一定是大于等于 15 的整数，程序便会按照语句，

```
y = 5*x + 9;
```

来计算 y 的值，最后同样也是输出 y 的值并结束程序。

实例11 嵌套if语句

实例说明

通过前两个实例的学习，读者对 if 语句三种形式应该有了较全面的了解。在此，还将向读者介绍一下嵌套的 if 语句。在 if 语句中又包含一个或多个 if 语句称为语句的嵌套，实际上是三种形式的 if 语句混合使用的一种情况。

本实例将要实现的功能是，根据给出的输血者的资料（包括性别和体重），程序判断输出输血者对应的输血量。

知识要点

嵌套 if 语句的一般形式如下：

```
if ()
    if ()  语句1
    else   语句2
else
    if ()  语句3
    else   语句4
```

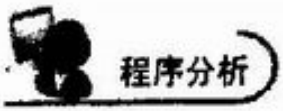
注意 if 和 else 的配对关系。从最内层开始，else 总是与它上面最近的（未曾配对的）if 配对。如果 if 与 else 的数目不一样，为实现程序设计者的意图，可以加花括号来确定配对关系。

程序源码

该应用程序的源代码如下：

```
    # include <stdio.h>

void main()
{
        // sex 代表输血者的性别，weight 代表输血者的体重，cubage 代表输血量
        int sex, weight, cubage;
        printf("请给出输血者的性别和体重：");
        scanf("%d,%d", &sex, &weight);

        if(sex >= 0)    // 若变量 sex 的数值为非负数，则表示为男性
        {
```

```
            if(weight >= 120)
            {
            cubage = 200;
                printf("此人应该输血：%d 毫升\n", cubage);
            }
            else
            {
                cubage = 180;
                printf("此人应该输血：%d 毫升\n", cubage);
            }
        }
        else    // 否则，表示为女性
        {
            if(weight >= 100)
            {
               cubage = 150;
                printf("此人应该输血：%d 毫升\n", cubage);
            }
            else
            {
                cubage = 120;
                printf("此人应该输血：%d 毫升\n", cubage);
            }
        }
}
```



程序根据输血者的性别和体重来判断他们的输血量。对于男性，体重超过 120 公斤的输血量为 200 毫升，低于 120 公斤的输血量为 180 毫升。对于女性，体重超过 100 公斤的将输血 150 毫升，否则输血量为 120 毫升。

实例 12 switch 语句

实例说明

C 语言的一个多分支选择语句是 switch 语句，这种语句把一个表达式的值和一个整数或字符常量表中的元素逐一比较，发现匹配时，与匹配常数关联的域将被执行。

本实例将使用 switch 语句解决一个有趣的小问题，问题表述如下：给出一个不多于 5 位的正整数，要求：①求它是几位数；②分别打印出每一位数字；③按逆序打印出各位数字。例如，原数为 489，应输出 984。

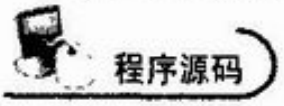
知识要点

前面介绍的 if 语句只能处理从两者间选择其一，当要实现几种可能之一时，就要用 if...else if 甚至多重的嵌套 if 来实现，当分支较多时，程序变得复杂冗长，可读性降低。C 语言提供了 switch 选择语句专门处理多路分支的情形，使程序变得简洁。

switch 语句的一般形式是：

```
switch(expression)
{
   case constant1:
      statement sequence;
      break;
   case constant2:
      statement sequence;
      break;
   case constant3:
      statement sequence;
      break;
   ……
   default:
      statement sequence;
}
```

表达式 expression 必须对整数求值，因此可使用字符或整数值，但不能使用浮点表达式。表达式 expression 的值顺序与 case 语句中的常量逐一比较，发现匹配时，与该 case 语句关联的语句序列被执行，直到遇到 break 语句或达到 switch 语句结尾时停止。若未发现匹配，则执行 default 语句。default 是可选的，若未选中，则不发生任何操作。



程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
    int num;
    // 下面定义的各变量，分别代表个位，十位，百位，千位，万位，十万位以及位数
    int indiv, ten, hundred, thousand;
    int ten_thousand, hundred_thousand, place;

    printf("请输入一个整数(0～999999)：");
    scanf("%d", &num);

    // 判断变量 num 的位数
    if(num > 99999)
        place = 6;
    else if(num > 9999)
        place = 5;
    else if(num > 999)
        place = 4;
    else if(num > 99)
        place = 3;
    else if(num > 9)
        place = 2;
    else
        place = 1;
    printf("place = %d\n", place);

    printf("每位数字为：");

    // 求出 num 在各位上的值
    hundred_thousand = num/100000;
    ten_thousand = (num - hundred_thousand*100000)/10000;
    thousand = (num - hundred_thousand*100000 - ten_thousand*10000)/1000;
    hundred = (num - hundred_thousand*100000 - ten_thousand*10000
            - thousand*1000)/100;
    ten = (num - hundred_thousand*100000 - ten_thousand*10000
          - thousand*1000 - hundred*100)/10;
    indiv = num - hundred_thousand*100000 - ten_thousand*10000
           - thousand*1000 - hundred*100 - ten*10;
```

```
    // 判断变量 num 的位数，并根据位数做出相应的输出
    switch(place)
    {
    case 1: printf("%d", indiv);
            printf("\n 反序数字为: ");
            printf("%d\n", indiv);
            break;
    case 2: printf("%d, %d", ten, indiv);
            printf("\n 反序数字为: ");
            printf("%d%d\n", indiv, ten);
            break;
    case 3: printf("%d, %d, %d", hundred, ten, indiv);
            printf("\n 反序数字为: ");
            printf("%d%d%d\n", indiv, ten, hundred);
            break;
    case 4: printf("%d, %d, %d, %d", thousand, hundred, ten, indiv);
            printf("\n 反序数字为: ");
            printf("%d%d%d%d\n", indiv, ten, hundred, thousand);
            break;
    case 5: printf("%d, %d, %d, %d, %d", ten_thousand, thousand,
                   hundred, ten, indiv);
            printf("\n 反序数字为: ");
            printf("%d%d%d%d%d\n", indiv, ten, hundred,
                   thousand, ten_thousand);
            break;
    case 6: printf("%d, %d, %d, %d, %d, %d", hundred_thousand,
                   ten_thousand, thousand, hundred, ten, indiv);
            printf("\n 反序数字为: ");
            printf("%d%d%d%d%d%d\n", indiv, ten, hundred, thousand,
                   ten_thousand, hundred_thousand);
            break;
    default: printf("Not find.\n");
            break;
    }
}
```

程序分析

本例是读者阅读本书以来所遇到的最长的一个程序，请不要惊慌，本例代码虽然较长，但在逻辑上还是比较容易理解的。

本例的流程可简单表示如下所示：

输入一个正整数（不低于五位）；判断所输入整数的位数并输出；求出整数在各位上的值；使用 switch 语句判断所输入整数的位数，然后输出整数在各位上的值；最后输出所输入整数

的反序数字。

程序中的 break 是 C 的跳转语句之一，既可用于循环当中，也可用于 switch 中。在 switch 中遇到 break 时，程序会跳转到 switch 之后的第一个代码行。

关于 switch 语句的三个要点：

（1）switch 语句和 if 语句不同，switch 只能测试是否相等，而 if 语句还能够测试关系表达式和逻辑表达式。

（2）各个 case 常数必须各异。不过，当外层 switch 中嵌套有内层 switch，那么内层 switch 中可以有同外层 case 常数相等的内层 case 常数。

（3）switch 语句中使用字符常数时，这些常数都会被自动转换成整数。

实例13 for语句

实例说明

C语言的for语句使用最灵活，不仅可以用于循环次数已经确定的情况，还可以用于循环次数不确定而只给出循环结束条件的情况。

在此程序中，我们使用了嵌套的for语句，用来在屏幕上输出一个菱形，此菱形由星号(*)组成，共九行九列。本例逻辑比较复杂，对初学for语句者在理解上可能有点困难，希望能够反复阅读本例，理清其中的逻辑关系。

此外，希望通过对此例的学习能够熟悉for语句的使用，特别是对for语句的一些特殊用法能有清楚的认识。

知识要点

for语句是循环控制结构中使用最广泛的一种循环控制语句，特别适合已知循环次数的情况。它的一般形式如下所示：

for (<表达式1>；<表达式2>；<表达式3>)

for语句很好地体现了正确表达循环结构应注意的三个问题：

（1）控制变量的初始化。

（2）循环的条件。

（3）循环控制变量的更新。

表达式1：一般为赋值表达式，给控制变量赋初值；

表达式2：关系表达式或逻辑表达式，循环控制条件；

表达式3：一般为赋值表达式，给控制变量增量或减量。

语句：循环体，当有多条语句时，必须使用复合语句。

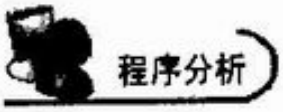
for循环的执行过程如下：

首先计算表达式1，然后计算表达式2，若表达式2为真，则执行循环体。否则，退出for循环，执行for循环后的语句。如果执行了循环体，则循环体每执行一次，都计算表达式3，然后重新计算表达式2，依此循环，直至表达式2的值为假，退出循环。

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>

void main()
{
        int i, j, k;
        // 变量 i 从 0 到 4，表示所画菱形图的第一至第五行
        for(i = 0; i <= 4; i++)
        {
            // 当行数为 i 时，空格数是 i 的函数，为 4-i 个
            for(j = 0; j <= 3-i; j++)
                printf(" ");
            // 星号数也是 i 的函数，为 2i+1 个
            for(k = 0; k <= 2*i; k++)
                printf("*");
            printf("\n");
        }
        // 变量 i 从 0 到 3，表示所画菱形图的第六至第九行
        for(i = 0; i <= 3; i++)
        {
            // 当行数为 i 时，空格数是 i 的函数，此时为 i 个
            for(j = 0; j <= i; j++)
                printf(" ");
            // 星号数也是 i 的函数，此时为 7-2i 个
            for(k = 0; k <= 6-2*i; k++)
                printf("*");
            printf("\n");
        }
}
```



本例中定义了三个整型变量 i、j 和 k，变量 i 和所画菱形的行数（第几行）相关，变量 j 代表在每行中应出现的空格数，而变量 k 代表每行中应出现的星号数。本例的关键在于找出 i 和 j，以及 i 和 k 之间的关系，根据它们之间的关系，便可顺利画出菱形图。

实例 14 while 语句

实例说明

C 语言中的第二种循环就是 while 循环，与 for 循环类似，while 循环也是在循环顶部测试条件，即循环体可能得不到执行，消除了循环之前的单独测试。

作为循环结构的重要组成部分之一，while 语句是向读者重点介绍的内容。在本例中，要求读者输入两个正整数，程序将输出它们的最大公约数和最小公倍数。

知识要点

while 语句是“当型”循环控制语句，一般形式如下所示：

```
while (condition) statement;
```

当 condition 为非零值时，执行 while 语句中的内嵌语句。它的特点是：先判断表达式，后执行语句。

注意点：

1. 循环体如果包含一个以上的语句，应该用花括弧括起来，以复合语句的形式出现。如果不加花括弧，则 while 语句的范围只到 while 后面的第一个分号处。

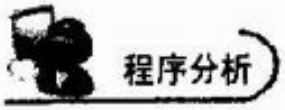
2. 在循环体中应该有使循环趋向于结束的语句。

程序源码

该应用程序的源代码如下：

```
   # include <stdio.h>

void main()
{
        int x, y, num1, num2, temp;
        printf("请输入两个正整数: \n");
        scanf("%d %d", &num1, &num2);

        if(num1 > num2)
        {
            temp = num1;    // 将两个整数互换
            num1 = num2;
            num2 = temp;
```

```
    }

    // 将求出的较大值存入 x 中，较小值存入 y 中
    x = num1;
    y = num2;
    // 求出它们的最小公约数
    while(y != 0)
    {
        temp = x%y;     // 求出 x/y 的余数
        x = y;
        y = temp;
    }

    printf("它们的最大公约数为：%d\n", x);
    printf("它们的最小公倍数为：%d\n", num1*num2/x);
}
```



在程序的起始部分，需要输入两个正整数，存入到变量 num1 和 num2 中。

接着，通过 if 判断语句，判断变量 num1 和 num2 的大小。如果出现变量 num1 小于 num2 的情况，则将它们的位置互换。经过上述的操作，在变量 num1 中便存放了所输入两个整数中的大者，而在 num2 中则存放了小的一个。

然后，程序执行了一个 while 循环。此循环的目的是找出变量 num1 和 num2 的最大公约数，并将此最大公约数存放到变量 x 中。

最后，在屏幕上输出两个整数最大公约数。根据最大公约数和最小公倍数的关系：

最大公倍数 = 原数 1 * 原数 2 / 最小公约数

输出对应的最小公倍数。

实例15 do-while 语句

实例说明

在C语句中，循环语句 do-while 是“直到型”的循环语句。在一般情况下，用 while 语句和 do-while 语句处理同一个问题时，若两者的循环部分一样，它们的结果也是一样的。但在 while 后面的表达式一开始就为假时（0 值），这两种循环的结果是不同的。

本例原题是：

计算 sin(x) = x- x3/3! + x5/5! - x7/7! + ...

直到最后一项的绝对值小于 1e-7 时为止。在本例中正是用到了 do-while 循环语句。

知识要点

“直到型”循环语句 do-while 的一般形式为：

```
do{
statement;
}while(condition);
```

其中的 statement 通常是复合的，称为循环体。

do-while 语句的基本特点是先执行后判断，所以，循环体至少被执行一次。

但需要注意的是，do-while 语句与标准的直到型循环有一个极为重要的区别，标准的直到型循环是当条件为真时结束循环，而 do-while 语句恰恰相反，它是当条件为真时循环，而一旦条件为假，立即结束循环。希望读者能够注意到 do-while 语句的这一特点。

程序源码

该应用程序的源代码如下：

```
# include <math.h>
# include <stdio.h>    // 数学函数库

void main()
{
    // 用 s 表示多项式的值，用 t 表示每一项的值
    double s, t, x;
    int n;
    printf("please input x: ");
    scanf("%lf", &x);
```

```
    // 赋初值
     t = x;
     n = 1;
     s = x;
    // 进行叠加运算
     do
     {
         n = n + 2;
         t = t * (-x*x)/((float)(n)-1)/(float)(n);
         s = s + t;
     } while (fabs(t)>=1e-8);
     printf("sin(%f) = %lf\n", x, s);
}
```

程序分析

本例我们使用递推方法来完成。

让多项式的每一项与变量n对应，n的值依次为1，3，5，7，...，从多项式的前一项算后一项，只需将前一项乘一个因子：

(-x*x)/((n-1)*n)

本例的运行结果如下所示：

```
运行
please input x: 2.369
sin(2.369000) = 0.697994
运行
please input x: 1.5753
sin(2.369000) = 0.999990
```

此外，在程序的开头包含了头文件math.h，它是一个数学函数库。函数fabs(double num)正是此函数库中的成员函数，它的功能是返回num的绝对值。

实例16 break 和 continue 语句

实例说明

有时，我们需要从循环体中提前跳出循环，或者在满足某种条件的情况下，不执行循环中剩下的语句，而从头开始新的一轮循环，这时就需要用到 break 语句和 continue 语句。

本例实现的功能是打印半径为 1 到 10 之间的圆的面积，在实例中同时用到了 break 语句和 continue 语句。希望通过实例，读者能够熟悉它们的用法及其在使用上的不同之处。

知识要点

在循环语句的执行过程中，除了通过测试从循环语句的顶部或底部正常退出外，有时从循环过程中直接退出来显得要更为方便一些。break 语句可用于从 for、while 和 do-while 语句中提前退出来，正如它可用于从 switch 语句中提前退出来一样。break 语句可以用于立即从最内层的循环语句或 switch 语句中退出。

注意：break 语句不能用于循环语句和 switch 语句之外的任何其他语句。

continue 语句与 break 语句相关，但较少用到。

continue 语句用于使其所在的 for、while 和 do-while 语句开始下一次循环。在 while 和 do-while 语句中，continue 语句的执行意味着立即执行测试部分；在 for 循环语句中，continue 语句的执行则意味着使控制传递到增量部分。continue 语句只能用于循环语句，不能用于 switch 语句。如果某个 continue 语句位于 switch 语句中，而后者又位于循环语句中，那么该 continue 语句用于控制下一次循环。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main( )
{
    int radius;    // 存放圆半径
    double area;   // 存放圆面积
    for(radius = 1; radius <= 10 ; radius++)
    {
        area = 3.1416 * radius * radius;
```

```
        // 若圆面积超过 120，则跳出 for 循环，不予输出
        if(area >= 120.0)
            break;
        printf("square = %f\n", area);
    }
    // 将最大圆面积的半径输出
    printf("now radius=%d\n\n", radius-1);

    for(radius = 1; radius <= 10 ; radius++)
    {
        area = 3.1416 * radius * radius;
        // 若圆面积没有超过 60，则不输出而是重新开始循环
        if(area < 120.0)
            continue;
        printf("square = %f\n", area);
    }
    // 将最大圆面积的半径输出
    printf("now radius=%d\n", radius-1);
}
```

程序分析

程序定义了圆面积和圆半径。在第一个 for 循环中，圆半径从 1 逐一增加到 10，程序分别输出相对应的圆面积。在 for 循环中，加入了一个 if 判断语句，语句判断圆面积是否超过 120，如果没有超出，则继续执行下面的语句，即输出圆面积；否则程序便会跳出 for 循环，输出最后一个圆面积所对应的圆半径。在这个输出模块中，输出的圆面积全部小于 120。

程序中的第二个 for 循环和第一个 for 循环基本相似，所不同的是，在第二个 for 循环中的 if 语句的执行语句是 continue。那么，此时程序所输出的便不再是面积小于 120 的圆的面积，而是输出面积大于 120 的圆的面积。一直执行到 for 循环的结尾，最后输出最大圆面积所对应的圆半径。

实例17 exit()函数

实例说明

尽管 exit()函数不是程序控制语句，但是我们还是要向读者对它作简短的介绍。与 break 语句中止循环相似，标准库函数 exit()终止程序的执行。当函数中出现 exit()时，该函数会立即结束全部程序，强制返回操作系统。exit()的作用类似于跳出整个程序。

本例是输入月数，程序则打印出 1999 年的该月有几天。在这个程序中，switch 语句是主结构，在 swtich 的分支选择语句中，用到了 exit()函数。

知识要点

exit()函数的一般形式是：

```
void exit(int return_code);
```

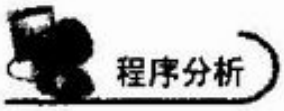
其中返回值 return_code 将送回调用过程，一般是操作系统。按照惯例，0 值一般表示正常结束，非 0 值则表示某种错误。exit()函数包含在头文件<stdlib.h>当中。

程序源码

该应用程序的源代码如下：

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

void main()
{
   // 定义变量 month 和 day 存放月数和天数
    int month;
    int day;

    printf("please input the month number: ");
    scanf("%d", &month);    // 输入月数
    switch (month)
    {
    // 当输入为 1、3、5、7、8、10 和 12 月时
case 1:
    case 3:
    case 5:
```

```
        case 7:
        case 8:
        case 10:
        case 12: day=31;
                 break;    // 每月天数都是 31 天，跳出循环
      // 当输入为 4、6、9 和 11 月时
        case 4:
        case 6:
        case 9:
        case 11: day=30;
                 break;    // 每月天数都是 30 天，跳出循环
        case 2:  day=28;
                 break;    // 二月天数为 28 天，跳出循环
      // 如果读者输入的数字不是 1～12，则会跳出程序
        default: exit(0);
        }
      // 打印出输入月数所对应的天数
        printf("1999.%d has %d days.\n", month, day);
}
```



程序分析

由于本实例的主结构是 switch 循环，而在前面的实例中，已经详细地介绍过 switch 语句的用法，所以读者应该能够独立分析出本实例的流程。在此，只向大家做简单的介绍。首先输入月数，如果月数是从 1～12 之间的整数，那么将输出月份所对应的天数。如果不是它们之间的整数，那么将执行语句 exit()，直接跳出整个程序。

需要注意的是，如果要在程序中使用函数 exit()，则必须在程序的开头包含头文件 <stdlib.h>，否则将不能够调用此函数。

实例 18 综合实例

实例说明

本例是有关循环结构的综合实例。在本例的程序中，同时包含三个循环结构，它们分别是 do-while 循环、for 循环以及 while 循环。此外，程序还含有 if 判断语句。通过对本例的学习，首先是复习一下前面所讲的知识点，其次是希望读者对这些知识点能够有更深的认识。

在实例中，我们要实现的功能是验证哥德巴赫猜想，即任一充分大的偶数，可以用两个素数之和表示。

知识要点

本例介绍的内容是程序控制结构。在 C 语言中，程序控制结构通常也叫作语句结构。从最普遍的意义上来讲，语句是可执行程序的一部分。实际上，语句说明了一种行为。

C 语言将语句分成了如下的几类：

- 选择语句（Selection）
- 重复语句（Iteration）
- 跳转语句（Jump）
- 标号（Label）
- 表达式（Expression）
- 块（Block）

选择语句包括 if 语句和 switch 语句。

重复语句包括 while 语句、for 语句和 do-while 语句。在 C 中重复语句也常被称为循环语句（loop）。

跳转语句有 break 语句、continue 语句、goto 语句（由于它很少被使用，所以在此没有介绍）以及 return 语句（在后面的实例中会碰到）。

标号语句有 case 和 default（与 switch 一起讨论），以及 label（常和 goto 语句一同使用）。

表达式语句由有效 C 表达式构成。

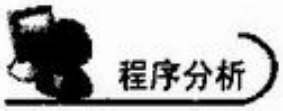
块语句就是代码块，由一对大括号包围。块语句通常也被称为复合语句。

程序源码

该应用程序的源代码如下：

```
# include <math.h>
# include <stdio.h>

void main( )
{
    int i, j, num;
    int p, q, flagp, flagq;
    printf("Please input a plus integer: ");
    scanf("%d", &num);

    // 代码 (num%2)!=0) 表示num不能被2整除
    if(((num%2)!=0) || (num<=4))
        printf("input data error!\n");
    else
    {
        p = 1;
        // do-while循环体
        do {
                p = p + 1;
                q = num - p;
                flagp = 1;
                flagq = 1;
                // for循环体
                for(i=2; i<=(int)(floor(sqrt((double)(p)))); i++)
                {
                    if((p%i) == 0)
                    {
                        flagp = 0;
                        break;
                    }
                }
                // while循环体
                j = 2;
                while(j <= (int)(floor(sqrt((double)(q)))))
                {
                    if ((q%j) == 0)
                    {
                        flagq = 0;
                        break;
                    }
                    j++;
                }
            } while (flagp*flagq == 0);
        printf("%d = %d + %d \n", num, p, q);
```

```
    }
}
```



程序分析

先不考虑怎样判断一个数是否为素数，而从整体上对这个问题进行考虑。我们可以这样做：读入一个偶数 num，将它分成 p 和 q 两部分，使得 num = p + q。那么怎样分呢？可以令 p 从 2 开始，每次增加 1，而令 q = n - p，如果 p、q 均为素数，则正为所求，否则令 p = p + q 再试。

基本算法如下：

（1）读入大于 3 的偶数 n。

（2）p = 1

（3）do {

（4）　　p = p+1; q = n-p;

（5）　　p 是素数吗？

（6）　　q 是素数吗？

（7）} while p、q 有一个不是素数。

（8）输出 n = p + q。

为了判断 p、q 是否为素数，设置了两个标志量 flagp 和 flagq，它们的初始值都为 0。若 p 是素数，令 flagp = 1，若 q 是素数，令 flagq = 1，于是第 7 步变成：

（7）} while (flagp*flagq==0);

接下来分析第 5、6 步，怎样判断一个数是否为素数。

素数就是除了 1 和它自身外，不能被任何数整除的整数。由定义可知：

2、3、5、7、11、13、17、19、23 等都是素数；

1、4、6、8、9、10、12、14、17 等都不是素数；

要判断 i 是否为素数，最简单的办法是用 2、3、4、..i-1 这些数依次去除 i，看能否除尽，若被其中之一除尽，则 i 不是素数，反之，i 是素数。

其实没必要用那么多的数去除，实际上，用反证法很容易证明。如果小于等于 i 的平方根的数都除不尽，则 i 必是素数。于是，上述算法中的第 5、6 步可以细化为：

（5）p 是素数吗？

```
flagp = 1;
for(i=2; i<=(sqrt(p)); i++)
if p 除以 i 的余数为 0
{ flagp=0;
break; }
```

（6）q 是素数吗？

```
flagq = 1 ;
```

```
for(j=2; j<=(sqrt(q)); j++)
if q除以j的余数为0
{ flagq=0;
break; }
```

最后程序运行结果如下所示：

运行

```
please input a plus integer : 10
10 = 3 + 7
```

运行

```
please input a plus integer : 98
98 = 19 + 79
```

运行

```
please input a plus integer : 39
input data error!
```

此外，程序中用到了 floor(double num)函数，它包含在<math.h>的头文件中。函数 floor(double num)的功能是返回不大于 num 的最大整数（表示为浮点值）。例如，给定 1.08 时，floor()返回 1.0；若给定-2.3，floor()返回-3.0。

实例19 一维数组

实例说明

迄今为止，我们使用的都是基本类型（整形、字符型和实型）的数据，除此之外，C语言还提供了构造类型的数据，它们有：数组类型、结构类型和公用体类型。构造类型数据实际上是由基本类型数据按一定规则组成的，所以有时也称它们为导出类型。

下面，向读者介绍的正是构造类型中的数组类型。本例是使用选择法对十个整数进行排序。

知识要点

一维数组的一般说明形式如下：

```
type-specifier var_name[size];
```

在C语言中，数组必须显式说明，以便编译程序为它们分配内存空间。在上式中，类型说明符（type-specifier）指明了数组的类型，也就是数组中每一个元素的类型，而size则指明了数组的大小，即数组中元素的个数。

一维数组的总字节数可按下式计算：

sizeof（类型）* 数组长度 = 总字节数

程序源码

该应用程序的源代码如下：

```
// 使用选择法排序
# include <stdio.h>

void main()
{
    int i, j, min, temp;
    // 定义一个整型的一维数组
    int array[10];
    // 输入数据
    printf("Please input ten integer: \n");
    for(i=0; i<10; i++)
    {
        printf("array[%d] = ", i);
        scanf("%d", &array[i]);
```

```
    }
    printf("The array is: ");
    for(i=0; i<10; i++)
        printf("%d ", array[i]);
    printf("\n");
    // 排序
    for(i=0; i<9; i++)
    {
        min = i;
        for(j=i; j<10; j++)
            if(array[min]>array[j]) min = j;
        temp = array[i];
        array[i] = array[min];
        array[min] = temp;
    }
    // 输出
    printf("\nThe result: \n");
    for(i=0; i<10; i++)
        printf("%d ", array[i]);
    printf("\n");
}
```

程序分析

选择排序法的原理是：从一堆数中先找出最小的，将其抽出；然后在余下的数中再找出最小的，抽出并放在前一个抽出数的后面；依次执行，最后完成排序。

注意：在C语言中，所有数组都把0作为首元素的下标。因此，当书写：

int array[10];

时，实际上是声明了 array[0]到 array[9]这十个元素。

实例20 二维数组

实例说明

在 C 语言中允许使用多维数组，最简单的多维数组是二维数组。从本质上说，二维数组是以一维数组为元素构成的数组。

本例要求打印出一个魔方阵，所谓魔方阵指的是每一行、每一列以及对角线之和均相等。程序中的方阵使用二维数组来表示。希望读者通过对此例的学习，能够理清一维数组和二维数组的之间的关系。

知识要点

二维数组定义的一般形式如下所示：

```
type-specifier var_name[size][size];
```

例如：

```
int array1[5][10], float array2[2][4];
```

请留心上面的说明语句，C 不像其他大多数计算机语言那样使用逗号区分下标，而是用方括号将各维下标括起。此外，二维数组的下标和一维数组一样，均是从 0 开始。

二维数组以行—列矩阵的形式存储。第一个下标代表行，第二个下标代表列，这意味着按照在内存中的实际存储顺序访问数组元素时，右边的下标比左边的下标的变化快一些。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
    int array[16][16];
    int i, j, k, m, n;

    // 变量初始化
    m = 1;
    while(m == 1)
    {
        printf("请输入 n(0<n<=15), n 是奇数)\n");
```

```
        scanf("%d", &n);
        // 判断n是否为大于0小于等于15的奇数
        if((n!=0) && (n<=15) && (n%2!=0))
        {
            printf("矩阵阶数是 %d\n", n);
            m = 0;
        }
    }
    // 数组赋初值为0
    for(i=1; i<=n; i+-)
        for(j=1; j<=n; j++)
            array[i][j] = 0;

    // 建立魔方阵
    j = n/2 + 1;
    array[1][j] = 1;
    for(k=2; k<=n*n; k++)
    {
        i = i - 1;    // 行数减少1
        j = j + 1;    // 列数增加1
        if((i<1) && (j>n))
        {
            i = i + 2;    // 行数增加2
            j = j - 1;    // 列数减少1
        }
        else
        {
            if(i < 1)    // 如果行数小于1，则跳转到第n行
                i = n;
            if(j > n)    // 如果列数大于n，则回到第1列
                j = 1;
        }
        if(array[i][j] == 0)
            array[i][j] = k;
        else
        {
            i = i + 2;
            j = j - 1;
            array[i][j] = k;
        }
    }

    // 输出魔方阵
    for(i=1; i<=n; i++)
```

```
    {
        for(j=1; j<=n; j++)
            printf("%5d", array[i][j]);
        printf("\n");
    }
}
```

程序分析

魔方阵中各个数的排列规律如下所示：

（1）将 1 放在第一行中间的一列；

（2）从 2 开始直到 n*n 为止，各数依次按照下列规则存放：每一个数字存放的行比前一个数的行数减一，列数加一；

（3）如果上一个数的行数为 1，则下一个数的行数为 n（即最下一行）；

（4）当上一个数的列数为 n 时，下一个数的列数应该 1，行数减一；

（5）如果按照上面规则确定的位置上已有数，或者上一个数是第 1 行第 n 列时，则把下一个数放在上一个数的下面。

说明：魔方阵最大阶数只能为 15，所以定义数组 array 为 16 行 16 列，对第 0 行第 0 列不存放数据，而是从第 1 行第 1 列开始存放数，以符合读者的习惯。

实例21 字符数组

实例说明

程序设计中经常要用到字符数组，即用来存放字符数据的数组。字符数组中的一个元素存放一个字符。因为字符数组是 C 程序中最常用的数据类型，对它有些例外的约定，所以在此专门讨论。

本例一个以字符数组为基础做简单的文本编辑器，该程序输入文本行直至遇到一个空行为止，而后每次一个字符重新显示各行。

知识要点

在 C 语言中，文字信息作为字符串处理，而字符串又以字符数组的形式组织。字符数组需预先指定长度，以保证能存放足够长的文字信息。

字符数组的定义形式和其他数组的定义形式一样，例如：

```
char a_line[120];
```

表示数组 a_line[]有 120 个成分，每个成分都是字符型，能存放一个字符。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>
// 宏定义
# define MAX 100
# define LEN 80

// 一个非常简单的文本编辑器
void main()
{
    char text[MAX][LEN];
    register int t, i, j;    // 定义三个寄存器变量
    // 逐行输入字符串
    for(t=0; t<MAX; t++)
    {
        printf("%d: ", t);
        gets(text[t]);    // 输入函数
        if(!text[t][0])
```

```
            break;  // 空行退出
        }

        // 按行，逐个字符输出字符串
        for(i=0; i<t; i++)
        {
            for(j=0; text[i][j]; j++)
                putchar(text[i][j]);    // 输出函数
            putchar('\n');
        }
    }
```

程序分析

程序首先定义了一个字符类型的二维数组，然后定义了三个整型的寄存器变量。之所以要将它们定义成寄存器类型，是因为在程序中多次反复调用了它们，将它们定义成寄存器类型后，便可以在寄存器中直接调用了，而不用多次访问内存，因此节省了大量时间。

接着程序执行了第一个 for 循环，让读者逐行输入字符串，并存入到刚定义的二维字符型数组中，在读者输入空行后，程序跳出第一个 for 循环，随即又进入到第二个 for 循环中，程序会逐行的输出读者刚才所输入的内容（先后顺序不便），最后跳出程序。

在程序中，用到了函数 gets()，它的原型如下所示：

```
char *gets(char *str);
```

函数 gets()从 stdin 中读字符串，读入结果存放到 str 指向的字符数组中，一直读到接收到新行符或 EOF（文件结束）为止。新行符不作为读入串的内容，读入的新行符变成 NULL 值，由此结束字符串。如果成功，gets()返回 str；如失败，返回空指针。

函数 putchar 的作用是向终端输出一个字符，例如：

```
putchar(c)
```

输出字符变量 c 的值，c 可以是字符型变量或整形变量。

实例22 数组初始化

实例说明

C语言允许在声明数组的同时对其进行初始化，由于一维数组、二维数组以及无尺寸数组的初始化规则不尽相同，所以有必要将这部分内容单独提出并作详细说明。

本实例将会对上述的各种情况都做出详细说明，希望读者能够掌握相关的知识点。

知识要点

数组初始化的一般形式和其他变量类似，形式如下所示：

```
type_specifer array_name[size1]…[sizeN] = {value_list};
```

value_list 是逗号分隔的常量表，常量类型必须与 type_specifer 兼容。第一个常量放入数组的第一个位置，第二个常量放入数组的第二个位置，依此类推。注意语句最后有一个分号(;)。

一般数组的初始化示例如下：

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

这是用 1 到 10 初始化 10 元素的整形数组，integer[0]的值为 1，integer[9]的值为 10；

放字符串的字符数组允许如下形式的初始化：

```
char array_name[size] = "string ";
```

例如，下面的代码是把 string 初始化为"How are you!"

```
char string[13] = "How are you!";
```

它和以下的书写形式等价，

```
char string[13] = { 'H','o ','w',' ','a ', 'r ','e ',' ', 'y','o','u','\0'};
```

由于 C 的所有串都以 null 符结尾，所以定义字符串数组必须留出一个存放 null 的位置。

此外，还可以写成：

```
char string[] = "How are you!";
```

这便是无尺寸数组的初始化。

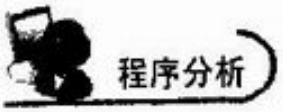
当用手工计算字符个数来决定正确的数组大小时，工作冗长且易出错，而使用无尺寸数组，C 自动完成计算数组大小的工作。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>
```

```
void main()
{
    int  array1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    char array2[13] = "How are you!";
    char array3[13] = {'H','o','w',' ','a','r','e',' ','y','o','u','!'};/
    int array4[4][4] =
    {
        12, 18,  6, 25,
       23, 10, 32, 16,
        25, 63,  1, 63,
         0,  0, 27, 98
    };
    char array5[] = "How are you!";
    int array6[][2] = {
        { 1,50},
        {45, 2},
        { 2, 0},
        {12,32},
        {42,33},
        {15,18}
    };
}
```



程序中，对各种类型的数组初始化都举出了实例，其中数组 array1、array2 和 array5 是完全相同的。数组 array6 的初始化属于无尺寸数组的初始化，希望读者能够注意其初始化形式，对于二维无尺寸数组的初始化，数组第二个下标的尺寸是必须给出的。

实例23 数组应用

实例说明

本例是一个有关数组应用的综合实例。

本程序是一个简易的学生成绩查询系统。通过这个例子，希望读者能够清楚，当涉及二维数组时，通常用两重 for 循环存取元素。

知识要点

在本例当中，分别用到了循环结构中的 do-while 语句和 for 语句，以及选择结构中的 switch 语句，这些都是对前面内容的学习。此外，通过对本例的学习，读者也能够对循环部分的知识有一个全面的复习和总结。

程序源码

该应用程序的源代码如下：

```
// 学生成绩查询系统
# include <stdio.h>
# include <stdlib.h>

void main( )
{
    int select;
    int i, j;
    int score[5][7];
   // 赋初值
    int average = 0;
    int sum = 0;
    do{
        printf("本程序有 4 项功能: \n");
        printf(" 1. 根据学号查询学生成绩\n");
        printf(" 2. 根据考试号统计成绩\n") ;
        printf(" 3. 根据考试号和学号查询成绩\n");
        printf(" 4. 成绩录入\n");
        printf(" 0. 退出\n");
        printf(" 请输入选择(0 - 4): ");
        scanf("%d", &select);
```

```
        switch(select)
        {
        case 0:
            printf("OK\n");
            exit(0);
            break;
        case 1:
            printf("输入学号: ");
            scanf("%d\n", &i);
            for(j=1; j<7; j++)
            {
                printf("第%d科成绩是%d\n", j, score[i][j]);
                sum += score[i][j];
            }
            average = sum/6;
            printf("学生的平均成绩是%d\n", average);
            break;
        case 2:
            printf("输入考试号: ");
            scanf("%d\n", &j);
            for(i=1; i<5; i++)
            {
                printf("第%d号学生本科成绩是%d\n", i, score[i][j]);
                sum += score[i][j];
            }
            average = sum/4;
            printf("本科平均成绩是%d\n", average);
            break;
        case 3:
            printf("输入学号和考试号: ");
            scanf("%d %d\n", &i, &j);
            printf("第%d号学生的第%d科考试成绩是%d\n", i, j, score[i][j]);
            break;
        case 4:
            printf("请输入成绩\n");
            for(i=1; i<5; i++)
                for(j=1; j<7; j++)
                    scanf("%d\n", &score[i][j]);
            break;
        default:
            break;
        }
    }while(1);
}
```

程序分析

本例将要完成的具体功能如下所示：

（1）根据输入的学生学号给出各次考试成绩及平均成绩；

（2）根据输入考试的次数打印出该次考试中每个学生的成绩，并给出平均分；

（3）根据学号查出学生某次考试成绩；

（4）录入考试成绩，如下所示。

考试学号	1	2	3	4	5	6
1	85	78	82	96	45	83
2	74	58	66	85	69	81
3	96	61	61	89	55	83
4	66	78	99	100	51	78

程序首先需要读者选择（4），将学生的成绩分别录入，方可进行其他操作。

实例 24 函数的值调用

实例说明

在 C 语言中，函数是程序的基本组成单位，因此可以很方便地用函数作为程序模块来实现 C 语言程序。在以下的几个实例中，读者将会对函数作系统的学习。

在计算机语言中，向子程序传递变元的方法有两种。在本例中，介绍的是第一种，称为值调用，即是把变元（实参）复制到形参中。在这种情况下，形参的修改对变元没影响。

知识要点

函数的一般形式为：

```
ret-type function-name(parameter list)
{
   body of the function
}
```

其中，ret-typ 说明函数返回的数据类型可以是除数组外的任何类型。parameter list（参数表）是用逗号分割的一系列变量名字和它们相关类型的表，参数接受该函数调用时由调用者传入变元值，函数也可以没参数，即参数表为零。此时须将关键词 void 放在括号中来说明指定空参数表。

函数的参数必须分别声明，且类型和变量名成对出现，这一点希望读者注意。

在函数进行值调用时，进行的是单向传递，即只是将变量的值传入到参数中，而参数的值不会再回传到变量中，无论参数的值是否变化。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

// 子函数声明
int square(int x);
int cube(int y);

void main()
{
    int m = 12;
```

```
    int n = 4;
    printf("%d %d\n", square(m), m);
    printf("%d %d\n", cube(n), n);
}

// 定义子函数
int square(int x)
{
    x = x * x;    // 求参数的平方根
    return x;
}

int cube(int y)
{
    y = y * y * y;    // 求参数的立方根
    return y;
}
```

程序分析

本例中除主函数 main()外还含有两个子函数，分别是 square()和 cube()。子函数 square 的功能是求出参数的平方并返回，而子函数 cube()的功能是求出参数的立方并返回。

在本程序中，由于子函数的定义是在主函数之后完成的，所以我们在主函数之前对它们作出声明，否则主函数在对子函数进行调用时会出现错误。当然，读者也可以在主函数之前就完成对子函数的定义，这样，我们便不再需要对子函数作出声明。

由于，在本例中的两个子函数在程序中的地位和用法是相同的，所以在此只对其中之一 cube()函数作出详细讲解。程序传入 cube()的变元值 4 复制到参量 y 中。当执行符值语句

y = y * y * y 时，只修改了局部变量 y。调用子函数 cube()时，使用的变量 y 值不变，仍然保存为 4，而子函数返回值是参数 y 的立方，即为 64，所以输出的是 64　4。

切记，传入函数的是变元值的复制，函数内的操作不影响调用时使用的变元变量。

实例 25 函数的引用调用

实例说明

在 C 语言中，向子程序传递变元的方法除了值调用外，便是引用调用。引用调用就是把变元的地址复制到子程序的形参中，子程序通过该地址访问实际变量。这样，通过参数进行的修改，便可以影响到子程序调用的变元值。

在本例中，子程序传递变元时便是采用了引用调用的方法。此时若是改变了函数参数的值，那么原变量的值也会跟着改变。

知识要点

虽然 C 参数传递的规范是值调用，但通过传递变元到指针可以生成引用调用。因为变元地址传入了函数，所以函数内的代码可以改变函数外的变元的值。

指针像其他值一样可传入到函数中，复制进相应的形参中。当然，形参必须声明成指针类型。例如本例中的程序，函数 swap()交换两个整数变元的值，示范怎样生成引用调用。

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

   temp = *x;     // 存储变量 x 的值
   *x = *y;       // 将 y 的值放入到 x 中
   *y = temp;     // 将 x 的值放入到 y 中
}
```

函数 swap()可以改变由 x 和 y 指向的两个变量的值，因为传递的是它们的地址而不是它们的值。这样，在该函数中，便可以使用标准的指针操作来访问这些变量的内容。

调用使用指针参数的函数时，必须使用变元地址。在下面的程序中，将向读者示范正确调用函数 swap()的方法。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void swap(int *x, int *y);
```

```
void main()
{
    int i, j;

    i = 12;
    j = 36;

    printf("i and j before swapping: %d %d\n", i, j);

    swap(&i, &j);    // 传递变量 i 和 j 的地址

    printf("i and j after swapping: %d %d\n", i, j);
}

void swap(int *x, int *y)
{
    int temp;

    temp = *x;    // 存储变量 x 的值
    *x = *y;      // 将 y 的值放入到 x 中
    *y = temp;    // 将 x 的值放入到 y 中
}
```

程序输出如下所示：

```
i and j before swapping: 12 36
i and j after swapping: 36 12
```

在此程序中，i 和 j 分别赋值 10 和 20，用 i 和 j 的地址调用子函数 swap()后，一元操作符“&”取变量的地址，这样变量 x 和 y 的地址被传入到函数 swap()中。

实例26 数组函数的调用

实例说明

在上面的例子中，我们已经对数组进行了讨论。但由于向函数传递数组是对标准值调用的一个例子，所以在这里我们再次讨论它。

当数组作为参数时，一般有两种情况，第一种情况是数组元素作为函数实参，在这种情况下，和一般变量作为函数参数区别不大；第二种情况是用数组名作为函数参数，此时实参和形参都应该用数组名或者数组指针。

在本实例中，编写了一个函数，使给定的一个二维数组转置，即行和列的互换。以数组名作为函数参数是我们要介绍的重点。

知识要点

当数组作为函数变元时，传入函数的只是数组的地址，这是 C 语言的标准调用规则的一个例外。在这种情况下，函数中的代码是对实际数组操作的，如果发生修改，修改的是实际数组的实际内容。

此外，还需要读者注意三点：

（1）用数组作为函数参数时，应该在主调函数和被调函数中分别定义数组。例如本例中，array 是实参数组名，而 element 是形参数组名，它们应该分别在所在的函数中声明，而不能只在一方定义。

（2）实参数组和形参数组的的类型应该一致，如果不一致，结果将会出错。

（3）实参数组和形参数组大小可以一致也可以不一致，C 编译对形参数组大小不做检查，只是将实参数组的首地址传给形参数组。不过，如果要求形参数组得到实参数组全部的元素值，则应当指出形参数组和实参数组大小一致。

程序源码

该应用程序的源代码如下：

```
// 矩阵的转置
# include <stdio.h>
# define N 3

// 转置函数声明
```

```
void convert(int element[N][N]);

void main()
{
    // 定义一个二维数组
    int array[N][N];
  int i, j;
    // 给数组赋初值
    printf("输入数组元素: \n");
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            scanf("%d", &array[i][j]);
    printf("\n 数组是: \n");
    for(i=0; i<N; i++)
    {
        for(j=0; j<N; j++)
            printf("%5d", array[i][j]);
        printf("\n");
    }

    // 对数组进行转置工作
    convert(array);
    printf("转置数组是: \n");
    for(i=0; i<N; i++)
    {
        for(j=0; j<N; j++)
            printf("%5d", array[i][j]);
        printf("\n");
    }
}

// 转置函数定义
void convert(int element[N][N])
{
    int i, j, t;
    for(i=0; i<N; i++)
        for(j=i+1; j<N; j++)
        {
            t = element[i][j];
            element[i][j] = element[j][i];
            element[j][i] = t;
        }
}
```

程序分析

本例的运行结果如下所示：

输入数组的元素：

1（回车）

2（回车）

3（回车）

4（回车）

5（回车）

6（回车）

7（回车）

8（回车）

9（回车）

数组是：

1　2　3

4　5　6

7　8　9

转置数组是：

1　4　7

2　5　8

3　6　9

实例27 命令行变元

实例说明

有时，我们需要向程序传入信息。这时，是通过命令行变元向主函数 main()传递信息的。命令行变元（command line argument）是操作系统命令中执行程序名字之后的信息。例如，在编译 C 程序时，在提示符下键入下面这样的内容：

```
cc program_name
```

其中，program_name 是命令行变元，它指定准备编译的程序名。

在本实例中，将使用命令行变元编写一个小程序 countdown。程序对来自命令行的值连续减量，达到零时产生蜂鸣。

知识要点

在 C 中，有两个内设参量用于接受命令行变元，一个是 argc，另一个是 argv。Argc 是整型变量，其中放置命令行中变元的总数，由于程序名本身也计算在内，所以 argc 的值至少为 1。形参 argv 是指针，指向由字符串指针组成的数组，数组中每一个元素指向一个命令行变元。

所有的命令行变元都是字符串，任何数字都将由程序转换成相应的二进制格式。

必须正确地声明 argv。最常用的方法是：

```
char *argv[];
```

一对方括号说明无尺寸数组。此时可用下标访问 argv 的元素。例如，argv[0]指向第一个串，它总是程序的名字；argv[1]指向第二个串，是一个命令行变元，依此类推。

通常，程序通过 argc 和 argv 取得操作者的初始命令。例如，命令行变元常用来指定文件名、选项或可选的行为。命令行变元使程序显出专业水平，便于程序在批命令文件中使用。

argc 和 argv 的取名来自于惯例，实际上也可以取其他的名字。

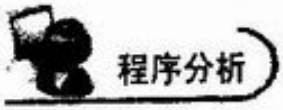
当应用程序不需要命令行参数时，一般都用参数表中的关键字 void 把 main()声明成无参数，即是在 main 的括号中使用 void 关键字，或者省略不写。

程序源码

该应用程序的源代码如下：

```
// Countdown program.
# include <stdio.h>
# include <stdlib.h>
```

```
# include <ctype.h>
# include <string.h>
void main(int argc, char* argv[])
{
    int disp, count;
    if(argc < 2)
    {
        printf("You must enter the length of the count\n");
        printf("on the command line. Try again\n");
        exit(1);    // 非正常跳出程序
    }
    if(argc==3 && !strcmp(argv[2], "display"))
        disp = 1;
    else
        disp = 0;
    for(count = atoi(argv[1]); count; --count)
        if(disp)
            printf("%d\n", count);
    putchar('\a');    // 将产生蜂鸣
    printf("Down");
    return;
}
```



程序中使用了函数 atoi()，它包含在头文件<stdlib.h>中。函数的使用形式如下：

```
# include <stdlib.h>
int atoi(const char *str);
```

函数 atoi()把 str 指向的串转换为整型值。串中必须含合法的整形数，否则返回的值无定义。串中的整数内容可由任何不是该整数的一部分的字符终止，如空白符、标点符或字符等。也就是说，用 123.23 调用 atoi()时，返回整形值 123，忽略 0.23。

在程序中，如果串"display"作为命令行的第二个变元，则减量过程显示在屏幕上。

注意：未使用任何命令行变元时，程序会给出一条错误信息。实践中，需要命令行变元的程序，常在用户未给出变元时显示出相应操作的提示。

实例 28 函数的返回值

实例说明

除了 void 类型之外，所有函数都返回一个值，返回值可以由 return 语句实现。非 void 函数必须使用有返回值的 return 语句。如果指定函数返回值，则其中的任何 return 语句必须有与之相关的值。然而，如果执行到达非 void 函数的结尾（遇到函数结束的花括号）时，则返回无用值。

在本例中，函数 find_substr()返回串内一字符串的开始位置或未发现匹配时返回-1。为了简化程序代码这里使用了两个 return 语句。

知识要点

有关函数的返回值，有以下几点说明：

（1）函数的返回值是通过函数中的 return 语句获得的。return 语句将被调用函数中的一个确定值带回主调函数中。一个函数中可以有一个以上的 return 语句，执行到哪一个 return 语句，哪一个语句起作用。

（2）函数的返回值应当属于某一个确定的类型，应当在定义时指定函数值的类型。C 语言规定，凡不加类型说明的函数，一律自动按整数类型处理。

（3）如果函数值的类型和 return 语句中的表达式不一致，则以函数类型为准。对数值类型数据，可以自动进行类型转换。

（4）如果被调用函数中没有 return 语句，并不表示函数不带回值，只是不带回有用的值，带回的是一个不确定的值。

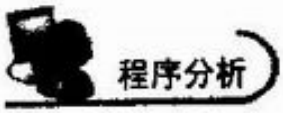
（5）为了明确地表示“不带回值”，可以用 void 定义“无类型”。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

int find_substr(char* s1, char* s2);

void main()
{
```

```
    if(find_substr("C is fun", "is") != -1)
        printf("Substring is found.\n");
}

// 定义子函数
int find_substr(char* s1, char* s2)
{
    register int t;
    char *p, *p2;

    for(t=0; s1[t]; t++)
    {
        p = &s1[t];
        p2 = s2;

        while(*p2 && *p2==*p)
        {
            p++;
            p2++;
        }
        if(! *p2)
            return t;
    }
    return -1;
}
```



本程序的大部分内容都是在子函数 find_substr 中完成的，子函数 find_substr 的功能是返回在第一个参数内，第二个参数的开始位置；如果在第一个参数中没有发现第二个参数，则返回-1。

实例29 函数的嵌套调用

实例说明

C语句不能嵌套定义函数，但可以嵌套调用函数，也就是说，在调用一个函数过程中有调用另一个函数。

我们将使用函数的嵌套调用功能完成对一个一元三次方程的求解。在本实例中需要求解的一元三次方程如下所示：

$$x^3-8x^2+12x-30=0$$

知识要点

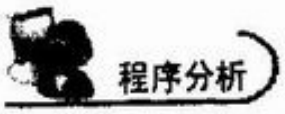
C语言的函数定义都是互相平行的、独立的，也就是说在定义函数时，一个函数不能包含另一个函数，这是C语言和PASCAL语言不同的地方（在PASCAL中，允许在定义一个函数时，其函数体内又包含另一个函数的完整定义，这就叫做嵌套定义。这个内嵌的函数只能被包含它的函数调用，其他函数都不能调用）。

程序源码

该应用程序的源代码如下：

```
// 用弦截法求解方程的根
# include <stdio.h>
# include <math.h>

// 定义函数 f，以实现 x^3-8x^2+12x-30=0
float f(float x)
{
    float y;
    y = ((x-8.0)*x+12.0)*x - 30.0;
    return y;
}

// 定义函数 xpoint，求出弦与 x 轴的焦点横坐标
float xpoint(float x1, float x2)
{
    float y;
    y = (x1*f(x2)-x2*f(x1)) / (f(x2)-f(x1));
```

```
    return y;
}

// 定义函数root，求解区间(x1,x2)的实根
float root(float x1, float x2)
{
    float x, y, y1;
    y1 = f(x1);
    do
    {
        x = xpoint(x1, x2);
        y = f(x);
        if(y*y1 > 0)    // f(x)和f(x1)同符号
        {
            y1 = y;
            x1 = x;
        }
        else
            x2 = x;
    } while(fabs(y) >= 0.0001);
    return x;
}

// 主函数体
void main()
{
    float x1, x2, f1, f2, x;
    do
    {
        printf("Please input x1, x2:\n");
        scanf("%f, %f", &x1, &x2);
        f1 = f(x1);
        f2 = f(x2);
    } while(f1*f2 > 0);
    x = root(x1, x2);
    printf("A root of equation is %9.6f\n", x);
}
```



程序分析

本实例是用弦截法求解一元三次方程的，具体方法如下：

第一步：取两个不同点x1和x2，如果f(x1)和f(x2)符号相反，则(x1,x2)区间内必然有一个根。如果f(x1)和f(x2)符号相同，则应该改变x1和x2，直到f(x1)和f(x2)符号相异为

止。读者请注意，为保证在区间(x1,x2)内只有一个根，x1 和 x2 的值不应相差太大。

第二步：连接 f(x1)和 f(x2)两点，由于 f(x1)和 f(x2)符号相反，连线必然和 x 轴有一个交点。交点的横坐标可用下式求出：

$$x=\frac{x_1 f(x_2)-x_2 f(x_1)}{f(x_2)-f(x_1)}$$

再从 x 求出 f(x)。

第三步：若 f(x)和 f(x1)同符号，则根必然在区间(x1,x2)中，此时将 x 作为新的 x1。如果 f(x)和 f(x1)同号，则表示根在(x1,x)区间中，将 x 作为新的 x2。

第四步：重复上述的步骤 2 和 3，直到 $|f(x)<\varepsilon|$为止，ε是一个很小的数，例如在本程序中取为 10^{-4}，此时近似认为 f(x)=0。

程序运行结果如下所示：

```
Please input x1, x2:
6,8
A root of equation is 6.890316
```

实例 30 函数的递归调用

实例说明

C 语言的特点之一就是允许函数的递归调用。

本例将使用递归的方法将一个整数 n 转换成为字符串。例如输入 256，应该输出字符串“256”，n 的位数不确定，可以是任意位数的整数。

知识要点

在调用一个函数的过程中出现直接或间接的调用该函数本身，称为函数的递归调用。函数的递归调用有直接和间接之分。直接的递归如下所示：

```
int f(int x);
{int y, z;
……
     z = f(y);
       ……
     return (2*z);
}
```

即在调用函数 f 的过程中又要调用 f 函数。

间接调用如下所示：

```
int f1(int x)          int f2(int a)
{int y, z;             {int b, c;
  ……                   ……
 z = f2(y);             c = f1(b);
  ……                   ……
 return (2*z);          return (11+c);
}                      }
```

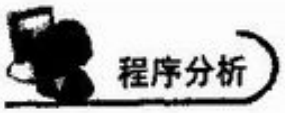
即在调用 f1 函数的过程中调用函数 f2，而在调用 f2 函数的过程中调用函数 f1。

从上面看，这两种函数的调用都是无终止的自身调用。显然，在程序中是不应该出现这种情况的，而只应该出现有限次数的、有终止的递归调用，这里可以使用 if 语句来控制，只有在某一条件成立时才继续执行递归调用，否则跳出。

程序源码

该应用程序的源代码如下：

```
// 递归法将整数转换成字符
# include <stdio.h>

void convert(int n)
{
    int i;
    if((i=n/10) != 0)
        convert(i);
    putchar(n%10+'0');
}

void main()
{
    int number;
    printf("输入整数: ");
    scanf("%d", &number);
    printf("输出是: ");
    if(number < 0)
    {
        putchar('-');
        number = -number;
    }
    convert(number);
    putchar('\n');
}
```



递归函数将参数以字符串的形式输出，输出大致过程如下：

如果整数是负数，则先输出负号。然后，程序是按照这样的顺序输出的，先找出整数的最高位，将其转换成字符输出，依次以字符型输出下一位。

实例 31 局部和全局变量

实例说明

从变量的作用域原则出发，可以将变量分为局部变量和全局变量。在本实例中将向读者介绍局部变量和全局变量的使用，希望读者通过本例的学习，能够对 C 程序中的局部变量和全局变量作出明确的区分，并熟悉它们的使用规则。

知识要点

下面将要向读者详细介绍局部变量和全局变量。

在一个函数内部定义的变量是内部变量，它只在本函数的内部有效，而在此函数的外部不能使用这些变量，通常也称内部变量为局部变量。例如：

```
void fun(int a, int b)    // 子函数 fun
{int c, int d;
……
}    // 变量在子函数 fun 内有效
void main()    // 主函数
{int x, y, z;
……
}    // 变量在主函数内有效
```

对于局部变量有几点说明如下所示：

（1）主函数 main()中定义的变量 x、y 和 z 也只有在主函数中有效，而不是因为在主函数中定义而在整个文件或程序中都有效。主函数也不能使用其他函数中定义的变量。

（2）不同函数中的变量可以起相同的名字，它们代表不同的对象，互不干扰。

（3）形式参数也属于局部变量。

（4）在一个函数的内部，可以在复合语句中定义变量，这些变量也只在本复合语句中有效，这种复合语句称之为分程序或程序块。如下所示的变量 z 只在复合语句（分程序）内有效，离开该复合语句该变量无效，释放内存。

```
void main()
{int x, y;    // 变量 x 和 y 在此范围（指第一对大括号）内有效
……
    {int z;    // 变量 z 只在此范围（指第二对大括号）内有效
        z = x * y;
        ……
}
```

```
}
```

在函数内部定义的变量称之为局部变量，而在函数外部定义的变量则称之为外部变量，外部变量就是全局变量。全局变量可以为本文件中其他函数所共用。它的有效范围是从定义变量的位置开始到源文件结尾。如下所示：

```
int m = 5, n = 10;    // 外部变量
void fun(int a, int b)    // 定义函数 fun
{int c, int d;
 ……
}
int p = 4, int q = 8;    // 外部变量
void main()    // 主函数
{int x, y, z;
 ……
}
```

在上面的例子中，m、n、p、q 都是全局变量，但是它们的作用范围不同，在 main 函数中可以使用全局变量 m、n、p、q，但在子函数 fun 中只能使用全局变量 m 和 n，而不能使用 p 和 q。

在一个函数中，既可以使用本函数中的局部变量，也可以使用有效的全局变量。

对全局变量的使用有如下几点说明：

（1）全局变量的使用增加了函数间数据联系的渠道。

（2）建议不要随便使用全局变量，因为全局变量的使用极易出错。

（3）如果全局变量在文件的开头定义，则在整个文件范围内都可以使用它们，如果不是在文件开头的部分定义，则按上面的规定作用范围只限于定义点到文件终了。

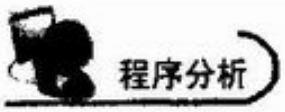
（4）如果在同一个源文件中，全局变量和局部变量通明，则在局部变量的作用范围内，外部变量不起作用。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

int count;    // count 是全局变量
vcid funcl();    // 函数声明
void func2();

void main()
{
    count = 100;
    funcl();
}
```

```
void func1()    // 函数定义
{
    int temp;    // temp 是局部变量
    temp = count;
    func2();
    printf("   count is %d\n", count);    // 打印 100
    func2();
}

void func2()
{
    int count;    // 定义局部变量 count
    for(count = 1; count < 20; count++)
        printf(".");    // 打印出"."
    printf("\n");
}
```



在仔细研究过此程序后，可以发现变量 count 既不是在 main()中也不是 func1()中定义的，但两者都可以使用它。在函数 func2()中也定义了一个局部变量 count。当 func2()访问 count 时，它仅访问自己定义的局部变量 count，而不是那个全局变量 count。

切记，全局变量和某一函数的局部变量同名时，该函数对该名的所有访问仅针对局部变量，对全局变量无影响，这是很方便的。然而，如果忘记了这点，即使程序看起来是正确的，也可能导致运行时的奇异行为。

实例32 变量的存储类别

实例说明

上一个实例介绍了变量可以分为局部变量和全局变量。从另一个角度，即从变量值存在的时间角度来分，又可以分为静态存储变量和动态存储变量。静态存储变量是程序中整个运行时间都存在，而动态存储变量是在调用函数时临时分配单元。

静态存储变量和动态存储变量的使用也是 C 中一个难点和重点，因此，在本例中对它们作详细的阐述。需要注意的事项有，掌握静态存储变量和动态存储变量概念，并能够使用它们。

知识要点

在编译时分配存储空间的变量称为静态存储变量，其定义形式为在变量定义的前面加上关键字 static，例如：

```
static int a=8;
```

定义的静态存储变量无论是做全程变量或是局部变量，定义和初始化在程序编译时进行。作为局部变量，调用函数结束时，静态存储变量不消失并且保留原值。

动态存储方式是指在程序运行期间分配固定的存储空间的方式。

动态存储变量可以是函数的形式参数、局部变量、函数调用时的现场保护和返回地址。这些动态存储变量在函数调用时分配存储空间，函数结束时释放存储空间。动态存储变量的定义形式为在变量定义的前面加上关键字 auto，例如：

```
auto int a, b, c;
```

“auto”也可以省略不写。事实上，已经使用的变量均为省略了关键字 auto 的动态存储变量。

程序源码

该应用程序的源代码如下：

```
// 给出年、月、日，计算该日是该年的第几天
# include <stdio.h>
int sum_day(int month, int day);
int leap(int year);
void main()
{
    int year, month, day;
```

```
    int days;
    printf("请输入日期(年, 月, 日): ");
    scanf("%d, %d, %d", &year, &month, &day);
    printf("%d年%d月%d日", year, month, day);
    days = sum_day(month, day);    // 调用函数 sum_day()
    if(leap(year) && month>=3)    // 调用函数 leap()
        days = days + 1;
    printf("是该年的第%d天.\n", days);
}

// 定义静态存储变量并赋初值
static int day_tab[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int sum_day(int month, int day)    // 计算日期
{
    int i;
    for(i=1; i<month; i++)
        day = day + day_tab[i];
    return day;
}
// 判断是否为闰年(能够被 400 整除或者同时能被 4 和 100 整除的年份都是闰年)
int leap(int year)
{
    int leap;
    leap = (year%4==0&&year%100!=0)||(year%400==0);
    return leap;
}
```

程序分析

在本程序中，主函数接受从键盘输入的日期，并调用 sum_day 和 leap 函数计算天数，函数 sum_day 计算输入日期的天数。函数 leap 返回是否为闰年的信息。

实例33 内部和外部函数

实例说明

函数本质上是全局的，因为一个函数要被另外的函数调用，但是，根据函数是否能被其他的源文件调用，可以将函数分为内部函数和外部函数。

本程序实际上可以用普通的调用子函数的方式（即内部函数的方式）来完成，但在此为了向读者介绍外部函数的用法，将会使用外部函数来实现。

知识要点

如果一个函数只能够被本文件中的其他函数调用，则称它为内部函数。在定义内部函数时，在函数名和函数类型的前面加 static。形式如下：

```
static int function(int x, int y)
```

内部函数又称为静态函数。使用内部函数，可以使函数只局限于所在文件，如果在不同的文件中有同名的内部函数，它们互不干扰。在一个大的工程项目中分工合作时，就不必担心自己所用的函数是否会和其他文件中的函数同名，通常把只由同一文件使用的函数和外部变量放在一个文件中，冠以 static 使之局部化，其他文件不能使用。

在定义函数时，如果冠以关键词 extern，则表示此函数是外部函数。如：

```
extern int function(int x, int y)
```

函数 function 可以为其他的文件调用，如果在定义函数时省略 extern，则隐含为外部函数。本实例前面例子中的所有函数都作为外部函数。

在需要调用此函数的文件中，一般需要用 extern 说明所用函数是外部函数。

程序源码

该应用程序的源代码如下：

extern.c（文件 1）

```
# include <stdio.h>
void  main()
{
    // 说明本文件将要使用其他文件中的函数
extern int multiply();
    extern int sum();
  int a, b;
```

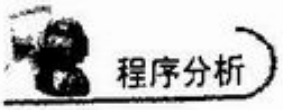
```
    int result;
    printf("Please input a and b: ");
    scanf("%d, %d", &a, &b);
    result = multiply(a, b);    // 调用外部函数 multiply 求解 a 和 b 的乘积
    printf("The result of multiply is: %d", result);
    result = sum(a, b);    // 调用外部函数 sum 求解 a 和 b 的和
    printf("\nThe result of sum is: %d\n", result);

}
```

file1.c (文件 2)

```
    # include <stdio.h>

extern int multiply(int a, int b)    // 定义外部函数 multiply()
{
        int c;
        c = a * b;
        return c;    // 返回参数的乘积
}
```

file2.c (文件 3)

```
    # include <stdio.h>

extern int sum(int a, int b)    // 定义外部函数 sum()
{
        int c;
        c = a + b;
        return c;    // 返回参数的商
}
```



整个程序由三个文件组成。每个文件包含一个函数。在 main()函数中调用了外部函数 multiply()和 sum()，它们所实现的功能非常简单，只是相乘和相加。希望读者通过此例能够掌握外部函数的定义和使用规则。

实例34 综合实例1

实例说明

下面将会给出两个有关函数使用的综合实例，本例是第一个综合实例。在实例中，侧重于帮助读者复习有关全局变量与局部变量的内容。此外，本例还涉及到多个函数的调用。

知识要点

在此，补充一些有关外部变量初始化的知识点。

首先，同函数一样，外部变量的定义和外部变量的说明并不是一回事。外部变量的定义只有一次，它的位置必须在所有函数之外，而同一文件中的外部变量的说明可以有许多次，位置是在函数之内（哪个函数要用到便在哪个函数中声明）。

对外部变量的初始化只能在“定义”中进行，而不能在“说明”中完成。

此外，C语言中规定，外部数组可以赋初值，而局部数组则不能赋初值。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

// 声明三个子函数
void head1();
void head2();
void head3();

int count;    // 全局变量
void main()
{
    register int index;    // 定义为主函数寄存器变量
                           // 这样我们在调用变量 index 时，将会节省很多时间
    head1();
    head2();
    head3();
    for (index=8; index>0; index--)    // 主函数 for 循环
    {
        int stuff;    // 局部变量
```

```
                    // 这种变量的定义方法在 Turbo C 中是不允许的
                    // stuff 的可见范围只在当前循环体内

        for(stuff=0; stuff<=6; stuff++)
            printf("%d ", stuff);
        printf("index is now %d\n", index);
    }
}

int counter;    // 全局变量
                // 可见范围为从定义之处到源程序结尾

// 子函数定义
void head1()
{
    int index;    // 此变量只用于 head1

    index = 23;
    printf("The header1 value is %d\n", index);
}

void head2()
{
    int count;    // 此变量是函数 head2()的局部变量
                  // 此变量名与全局变量 count 重名
                  // 故全局变量 count 不能在函数 head2()中使用

    count = 53;
    printf("The header2 value is %d\n", count);
    counter = 77;
}

void head3()
{
    printf("The header3 value is %d\n", counter);
}
```

程序分析

程序在屏幕的运行结果如下：

```
The header1 value is 23
The header2 value is 53
The header3 value is 77
0 1 2 3 4 5 6 index is now 8
```

```
0 1 2 3 4 5 6 index is now 7
0 1 2 3 4 5 6 index is now 6
0 1 2 3 4 5 6 index is now 5
0 1 2 3 4 5 6 index is now 4
0 1 2 3 4 5 6 index is now 3
0 1 2 3 4 5 6 index is now 2
0 1 2 3 4 5 6 index is now 1
```

本程序中有多个变量的重名，在此将详细说明它们的区别，希望读者通过程序的分析能够理清它们之间的关系。

在主函数 main()中定义了寄存器变量 index，而在子函数 head1()中也定义了一个普通变量 index。首先应该清楚，它们是不同的变量，占用着不同的内存区域。寄存器变量 index 的作用域是整个 main()函数，普通变量 index 只在函数 head1()中起作用。在主函数中调用子函数 head1()时，寄存器变量 index 必然会传入到 head1()中，但此时在子函数中起作用的仍然是它自身的变量 index，寄存器变量 index 将不会起作用。

此外在整个文件中和函数 head2()都定义了变量 count，显然它们也是不同的变量，当在函数 head2()作用域之外时，全局变量 count 起作用，而到了函数 head2()内部时，由于它自身也有变量 count，所以此时全局变量 count 便不再起作用。

实例35 综合实例2

实例说明

本综合实例是一个统计学生成绩的程序，要求输入10个学生5门功课的成绩，分别用子函数求出：①每个学生的平均分；②每门功课的平均分；③找出最高分所对应的学生和功课；④求出平均分方差，方差公式如下所示：

$$\sigma = \frac{1}{n}\sum x_i^2 - (\frac{\sum x_i}{n})^2$$

公式中 x_i 代表某一个学生的平均分。

知识要点

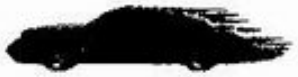
本程序虽然较长，但逻辑却非常明确，模块也非常清楚。通过对本程序的分析学习，旨在帮助读者建立一种编写较大程序时的思维逻辑。

程序源码

该应用程序的源代码如下：

```
// 学生成绩统计
# include <stdio.h>
# define M 5
# define N 10

float score[N][M];
float a_stu[N], a_cor[M];

// 声明子函数
void input_stu();
void avr_stu();
void avr_cor();
float highest(int *r, int *c);
float s_diff();

void main()    // 主函数
{
    int i, j, r, c;
```

```
    float h;
    r = 0;
    c = 1;
    input_stu();    // 调用函数 input_stu，输入学生各门功课的成绩
    avr_stu();    // 调用函数 avr_stu，求出每个学生的平均分
    avr_cor();    // 调用函数 avr_cor，找出学生成绩中的最高分
    printf("\n  序号    课程 1    课程 2    课程 3    课程 4    课程 5    平均分");
    for(i=0; i<N; i++)
    {
        printf("\n  NO%2d", i+1);
        for(j=0; j<M; j++)
            printf("%8.2f", score[i][j]);
        printf("%8.2f", a_stu[i]);
    }
    printf("\n课平均");
    for(j=0; j<M; j++)
        printf("%8.2f", a_cor[j]);
    h = highest(&r, &c);
    printf("\n\n最高分%8.2f 是 %d 号学生的第%d 门课\n", h, r, c);
    printf("  方差 %8.2f\n", s_diff());
}

void input_stu()    // 输入学生的成绩
{
    int i, j;
    for(i=0; i<N; i++)
    {
        printf("请输入学生%2d 的 5 个成绩:\n", i+1);
        for(j=0; j<M; j++)
            scanf("%f", &score[i][j]);
    }
}

void avr_stu()    // 计算学生的平均分
{
    int i, j;
    float s;
    for(i=0; i<N; i++)
    {
        s = 0;
        for(j=0; j<M; j++)
            s = s + score[i][j];
        a_stu[i] = s/M;
    }
```



```
}

void avr_cor()    // 计算课程的平均分
{
    int i, j;
    float s;
    for(j=0; j<M ;j++)
    {
        s = 0;
        for(i=0; i<N; i++)
            s = s + score[i][j];
        a_cor[j] = s/(float)N;
    }
}

float highest(int *r, int *c)    // 找最高分
{
    float high;
    int i, j;
    high = score[0][0];
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            if(score[i][j]>high)
            {
                high = score[i][j];
                *r = i + 1;
                *c = j + 1;
            }
    return high;
}

float s_diff()    // 求方差
{
    int i;
    float sumx, sumxn;
    sumx = 0.0;
    sumxn = 0.0;
    for(i=0; i<N; i++)
    {
        sumx = sumx + a_stu[i]*a_stu[i];
        sumxn = sumxn + a_stu[i];
    }
    return (sumx/N-(sumxn/N)*(sumxn/N));
}
```

程序分析

函数 input_stu 的执行结果是给全程变量成绩数组 score 各个元素输入初值。

函数 avr_stu 的作用是计算每个学生的平均分，并将结果赋给全程变量数组 a_stu 中的各个元素。

函数 avr_cor 的作用是计算每门课程的平均成绩，计算结果存入全程变量数组 a_cor。

函数 highest(r, c)的返回值是所有学生所有成绩中的最高分，r、c 两个参数分别是最高分所在行、列的序号。

本程序的运行结果如下所示：

```
请输入学生 1 的 5 个成绩：
78  88  69  90  96
请输入学生 2 的 5 个成绩：
85  87  89  98
请输入学生 3 的 5 个成绩：
60  78  71  80
请输入学生 4 的 5 个成绩：
79  64  58  66
请输入学生 5 的 5 个成绩：
97  90  92  89
请输入学生 6 的 5 个成绩：
71  79  96  90
请输入学生 7 的 5 个成绩：
63  67  60  66
请输入学生 8 的 5 个成绩：
86  78  79  98
请输入学生 9 的 5 个成绩：
95  75  79  78
请输入学生 10 的 5 个成绩：
78  59  68  90
```

序号	课程 1	课程 2	课程 3	课程 4	课程 5	平均分
NO 1	78.00	88.00	69.00	90.00	96.00	84.20
NO 2	81.00	85.00	87.00	89.00	98.00	88.00
NO 3	69.00	60.00	78.00	71.00	80.00	71.00
NO 4	78.00	79.00	64.00	58.00	66.00	69.00
NO 5	99.00	97.00	90.00	92.00	89.00	93.40
NO 6	79.00	71.00	79.00	96.00	90.00	83.20
NO 7	69.00	63.00	67.00	60.00	66.00	65.00
NO 8	85.00	86.00	78.00	79.00	98.00	82.00
NO 9	78.00	95.00	75.00	79.00	78.00	78.40
NO10	96.00	78.00	59.00	68.00	90.00	75.80
课平均	79.50	82.00	76.50	77.30	81.60	

```
最高分    99.00 是 5 号学生的第 1 门课
  方差    73.14
```

实例36 变量的指针

实例说明

有关指针的内容，在前面的例子中已向大家做过简单的介绍，本实例将要向读者详细地、系统地介绍指针这部分内容。

首先，读者接触到的内容是变量的指针，这是指针中最简单也是最常用到的部分，希望读者能够认真学习，为后面掌握更深层次的内容打好基础。

知识要点

变量的指针就是指变量的地址。

在C程序中，存放地址的指针变量需专门定义：

```
int *ptr1;
float *ptr2;
char *ptr3;
```

以上代码表示定义了三个指针变量ptr1、ptr2、ptr3。ptr1指向一个整型变量，ptr2指向一个实型变量，ptr3指向一个字符型变量，换句话说，ptr1、ptr2、ptr3可以分别存放整型变量的地址、实型变量的地址、字符型变量的地址。

在定义了指针变量后，才可以写入指向对应数据类型的变量的地址，或者说是为指针变量赋初值：

```
int *ptr1, m= 3;
float *ptr2, f=4.5;
char *ptr3, ch='a';
ptr1 = &m;
ptr2 = &f;
ptr3 = &ch;
```

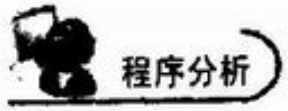
上述赋值语句ptr1 = &m表示将变量m的地址赋给指针ptr1，此时ptr1就指向m。三条赋值语句产生的效果是ptr1指向m；ptr2指向f；ptr3指向ch。

需要说明的是，指针变量可以指向任何类型的变量，当定义指针变量时，指针变量的值是随机的，不能确定它具体的指向，必须为其赋值，才有意义。

程序源码

该应用程序的源代码如下：

```
// 输入x、y和z三个整数，按大小顺序输出
# include <stdio.h>

void swap(int *pt1, int *pt2);
void exchange(int *q1, int *q2, int *q3);
void main()
{
        int x, y, z, *p1, *p2, *p3;
        printf("请输入三个整数：");
        scanf("%d, %d, %d", &x, &y, &z);  // 输入整数时以逗号隔开
        p1 = &x; p2 = &y; p3 = &z;
        exchange(p1, p2, p3);
        printf("按大小排序后的三个整数为：");
        printf("%d, %d, %d\n", x, y, z);
}
void swap(int *pt1, int *pt2)
{
        int p;
        p = *pt1;   *pt1 = *pt2;    *pt2 = p;
}
void exchange(int *q1, int *q2, int *q3)
{
        if(*q1 < *q2)   swap(q1, q2);
        if(*q1 < *q3)   swap(q1, q3);
        if(*q2 < *q3)   swap(q2, q3);
}
```



程序中用到了函数的嵌套调用，首先定义一个子函数，完成两个数的互换；然后再定义一个，完成三个数互换的子函数，在它的内部嵌套调用了前一个子函数。在这些子函数中，使用的参数都是指针变量，实现了传址的功能，即形参改变的同时，实参也被改变。

实例 37 一维数组指针

实例说明

前面我们介绍了变量的指针，即指向一个常用数据类型变量的指针。在这将要向读者介绍数组指针（本实例重点介绍一维数组指针）。

数组指针本质上就是变量指针，但它是一种较为特殊的变量指针。在本实例中，将要向读者介绍数组指针的特征及用法。

知识要点

假设定义一个一维数组，该数组在内存会有系统分配的一个存储空间，其数组的名字就是数组在内存中的首地址。若再定义一个指针变量，并将数组的首址传给指针变量，则该指针就指向这个一维数组。数组名是数组的首地址，也就是数组的指针，而定义的指针变量就是指向该数组的指针变量。对一维数组的引用，既可以用传统的数组元素的下标法，也可使用指针的表示方法。

```
int a[10] , *ptr; // 定义数组与指针变量* /
```

做赋值操作：ptr=a; 或 ptr = &a[0];

则 ptr 就得到了数组的首地址。其中，a 是数组的首地址，&a[0]是数组元素 a[0]的地址，由于 a[0]的地址就是数组的首地址，所以，两条赋值操作效果完全相同。指针变量 ptr 就是指向数组 a 的指针变量。

若 ptr 指向了一维数组，现在看一下 C 规定指针对数组的表示方法：

（1）ptr+n 与 a+n 表示数组元素 a[n]的地址，即&a[n] 。对整个 a 数组来说，共有 10 个元素，n 的取值为 0～9，则数组元素的地址就可以表示为 ptr+0～ptr+9 或 a+0～a+9，这与&a[0]～&a[9]是一致的。

（2）*(ptr+n)和*(a+n)表示数组 a 的各个元素，等效于 a[n]，这是数组元素地址的另一种表示方法。

（3）指向数组的指针变量也可用数组的下标形式表示为 ptr[n]，其效果相当于*(ptr+n)。

程序源码

该应用程序的源代码如下：

```
// 将数组 array 中的各个整数按照逆序存放
# include <stdio.h>
```

```
void inv(int *x, int n);

void main()
{
    int i;
    int array[10] = {1, 3, 9, 11, 0, 8, 5, 6, 14, 98};
    printf("原始数组是:\n");
    for(i=0; i<10; i++)
        printf("%d " , array[i]);
    printf("\n");
    inv(array, 10);
    printf("按相反次序存放后的数组为:\n");
    for(i=0; i<10; i++)
        printf("%d ", array[i]);
    printf("\n");
}

void inv(int *x, int n)
{
    int *p, *i, *j;
    int t;
    int m = (n-1)/2;
    i = x; j = x + n - 1; p = x + m;
    for(; i<=p; i++, j--)
    {
        t = *i; *i = *j; *j = t;
    }
}
```

程序分析

求解此题的算法是：将 array[0]和 array[n-1]对换，再将 array[1]和 array[n-2]对换，依次类推，直到将 array[(n-1)/2]和 array[n-int((n-1)/2)]对换。在程序中，子函数 inv()的形参 x 是指针变量，实参是数组名 array。

实例 38 二维数组指针

实例说明

用指针可以指向一维数组，也可以指向多维数组。但在概念的理解上和具体使用上，多维数组的指针要比一维数组的指针要复杂一些。

在本例中将要向读者介绍多维数组的指针，主要是二维数组的指针。希望通过对本实例的学习，能够加深读者对多维数组指针概念的理解。

实例原题是：有一个班级，共三个学生，各学四门课，要求计算出这个班级的总平均分，以及第 n 个学生的成绩。

知识要点

定义一个二维数组：

```
int a[3][4];
```

表示二维数组有三行四列共 12 个元素，在内存中按行存放，其中 a 是二维数组的首地址，& a[0][0]既可以看作数组第 0 行 0 列元素的首地址，同样还可以看作是二维数组的首地址，a[0]是第 0 行的首地址，当然也是数组的首地址。同理 a[n]就是第 n 行的首址，&a[n][m]就是数组元素 a[n][m]的地址。

既然二维数组每行的首地址都可以用 a[n]来表示，就可以把二维数组看成是由 n 个一维数组构成，将每行的首地址传递给指针变量，行中的其余元素均可以由指针来表示。

用地址法来表示数组各元素的地址。对元素 a[1][2]，&a[1][2]是其地址，a[1]+2 也是其地址。分析 a[1]+1 与 a[1]+2 的地址关系，它们地址的差并非整数 1，而是一个数组元素的所占位置 2，原因是每个数组元素占两个字节。

对第 0 行首地址 a 和第一行首地址 a+1 来说，地址的差同样也并非整数 1，而是一行中四个元素所占字节数的总和 8。由于数组元素在内存中连续存放。将数组的首地址传递给指向整型变量的指针，则该指针指向二维数组。

```
int *ptr, a[3][4];
```

若有赋值操作：ptr=a；则用 ptr++就能访问数组的各元素。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>
```

```
void main()
{
    int num;
    // 声明子函数
    void average(float *point, int n);
    void search(float(*point)[4], int n);

    // 定义一个静态存储数组并赋初值
    static float score[4][4] = {{76, 90, 92, 87}, {68, 78, 69, 94},
                                {89, 82, 81, 60}, {81, 68, 60, 97}};

    printf("班级的总平均分: ");
    average(*score, 16);    // 调用函数 average 求 12 个分数的平均分
    printf("请输入学生的学号(0-3): ");
    scanf("%d", &num);
    search(score, num);    // 求出第四个学生的成绩
}

// 子函数定义
void average(float *point, int n)
{
    float *p_end;
    float aver;
    float sum = 0;
    p_end = point + n -1;
    for(; point<=p_end; point++)
        sum = sum + (*pcint);
    aver = sum/n;
    printf("%5.2f\n", aver);
}

void search(float(*point)[4], int n)
{
    int i;
    for(i=0; i<4; i++)
        printf("%5.2f ", *(*(point+n)+i));
    printf("\n");
}
```

程序分析

程序的运行结果如下：

```
班级的总平均分: 79.50
```

```
请输入学生的学号(0-3)：1
68.00  78.00  69.00  94.00
```

在主函数 main()中，调用子函数 average()求取全班的总平均分。在子函数 average()中形参 point 被声明为指向一个实形变量的指针变量，用指针 point 指向二维数组的各个元素，point 每加 1 就改为指向下一个元素。如下图所示：

p→76
p+1→90
92
87
68
78
……

相应的实参是*score，即 score[0]。形参 n 是元素的总个数，实参 16 表示需要求出 16 个元素的平均值。函数 average()中的指针变量 point 指向 score 数组的某一个元素（元素值为一门课的成绩）。变量 sum 是累计总分，aver 是平均分。由于在函数中已经输出了 aver 的值，所以函数无返回值。

函数 search()的形参 point 不是指向一般实形变量的指针变量，而是指向包含四个元素的一维数组的指针变量。实参传给形参 n 的值为 sum（1），即寻找序号为 1 的学生的成绩（四个学生的学号分别为 0、1、2、3）。

函数开始调用时，实参 score 代表该数组的第 0 行首地址，传给 point，使 point 也指向 score[0]。p+n 指向 score[n]，*(p+n)+i 是 score[n][i]的地址，*(*(p+n)+i)是 score[n][i]的值。现在 num=3，i 由 0 变到 3，for 循环输出 score[2][0]到 score[3][3]的值。

实例39 字符串指针

实例说明

在 C 程序中，可以用两种方法实现一个字符串，分别是用字符数组实现和用字符指针实现。其中用字符数组实现的方法，在前面的例子中已经作过讲解。在本例中，将向读者详细介绍用字符指针实现的方法。

知识要点

在表示一个字符串时，可以不定义字符数组，而定义一个字符指针。用字符指针来指向字符串中的字符。例如：

```
# include <stdio.h>
void main()
{char *string = "I am a student!";
 printf("%s\n", string);
}
```

在上面例子中，我们并没有定义字符数组。由于 C 语言对字符串常量是按照字符数组处理的，所以实际上是程序在内存中开辟了一个字符数组用来存放字符串常量。在程序中定义了一个字符指针变量 string。并把字符串首地址赋给它。

语句

```
char *string = "I am a student!";
```

等价于下面的两行语句：

```
char *string;
*string = "I am a student!";
```

可以看到 string 被定义成一个指针变量，它指向字符型数据，需要读者注意的是，只是把"I am a student!"的首地址赋给了指针变量 string，而不是把字符串赋给*string。

在输出时，用如下语句：

```
printf("%s\n", string);
```

%s 表示输出一个字符串，给出字符指针变量名为 string，系统首先输出它所指向的一个字符数据，然后自动使 string 加 1，使之指向下一个字符并再输出，依此类推，直到遇到字符串结束标志'\0'为止。在内存中，字符串的最后自动加了一个'\0'，所以在输出时能够确定字符串的终止位置。

程序源码

该应用程序的源代码如下：

```
// 将字符串 a 复制到字符串 b
# include <stdio.h>

void main()
{
    int i;
    char a[] = "I am a student.";
    char b[20];
    char *p1, *p2;

    p1 = a;    // 将数组 a 的首地址赋给字符型指针 p1
p2 = b;    // 将数组 b 的首地址赋给字符型指针 p2
    for(; *p1!='\0'; p1++, p2++)
        *p2 = *p1;
    *p2 = '\0';
    printf("string a is: %s\n", a);
    printf("string b is: ");
    for(i=0; b[i]!='\0'; i++)
        printf("%c", b[i]);
    printf("\n");
}
```

程序分析

程序中 p1、p2 是两个指针变量，它们都指向字符型数据。先使 p1 和 p2 的值分别为字符串 a 和 b 的首地址。*p1 最初的值为 'I'，赋值语句 "*p2 = *p1;" 的作用是将字符 'I'（a 串中的第一个字符）赋给 p2 所指向的数组元素，即 b[0]。然后，p1 和 p2 分别加 1，使它们分别指向下面的一个元素，直到*p1 的值为'\0'为止。

需要注意的是，程序中变量 p1 和 p2 的值在不断改变，程序必须保证 p1 和 p2 同步移动。

实例40 函数指针

实例说明

在 C 语言中，一个特别难理解却强有力的特性便是函数指针（function pointer）。不带括号和变元的函数名代表函数的地址和不带下标的数组名代表数组的地址相类似。

为了使读者能够理解函数指针的概念，我们将在本实例中研究如下程序，它的功能是比较用户输入的两个串。

知识要点

函数具有可赋给指针的物理内存地址。一个函数的地址是该函数的进入点，也是调用函数的地址。一旦指针指向某一函数，该函数可以通过该指针来调用。函数指针还允许将函数作为变元传递个其他函数。

有关函数指针有如下几点说明。

（1）指向函数的指针变量的一般定义形式为：

数据类型标识符 (*指针变量名)();

这里的“数据类型标识符”是指向函数返回值的类型。

（2）可以通过函数名调用函数，也可以通过函数指针调用（即用指向函数的指针变量表示）。

（3）(*point)()表示定义一个指向函数的指针变量，它不是固定指向哪一个函数，而是表示定义了这样一个类型的变量，它专门用来存放函数的入口地址。在一个程序中，一个指针变量可以先后指向不同的函数。

（4）在给函数指针变量赋值时，只需要给出函数名而不必给出参数，例如：

```
point = address;
```

因为是将函数入口地址赋给 point，而不牵涉到实参和形参的结合问题。

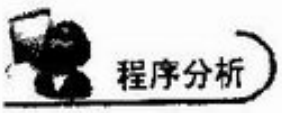
（5）用函数指针变量调用函数时，只需要将(*point)代替函数名即可（point 为指针变量名），在（*point）之后的括弧中根据需要写上实参。

（6）对于指向函数的指针变量，像 point + n、point + +、point - -等运算都是无意义的。

程序源码

该应用程序的源代码如下：

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

void check(char *a, char *b, int(*cmp)(const char *, const char *));

void main()
{
    char s1[80];    // 定义字符串数组一
char s2[80];    // 定义字符串数组二
    int(*p)(const char *, const char *);    // 定义一个函数指针

  // 函数 strcmp 包含在头文件<string.h>中，功能是比较它的
// 两个参数（两个字符串）是否相等
    p = strcmp;    // 将函数 strcmp 的地址赋给函数指针 p

    printf("输入两个字符串：\n");
    gets(s1);    // 输入字符串 1
    gets(s2);    // 输入字符串 2

  // 调用字符串对 s1 和 s2 进行比较
    check(s1, s2, p);    // 通过指针变量 p 传递函数 strcmp 的地址
}

void check(char *a, char *b, int(*cmp)(const char *, const char *))
{
    printf("测试是否相等\n");
    if(!(*cmp)(a, b))
        printf("结果：相等\n");
    else
        printf("结果：不相等\n");
}
```



我们首先观察一下主函数中的变量 p 的声明，如下所示：

```
int(*p)(const char *, const char *);
```

该声明告诉编译器，p 是一个指针变量，所指向的函数带有两个 const char*类型的参数，并且函数返回 int 类型的结果。包围 p 的括号是必不可少的，以便编译器能够正确地解释这个声明。

其次，来考察一下函数 check()。它声明了三个参数：两个字符类型的指针 a 和 b，以及一个函数指针 cmp。由于函数指针在子函数中的声明方式和在主函数中的声明方式相同，所以 cmp 可以接受带有两个 const char*类型的函数参数并且返回值是 int 类型的函数指针。像变量

p 的声明一样，包围*cmp 的括号是必不可少的，以便编译器能够正确地解释这个语句。

当这个程序开始执行时，将标准的串比较函数 strcmp()的地址赋给函数指针变量 p，然后程序将会提示用户输入两个字符串，并把指针（连同指向 check()的指针 p）传递给这两个串，比较串是否相等。

在函数 check()中，表达式(* cmp)(a, b)调用了函数 strcmp()，由 cmp 指向函数 strcmp()，由 a 和 b 作为调用 strcmp()时的变元。在调用时，和声明的情况类似，必须在* cmp 周围使用一对括号，使编译程序正确操作。这是通过指针调用函数的一种方法。也可以通过较为简单的语法来实现，如下所示：

```
cmp(a, b);
```

读者经常会看到第一种风格，因为它可以帮助你阅读用户的代码，函数是通过指针调用的（也就是说，cmp 是函数指针，不是函数名）。而且，第一种风格是 C 最初的指定格式。直接用 strcmp 作为变元调用 check 的代码如下所示：

```
check(s1, s2, strcmp);
```

当我们直接调用函数 strcmp()时，可以消除额外增加的指针变量的要求，也大大增加了代码的可读性。

既然上述直接调用函数 strcmp()的方法简单易懂，而使用函数指针却是既无好处又容易混淆，那么在程序当中为什么又要使用函数指针呢？那是因为常常需要向过程传入任意的函数，有时需要使用函数指针构成的数组。以编写解释程序为例，在解释程序的运行中，常需要被解释语句调用各种函数，例如求正弦、余弦或者正切等等。这时，用一个函数指针构成的数组取代大型的 switch 语句，由数组下标实施调用。

实例41 指针数组

实例说明

前面介绍了指向不同类型变量指针的定义和使用，可以让指针指向某类变量，并替代该变量在程序中使用；也可以让指针指向一维、二维数组或字符数组，来替代这些数组在程序中使用，在编程时带来许多方便。

下面要讨论的是一种特殊的数组，这类数组存放的全部是指针，分别用于指向某类变量，以替代这些变量在程序中的使用，增加灵活性。我们称这种数组为指针数组。

本实例的原题是：在一个已排好序的字符串数组中，插入一个键盘输入的字符串，使其继续保持有序。

知识要点

一个数组，其元素均为指针类型数据，称为指针数组，也就是说，指针数组中的每一个元素都是指针变量。指针数组的定义形式为：

类型标志*数组名[数组长度说明]

例如：char *point[10];

由于[]的优先级比*高，所以上面的代码实际上可以写成 char *(point[10])，即 point 先和[10]结合，形成 point[10]的形式，这是一个数组格式，共有 10 个元素。然后再和 point 前面的*结合，*表示数组是指针类型的，每个数组元素（指针变量）都指向一个字符型数据。

请注意，千万不要写成 char (*point)[10]，这是一个指向一维数组的指针变量。

使用指针数组的理由很简单，就是它比较适合于指向若干个字符串，使字符串的处理更加方便。

程序源码

该应用程序的源代码如下：

```
# include <stdlib.h>
# include <string.h>
# include <stdio.h>
void main()
{
    // 声明子函数
```

```
    int binary(char *ptr[], char *str, int n);    // 查找函数声明
    void insert(char *ptr[], char *str, int n, int i);    // 插入函数声明

    char *temp, *ptr1[6];
    int i;
    printf("请为字符形指针数组赋初值: \n");
    for (i=0; i<5; i++)
    {
        ptr1[i] = (char *)malloc(20);    // 为指针分配地址后
        gets(ptr1[i]);    // 输入字符串
    }
    ptr1[5] = (char *)malloc(20);
    printf("\n");
    printf("original string:\n");
    for(i=0; i<5; i++)    // 输出指针数组各字符串
        printf("%s\n", ptr1[i]);

    printf("\ninput search string:\n");
    temp = (char *)malloc(20);
    gets(temp);    // 输入被插字符串
    i=binary(ptr1, temp, 5);    // 寻找插入位置 i
    printf("i = %d\n", i);
   insert(ptr1, temp, 5, i);    // 在插入位置 i 处插入字符串
    printf("output strings:\n");

    for(i=0; i<6; i++)    // 输出指针数组的全部字符串
        printf("%s\n", ptr1[i]);
}

int binary(char *ptr[], char *str, int n)
{
    // 折半查找插入位置
    int hig, low, mid;
    low = 0;
    hig = n-1;
    if(strcmp(str,ptr[0]) < 0)
        return 0;
    // 若插入字符串比字符串数组的第 0 个小，则插入位置为 0
    if(strcmp(str,ptr[hig]) > 0)
        return n;
    // 若插入字符串比字符串数组的最后一个大，则应插入字符串数组的尾部
    while(low <= hig)
    {
        mid = (low + hig)/2 ;
```

```
        if (strcmp(str,ptr[mid]) < 0)
            hig = mid - 1;
        else if(strcmp(str,ptr[mid]) > 0)
            low = mid + 1;
        else
            return mid;    // 插入字符串与字符串数组的某个字符串相同
    }
    return low;    // 插入的位置在字符串数组中间
}

void insert(char *ptr[], char *str, int n, int i)
{
    int j;
    for(j=n; j>i; j--)    // 将插入位置之后的字符串后移
        strcpy(ptr[j], ptr[j-1]);
    strcpy(ptr[i], str);    // 将被插字符串按字典顺序插入字符串数组
}
```

程序分析

在已完成程序查找的基础上，将该字符串插入到字符数组中。插入的位置可以是数组头、中间或数组尾。查找的算法采用折半算法，找到插入位置后，将字符串插入。

在程序中，字符串数组中的六个指针变量均分配了存放 20 字节的有效地址。语句 ptr1[5] = malloc(20)保证插入字符串后，也具有安全的存储空间，字符串的长度以串中最长的为基准向系统申请存储空间，以保证在串的移动中有足够的存储空间。

实例 42 二维指针

实例说明

一个指针变量可以指向整型变量、实型变量、字符类型变量，当然也可以指向指针类型变量。当这种指针变量用于指向指针类型变量时，我们称之为指向指针的指针变量，即二维指针变量。

首次接触到二维指针这个概念时，可能会感到较难理解。实际上我们可以这样理解它，指针变量指向的地址中存放的也是指针。下面我们将通过一个实例帮助读者加深对二维指针概念的理解。

知识要点

指向指针的指针变量的定义格式如下：

类型标识符 ** 指针变量名

例如：float **ptr;

其含义为定义一个指针变量 ptr，它指向另一个指针变量（该指针变量又指向一个实型变量）。由于指针运算符“ *”是自右至左结合，所以上述定义相当于：

```
float *(*ptr);
```

下面将要对这个二维指针进行赋初值，在赋初值的过程中，将会有助于读者对二维指针概念的理解。

```
float f = 5.0;     // 定义一个实形变量并赋初值为 5
float *point = &f;   // 定义一个实形的指针变量 point，并用变量 f 的地址对它进行初始化
float **ptr = &point;   // 定义一个二维的指针变量 ptr，并用变量 point 的地址
                        // 对它进行初始化
```

从上面初始化的过程中不难看出，二维指针变量实际上是一个特殊的指针变量，因为它里面存放的不是常用数据类型变量的地址，而是指针变量的地址。也就是说，二维指针指向的不是一个变量，而是指向一个一维指针。

程序源码

该应用程序的源代码如下：

```
// 用指向指针的指针变量访问一维和二维数组
// 用指向指针的指针变量访问一维和二维数组
# include <stdio.h>
```

```
#include <stdlib.h>

void main()
{
    int a[10], b[3][4];
    int *p1, *p2, **p3;    // p3是指向指针的指针变量
    int i,j;

    printf("请输入一维数组（10个元素）：\n");
    for(i=0; i<10; i++)
        scanf("%d", &a[i]);    // 一维数组的输入
    printf("请输入二维数组（三行四列）：\n");
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            scanf("%d", &b[i][j]);    // 二维数组输入
    printf("\n");

   // 分别给一维指针变量p1和二维指针变量p3赋初值
    for(p1=a, p3=&p1, i=0; i<10; i++)
        printf("%4d", *(*p3+i));    // 用二维指针变量输出一维数组
    printf("\n");
    for(p1=a; p1-a<10; p1++)    // 用二维指针变量输出一维数组
    {
        p3 = &p1;
        printf("%4d", **p3);
    }
    printf("\n");

    for(i=0; i<3; i++)    // 用二维指针变量输出二维数组
    {
        p2 = b[i];
        p3 = &p2;
        for(j=0; j<4; j++)
            printf("%4d", *(*p3+j));
        printf("\n");
    }
    for(i=0; i<3; i++)    // 用二维指针变量输出二维数组
    {
        p2 = b[i];
        for(p2=b[i]; p2-b[i]<4; p2++)
        {
            p3 = &p2;
            printf("%4d", **p3);
        }
```

```
        printf("\n");
        }
}
```

程序分析

程序的运行结果如下所示：

```
请输入一维数组（10个元素）：
1 2 3 4 5 6 7 8 9 0
"请输入二维数组（三行四列）：
1 3 5 7
2 4 6 8
5 7 9 2

1 2 3 4 5 6 7 8 9 0
1 2 3 4 5 6 7 8 9 0
1 3 5 7
2 4 6 8
5 7 9 2
1 3 5 7
2 4 6 8
5 7 9 2
```

实例43 指针的初始化

实例说明

本实例是一个有关指针初始化的 C 程序实例。非静态的局部指针在已声明但未赋值前，其值是不确定的（全局指针和静态局部指针自动初始化为零），如果在赋值前试图使用指针，不仅可能导致程序瘫痪，也可能使计算机的操作系统垮掉，错误非常严重。

大多数 C 程序员使用时的习惯是：对于当前没有指向合法的内存位置的指针，为其赋值 null（零）。C 语言规定，空指针即意味着它不指向任何对象。

知识要点

给指针赋空值的一种方法是为它赋值为零。例如，下面的语句将 p 初始化为空。

```
char* p = 0;
```

利用下面的语句为指针赋予空值也是很常用的方法：

```
p = NULL;
```

我们可以利用 null 简化编码并提高效率。用 null 指针标志指针数组的结尾，使访问该数组的子程序遇到 null 值时，认为数组内容已结束。下面实例中的函数 search()描述了这种方法。

程序源码

该应用程序的源代码如下：

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

int search(char* p[], char* name);

// 给字符型的指针数组赋初值
char* names[] = {
    "Herb",
    "Rex",
    "Dennis",
    "John",
    NULL};    // NULL指针标志数组内容的结束

void main()
{
```

```
    if(search(names, "Herb") != -1)
        printf("Herb is in list.\n");

    if(search(names, "Mary") == -1)
        printf("Mary not found.\n");
}

int search(char* p[], char* name)
{
    register int t;

    for(t = 0; p[t]; ++t)
        if(!strcmp(p[t], name)) return t;
    return -1;
}
```

程序分析

在函数 search()中使用的方法是：用 null 指针标志指针数组的结尾，使访问该数组的子程序遇到 null 时认为数组内容已经结束。

在函数 search()中共有两个参数。第一个参数是 char* 指针数组 p，它指向包含这些名字的串；第二个参数是 name 指针，指向被查找的名字。search()函数搜索整个指针列表，查找与 name 指向的字符串相匹配的字符串。search()中的 for 循环一直运行到发现匹配或遇到 null 指针。由于用 null 标志数组的结尾，到达 null 处时，控制条件失败，退出 for 循环。也就是说，p[t]为空时即为假。在本例中，当用 Bill 这个名字试过后即发生这种情况，因为它不在名单之中。

实例 44 综合实例

实例说明

本实例原题如下：

一个班级有 4 个学生，共学习 5 门课。要求编写程序完成下面的三个功能。

（1）求出第 5 门功课的平均分；

（2）找出有两门以上功课不及格的学生，输出他们的学号和全部课程成绩及平均成绩；

（3）找出平均成绩在 90 分以上或者全部课程成绩在 85 分以上的学生。

知识要点

指针的作用特别强，对许多问题都是必要的。同时，偶然错用指针时，可能引入最难排除的错误。

指针错误难于定位，因为指针本身并没有问题。问题在于通过错误的指针操作时，程序将会对未知内存区域进行读或写操作。读操作时，最坏情况是取得无用的数据；而写操作时，可能冲掉其他代码或数据。这种错误可能到程序执行相当一段时间后才出现，所以我们很难找出它们。

指针错误的性质特别严重，读者应该尽力防止。

程序源码

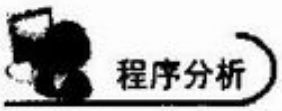
该应用程序的源代码如下：

```
# include <stdio.h>

void avscc(float *psco, float *pave);
void avcour5(char *pcou, float *psco);
void fali2(char *pcou, int *pnum, float *psco, float *pave);
void excellence(char *pcou, int *pnum, float *psco, float *pave);

void main()
{
    // 数组 num 用于存放每位学生的学号
    int i, j, *pnum, num[4];
    // 数组 aver 存放每位学生的平均分，二维数组 score 用于存放学生成绩
    float score[4][5], aver[4], *psco, *pave;
```

```
    // 数组 course 存放 5 门课程的名称
    char course[5][10], *pcou;

    printf("请按行输入 5 门功课的名称：\n");
    pcou = course[0];  // 指针变量 pcou 用来存放数组 course 的首地址
                       // 从首地址开始，每十个字节存放一个课程的名称
    for(i=0; i<5; i++)
        scanf("%s", pcou+10*i);  // 以空格为间隔输入五门课程的名称

    printf("请按下面的格式输入 4 个学生的学号和各科成绩：\n");
    printf("学号");
    for(i=0; i<5; i++)
        printf(",%s", pcou+10*i);  // 输出各门课程的名称
    printf("\n");
    psco = &score[0][0];  // 指针 psco 指向数组 score 中的第一个元素
                          // 即指向第一个学生第一门课程的成绩
    pnum = &num[0];
    for(i=0; i<4; i++)
    {
        scanf("%d", pnum+i);  // 输入学号
        for(j=0; j<5; j++)
            scanf(",%f", psco+5*i+j);  // 以逗号为间隔输入学生成绩
    }

    pave = &aver[0];  // 将数组 aver 的首地址赋给指针 pave
    printf("\n\n");  // 空行
    avsco(psco, pave);
    avcour5(pcou, psco);
    printf("\n\n");  // 空行
    fali2(pcou, pnum, psco, pave);
    printf("\n\n");  // 空行

    excellence(pcou, pnum, psco, pave);
}

void avsco(float *psco, float *pave)  // 求每个学生的平均成绩
{
    int i, j;
    float sum, average;
    for(i=0; i<4; i++)  // i 代表学生的序号，表示第 i 个学生
    {
        sum = 0.0;
        for(j=0; j<5; j++)  // j 代表课程的序号，表示第 j 门课程
            sum = sum + (*(psco+5*i+j));  // 累计每个学生的各科成绩
        average = sum/5;  // 计算第 i 个学生平均成绩
        *(pave+i) = average;
```

```
    }
}

void avcour5(char *pcou, float *psco)  // 求第五门课程的平均成绩
{
    int i;
    float sum, average5;
    sum = 0.0;
    for(i=0; i<4; i++)
        sum = sum + (*(psco+5*i+4));  // 累计每个学生第五门课的得分
    average5 = sum/4;    // 计算第五门课程的平均成绩
    printf("第 5 门课程%s 的平均成绩为%5.2f.\n", pcou, average5);
}

void fali2(char *pcou, int *pnum, float *psco, float *pave)
{
    int i, j, k, label;
    printf("      =====两门以上课程不及格的学生=====      \n");
    printf("  学号  ");
    for(i=0; i<5; i++)
        printf(" %-8s", pcou+10*i);  // 输出课程名称
    printf("  平均分\n");
    for(i=0; i<4; i++)
    {
        label = 0;
        for(j=0; j<5; j++)
            if(*(psco+5*i+j) < 60.0)
                label++;  // 计算第 i 个学生不及格课程的门数
        if(label >= 2)
        {
            printf("%-8d", *(pnum+i));  // 输出学号
          // 输出符合条件学生的各科成绩
            for(k=0; k<5; k++)
                printf("  %-8.2f", *(psco+5*i+k));
          // 输出符合条件学生的平均分
            printf("  %-8.2f\n", *(pave+i));
        }
    }
}

// 程序结构和上一个子函数 fali2 类似
void excellence(char *pcou, int *pnum, float *psco, float *pave)
{
    int i, j, k, label;
    printf("      =====成绩优秀学生=====\n");
    printf("  学号  ");
```

```
    for(i=0; i<5; i++)
        printf("  %-8s", pcou+10*i);
    printf("  平均分\n");
    for(i=0; i<4; i++)
    {
        label = 0;
        for(j=0; j<5; j++)
            if(*(psco+5*i+j) >= 85.0)
                label++;
        if((label>=5)||(*(pnum+i)>=90))
        {
            printf("%-8d", *(pnum+i));
            for(k=0; k<5; k++)
                printf("  %-8.2f", *(psco+5*i+k));
            printf("  %-8.2f\n", *(pave+i));
        }
    }
}
```



程序分析

程序运行结果如下所示：

```
请按行输入 5 门功课的名称：
数学
物理
化学
语文
英语
请按下面的格式输入 4 个学生的学号和各科成绩：
学号，数学，物理，化学，语文，英语
7981, 78, 76, 96, 63, 60
7982, 89, 79, 98, 87, 100
7983, 92, 90, 93, 85, 89
7984, 56, 87, 60, 51, 45

第 5 门课程数学的平均成绩为 80.25.

      =====两门以上课程不及格的学生=====
学号     数学    物理    化学    语文    英语    平均分
7984   56.00  87.00  60.00  51.00  45.00  59.80

            =====成绩优秀学生=====
学号     数学    物理    化学    语文    英语    平均分
7982   89.00  79.00  98.00  87.00  100.00  90.60
7983   92.00  90.00  93.00  85.00  89.00   89.80
```

第二篇
深入提高篇

通过对上一篇的学习，读者已不再是新手了，应该可以编写一些较小的 C 程序，但要想成为一名合格的 C 程序员，这是远远不够的，读者还需要掌握更多的、更深的东西。这也正是“深入提高篇”将要完成的工作。

本篇以“基础知识篇”为基础，向读者介绍了一些常用数据结构的运用。值得注意的是，这部分内容也是算法篇的基础。

C 语言的 I/O 系统是漂亮的工程成果，提供了在设备间传递数据的机制，既灵活又规范，所以控制台 I/O 和文件 I/O 是本篇的重点。通过本篇的学习，旨在使读者能够分清两者的概念并理清关系，从而能够熟练使用 I/O。

为便于读者编程方便，在此还介绍了一些常用的 C 函数。

在本篇的例子中，读者能够不断复习到前面的知识，有助于读者加深巩固。学习完本篇后，读者再动手编写程序，很快就会成为一名合格的 C 程序。

实例45 结构体变量

实例说明

到目前为止，读者已经学习了基本类型的变量（如整形、实形、字符型变量等等），也学习了一种新的构造类型数据——数组，数组中的各个元素是属于同一种类型的。

但仅仅这些是不够的，因为我们有时需要将不同类型的却又互相联系的数据组合在一起，以便于使用。这就是将要向读者介绍的新的类型——结构体类型。

在本例中，定义了一个结构变量(包括年、月、日)，计算该日是本年中的第几日。

知识要点

结构是在一个名下引用多种变量的集合，提供了一种把相关数据组合到一起的方便手段。结构定义形成一个样板，用于生成结构变量。构成结构的变量称为成员，成员一般也称为元素或域。

通常结构中的各个成员都是逻辑相关的。例如个人资料中的姓名和家庭住址一般应该放入到一个结构中。下面的代码将示范怎样定义结构，由此定义机构的各个成员。关键字 struct 通知编译程序正在定义结构。

```
struct person
{
    char name[20];    // 姓名
   int  age;         // 年龄
   char sex[10];     // 性别
   char job[20];     // 职业
   char address[50]; // 住址
};
```

读者注意，结构定义的结尾必须使用分号，这是因为结构定义就是 C 语句。结构标志 person 标志这一特定的数据结构，是结构的类型标识符。

定义结构时，我们实际上是声明了一种复杂的数据类型，并未生成任何变量。为了声明 person 类型的变量，可以书写成如下的形式：

```
struct person per;
```

以定义 person 类型的结构变量 per。这样，person 描述结构类型，而 per 是结构的变量。定义了结构变量之后，C 编译程序自动为结构的所有成员分配足够的内存。

此外我们还可以在定义结构的同时声明变量，如下所示：

```
struct data
```

```
{
int day;
int month;
int year;
} time1,time2;
```

在这种情况下，我们可以将结构名 data 省略不写。变量 time1、time2 各成员的引用形式为： time1.day，time1.month，time1.year，time2.day，time2.month，time2.year。

程序源码

该应用程序的源代码如下：

```
// 计算天数
# include <stdio.h>

struct
{
    int year;
    int month;
    int day;
} data;    // 定义一个结构并声明对象为 data

void main()
{
    int days;
    printf("请输入日期(年、月、日): ");
    scanf("%d, %d, %d", &data.year, &data.month, &data.day);
    switch(data.month)
    {
    case 1:  days = data.day;
             break;
    case 2:  days = data.day-31;
             break;
    case 3:  days = data.day+59;
             break;
    case 4:  days = data.day+90;
             break;
    case 5:  days = data.day+120;
             break;
    case 6:  days = data.day+151;
             break;
    case 7:  days = data.day+181;
             break;
    case 8:  days = data.day+212;
```

```
            break;
    case 9:  days = data.day+243;
            break;
    case 10: days = data.day+273;
            break;
    case 11: days = data.day+304;
            break;
    case 12: days = data.day+334;
            break;
    }
    if(data.year%4==0&&data.year%100!=0 || data.year%400==0) //判断是否为闰年
        if(data.month>=3)  //判断月份是否超过2月
            days = days + 1;  // 如果上述条件同时满足，则加1
    printf("%d月%d日是%d年的第%d天.\n", data.month, data.day,
data.year, days);
}
```

程序分析

程序的运行结果是：

```
请输入日期(年、月、日)：1998, 9, 25
9月25日是1998年的第268天
```

实例 46 结构体数组

实例说明

单个结构体类型变量在解决实际问题时的作用并不大，在程序中一般以结构体类型数组的形式出现。下面将通过实例来给读者讲解结构体数组的用法和一些注意事项。

程序原题是：编写一个函数 output，打印出一个学生的成绩数组，该数组中有 3 个学生的数据记录，每个记录包括 number、name、score[3]，用 main 函数输入这些记录，用 output 函数输出纪录。

知识要点

定义结构数组时，必须先定义结构，然后再定义该类型结构的数组。

结构体类型数组的定义形式为：

```
struct stu    // 定义学生结构体类型
{
char name[20];  // 学生姓名
char sex;       // 性别
long num;       // 学号
float score[3]; // 三科考试成绩
};
struct stu stud[20]; // 定义结构体类型数组 stud
// 该数组有 20 个结构体类型元素
   其数组元素各成员的引用形式为：
stud[0].name、stud[0].sex、stud[0].score[i];
stud[1].name、stud[1].sex、stud[1].score[i];
stud[2].name、stud[2].sex、stud[2].score[i];
.....
.....
stud[19].name、stud[19].sex、stud[19].score[i];
```

我们在定义数组 stud 的时候，元素个数可以不指定，即可以写成以下的形式：

```
   struct stu stud[];
```

但此时，必须同时给数组赋初值。

程序源码

该应用程序的源代码如下：

```
// 输入学生成绩并显示
# include <stdio.h>

struct student
{
    char number[6];
    char name[6];
    int  score[3];
} stu[2];   // 定义结构体，并同时声明一个数组变量

void output(struct student stu[2]);   // 声明子函数

void main()
{
    int i, j;
    for(i=0; i<2; i++)
    {
        printf("请输入学生%d 的成绩: \n", i+1);
        printf("学号: ");
        scanf("%s", stu[i].number);
        printf("姓名: ");
        scanf("%s", stu[i].name);
        for(j=0; j<3; j++)
        {
            printf("成绩 %d. ", j+1);
            scanf("%d", &stu[i].score[j]);
        }
        printf("\n");
    }
    output(stu);
}

void output(struct student stu[2])   // 定义子函数
{
    int i, j;
    printf("学号  姓名  成绩1  成绩2  成绩3\n");
    for(i=0; i<2; i++)
    {
        printf("%-6s%-6s", stu[i].number, stu[i].name);
        for(j=0; j<3; j++)
            printf("%-8d", stu[i].score[j]);
        printf("\n");
    }
}
```

程序分析

程序的运行结果如下所示：

```
请输入学生 1 的成绩：
学 号：101
姓 名：tong
成绩 1.  80
成绩 2.  90
成绩 3.  85

请输入学生 2 的成绩：
学 号：102
姓 名：wang
成绩 1.  81
成绩 2.  78
成绩 3.  90

学号    姓名   成绩 1   成绩 2   成绩 3
101     tong    80       90       85
102     wang    81       78       90
```

实例47 结构体指针变量

实例说明

指针变量非常灵活方便，可以指向任一类型的变量，若定义指针变量指向结构体类型变量，则可以通过指针来引用结构体类型变量。

在这里，我们将通过一个简单的实例向读者说明指向结构体变量的指针变量的应用。

知识要点

定义结构体指针变量的步骤如下：

首先定义结构体：

```
struct stu
{
char name[20];
long number;
float score[4];
};
```

再定义指向结构体类型变量的指针变量：

```
struct stu *p1, *p2;
```

定义指针变量 p1、p2，分别指向结构体类型变量。引用形式为：指针变量→成员。

程序源码

该应用程序的源代码如下：

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

void main()
{
    struct student
    {
        long num;
        char name[30];
        char sex[10];
        float score;
    };
```

```
    struct student stu;   // 定义结构变量
    struct student *p;    // 定义结构指针
    p = &stu;     // 给结构指针赋初值
  // 结构变量 stu 初始化
    stu.num = 97032;
    strcpy(stu.name, "小明");
    strcpy(stu.sex, "男");
    stu.score = 98.5;
  // 通过结构变量输出
    printf("学号: %ld\n 姓名: %s\n 性别: %s\n 分数: %4.2f\n",
           stu.num, stu.name, stu.sex, stu.score);
    printf("\n");
// 通过结构指针输出
    printf("学号: %ld\n 姓名: %s\n 性别: %s\n 分数: %4.2f\n",
           (*p).num, (*p).name, (*p).sex, (*p).score);
}
```

程序分析

在主函数中定义了 struct student 类型，然后定义了一个 struct student 类型的变量 stu，同时又定义了一个指针变量 p，指向一个 struct student 类型的数据。在函数的执行部分将 stu 的起始地址赋给了指针变量 p，然后对 stu 的各个成员赋初值。

第一个 printf 函数输出 stu 的各个成员的值。用 stu.num 表示 stu 中的成员 num，以此类推。第二个 printf 函数也是输出 stu 各成员的值，但是使用了(*p).num 的形式。(*p)表示 p 指向的结构体变量，(*p).num 是 p 指向的机构体变量中的成员 num。由于成员运算符“.”的优先级高于“*”，所以(*p).num 中的括号不能够省略。

程序的输出结果如下：

```
学号：97032          学号：97032
姓名：小明           姓名：小明
性别：男             性别：男
分数：98.50          分数：98.50
```

实例48 结构体指针数组

实例说明

前面已经介绍过，可以使用指向数组或数组元素的指针和指针变量。同样，对结构体数组以及其元素，也可以用指针或指针变量来表示。

知识要点

定义一个结构体类型数组，其数组名是数组的首地址，这一点前面的实例已介绍得很清楚。定义结构体类型的指针，既可以指向数组的元素，也可以指向数组，在使用时要加以区分。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

// 定义一个全局的结构体
struct student
{
    long num;
    char name[20];
    char sex;
    int age;
};

// 声明结构体数组并赋初值
struct student stu[4] = {{97032, "xiao ming", 'M', 20},
                         {97033, "xiao wang", 'M', 20},
                             {97034, "xiao tong", 'M', 21},
                         {97035, "xiao shui", 'F', 18}};

void main()
{
    // 定义一个结构体指针变量
    struct student *p;

    printf(" 学号     姓名     性别     年龄\n");
```

```
    for(p=stu; p<stu+4; p++)
        printf("%-8ld%-12s%-10c%-3d\n", p->num, p->name,
                                        p->sex, p->age);
}
```

程序分析

程序的运行结果如下所示：

学号	姓名	性别	年龄
97032	xiao ming	M	20
97033	xiao wan	M	20
97034	xiao tong	M	21
97035	xiao shui	F	18

程序中 p 是指向 struct student 结构体类型的指针变量。在 for 循环语句当中，先使 p 的初值为 stu，即结构体数组 stu 的首地址。

在第一次循环中输出 stu[0]的各个成员值。然后执行 p++，使 p 自加 1。p 加 1 意味着增加的地址值为结构体类型数组 stu 的一个元素所占得字节数（在本例中是 4+20+1+2=27 个字节），p++使 p 指向 stu[1]的起始地址。所以在第二次循环中，输出的是 stu[1]中的各个成员值。再一次执行 p++后，p 的值等于 stu+2，它指向 stu[2]，再输出 stu[2]的各个值。一直执行到数组结束。

注意：

（1）如果 p 的初值为 stu，即指向第一个元素，则 p+1 后指向下一个元素的起始地址。

（2）指针 p 已定义为指向 struct student 类型的数据，它只能指向一个结构体类型的数据（也就是 p 的值是 stu 数组的一个元素的起始地址），而不能指向某一个元素中的某一个成员（即 p 的地址不能是成员的地址）。

千万不能认为，指针 p 是存放地址的，所以可以将任何地址赋给它，即使不匹配，也可以使用类型强制转换来完成。这种想法是错误的。

实例49 共用体变量

实例说明

有时，需要将几种不同类型的变量存放在同一段内存单元中。它们在内存中所占的字节数并不相同，但都从同一地址开始存放。也就是说，这几个变量是相互覆盖的。这时，我们便需要用到共用体。

本例是一个共用体使用的小程序，旨在使读者了解共用体的用法。

知识要点

共用体是多种变量共享的同一片内存，它提供了以多种方式解释同一种模式的方法。共用体的定义和结构类似，一般形式如下：

```
union tag
{
    type member-name;
    type member-name;
    ……
} union-variables;
```

例如以下代码：

```
union data
{
    int i;
    char ch;
    float f;
};
```

共用体的定义并不声明变量。变量的声明既可以通过把变量名放到定义的末尾进行，也可以用共用体类型进行定义。用刚刚定义的 data 可以声明 data 类型的联合变元 exa，如下所示：

```
union data exa;
```

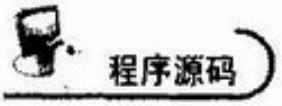
在 exa 中，整形变量 i 和字符型变量 ch，以及实形变量 f 共享同一片内存。i 占两个字节，ch 占一个字节，f 占四个字节。共用体变量所占用的内存长度等于最长的成员的长度，所以，exa 所占的内存长度为 4 个字节。

访问共用体中成员的语法和访问结构体中成员的语法相同，也是用圆点(.)和箭头(->)操作符。通过共用体变量名访问其元素时用圆点；通过共用体指针访问其元素时用箭头。

赋值语句如下所示：

```
exa.i = 15;
```

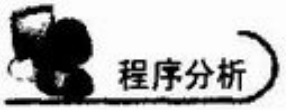
共用体常用于需要频繁进行类型转换的场合，这时程序便可以使用联合中的数据。



程序源码

该应用程序的源代码如下：

```
# include <stdio.h>
union data
    {
        int a;
        float b;
        double c;
        char d;
    } exa;
void main()
{
    exa.a = 6;  printf("%d\n", exa.a);
    exa.c = 67.2;  printf("%5.1f\n", exa.c);
    exa.d = 'W';  exa.b = 34.2;
    printf("%5.1f, %c\n", exa.b, exa.d);
}
```



程序分析

程序输出为：

```
6
     67.2
     34.2, ?
```

程序最后一行的输出是我们无法预料的。其原因是连续做了 exa.d = 'W'和 exa.b = 34.2 的赋值操作，两个连续的赋值语句最终使共用体变量的成员 exa.b 所占的四字节写入 34.2，而写入的字符被覆盖了，输出的字符便成了符号“？”。事实上，字符的输出是无法得知的，由写入内存的数据决定。例子虽然很简单，但说明了共用体变量的正确用法。

实例50 枚举类型

实例说明

枚举(enumeration)是一系列命名的整形常量，它是 ANSI C 新标准中增加的内容。生活中枚举的例子很多，如美国硬币的枚举是：

penny，nickel，dime，quarter，half-dollar，dollar

如果一个变量只有几种可能的值，那么便可以定义为枚举类型。

下面将以一个实例向读者讲解枚举类型的用法。实例原题是：口袋中有蓝、红、黄、紫、黑五种颜色的铅笔若干支，每次从口袋中取出三支，问得到三种不同颜色的铅笔的可能取法，打印出每种组合中的三种颜色。

知识要点

所谓枚举，就是将变量的值一一列举出来，变量的值只限于列举出来的值的范围内。枚举的定义很像结构，关键字 enum 表示开始定义。枚举定义的一般形式如下所示：

```
enum tag {enumeration list} variable_list;
```

其中，枚举标志(tag)和变量表(variable_list)都是可选的（至少要出现一个）。以下代码段便定义了 money 的枚举类型：

```
enum money {penny, nickel, dime, quarter, half_dollar, dollar};
```

枚举的标志名用于声明该类型的变量。以下代码声明 coin 为类型 money 的变量：

```
enum money coin;
```

这样，以下形式的语法完全合法：

```
coin = penny;
if(coin == nickel)
    printf("coin is a nickel.\n");
```

对于枚举，希望读者能够理解它的要点，就是每个符号都代表一个整数值。所以，个个符号都可以用于整数值的任何场合。每个符号值都取大于其前一个符号，第一个符号值是 0。所以，

```
printf("%d %d %d", penny, dime, dollar );
```

在屏幕上显示出 0、2 和 5。

通过初始化操作，指定一个或者多个符号的值。例如以下代码将值 100 赋给符号 dime，将 200 赋给符号 half_dollar：

```
enum money {penny, nickel, dime=100, quarter, half_dollar=200, dollar};
```

现在，各个符号的值如下所示：

```
penny          0
nickel         1
dime         100
quarter      101
half_dollar  200
dollar       201
```

还需要注意的是，枚举符号不可以直接输入和输出。这是因为，枚举符号本质上是整数值，而不是字符串。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
    // 经过下面的定义后，默认有：blue=0 red=1 ... black=4
    enum color {blue, red, yellow, purple, black};
    enum color i, j, k, pri;
    int n, loop;
    n = 0;

    for(i=blue; i<=black; i++)  // i 代表第一次所取铅笔的颜色
        for(j=blue; j<=black; j++)  // j 代表第二次所取铅笔的颜色
            if(i!=j)  // 第一次和第二次所取铅笔颜色不同
            {
                for(k=blue; k<=black; k++)  // k 代表第三次所取铅笔的颜色
                    if((k!=i)&&(k!=j))  // 三次所取铅笔颜色各不相同
                    {
                        n++;  // 能得到三种不同颜色铅笔的可能取法加 1
                        printf("%-6d", n);
                        // 将当前 i、j、k 所对应的颜色依次输出
                        for(loop=1; loop<=3; loop++)
                        {
                            switch(loop)
                            {
                            case 1: pri = i;
                                    break;
                            case 2: pri = j;
                                    break;
                            case 3: pri = k;
                                    break;
```

```
                default:
                        break;
                }
                switch(pri)
                {
                case blue:   printf("%-10s", "blue");
                             break;
                case red:    printf("%-10s", "red");
                             break;
                case yellow: printf("%-10s", "yellow");
                             break;
                case purple: printf("%-10s", "purple");
                             break;
                case black:  printf("%-10s", "black");
                             break;
                default:
                             break;
                }
            }
            printf("\n");
        }
    }
    printf("total: %5d\n", n);
}
```

程序分析

铅笔的颜色只可能是五色之一，要判断铅笔的是否相同颜色，就应该用枚举变量处理。设取出的铅笔为 i、j 和 k。由原题可知，i、j、k 分别是五种颜色之一，并且要求三支铅笔颜色互不相同，即 i≠j≠k。可以使用列举法，即尝试每一种方法，看看哪一种方法符合条件。

程序中用 n 累计取出的三种不同颜色铅笔的次数。第一次 for 循环（外循环）使第一支铅笔的颜色 i 从 blue 到 black。第二次 for 循环（中循环）是第二个铅笔的颜色 j 从 blue 到 black。如果 i 和 j 颜色相同则不可取，只有在 i 和 j 的颜色不相同时（i 不等于 j）才需要继续找第三个铅笔，此时第三个铅笔 k 也有五种可能的颜色（从 blue 到 black），但要求第三个铅笔不能和第一支铅笔或第二支铅笔同色（即 k 不等于 i，且不等于 j）。满足了这个条件，就得到了三种不同颜色的铅笔，使 n 加 1 并将这三种颜色输出。等外循环执行完毕后，全部可能的方案都已输出。最后输出总数 n。

下面要讨论的问题是如何输出一种取法，即如何输出各种颜色，包括 blue、red 等等。我们不能直接用 pirntf 函数来输出 blue 这个字符串，而采用了下面的方法。

为了输出三种铅笔的颜色，很显然应该经过三次循环，第一次输出 i 的颜色，第二次输出 j 的颜色，最后输出 k 的颜色。在三次循环中先后将 i、j、k 赋给变量 pri。然后根据 pri 的值

输出颜色信息。在第一次循环中，pri 的值为 i，如果 i 的值为 blue，则输出字符串 blue，其他依此类推。

现将程序的运行结果部分显示如下：

```
1        blue        red          yellow
2        blue        red          purple
3        blue        red          black
4        blue        yellow       red
5        blue        yellow       purple
6        blue        yellow       black
7        blue        purple       red
8        blue        purple       yellow
9        blue        purple       black
.........
.........
.........
60       black       purple       yellow
total:   60
```

实例51 读写字符

实例说明

在C语言中，最简单的I/O函数是getchar()和putchar()，两者的功能分别是从键盘读入一个字符和向显示屏显示一个字符。getchar()函数等待击键，发生击键后将读入值返回，并自动把击键结果显示在屏幕上。putchar()把传入的字符写在屏幕的当前光标处。

本例向读者介绍的正是getchar()和putchar()这两个函数。该程序是从键盘读入字符，改变字符的大小后再显示到显示屏上。当键入圆点(.)后，程序退出。

知识要点

函数getchar()和putchar()的原型如下所示：

```
int getchar(void);
int putchar(int c);
```

正如上述的原型所示，函数getchar()被定义为返回整数。当然，也可以将这个值按通常的方法付给变量char，因为该字符包含在低字节中，而高字节通常为零。如果函数调用失败，则返回EOF（-1）。

在函数putchar()中，参数被定义为整型，但也可以用字符型变量来调用它，它只向显示屏写其中的低字节。调用成功时，putchar()返回写出的字符；失败时，则返回EOF。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>
# include <ctype.h>

void main()
{
    char ch;

    printf("Please enter some text(input a point to quit).\n");
    do{
        ch = getchar();

        if(islower(ch))  // 察看ch是否为小些字符
```

```
            ch = toupper(ch);
        else
            ch = tolower(ch);
        putchar(ch);
    } while(ch != '.');
}
```

程序分析

程序的输出结果如下所示：

```
Please enter some text(input a point to quit).
Hello, I am a student.
hELLO, i AM A STUDENT.
```

程序中调用了函数 toupper()和 tolower()，它们都是包含在头文件<ctype.h>中的。

函数 toupper()的原型如下所示：

```
int toupper(int ch);
```

它的功能是在 ch 为字母时，返回等价的大写字母；否则返回的 ch 值不变。

函数 tolower()和 toupper()相似，只是返回的是等价的小写字母。

在我们使用 getchar()函数时，需要注意几个潜在的问题。函数 getchar()的原始形式中，输入先被缓冲，直到键入回车时才返回。这就是所谓的行缓冲输入。由于 getchar()每次只输入一个调用的字符，所以，行缓冲使得输入队列只遗留若干字符，这在交互环境中很不方便。

为了解决上述函数 getchar()的不能交互问题，在 C 中提供了另外两个函数，它们是 getch()和 getche()。它们的原型是：

```
int getch(void);
int getche(void);
```

它们一般包含在头文件<conio.h>中，getch()函数在键击之后立即返回，不向屏幕回显字符。getche()函数和 getch()函数功能相似，但是回显字符。在交互式程序中，当需要从键盘读入字符时，常常需要用函数 getche()或者 getch()来代替 getchar()。

实例52 读写字符串

实例说明

用于读写字符串的的函数是 gets()和 puts()。这两个函数允许程序从控制台读字符串和向控制台写字符串。

我们的程序是一个简单的计算机化字典，通过这个程序实例向读者示范怎样使用读写字符串的 I/O 函数。程序原题：让用户输入一个字，然后在内部数据库中查找它，如果发生匹配，程序便打印出该字的意义；否则，指出该字不在字典中。

知识要点

函数 gets()是将键盘输入的一个字符串读入，把读入结果放到指针变元指向的地址。用户键入字符直到回车，函数读入键入的字符，忽略回车符，然后在读入串的尾部自动加上 null（其值一般为 0）终止符。和函数 getchar()不同的是不能通过 gets()读回车符。函数 gets()的原型是：

```
char *gets(char *string)
```

其中，string 是接受用户键入串的字符数组。函数返回值也是同一个 string，然后打印出它的长度。

在使用 gets()时必须小心，这是因为它不对输入的数组进行边界检查。在这种情况下，用户键入的字符数可能多于数组能够容纳的字符数。由于这个缺陷，所以我们在正式的商业代码中很少使用它。

函数 puts()是把变元串写到显示屏上，然后再写一个新的行符。它的原型如下所示：

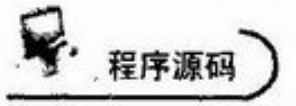
```
int puts(const char * string);
```

和 printf 函数相同，puts()也识别反斜线码，例如代表制表符的\t 等等。由于函数 puts()只处理字符串，所以调用 puts()时的开销要远远小于调用 printf()。这样，函数 puts()所占用的空间要比 printf()小，速度也更加快。所以，当输出时，若不需要格式转换，常常使用函数 puts()。

puts()调用成功时返回非零值；否则，返回 EOF。由于向控制台写出的时候不太可能会出错，所以在程序中很少检查 puts()的返回值。下面的语句

```
puts("Sorry!");
```

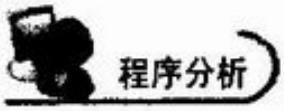
在屏幕上将显示出字符串 Sorry!。



该应用程序的源代码如下：

```
// A simple dictionary
# include <stdio.h>
# include <string.h>
# include <ctype.h>

char *dic[][40] = {
    "luster", "A bright shine on the surface.",
    "disgrase", "Loss of honor and respect.",
    "glamour", "Strong attraction.",
    "tomb", "The place where a dead person is buried.",
    "garbage", "Unwanted or spoiled food.",
    "bliss", "Great happiness or joy.",
    "commend", "Speak favorably of.",
    " ", " "    // null end the list
};
void main()
{
    char word[80], ch;
    char **point;    // 定义一个二维指针

    do{
        puts("Please enter word: ");
        scanf("%s", word);

        // 将二维数组首地址赋给二维指针 p
        point = (char **)dic;

        // 查看字典中是否存在输入的单词，存在则输出解释
        do{
            if(!strcmp(*point, word))
            {
                puts("The meaning of the word is: ");
                puts(*(point+1));    // 输出单词的意义
                break;
            }
            if(!strcmp(*point, word))
                break;
            point = point + 2;
        } while(*point);    // 判断字典是否结束
```

```
        if(!*point)
            puts("The word is not in dictionary.");

        printf("Another? (y/n):");
      // 下面的%*c 表示，scanf()读入该区域但不向任何变量赋值
        scanf("%c%*c", &ch);
    } while(toupper(ch)!='N');    // 判断输入是否为 n
}
```



程序的输出结果如下：

```
Please enter a word:
cluster
The meaning of the word is:
A bright shine on the surface.
Another? (y/n):y
Please enter a word:
car
The word is not in dictionary.
Another? (y/n):n
```

程序将输入的单词分别和字符串数组中的第 1、3、5...13 个字符串比较，若发现匹配，则输出其后的字符串（即单词解释）；否则输出语句：

```
The word is not in dictionary.
```

然后输出语句：

```
Another? (y/n):
```

若输入 y，则继续查找，否则将会结束程序。

实例53 格式化输出函数

实例说明

函数 printf()实现输出操作，即在程序设计者的控制下以各种格式写控制台。它是向显示器写数据的。printf()函数和前面介绍的函数 putchar()不同，putchar()只能输出字符，而 printf()可以输出各种类型的数据。

本实例将示范格式化输出函数 putchar()的几种主要输出格式，希望读者能够通过对实例的学习掌握它们。

知识要点

printf()函数的原型是：

```
int printf(const char * control_string, …);
```

函数成功时，返回输出的字符数；失败时，返回负值。

函数原型中的字符串 control_string 是由两类项目组成的：第一类是在屏幕上的字符，而第二类就是定义相应变元显示格式的格式说明符。由原型不难看出，格式说明符是由%开始，然后紧跟着格式码。格式说明符的数量必须和变元表 argument_list 中的变元数量严格一致，两者对应的顺序是从左到右一一对应。例如，printf()调用：

```
printf("I am %c %s", 'a', "student!");
```

在屏幕上将会输出：

```
I am a student!
```

在上面的语句中，格式说明符%c 匹配字符'a'，而%s 匹配字符串"student!"。

函数 printf()具有多种格式说明符。现将常用的部分列举如下：

printf()格式说明符	说明
%c	输出单字符
%d	%i 输出带符号的十进制整数
%e	%E 科学表示法（表示指数部分）
%f	输出十进制浮点数
%s	输出字符串
%o	输出无符号八进制整数
%u	输出无符号十进制整数
%x	输出无符号十六进制整数（小写）

%X	输出无符号十六进制整数（大写）
%p	显示指针
%%	显示百分号

上面列出的都是一些常见的格式说明符，希望读者能够记住它们。

在使用格式说明符时，常常用到最小域宽说明符，在百分号和格式码之间的数即是起最小域宽说明符的作用。在程序中将会有详细的讲解。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
    unsigned number;
    double item = 1.23456;

    for(number=8; number<16; number++)
    {
        printf("%o   ", number);  // 以八进制格式输出 number
        printf("%x   ", number);  // 以十进制格式(小写)输出 number
        printf("%X\n", number);   // 以十进制格式(大写)输出 number
    }
    printf("\n");

    printf("%p\n\n", &item);  // 显示变量 item 的地址

    printf("%f\n", item);
    printf("%8.2f\n", item);  // 总域宽为 8，小数部分占 2
    printf("%-8.2f\n", item);  // 域中左对齐输出(默认右对齐)
}
```

程序分析

程序较简单，程序结果可以根据程序中的注释分析得出，在这就不给出了。

实例54 格式化输入函数

实例说明

前面介绍的 getchar()函数只能用来输入一个字符，而使用 scanf()函数可以输入任何类型的多个数据。在前面的例题中，已经接触过 scanf()函数，在这将对它作详细的讲解。

知识要点

scanf()函数是通用的终端输入函数，可以读入任何固有类型的数据并把数值变换成适当的机内格式。scanf()函数很像前面所讲 printf()函数的逆函数。

scanf()函数的原型如下所示：

```
int scanf(const char * control_string, …)
```

scanf()返回成功赋值的数据项数，若出错则返回 EOF。控制串 control_string 用来确定怎样把值读进变元表中的相应变量。

控制串由三类字符构成：

（1）格式化说明符；

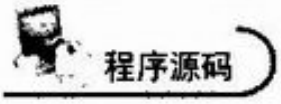
（2）空白符；

（3）非空白符。

格式化说明符是由%作先导，通知 scanf()随后应该读哪类数据。scanf()函数的格式码详列如下：

格式码	说明
%c	读单字符
%d %I	读一个十进制数
%e %f %g	读一个浮点数
%o	读一个八进制数
%s	读一个字符串
%x	读一个十六进制数
%p	读一个指针
%u	读一个无符号整数
%%	读一个百分号

希望读者能够牢记它们。



程序源码

该应用程序的源代码如下：

```
# include <stdio.h>

void main()
{
    int i, j, k;
    char str[80];
    char *p;

    // 输入的数将分别以十进制、八进制和十六进制读入程序
    scanf("%d %o %x", &i, &j, &k);
    printf("%d %d %d\n\n", i, j, k);  // 查看实际输入的数据

    printf("Enter a string: ");
    scanf("%s", str);
    printf("Here is your string: %s\n\n", str);

    printf("Enter an address: ");
    scanf("%p", &p);
    printf("Value at location %p is %c.\n", p, *p);
}
```

程序分析

程序中前两个输出模块的输出为：

```
20  20  20
20  16  32

Enter a string: Hello!
Here is your string: Hello!
```

程序的第三个输出模块只是介绍读地址的方法，但并不提倡这样做，因为大多数情况下对地址并不了解，如果任意输入地址让程序去读，那么很可能是内存中的未知地域，这对编程毫无意义。

实例55 打开和关闭文件

实例说明

下面介绍有关文件的操作。同其他高级语言一样，在对文件进行读写操作之前应该先打开该文件，在使用结束之后应该将它关闭。

本例介绍的正是文件的打开和关闭操作。它们是所有文件操作的基础，希望读者能够好好掌握。

知识要点

ANSI C 规定，在标准输入输出函数库中可以使用 fopen()函数打开文件。函数 fopen()的原型是：

```
FILE *fopen(const char *filename, const char *mode);
```

fopen()的调用方式通常如下所示：

```
FILE *fp;
fp = fopen(文件名, 文件使用方式);
```

例如：

```
FILE *fp;
fp = fopen("file1 ", "r");
```

上面语句表示要打开名字为 file1 的文件，使用的方式是“读入”（有关文件的使用方式，可查相关的书籍，本书由于篇幅有限，在这里就不作详细介绍了）。fopen()函数返回指向 file1 文件的指针并赋给 fp，这样 fp 就指向了文件 file1。如果函数 fopen()返回的是空指针，表示函数调用失败。

函数 fclose()的作用是用来关闭 fopen()所打开的文件。fclose()函数把遗留在缓冲区中的数据写入到文件中后，实施操作系统级的关闭操作。如果关闭失败会导致各种问题，如丢失数据、破坏文件和程序中的随机错。fclose()同时释放掉与流联系的文件控制块，以后可以再次使用这部分空间。

在大多数情况下，系统都限制同时处于打开状态的文件总数，所以，在打开文件前先关闭无用的文件是合理的。

函数 fclose()的原型是：

```
int fclose(FILE *fp);
```

其中，fp 是 fclose()返回的文件指针。fclose()返回零表示关闭文件，返回其他值表示关闭操作发生错误。一般而言，只有在驱动器中无盘或者盘空间不足时，才发生错误。

程序源码

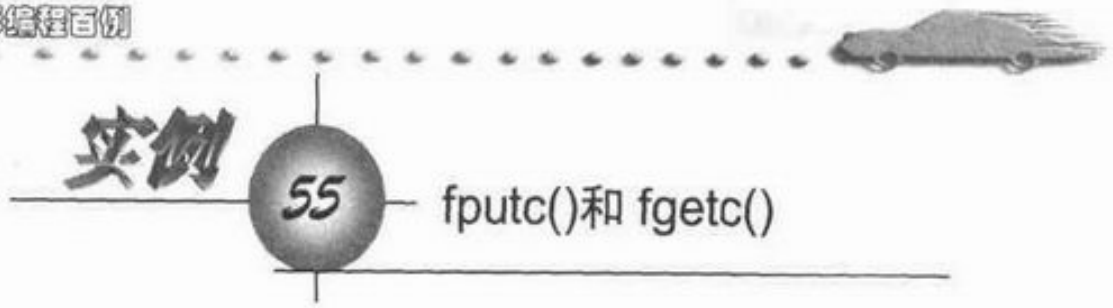
该应用程序的源代码如下：

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

void main()
{
    // 定义一个文件指针 fp
    FILE *fp;
    char ch, filename[10];

    printf("Please input the name of file: ");
    scanf("%s", filename);  // 输入字符串并赋给变量 filename

    // 以读的使用方式打开文件 filename
    if((fp=fopen(filename, "r")) == NULL)
    {
        printf("Cannot open the file.\n");
        exit(0);  // 正常跳出程序
    }

    // 关闭文件
    fclose(fp);
}
```

程序分析

本程序非常简单，首先要求用户输入文件的名字，然后使用函数 fopen()打开文件，如果文件打开不成功，则程序输出“Cannot open the file.”，然后跳出整个程序。如果打开成功，再将文件关闭，在文件的打开和关闭的操作过程中，没有做任何其他操作。

本例的目的仅仅是让读者熟悉和掌握文件打开和关闭的用法，不具有任何实际意义。



实例55 fputc()和fgetc()

实例说明

文件打开后，就可以对它进行读写操作了。常用的读写函数有fputc()函数和fgetc()函数。这两个函数对文件进行访问时，操作的对象都是字符，下面将通过实例对它们作详细介介绍。

本例原题是：从键盘输入一个字符串，将其中小写的字母全部转换成大写字母，然后再输出到磁盘文件“file”中保存。输入的字符串以“.”结尾。

知识要点

fputc()函数把一个字符写到磁盘文件上去，一般形式如下所示：

```
fputc(ch, fp);
```

参数ch表示要输出的字符，可以是一个字符型常量，也可以是一个字符型变量。fp代表文件指针变量，它从fopen()函数得到返回值。上述的fputc(ch, fp)函数的作用是将字符ch的值输入到fp所指的文件中去。函数fputc()有返回值，如果函数调用成功，那么返回值就是输出的字符；如果调用失败，那么返回EOF。

fgetc()函数执行的操作是从指定的文件读入一个字符，该文件必须是以读或者写的方式打开的。fgetc()的一般调用形式如下所示：

```
ch = fgetc(fp);
```

fp代表文件指针变量，ch代表字符型变量。fgetc()函数返回一个字符，赋给ch，当执行fgetc()遇到文件结束符时，函数会返回一个文件结束标志EOF。可以用如下的代码读文本文件，直到文本文件的结束：

```
do{
   ch = fgetc(fp);
} while(ch!=EOF);
```

当读入的字符为EOF(-1)时，表示读入的已不是正常的字符而是文件的结束标志（这是由于字符的ASCII码不可能出现-1，所以将-1作为文件结束标志是可取的）。需要注意的是，上述情况只适用于读文本文件。

程序源码

该应用程序的源代码如下：

```
# include <stdio.h>
```

```
# include <string.h>
# include <stdlib.h>
void main();
{
    FILE *fp;
    char str[100];
    int i;
    if((fp=fopen("file.txt", "w"))==NULL)
    {
        printf("无法打开文件\n");
        exit(0);
    }
    printf("请输入一个字符串: \n");
    gets(str);
    // 将字符串中的小写字符转换成大写字符，直到遇到"."为止
    for(i=0; str[i]!='.'; i++)
    {
        if(str[i]>='a' && str[i]<='z')
            str[i] = str[i] - 32;
        fputc(str[i], fp);  // 将转换后的字符存入文件
    }
    fclose(fp);
    fp = fopen("file.txt", "r");
    for(i=0; str[i]!='.'; i++)
        str[i] = fgetc(fp);  // 将文件中的字符串回传到 str 中去
    printf("%s\n", str);
    fclose(fp);
}
```

程序分析

程序的执行流程简述如下：从键盘输入字符串，将其中小写的字符转换成大写的字符后，存放到文件 file.txt 中；再从文件中将串读出，最后显示在屏幕上。

实例57 函数 rewind()

实例说明

当对文件进行操作时，文件定位函数 rewind()扮演着非常重要的作用。鉴于它的重要性，有必要将它单独列出作详细介绍，希望能够引起读者的充分的注意。

知识要点

函数 rewind()的功能是重置文件的位置，将其置到函数变元所指定的文件的开头部分，即该函数具有“重绕”文件的功能。函数 rewind()的原型如下所示：

```
void rewind(FILE *fp)
```

参数 fp 是有效的文件指针。函数 rewind()没有返回值。

程序源码

该应用程序的源代码如下：

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

void main()
{
    char str[80];
    FILE *fp;  // 定义一个文件类型的指针

    // 以写的方式打开文件 test
    if((fp=fopen("test.txt", "w"))==NULL)
    {
        printf("Cannot open file.\n");
        exit(0);
    }

    do{
        printf("Please enter a string: \n");
        gets(str);
        strcat(str, "\n");  // 增加一个新行
        fputs(str, fp);
```

```
    } while(*str!='\n');

    // 从文件中读出字符串,并将它们显示出来
    rewind(fp);  // 重置文件指针
    while(!feof(fp))
    {
        fgets(str, 79, fp);
        printf(str);
    }

    fclose(fp);
}
```

程序分析

在程序中调用了函数 strcat()，它是包含在头文件<string.h>中的。strcat()的函数原型如下所示：

```
char *strcat(char *str1, const char *str2);
```

函数执行的功能是将 str2 的一个副本连接到 str1 之后，在新形成的 str1 串的后面用一个‘\0’终止。其中 str1 的原终止符被 str2 第一个字符所覆盖，而 str2 的内容和操作无关。函数 strcat()返回指针 str1。

此外，在 while 循环中的结束条件处调用了函数 feof()。读者应该还记得，在前面讲到过，当访问文本文件时，可用 EOF 作为文件的结束标志。而在这用到了函数 feof()，它是 ANSI C 提供的正规的判断文件是否结束的函数。feof(fp)用来测试 fp 所指向的文件的当前状态是否为“文件结束”，如果文件结束，函数 feof()将返回值 1，否则返回值 0。函数 feof()也可用于判断非文本文件是否结束。

实例58 fread()和fwrite()

实例说明

前面所介绍的函数fgetc()和fputc()只可用来读取文件中的一个字符，但常常需要一次读入一组数据，ANSI C文件系统提供了fread()和fwrite()两个函数，它们可用于读写长于一个字节的数据类型。

在本程序中，通过函数fread()向文件中写入double、int和long类型的数据，然后再通过fwrite()函数读回。

知识要点

函数fread()和fwrite()的原型分别如下所示：

```
size_t fread(void *buffer, size_t num_bytes, size_t count, FILE *fp);
size_t fwrite(const void *buffer, size_t num_bytes, size_t count, FILE *fp);
```

对于函数fread()而言，变元buffer是存放所读入数据的内存区域的指针，而对于fwrite()函数而言，buffer是写入到那个文件的信息的指针。变元count的值确定将读写多少项，而每项的长度是由num_bytes决定的，变元num_bytes的类型是size_t，一般代表无符号整数。函数最后的变元fp是指针变量，是指向原先打开的文件。

函数fread()和fwrite()都有返回值。fread()返回读入的项数，如果出错或者达到文件的尾部时，返回值可能会小于count，fwrite()返回写出的项数，如果不出错，该值将等于count。

程序源码

该应用程序的源代码如下：

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

void main()
{
    FILE *fp;
    int i = 156;
    long l = 9701076L;
    double d = 3.456;

    if((fp=fopen("test", "w"))==NULL)
```

```
    {
        printf("不能打开文件.\n");
        exit(0);
    }

    // 通过 fread()函数，将数据写入文件
    fwrite(&i, sizeof(int), 1, fp);
    fwrite(&l, sizeof(long), 1, fp);
    fwrite(&d, sizeof(double), 1, fp);

    rewind(fp);

    // 通过 fwrite()函数，将数据读出文件
    fread(&i, sizeof(int), 1, fp);
    fread(&l, sizeof(long), 1, fp);
    fread(&d, sizeof(double), 1, fp);

    printf("i = %d\n", i);
    printf("l = %ld\n", l);
    printf("d = %f\n", d);

    fclose(fp);
}
```

程序分析

本程序的运行结果如下所示：

```
i = 156
l = 9701076
d = 3.456000
```

从上例不难看出，缓冲区也就是存放变量的内存本身。

实例59 fprintf()和 fscanf()

实例说明

除了前面讨论过的 I/O 函数之外，C 的 I/O 文件系统还包括 fprintf()和 fscanf()。除了操作的对象不同之外，这两个函数的操作和函数 printf()和 scanf()是完全一样的，都是格式化的读写函数。

下面将通过这样一个实例对 fprintf()和 fscanf()进行介绍。程序首先要求读者从键盘读入一个字符串和一个整数，然后将它们写入到磁盘文件中去，最后再从磁盘文件中把数据读出来，并显示在屏幕上。

知识要点

函数 fprintf()和 fscanf()的原型分别是：

```
int fprintf(FILE *fp, const char *control_string, …);
int fscanf(FILE *fp, const char *control_string, …);
```

变元 fp 是函数 fopen()返回的文件指针，而函数 fprintf()和 fscanf()是把 I/O 操作导向 fp 指明的文件。

例如：

```
int i = 3;
float t =2.56;
fprintf(fp, "i = %d, t = %4.2f", i, t);
```

代码表明了将整形变量 i 和实型变量 t 的值按照%d 和%4.2f 的格式输出到 fp 所指向的文件中去。在文件中可得到：i = 3, t = 2.560。

使用 fprintf()和 fscanf()时，需要在 ASCII 码和二进制码之间进行转换，这将花费较多的时间。所以，在内存和磁盘频繁交换数据的情况下，最好不用 fprintf()和 fscanf()函数，而使用 fread()和 fwrite()函数。

程序源码

该应用程序的源代码如下：

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

```
void main()
{
    FILE *fp;
    char str[80];
    int i;
    if((fp=fopen("test", "w"))==NULL)
    {
        printf("不能打开文件.\n");
        exit(0);
    }
    printf("Please enter a string and a number: \n");
    fscanf(stdin, "%s %d", str, &i);  // 参数 stdin 表示从键盘读入
    fprintf(fp, "%s %d", str, i);
    fclose(fp);

    if((fp=fopen("test", "r"))==NULL)
    {
        printf("不能打开文件.\n");
        exit(0);
    }
    fscanf(fp, "%s %d", str, &i);
    fprintf(stdout, "%s %d\n", str, i);  // 参数 stdout 表示写向屏幕
    fclose(fp);
}
```

程序分析

本例的工作流程如下：

以写的方式打开文件，从键盘输入数据并存储到文件中去，存储完毕后将文件关闭，再以读的方式打开文件，将数据从文件中读出，并在屏幕上输出，最后将文件关闭。

程序输出结果如下所示：

```
Hello!  1578
Hello!  1578
```

实例60 随机存取

实例说明

对于以流格式存储的文件可以进行顺序读写，也可以进行随机读写，关键是在于控制文件的位置指针，如果位置指针是按照字节位置顺序移动的，就是顺序读写，如果可以将位置指针按我们的需要任意移动，就可以实现随机读写。

借助于 fseek()函数就可以设置文件的位置，再使用 C 的 I/O 系统便可以完成对文件的随机读写操作。

在本例中示范了函数 fseek()的用法，程序在给定的文件中找到并显示字节。文件名和字节的位置是在命令行中给定的。

知识要点

所谓随机读取，就是指读写完上一个字符后，并不一定要读写其后续的字符，而是可以读写文件中任意所需的字符。fseek()函数的原型如下：

```
int fseek(FILE *fp, long int numbytes, int origin);
```

变元 fp 是函数 fopen()返回的指针，长整数 numbytes 是从原点 origin 到新位置的字节数，而变元 origin 取自<stdio.h>中定义的下列宏之一：

原点	宏名字	用数字代表
文件开始	SEEK_SET	0
文件当前位置	SEEK_CUR	1
文件末尾	SEEK_END	2

所以，当从起始位置寻找时，应该使用 SEEK_SET 宏。此时，函数 fseek()的操作是以原点为基点，向前移动 numbytes 个字节。若从当前位置寻找，则应该使用 SEEK_CUR 宏，而要从文件末尾进行寻找时，则需要使用 SEEK_END 宏。

下面是 fseek()函数调用的几个例子：

```
fseek(fp, 100L, 0);  // 将位置指针移到距文件头 100 个字节处
fseek(fp, 120L, 1);  // 将位置指针移到距文件当前位置 120 个字节处
fseek(fp, 20L, 2);   // 将位置指针从文件末尾处向后退 20 个字节
```

函数 fseek()具有返回值。当函数调用成功时，fseek()将返回零值，否则，返回非零值。

程序源码

该应用程序的源代码如下：

```
void main(int argc, char *argv[])
{
    FILE *fp;

    if(argc!=3)
    {
        printf("缺少字节位置，无法进行操作。\n");
        exit(0);
    }

    if((fp=fopen(argv[1], "rb"))==NULL)
    {
        printf("无法打开文件。\n");
        exit(0);
    }

    if(fseek(fp, atol(argv[2]), 0))
    {
        printf("寻找出现错误。\n");
        exit(0);
    }

    printf("在%ld处的字符是%c。\n", atol(argv[2]), getc(fp));
    fclose(fp);
}
```

程序分析

程序中函数atol()的原型是：

```
long int atol(const char *str);
```

它的功能是把str指向的串变成长整形的值。但串中必须含有合法的整形数据，否则所返回的值可能无法定义。本程序较简单，所以就不作详细分析。

实例 61 错误处理

实例说明

在调用各种输入输出函数时，如果出现错误，除了在函数返回值上有反映外，在 C 语言中，还专门提供了一些函数用以检查这些错误。

这样的函数主要包括 ferror()和 clearerr()。由于函数 clearerr()现在已很少使用，所以在这里只介绍 ferror()函数。

本程序所完成的功能是将文件中的制表符(tab)换成恰当数目的空格，每次读写操作后都调用 ferror()函数。

知识要点

函数 ferror()的原型是：

```
int ferror(FILE *fp);
```

原型中的变元 fp 是有效的文件指针。如果文件操作出错，那么函数将返回真值(1)，否则返回假值(0)。由于每个文件操作都置错误条件，所以在调用文件函数后应该立即调用 ferror()函数，以防止丢失错误状态。函数 ferror()是包含在头文件<stdio.h>中的。

注意：对同一个文件每一次调用输入输出函数时，都会产生一个新的 ferror 函数值，而并不是只产生一个。

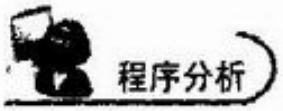
程序源码

该应用程序的源代码如下：

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

# define TAB_NUM 8    // 定义应换的空格数
# define IN 0
# define OUT 1

void error(int e)
{
    if(e == IN)
        printf("输入错误。\n");
    else
```

```
        printf("输出错误。\n");
    exit(1);    // 跳出程序
}

// 为使用该程序，应该指定命令行中的输入和输出文件名
void main(int argc, char *argv[])
{
    FILE *in, *out;
    int tab, i;
    char ch;

    if(argc != 3)
    {
        printf("用法出错。\n");
        exit(1);
    }

    if((out=fopen(argv[1], "wb"))==NULL)
    {
        printf("不能打开输入文件%s。\n", argv[1]);
        exit(1);
    }

    if((out=fopen(argv[2], "wb"))==NULL)
    {
        printf("不能打开输出文件%s。\n", argv[2]);
        exit(1);
    }

    tab = 0;
    do{
        ch = getc(in);
        if(ferror(in))
            error(IN);

        // 如果发现制表符，则输出相同数目的空格符
        if(ch == '\t')
        {
            for(i=tab; i<8; i++)
            {
                putc(' ', out);
                if(ferror(out))
                    error(OUT);
            }
```

```
            tab = 0;
        }
        else
        {
            putc(ch, out);
            if(ferror(out))
                error(OUT);
            tab++;
            if(tab==TAB_NUM)
                tab = 0;
            if(ch=='\n'||ch=='\r')
                tab = 0;
        }
    } while(!feof(in));

    fclose(in);    // 关闭输入文件
    fclose(out);   // 关闭输出文件
}
```



程序流程比较简单，在这就不介绍了。请读者注意每一次文件操作后是如何调用 ferror() 函数的。

实例62 综合实例

实例说明

本例开发了一个简单的通信录程序，主要用以示范用函数 fread()和 fwrite()方便地读写大量的数据，此外也涉及到其他一些知识点。

知识要点

有关文件操作部分的内容非常重要，许多可供实际使用的 C 程序都包含文件处理。希望读者能够在实践中掌握文件的使用。

程序源码

该应用程序的源代码如下：

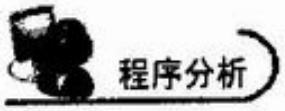
```
# include <stdio.h>
# include <stdlib.h>

# define MAX 100

struct addr
{
    char name[20];
    char street[40];
    char city[20];
    char state[4];
    unsigned long zip;
} addr_list[MAX];

void init_list(void);
void enter(void);
void dele(void);
void list(void);
void save(void);
void load(void);
int menu_select(void);
int find_free(void);
```

```
void main()
{
    char choice;

    init_list();
    for( ; ; )
    {
        choice = menu_select();
        switch(choice)
        {
        case 1: enter();
                break;
        case 2: dele();
                break;
        case 3: list();
                break;
        case 4: save();
                break;
        case 5: load();
                break;
        case 6: exit(0);
        }
    }
}

void init_list(void)
{
    register int t;

    for(t=0; t<MAX; t++)
        addr_list[t].name[0] = '\0';
}

void enter(void)
{
    int slot;
    char str[80];

    slot = find_free();

    if(slot == -1)
        printf("\nList Full");

    printf("Enter name: ");
```

```
    gets(addr_list[slot].name);

    printf("Enter street: ");
    gets(addr_list[slot].street);

    printf("Enter city: ");
    gets(addr_list[slot].city);

    printf("Enter state: ");
    gets(addr_list[slot].state);

    printf("Enter zip: ");
    gets(str);
    addr_list[slot].zip = strtoul(str, '\0', 10);
}

void dele(void)
{
    register int slot;
    char str[80];

    printf("Enter record #: ");
    gets(str);
    slot = atoi(str);

    if(slot>=0 && slot<MAX)
        addr_list[slot].name[0] = '\0';
}

void list(void)
{
    register int t;

    for(t=0; t<MAX; t++)
    {
        if(addr_list[t].name[0])
        {
            printf("%s\n", addr_list[t].name);
            printf("%s\n", addr_list[t].street);
            printf("%s\n", addr_list[t].city);
            printf("%s\n", addr_list[t].state);
            printf("%s\n\n", addr_list[t].zip);
        }
    }
```

```
    printf("\n\n");
}

void save(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "wb"))==NULL)
        printf("Cannot open file.\n");

    for(i=0; i<MAX; i++)
        if(*addr_list[i].name)
            if(fwrite(&addr_list[i], sizeof(struct addr), 1, fp)!=1)
                printf("File write error.\n");

    fclose(fp);
}

void load(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "rb"))==NULL)
        printf("Cannot open file.\n");

   init_list();
    for(i=0; i<MAX; i++)
        if(fread(&addr_list[i], sizeof(struct addr), 1, fp)!=1)
        {
            if(feof(fp))
                break;
            printf("File read error.\n");
        }

    fclose(fp);
}

int menu_select(void)
{
    char str[80];
    int c;
```

```
    printf("1. Enter a name\n");
    printf("2. Delete a name\n");
    printf("3. List the file\n");
    printf("4. Save the file\n");
    printf("5. Load the file\n");
    printf("6. Quit\n");

    do{
        printf("\nEnter your choice: ");
        gets(str);
        c = atoi(str);
    } while(c<0 || c>6);

    return c;
}

int find_free(void)
{
    register int t;

    for(t=0; addr_list[t].name[0]&&t<MAX; t++);

    if(t==MAX)
        return -1;

    return t;
}
```



程序中，通信地址是存储在结构类型树组 addr 中的，而 MAX 的值定义了表中存放的地址数。开始执行时，程序先把每个结构的 name 域置为 null，这是因为程序遵照惯例，认为名字长度为零的结构是未被使用的。

save()和 load()函数分别用来保存和装入通信录数据库。在这两个函数中，也检查 fread()和 fwrite()的返回值，由此确定文件操作是否成功。load()还必须直接使用 feof()函数查找文件结尾，因为不管是到达文件结尾还是出错，fread()的返回值都是一样的。

实例63 动态分配函数

实例说明

动态分配函数的核心是malloc()和free()。每次调用malloc()时，均分配剩余空内存的一部分；每次调用 free()时，均向系统返回内存。被分配的空内存区中的内存叫做堆(heap)。动态分配函数的原型在<stdlib.h>中。

本程序为用户输入的字符串分配内存，并释放该内存空间。

知识要点

函数 malloc()的原型如下所示：

```
void *malloc(size_t size);
```

函数返回一个指针，指向从堆中分配的内存区域（参数 size 的值）的首字节。当堆中的内存不能满足分配请求时，malloc()返回空指针。使用前，必须核实返回的指针不为空，否则将导致系统瘫痪。

函数 free()的原型为：

```
void free(void *ptr);
```

函数 free()向堆中返回 ptr 指向的内存，使内存可供将来再分配。

调用 free()的指针必须是先前使用动态分配系统函数分配而得到的，用无效的指针调用free()可能摧毁内存管理机制，使系统瘫痪。如果传递一个空指针，free()不做操作。

程序源码

该应用程序的源代码如下：

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

# define NUM 10

int main()
{
    char *str[NUM];  // 定义一个字符型的指针数组
    int t;

    // 为数组中的每个指针分配内存
```

```
    for(t=0; t<NUM; t++)
    {
        if((str[t]=(char *)malloc(128))==NULL)
        {
            printf("Allocation Error.\n");
            exit(1);
        }

        // 在分配的内存中存放字符串
        printf("Enter string %d: ", t);
        gets(str[t]);
    }

    // 释放内存
    for(t=0; t<NUM; t++)
        free(str[t]);

    // 由于主函数有返回值，故返回 0 值
    return 0;
}
```

程序分析

程序执行流程：

定义一个字符型指针数组 str，它里面放有 NUM(10)个字符型的指针。使用 for 循环为数组中的各个指针分配内存地址，大小都为 128 个字节。在分配内存的过程中，如果发现内存分配不成功（即函数 malloc()的返回值为 null），则给出警告语句“Allocation Error.”。

内存分配完毕后，通过输入字符串为每个指针赋初值。

最后，将所有的分配的内存释放。

实例 64 常用时间函数

实例说明

在 C 的标准库函数中定义了一些处理时间的函数。在这里将要介绍的是其中一些较常用的时间函数，它们分别是函数 time()、localtime()、asctime()和 gmtime()。希望读者能够掌握并在实际编程当中正确使用它们。

本程序是用于输出系统的本地时间和 UTC 时间。

知识要点

函数 time()的原型为：

```
time_t time(time_t *time);
```

函数 time()返回系统的当前日历时间，如果系统丢失时间设置，则函数返回-1。

对函数 time()的调用，既可以使用空指针，也可以使用指向 time_t 类型变量的指针。

函数 localtime()的原型为：

```
struct tm *localtime(const time_t *time);
```

函数 localtime()返回指向以 tm 结构形式的分解形式 time（时间）的一个指针。该事件表示为本地时间（即所用计算机上的时间）。变元 time 指针一般通过调用函数 time()获得。

函数 asctime()的原型为：

```
char *asctime(const struct tm *ptr);
```

函数 asctime()返回指向一个串的指针，其中保存 ptr 所指结构中存储的信息的变换形式，具体格式如下：

```
day month date hours:minutes:seconds year \n \0
```

例如：

```
Fri Apr 15 9:15:27 2001
```

由 ptr 指向的结构一般是通过调用 localtime()或 gmtime()得到的。

保存 asctime()返回的格式化时间串空间是静态空间变量，因此每次调用 asctime()时都用新串冲掉该静态字符数组中的原值。希望保存以前的结果时，应该复制它到别处。

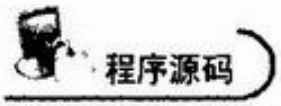
函数 gmtime()的原型为：

```
struct tm *gmtime(const time_t *time);
```

函数 gmtime()返回一个指针，指针指向以 tm 结构形式的分解格式 time。时间用 UTC(Coordinated Universal Time)即格林尼治时间表示，time 指针一般是通过调用 time()取得。如果系统不支持 UTC，则该函数返回空指针。

gmtime()函数用来存放分解时间的结构变量是静态分配的，每次调用 gmtime()时都需重写。希望保存结构中的内容时，必须把它复制到其他地方。

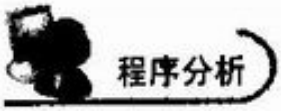
上述这些函数都是包含在头文件<time.h>中。



程序源码

该应用程序的源代码如下：

```
# include <time.h>
# include <stdio.h>

int main()
{
    struct tm *local;
    time_t tm;

    tm = time(NULL);
    local = localtime(&tm);
    printf("Local time and date: %s\n", asctime(local));

    local = gmtime(&tm);
    printf("UTC time and date: %s\n", asctime(local));

    return 0;
}
```



程序分析

程序的运行结果如下所示：

```
Local time and date: Sat May 10 16:05:18 2003

UTC time and date: Sat May 10 08:05:18 2003
```

实例 65 转换函数

实例说明

标准函数库中定义了一些工具函数，其中包括各种转换函数。在此，将要介绍这些转换函数，它们是函数 atof()、atoi()和 atol()。

知识要点

函数 atof()、atoi()和 atol()的原型分别如下所示：

```
double atof(const char *str);
int atoi(const char *str);
long int atol(const char *str);
```

函数 atof()把 str 指向的串转换成双精度浮点值。串中必须含有合法的浮点值，否则返回值将不准确。

函数 atoi()把 str 指向的串转换成整型值。串中必须含有合法的整型值，否则返回值将无意义。

函数 atol()把 str 指向的串转换成长整型值。串中必须含有合法的整型值，否则返回值将无定义。

变元 str 中的数可以由非有效浮点数中的任何字符结束，如空白符、除句号外的任何标点符号和 E 或 e 之外的任何字符等。因此，用“120.00HELLO”调用 atof()时，将返回值 120.00。

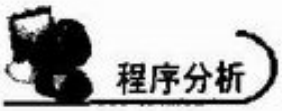
上述函数都是包含在头文件<stdlib>中的。

程序源码

该应用程序的源代码如下：

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

int main()
{
    char num1[80], num2[80];
    double sum1;
    int sum2;
    long sum3;
```

```
    printf("Enter first: ");
    gets(num1);
    printf("Enter second: ");
    gets(num2);
    sum1 = atof(num1)+atof(num2);
    printf("The sum is: %f\n", sum1);

    printf("Enter three: ");
    gets(num1);
    printf("Enter four: ");
    gets(num2);
    sum2 = atoi(num1)+atoi(num2);
    printf("The sum is: %d\n", sum2);

    printf("Enter five: ");
    gets(num1);
    printf("Enter six: ");
    gets(num2);
    sum3 = atol(num1)+atol(num2);
    printf("The sum is: %ld\n", sum3);

  return 0;
}
```



程序分析

程序共实现三个功能，分别是：

（1）输入两个字符串，将它们转换成浮点型数据，相加并输出。

（2）输入两个字符串，将它们转换成整型数据，相加并输出。

（3）输入两个字符串，将它们转换成长整型数据，相加并输出。

实例66 查找函数

实例说明

在 C 的标准函数库中，定义了查找函数，这就是 bsearch()函数。在本例中，将要介绍这个函数的用法。

本程序完成的功能是读入从键盘上输入的字符，确定所读入的字符是否属于已给出的字母数字字符表。

知识要点

函数 bsearch()是包含在头文件<stdlib.h>中的，它的原型如下所示：

```
void *bsearch(const void *key, const void *buf, size_t num, size_t size,
              int(*compare)(const void *, const void *));
```

函数 bsearch()对 buf 所指向的已排序数组实施对分查找(binary search)，返回和 key 指向的关键字匹配的第一个成员的指针。变元 num 说明数组中的元素数目，size 指出每个元素占有的字节数。

compare 指向的函数把数组元素与关键字(key)比较，compare 函数的形式必须是：

```
int func_name(const void *arg1, const void *arg2);
```

其返回值如下所示：

比较	返回值
arg1<arg2	<0
arg1=arg2	=0
arg1>arg2	>0

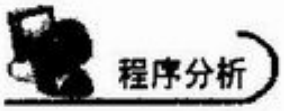
数组必须按照升序排列，最低地址处存放最低元素。如果数组中不含关键字(key)，函数返回空指针。

程序源码

该应用程序的源代码如下：

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

char *alpha = "abcdefghijklmnopqrstuvwxyz";
```

```
int comp(const void *ch, const void *s);

int main()
{
    char ch;
    char *p;

    printf("Enter a character: ");
    ch = getchar();
    ch = tolower(ch);    // 将变元 ch 转换成小写字符
    p = (char *)bsearch(&ch, alpha, 26, 1, comp);
    if(p)
        printf("%c is in alphabet\n", *p);
    else
        printf("is not in alphabet\n");

    return 0;
}

int comp(const void *ch, const void *s)
{
    return *(char *)ch -*(char *)s;
}
```



程序的运行结果如下所示：

```
Enter a character: a
a is in alphabet

Enter a character: 12
is not in alphabet
```

实例67 跳转函数

实例说明

在C的标准库中，函数setjmp()和longjmp()提供了一种在函数间跳转的手段，若要使用它们，必须用到头文件<setjmp.h>。

本实例中的程序介绍了上述跳转函数的用法，程序本身的功能很简单，只是输出1、3、5。

知识要点

函数setjmp()实际上是一个宏函数，它的原型为：

```
int setjmp(jmp_buf envbuf);
```

宏函数setjmp()在缓冲区envbuf中保存了系统堆栈的内容，以供函数longjum()以后使用。调用宏函数setjmp()时，返回值为零。然而函数longjmp()把一个变元传递给setjmp()，该值（恒为非零）就是调用longjmp()后出现的setjmp()的值。

函数longjmp()的原型如下所示：

```
void longjum(jmp_buf envbuf, int status);
```

函数longjmp()使程序在最近一次调用setjmp()处重新执行。longjmp()通过把堆栈复位成envbuf中描述的状态进行操作，envbuf的设置是由预先调用setjmp()生成的。这样使程序的执行在setjmp()调用后的下一个语句重新开始，使计算机认为从未离开调用setjmp()的函数。从效果上来看，函数longjmp()的调用似乎绕过内存回到了程序中的原点，不必执行正常的函数返回过程。

缓冲区envbuf具有<setjmp.h>中定义的jum_buf类型，它必须在调用longjum()前通过调用setjmp()来设置好。

值status变成setjmp()的返回值，由此确定长跳转的来处。不允许的唯一值是零。零时程序直接调用函数setjmp()是由该函数返回的，而不是间接通过执行函数longjmp()返回的。

函数longjmp()最常用于在一个错误发生时，从一组深层嵌套的实用程序中返回。

程序源码

该应用程序的源代码如下：

```
# include <setjmp.h>
# include <stdio.h>
```

```
jmp_buf ebuf;    // 类型在<setjmp.h>中定义
void fun(void);

int main()
{
    int i;

    printf("1 ");
    i = setjmp(ebuf);
    if(i == 0)
    {
        fun();
        printf("This will not be printed.");
    }
    printf("%d\n", i);

    return 0;
}

void fun(void)
{
    printf("3 ");
    longjmp(ebuf, 5);
}
```

程序分析

程序的执行流程如下所示：

首先输出数字 1，然后调用函数 setjmp()，由于这是直接调用，所以函数的返回值为 0。进入到判断语句后，执行子函数 fun()，它的功能是输出数字 3，再调用函数 longjmp()，由此会再次引发函数 setjmp()的调用，所不同的是此时函数 setjmp()的返回值不再是 0，而是 longjmp()的第二个参数 5，此时在执行判断语句便不再符合条件，所以直接执行主函数中的最后一个输出语句，输出数字 5。所以程序的输出结果为 1 3 5。

实例68 排序函数

实例说明

qsort()函数是C标准库中提供的排序函数。由于这个函数的存在，在很多情况下，不再需要自编写排序函数进行排序，而可以直接调用qsort()函数完成工作，这样能够很大地节省时间和精力。

因此，非常有必要介绍几个函数，使读者能够熟练使用它。

本程序是对一整数列表进行排序，并且显示出排序后的结果。

知识要点

函数qsort()的原型如下所示：

```
void qsort(void *buf, size_t num, size_t size,
           int(*compare)(const void *, const void *));
```

它是包含在头文件<stdlib.h>中的。

函数qsort()使用了Quicksort算法对buf指向的数组进行排序。Quicksort算法一般被认为是最佳的通用排序算法。变元num指出数组中元素的数目，而size说明每个元素的大小（按字节）。

compare指向的函数用于比较数组的两个元素，compare函数必须是如下的形式：

```
int func_name(const void *arg1, const void *arg2);
```

其返回值必须为：

比较	返回值
arg1<arg2	<0
arg1=arg2	=0
arg1>arg2	>0

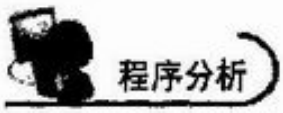
数组必须按照升序排列，最低地址处存放最低元素。如果数组中不含关键字(key)，函数返回空指针。

程序源码

该应用程序的源代码如下：

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

```
int num[12] = {
    14, 5, 9, 7, 6, 0, 91, 4, 1, 3, 2, 8
};

int comp(const void *, const void *);

int main()
{
    int i;

    printf("Original array: ");  // 输出原数组
    for(i=0; i<12; i++)
        printf("%d ", num[i]);
    printf("\n");

    qsort(num, 12, sizeof(int), comp);

    printf("Sorted array: ");  // 输出排序后的数组
    for(i=0; i<12; i++)
        printf("%d ", num[i]);
    printf("\n");

    return 0;
}

int comp(const void *i, const void *j)
{
    return *(int *)i - *(int *)j;
}
```



程序较简单，读者可自行分析。希望通过这个实例，能够掌握排序函数 qsort()。

实例 69 伪随机数生成

实例说明

标准函数库中定义了一些函数用来生成随机号码，它们就是本例中要讨论的函数 rand() 和 srand()。

本例中的程序是用系统时间通过使用 srand() 函数随机的初始化 rand() 函数。

知识要点

rand()和 srand()函数都包含在头文件<stdlib.h>中，所以在程序中如果要使用它们，就必须包含头文件<stdlib.h>。

rand()函数的原型为：

```
int rand(void);
```

函数 rand()产生伪随机数序列，每次它被调用时都返回一个 0 到 RAND_MAX 之间的整数。RAND_MAX 的值至少是 32767。

函数 srand()的原型为：

```
void srand(unsigned int seed);
```

函数 srand()为 rand()生成的伪随机数序列设置起点（rand()函数返回为随机数）。

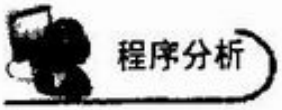
srand()一般用于多道程序运行时通过制定不同的起点而使用不同的伪随机数序列。相反，也可以使用 srand()反复地产生同一位随机数序列，其方法是在该序列开始以前用同一参数调用它。

程序源码

该应用程序的源代码如下：

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

// 利用系统时间寻找随机数，并将前十个随机数显示出来
int main()
{
    long time1;
    int i, time2;
```

```
    // 获得正确的日历时间
    time1 = time(NULL);    // 返回系统的当前日历时间
    printf("%ld\n", time1);

    time2 = (unsigned)time1/2;
    printf("%ld\n", time2);

    // 以系统时间为参数，为即将生成的伪随机数序列设置起点
    srand(time2);

    // 生成十个伪随机数序列
    for(i=0; i<10; i++)
        printf("%d ", rand());
    printf("\n");

    return 0;
}
```



程序的输出结果如下所示：

```
1052618131
526309065
22386 24218 19176 4316 11735 7183 3306 9994 29944 20210
```

在输出结果中，第一个数是系统当前的日历时间，它是系统内部对时间的一种表示方法。函数 srand()以系统的日历时间为参照，对伪随机数序列的起点进行设置。设置完毕后，在 for 循环中调用 rand()函数生成伪随机数，由于 for 循环只执行十次，所以程序只随机生成十个伪随机数，并将它们分别打印出来。

实例70 可变数目变元

实例说明

宏 va_arg()、va_start()和 va_end()一起作用，便可以完成向函数传入数目可变的变元操作。取可变数目变元的典型例子是函数 printf()。类型 va_list 是在<stdarg.h>中定义的。

下列程序是用 sum_series()求一系列的总和，sum_series()的第一个变元是数列项目数。本程序是求以下数列的前四个数之和：

$$\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}+\frac{1}{2^4}\cdots\cdots\frac{1}{2^N}$$

知识要点

上述的宏原型如下所示：

```
type va_arg(va_list argptr, type);
void va_end(va_list argptr);
void va_start(va_list argptr, last_parm);
```

它们都包含在头文件<stdarg.h>中。

创建一个能获取可变数目变元的函数的通用过程如下所示：在函数定义中，可变参数表之前必须有一个或多个已知参数，其中最右者为 last_parm。在调用 va_start()时，last_parm 名被用作第二个参数。

使用任何可变长度的变元被访问之前，必须先用 va_start()初始化变元指针 argptr。初始化 argptr 后，经过对 va_arg()的调用，以作为下一个参数类型的参数类型，返回参数。最后取完所有参数并从函数返回之前，必须调用 va_end()，由此确保堆栈的正确恢复。

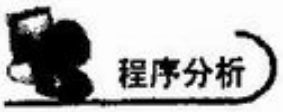
如果未正确使用 va_end()，程序可能瘫痪。

程序源码

该应用程序的源代码如下：

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

double sum_series(int num, ...);

int main()
```

```
{
    double d;
   // 在子函数实际调用中共有五个参数，第一个为序列个数，其后为相加的各个数
    d = sum_series(4, 0.5, 0.25, 0.125, 0.06254);
    printf("Sum of series is %f.\n", d);

    return 0;
}

double sum_series(int num, ...)
{
    double sum = 0.0, t;
    va_list argptr;  // 定义参变量

    // 初始化 argptr
    va_start(argptr, num);

    // 计算序列之和
    for( ; num; num--)
    {
        t = va_arg(argptr, double);
        sum = sum + t;
    }

    va_end(argptr);  // 序列关闭
    return sum;
}
```



程序的输出结果是 0.937500。

第三篇
常用算法篇

前两篇通过一系列实例对 C 语言的一些基础知识和常用的语法做了详细的阐述，在这一篇中将重点讲述如何用 C 语言实现一些常用的算法。

程序是由算法和数据结构组成，一个优秀的程序是两者的完美结合。通常，由编程问题的性质决定信息组织的方法。程序设计者应该掌握正确存储和检索数据的多种方法，以适应各种情况的需要。因此在本篇开始部分，首先对链表，队列，堆栈和树等 C 语言中常用到的数据结构做一个详尽的介绍，通过几个简单的例子使读者对这些不同数据引擎的使用和特点有一个具体的概念。接下来再通过具体实例讲述如何用 C 语言实现排序、插值、拟合等一些典型的算法，在此过程中希望读者可以对如何用 C 语言实现某种数学算法有一个清晰的思路，并能够仿照本篇的实例，用 C 语言实现其他更复杂的数学运算，达到举一反三的效果。

实例71 链表的建立

实例说明

该实例旨在通过建立一个简单的链表，让读者对链表这个 C 语言中常用的数据结构有一个直观的概念，并通过对本例的模仿，掌握新建一个链表的基本思路和方法。

本例讲述的是一个输入整数序列，并按整数的输入顺序建立一个整数链表。本例用一个函数实现建立链表的过程。函数采用以下算法思想：从空链表开始，每输入一个整数，向系统申请一个表元存储空间，将输入整数存入新表元并将新表元接在链表末尾。当不能输入一个整数时，函数结束建立链表的过程，返回链表的头指针值。

知识要点

在 C 语言中，程序可根据需要申请和释放存储空间来建立各种动态数据结构。所谓动态数据结构，是指一组逻辑相关的数据对象，其包含的数据对象的个数及数据对象之间的关系可按需要改变。反映一个运动着的世界，动态数据结构是强有力的描述手段。链表就是一个典型的动态数据结构。

链表中有一个指针指向链表的第一个表元，习惯称该指针为“头指针”或“链表头”。链表中的每个表元包含两部分内容：表元的实际数据信息和后继表元指针。头指针指向第一个表元，第一个表元又指向第二个表元……直到最后一个表元，最后一个表元不再指向其他表元。习惯称链表的最后一个表元为“表尾”。

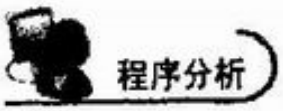
在链表结构中，一个表元的后继表元是由自己所包含的指针指出的，与后继表元在内存中的存放位置无关。

程序源码

该应用程序的源代码如下：

```
//建立一个整数链表
#include <stdio.h>
#include <stdlib.h>
struct chain
{
    int value;
    struct chain *next;
};
```

```
struct chain *create()
{
    struct chain *head, *tail, *p;
    int x;
    head = tail = NULL;
    printf("Input data.\n");
    while (scanf("%d",&x) == 1) //如果输入的是一个整型的数据，那么向下执行
    {
p = (struct chain *)malloc (sizeof (struct chain));
//首先为要新创建的表元 p 开辟一个内存空间
        p->value = x;
        p->next = NULL;
        if(head == NULL)
            head = tail = p;
        else
//tail 为倒数第二个表元指针，tail->始终指向最后一个表元
            tail = tail ->next;
            tail ->next = p;
    }
    return head;
}

void main(){
    struct chain *p,*q;
    q = create();
    while(q) {
        printf("%d\n",q->value);
        p = q->next;
        free(q);
        q = p;
    }
}
```



上程序实现链表的基本思路如下：

首先建立一个链表的结构，它由两部分组成，第一部分为链表要存储的数据，也就是链表中的信息域，本例中一个链表存储一个整型的数据（value）；另一部分为链指针（next），用于指向后一表元。

链表的建立是由一个函数 creat（）完成的，该函数的基本实现思路如下所示：

```
{
    建立一个空链表;
    while(scanf(" %d", &x) == 1)
```

```
    {//能为x读入整数循环
        申请新表元存储空间;
        将输入值存入新表元;
        将新表元接在链表末尾;
    }
    返回链表首指针;
}
```

函数 creat()将新表元接在链表末尾，如果表元接入链表的位置没有要求，最简单的办法是将新表元插在链表首表元之前，即让新表元作为链表的新的首表元。若按此改写函数 creat()，变量 tail 就不再需要了。另外，语句“p->next=NULL;”和其后的 if 控制结构可用以下两个赋值代替：

```
p->next = h;
h       = p;
```

该例子的 creat()函数是通过判断 scanf()的返回值决定是否继续建立新的表元。当输入一个整型数据时，scanf()函数返回值是 1；当发生错误操作，如输入的数据不是整型数据而是一个字符时，函数 scanf()返回值为 0。所以可以通过输入一个非整型的数据，如“a”来结束 creat()函数的执行。

本示例中，creat()函数通过语句 malloc (sizeof (struct chain))为一个新的表元分配一块内存区，当该函数创建的新的表元不再使用时，一定不要忘记用函数 free()释放内存，如果忽略这一步，malloc()函数所开辟的内存空间将永远不能释放，直到重新启动计算机。本例是在 main()函数中显示各个表元后立即释放表元占用的内存空间。

实例72 链表的基本操作

实例说明

上例讲述了链表的建立方法，在本例中，将重点学习链表的插入和删除等基本操作。希望读者通过学习链表的简单操作，能够对链表这种数据结构有更深一步的理解。

本例是在上例建立的一个链表的基础上，完成在指定位置上增添和删除一个链表节点的操作，并在主函数中将被修改的链表输出。链表的添加和删除主要是通过两个函数完成的：inlink（）和 dellink（）。希望读者在阅读本程序时将注意力重点集中到这两个函数，领悟其实现的思想。

知识要点

这里将重点介绍一下链表插入和删除一个表单元实现的思想。

节点的插入：设有数据（$a_1,a_2,\ldots,a_i,\ldots,a_n$）,用链表进行存储，表头指针为 head，要求在数据域值为 a 的表元之前插入一个数据域值为 b 的新节点。设存储数据域值为 a 的节点为 p，其直接前趋节点为 q。插入时首先调用一个新节点 s，在 s 节点的数据域中存入 b，再令 s 节点的指针指向 p 节点，然后使 q 节点的指针指向 s 节点。可见这种插入操作只改变了两个指针域的值，并未对数据元素做任何移动。

节点的删除：设有一个以 head 为头指针的线性链表，要求在该链表中删除数据域等于 a 的节点，若无此节点则给出相应的信息。删除操作和插入操作一样，需要搜索链表以找到指定的节点。找到被删除的节点后，只要将其直接前趋节点的指针改为指向被删除节点的直接后继节点就可以了。

程序源码

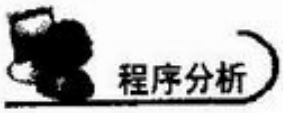
该应用程序的源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
struct chain
{
    int value;
    struct chain *next;
};
```

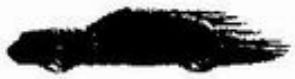
```
struct chain *create()
{
    struct chain *head, *tail, *p;
    int x,i;
    head = tail = NULL;
    printf("请输入四个整型数据，然后回车:\n");
    for(i= 0;i < 4; i++)
    {
        scanf("%d",&x);
        p = (struct chain *)malloc (sizeof (struct chain));
        p->value = x;
        p->next = NULL;
        if(head == NULL)
            head = tail = p;
        else
            tail = tail ->next = p;
    }
    return head;
}

struct chain *inlink(head,a,b)
struct chain *head;
int a, b;
{
    struct chain *p, *q, *s;
    s = (struct chain *)malloc(sizeof(struct chain));
    s->value = b;
    //插入空表
    if(head == NULL)
    {
        head = s;
        s->next = NULL;
    }
    //插入 s 节点作为新表头
    if(head->value == a)
    {   s->next = head;
        head = s;
    }
    else
    {
        p = head;
    //遍历链表，寻找数据域值为 a 的节点
        while ((p->value != a)&&(p->next != NULL))
        {
```

```
            q = p;
            p = p->next;
        }
        if(p->value == a)              //找到数据域为a的节点
        {
            q->next = s;
            s->next = p;
        }
    //插入节点s作为表尾
        else
        {
            p->next = s;
            s->next = NULL;
        }
    }
    return(head);
}

struct chain *dellink(head,a)
struct chain *head;
int a;
{   struct chain *p,*q;
    if(head == NULL)
        printf("空链表\n");
    else if(head ->value == a)
    //链表的第一个节点即为a节点
    {
p = head;
        head = head->next;
    }
    else
    {
        p = head;
        while ((p->value != a)&&(p->next != NULL))
        //在链表中搜索数据为a的节点
        {
            q = p;
            p = p->next;
        }
        if(p->value != a)
        //在链表中无数据值为a的节点
            printf("没有要删除的数据 %d\n",a);
        else
        {
```

```
            q ->next = p->next;
            free(p);
        }
    }
    return(head);
}

void main()
{
    struct chain *q,*head;
    int a, b;
    q = create();
    head = q;
    while(q)                          //显示链表
    {
        printf("%d\n",q->value);
        q = q->next;
    }
    printf("请输入新插入的表元数据位于那个数据之前：");
    scanf("%d",&a);
    printf("\n 请输入要插入的表元数据： ");
    scanf("%d",&b);
        q = inlink(head,a,b);
    head = q;
    while(q)                          //显示链表
    {
        printf("%d\n",q->value);
        q = q->next;
    }
    printf("请输入要删除表元的数据： ");
    scanf("%d",&a);
    q = dellink(head,a);
    while(q)                          //显示链表
    {
        printf("%d\n",q->value);
        q = q->next;
    }
}
```



为了保持连贯性，该程序沿用上一个实例中链表的建立方法，由用户输入自己建立一个整型数据的链表，建立链表的过程由 creat（）函数完成，其思路与上例基本相同，只是在本例中为了方便介双绍其他内容，规定刚开始输入的链表包含 4 个节点。



下面，看一下链表节点的插入函数 inlink（）。首先，通过语句 s = (struct chain *)malloc(sizeof(struct chain))建立一个新的节点，在这个节点的数据域中存放要插入的数据 b。一般情况下的插入操作，可能会遇到以下 4 种情况：

（1）链表是空链表（对应语句为 if(head == NULL)），则插入节点为表头。

（2）插入位置在表中第一个节点之前（if(head->value == a)，则插入节点为新的表头。

（3）插入节点在表的中央，则通过下面循环语句对链表中的各个节点的数据域逐个检索。

```
while ((p->value != a)&&(p->next != NULL))
{
    q = p;
    p = p->next;
}
```

如果当前节点 p 的数据域 value 不是要查找的值 a，且当前节点不是最后一个表单元，也就是 p->next 不是 NULL，那么通过语句 p = p->next 来检索下一个节点，直到找到包含要查找的值 a 的节点或者查找到链表的最后一个单元。所以，只有查找到插入节点，程序才跳出上面这个 while 循环，也就是由于不满足 while ((p->value != a)&&(p->next != NULL))中的(p->value != a)条件而跳出 while 循环。那么显然此时的 p 就为查找到的节点指针。由于在 while 循环中赋值语句 q = p 在语句 p = p->next 前面执行，所以此时指针 q 是指向数据域为 a 的节点的前一个节点，也就是 q->next 指向数据域为 a 的节点。此时通过赋值语句 p->next = s 和 s->next = p 把新建的 s 节点插入到链表中去。遍历整个链表，没有找到数据域为 a 的节点，也就是不满足 while ((p->value != a)&&(p->next != NULL))中的(p->next != NULL)条件而跳出 while 循环，那么同样易得此时 p 为链表最后一个节点的指针，所以通过语句 p->next = s 和 s->next = NULL 将新节点 s 作为链表的最后一个单元。

（4）如果链表中根本不存在指定的节点，则把插入节点做为新的表尾的节点，然后插入。

链表节点的删除跟链表节点的插入实现的思路基本一样，同样分四种情况，如果 head 为 NULL，则程序提示该链表为空链表；如果要删除的节点位于链表的表头，那么将下一个节点指针做为 head，并用函数 free（）将要删除的链表节点占用的内存空间释放；如果是删除链表中间的一个节点，将其找到后，把它的前趋节点的指针改为指向被删除节点的直接后继节点，并释放删除节点所占据的内存空间；如果链表中没有找到要删除的节点，则程序提示无相应节点。其具体实现方式请参看程序源码及其对应的注释。

实例 73 队列的应用

实例说明

该实例重点讲述的是 C 语言中另外一个常用的数据结构：队列。通过队列的插入和删除等基本操作，使读者对队列的结构和其如何实现有个清晰的概念，为今后灵活应用这种数据结构打下一定的基础。

本例中队列的基本操作是由四个函数完成的，它们是置空队列函数 SetNull()、判断队列是否为空函数 Empty()、入排操作函数 EnQueue()、出排操作函数 DelQueue()，在主函数中程序对其进行简单的调用，并显示调用效果，给读者一个更加直观的概念。

知识要点

队列（queue）是一种信息的线性列表，以先进先出的顺序访问。首先进队列的信息最先出队，第二进队的信息随后出队，依次类推。这是队列中存取的唯一方法，不许对其随机访问。

队列可用数组向量 q[Max]存储，队列所容许的最大容量为 Max，为了指示排头和排尾，需要引入两个指针：front－用于指向实际排头元素前一个位置；rear－用于指向实际排尾元素所在位置。由于 front 和 rear 是位置对应向量的下标，所以在进行具体的程序设计时，这两个变量应定义为整型变量，而不是指针变量。由 front 和 rear 的定义易得，当这两个指针相等时，队列为空。如果有 n 个数据入排，则尾指针 rear 的值自加 n，如果有 m 个数据出排，那么让头指针 front 的值自加 m。如果 front 或者 rear 的值达到队列的最大容量 Max，若 rear 为 Max，如果此时有一个数据再次入排，则让 rear 翻转为 0。在向量结构上采用这种方法存储的队列称为循环队列。

由上面的介绍易得，入排时，当 rear 加 1 后 rear = front，则为满排，也就是队列已经装满。而在出排时，让 front 加 1 后，如果 front = rear，则表示队列已经排空。

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>
#define Max 100                    //定义程序的最大容量为 100

void SetNull(front, rear)          //置空队列
int *front, *rear;
{
```

```
    *front = 0;
    *rear = 0;
}

int Empty(front,rear)                    //判空队列
int *front, *rear;
{
    if(*front == *rear)
        return(1);
    else
        return(0);
}

int EnQueue(q,x,front,rear)              //入排操作
int q[];                                 //存储队列的向量
int x;                                   //要插入队列的数据
int *front,*rear;
{
    *rear = (*rear+1) % Max;
    if(*front == *rear)
    {
        printf("队列发生上溢\n");
        return(-1);
    }
    else
    {
        q[*rear] = x;
        return(0);
    }
}

int DelQueue(q,y,front,rear)             //出排操作
int q[];                                 //存储队列的向量
int *y;                                  //提出队列的数据
int *front,*rear;
{
    *front = (*front +1)%Max;
    if(*front == *rear)
    {   printf("队列发生下溢\n");
        return(-1);
    }
    else
    {   *y = q[*front];
        return(0);
```

```
    }
}

void main()
{
    int q[Max];          //建立向量用来存储队列
    int f = 0, r = 0;    //f 和 r 分别对应队列的头和尾在整个队列存储区域的位置
    int i,x,m,n;
    int a;
    SetNull(&f,&r);          //清空队列
    printf("要输入队列的字符个数: \n");
    scanf("%d",&m);
    printf("输入队列的整型数据: \n");
    for (i=0; i<m; i++)
    {
        i=i;
        scanf("%d",&x);
        a = EnQueue(q,x,&f,&r);
        if(a == -1)
            break;           //如果发生上溢，则中断循环
    }
    printf("要提出队列的字符个数: ");
    scanf("%d",&n);
    printf("输出从队列中提取的数据: \n");
    for (i = 0; i<n; i++)
    {
        if(DelQueue(q,&x,&f,&r) == -1)      //如果发生下溢，则中断循环
            break;
        printf("%d\n",x);                   //逐个打印提取出队列的数据
    }
    if(Empty(&f,&r) == 1)
        printf("队列为空");
    else
        printf("队列中还有%d 个数据",(m-n));
}
```

程序分析

该实例通过 4 个函数完成对队列的操作：置空队列 SetNull()；判断队列是否为空 Empty()；入排操作 EnQueue()；出排操作 DelQueue()。SetNull()函数通过给 front 和 rear 指针指向对单元赋值为 0 实现对队列置空操作，而 Empty()函数则通过判断 front 和 rear 指针对应的数据是否相等来判断当前队列是否为空，其实现的数据都比较简单，这里就不再详述，由读者对照源程序以及注释自己分析。下面将重点讲述一下入排操作函数 EnQueue()和出排操作函数

DelQueue()。

主函数调用 EnQueue()时给它传递来四个参数，分别是存储队列的向量 q[]，其长度取决于调用该函数时外界传递给它队的参数。参数 x 为要插入队列的数据，*front 和*rear 分别为当前队列数据头尾在向量中的位置。设插入一个新的数据，由知识要点中队列实现的介绍可知，需要将*rear 的值自加 1，而如果此时*rear = Max，则将*rear 翻转为 0，在 EnQueue()函数中由语句*rear = (*rear+1) % Max 实现上述功能。当(*rear+1)小于 Max 时，其值对 Max 求余还是其本身，而当(*rear+1)等于 Max 时，对 Max 求余则将翻转为 0。然后 EnQueue()再通过语句 if(*front == *rear)判断存储在向量中的当前队列首尾指针是否相等，如果相等，则表示满排，系统做相应提示；否则，通过语句 q[*rear] = x 将新的数据 x 存入队列。

DelQueue()的实现跟 EnQueue()相似，请读者参看源程序以及注释自己分析。

在 main()函数内，首先用 SetNull()清空队列，然后调用 EnQueue()插入一定数量的数据，再用 DelQueue()将其中的部分数据提出并显示，最后通过函数 Empty()判断队列中是否还有数据，如果有，则将剩余数据的个数打印出来。

实例74 堆栈的应用

实例说明

本例通过一个简单的计算器程序，向读者展现堆栈这种重要的数据存储结构及其一些常用的操作。

多数现代计算器都接受称为插入法的表达式，表达式的一般输入形式是：操作数-操作符-操作数。形成鲜明对照的是许多早期计算器程序使用后插入法的表达式，也就是先输入两个操作数，再键入操作符。本例就是采用这种后插入法实现计算器功能。首先输入两个操作数，将它们分别压入堆栈；然后根据输入操作符，将压入堆栈的数据弹出，并对其进行相应的操作，最终将操作结果重新压入堆栈。

本例重点讲述堆栈的压入、弹出操作，它是由两个函数 push()和 pop()实现的。希望通过对这两个函数学习，掌握堆栈的实现和工作方式。

知识要点

堆栈（stack）和队列正好相反，采用后进先出的策略。堆栈可以形象地比作自助餐厅里的一摞盘子，最先放到桌子上的盘子最后取走。系统软件中大量使用堆栈，其中包括编译和解释程序。

对堆栈的基本操作是存和取，习惯上分别称作 push（压入）和 pop（弹出），因此用向栈上放元素的 push()函数和从栈上取元素的 pop()函数实现堆栈，同时还需要保存栈内容的内存区。与队列相同，如果不保存取操作得到的结果，那么将丢失这个栈元素。

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#define Max 100      //由其确定开辟内存区的大小

int *p;              //用于指向开辟的内存空间
int *tos;            //指向堆栈的顶端
int *bos;            //指向堆栈的底端

//添加一个数据放到堆栈顶端
```

```
void push(int i)
{
    if(p > bos)
    {
        printf("堆栈已满\n");
        return;
    }
    *p = i;                    //将数据存到开辟的内存区
    p++;                       //指针指向下一个单元
}

//从堆栈顶端取出一个数据
int pop(void)
{
    p--;                       //首先使指针指向上一个单元
    if(p < tos)
    {
        printf("堆栈下溢\n");
        return 0;
    }
    return *p;                 //返回值为取出的数据值
}

void main(void)
{
    int a,b;
    char s[80];                //用于存储操作符
    p = (int *)malloc(Max*sizeof(int));     //开辟内存空间
    if(!p)
    {
        printf("分配内存失败");
        exit(1);
    }
    tos = p;                           //栈顶指针赋值
    bos = p + Max -1;                  //栈底指针赋值
    printf("请输入第一个数据:\n");
    scanf("%d",&a);
    push(a);                           //将输入的值a压入堆栈
    printf("请输入第二个数据:\n");
    scanf("%d",&b);
    push(b);                           //将输入的值b压入堆栈
    printf("请输入操作符:\n");
    scanf("%s",s);
    switch (*s)
```

```
    {
    case '+':
        a = pop();          //将存于堆栈的数据弹出，并将该值赋给 a
        b = pop();          //将存于堆栈的数据弹出，并将该值赋给 b
        printf("结果是 a+b = %d\n",(a+b));
        push(a+b);          //将计算结构压入堆栈
        break;
    case '-':
        a = pop();
        b = pop();
        printf("结果是 a-b = %d\n",(a-b));
        push(a-b);
        break;
    case '*':
        a = pop();
        b = pop();
        printf("结果是 a*b = %d\n",(a*b));
        push(a*b);
        break;
    case '/':
        a = pop();
        b = pop();
        printf("结果是 a/b = %d\n",(a/b));
        push(a/b);
        break;
    default:
        printf("请输入正确操作符\n");
    }
}
```

程序分析

程序首先定义三个全局的整型指针，其中指针 p 开始时用于指向为保存堆栈内容而新开辟的内存空间，这是在主函数中通过语句 p = (int *)malloc(Max*sizeof(int))实现的。指针 tos 和 bos 分别指向堆栈的顶端和底端，同样也是在主函数中通过语句 tos = p 和 bos = p+Max-1 给其赋初值。在堆栈的应用过程中指针 p 指向堆栈最顶端数据的上一个单元，也就是*(p-1)为堆栈的最顶端的数据。

对堆栈的操作主要是通过两个函数 push()和 pop()完成的。Push()函数用于向堆栈中压入一个数据，当要调用 push()函数将一个新的数据压入堆栈时，首先判断此时指向堆栈当前数据上一单元的指针 p 是否比栈底指针 bos 针还大，如果 p>bos，则说明堆栈已经满了，不能再向该堆栈 push 数据了，此时程序作出相应提示：printf("堆栈已满\n")。如果堆栈还有空间，则通过语句*p = i 将要插入堆栈的数据 i 赋值给指针 p 指向的存储空间，并让当前指针 p 通过语句 p++

自加，保证指针 p 重新指向堆栈最顶端数据的上一个单元。Pop()函数实现的思路跟 push()基本相同，pop()函数用于将堆栈最顶端的数据弹出，所以首先要使指针 p 指向堆栈最顶端的数据，也就是让 p 自减 p--，然后判断此时指针 p 是否比栈顶指针还要小，如果 p < tos 说明此时堆栈已经为空，也就是在上次 pop()操作中已经把堆栈中的最后一个数据弹出了，所以此时作出相应的提示：printf("堆栈下溢\n")。如果堆栈不为空，则通过语句 return *p 将栈顶的数据作为函数 pop()的返回值返回。

在此过程中必须注意一点，对指针操作，例如 p++，并不是指 p 的值自加 1，而是指经过该操作后，p 为指向下一个单元的指针，其具体指针变化多少跟 C 的编译环境以及 p 指针指向一个什么类型的数据有关。

在主函数 main()中首先输入两个整型变量，并调用 push()函数将其压入堆栈，然后再通过 scanf()函数出入运算符，由程序通过 switch()函数判断要进行什么操作，然后再调用两次 pop()将数据从堆栈中提出，并做相应操作，然后将操作结果重新压入堆栈。其具体实现请参看上面的实例和对应的注释。

实例75 串的应用

实例说明

随着计算机技术的发展，计算机被越来越多地用于解决非数值处理问题，这些问题所涉及的主要操作对象是字符串，简称串。

本实例向读者演示的文本编辑程序就是一个典型的串应用的例子。在文本编辑软件中把用户输入的所有文本内容作为一个字符串。虽然文本编辑软件的功能有强弱差别，但是其基本功能都包含了串的输入、删除、和输出等。本例中分别使用函数 EnterData()，DeleteLine()和 List()完成串的输入、删除和输出。

知识要点

串（String）是由零个或多个字符组成的有限序列，一般记作 s="s_0 s_1 …s_{n-1}"其中 s 称作串的名，字符个数 n 称作串的长度，双引号括起来的字符序列是串的值。

对串的存储可以有两种处理方式：一种是将串定义成字符型数组，串的存储空间分配在编译时完成，不能更改，这种方式称为串的静态存储结构；另一种是串的存储空间在程序运行时动态分配，这种方式称为串的动态存储结构。

由于串是一种特殊的线性表，所以串的静态存储也采用顺序存储结构。串的动态存储结构有两种方式：一种是链式存储结构，另一种是称之为堆结构的存储方式。下面将重点介绍串的链式存储结构。

串的链式存储结构是包含数据域和指针域的节点结构。其中数据域用来存放字符，指针域存放指向下一个节点的指针，这样一个串就可以用一个单链表来表示。

用单链表存放串，若每个节点仅存放一个字符，则每个节点的指针域所占存储空间比数据域所占存储空间空间大数倍。为节省空间，可使每个节点存放若干个字符，这称之为块链结构。显然块链结构的存储密度高于一个节点存一个字符的链表结构。通常，串的链式存储结构多采用块链结构。如本实例中采用块链结构的文本编辑程序，一个节点存放 80 个字符。块链结构节点类型和头节点类型定义如下：

```
typedef struct node
{   char Data[80];
    struct node *Next;
}nodetype;                          //节点类型定义
typedef struct
{   int Length;
```

```
    nodetype *Next;
}headtype                    //头节点类型定义
```

程序源码

设计一个采用块链结构的简化文本编辑程序。程序简化为只可对一行进行操作，要求具有行输入、整行删除、文本显示和退出四项功能，程序功能由功能菜单选择。

该应用程序的源代码如下：

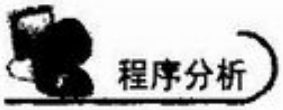
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100              //最大的行数

typedef struct node
{
    char Data[80];
    struct node *Next;
}nodetype;                   //节点类型

typedef struct head
{
    int Num;                 //行号
    int Len;                 //改行字符的个数
    nodetype *Next;
}headtype;                   //头节点类型

headtype Head[MAX];

void Initial();              //初始化各行头节点
int MenuSelect();            //菜单选择函数
void EnterData();            //输入数据函数
void DeleteLine();           //整行删除函数
void List();                 //显示各行数据函数
void ClearBuffer();          //清空缓存函数，与 scanf 配合使用

main()
{
    char choice;             //存放用户输入的选择参数
    Initial();               //初始化各行头节点
    while(1)
    {
        choice = MenuSelect();
        switch (choice)
        {
```

```
            case 1:EnterData();     //输入数据
                break;
            case 2:DeleteLine();    //整行删除
                break;
            case 3:List();          //显示各行数据
                break;
            case 4:exit(0);         //退出
        }
    }
}

void ClearBuffer()
{
    while(getchar()!='\n');
}

void Initial()
{
    int i;
    for(i=0;i<MAX;i++)
    {
        Head[i].Len=0;          //各行头节点长度参数置为零
    }
}

int MenuSelect()
{
    int i;
    i=0;
    printf(" 1. Enter\n");
    printf(" 2. Delete\n");
    printf(" 3. List\n");
    printf(" 4. Exit\n");
    while(i<=0||i>4)
    {
        printf("请输入菜单选择号\n");
        scanf("%d",&i);
        ClearBuffer();
    }
    return(i);
}

void EnterData()
{
```

```
    nodetype *p,*find();
    int i,j,m,LineNumber,k;
    char StrBuffer[100];
    while(1)
    {
        printf("输入数据要插入的行号(0~100):\n");
            scanf("%d",&LineNumber);
        ClearBuffer();
//输入参数不在0~100之间，跳出整个循环
        if(LineNumber<0||LineNumber>=MAX)
            return;
        printf("请输入要插入的数据，以@作为结束符号\n");
        i=LineNumber;
        Head[i].Num=LineNumber;                    //头节点赋值
        Head[i].Next=(nodetype *)malloc(sizeof(ncdetype));
        p=Head[i].Next;
        m = 1;                                     //当前节点数
        j = -1;
        StrBuffer[0] = 0;
        K = 0;
        do
        {
            j++;
            if(!StrBuffer[k])
            {
                scanf("%s",StrBuffer);  //输入字符串，最后数据肯定为\0
                k=0;
            }
            if(j>=80*m)       //如果满足此条件，说明数据要存储到下一个节点
            {
                m++;
                p->Next=(nodetype *)malloc(sizeof(nodetype));
                p=p->Next;
            }
            p->Data[j%80] = StrBuffer[k++];      //给节点数据域赋值
        }while(p->Data[j%80]!='@');              //当输入符号@时，跳出循环
        Head[i].Len = j;
    }
}

void DeleteLine()
{
    nodetype *p,*q;
    int i,j,m,LineNumber;
```

```
    while(1)
    {
        printf("输入要删除的行号(0～100)：\n");
        scanf("%d",&LineNumber);
        if(LineNumber<0||LineNumber>=MAX)
            return;
        i = LineNumber;
        p=Head[i].Next;
        m=0;                                    //当前的节点序号数
        if(Head[i].Len>0)
        {
            m=(Head[i].Len-1)/80+1;             //查找该行用到几个链表节点
        }
        for(j=0;j<m;j++)
        {
            q=p->Next;
            free(p);
            p=q;
        }
        Head[i].Len=0;                          //头指针赋参数赋值0，表明本行为空
        Head[i].Num=0;
    }
}

void List()
{
    nodetype *p;
    int i,j,m,n;
    for(i=0;i<MAX;i++)
    {
        if(Head[i].Len>0)                       //如果本行不为空
        {
            printf("第%d行有数据，它们是：\n",Head[i].Num);
            n=Head[i].Len;
            m=1;
            p=Head[i].Next;
            for(j=0;j<n;j++)                    //遍历本行，查找到数据在哪个节点
                if(j>=80*m)
                {
                    p=p->Next;
                    m++;
                }
                else
                printf("%c",p->Data[j%80]);     //依次打印出来
```

```
                printf("\n");
            }
        }
        printf("\n");
    }
```



该程序主函数首先调用函数 MenuSelect()，由用户输入一个整型数据（1～4），程序根据该数据判断下一步要调用的函数。

在输入函数 EnterData()中，首先要求用户输入要将数据插入到的行数，这个数值要在 0～100 之间，否则将调用 return 语句跳出该 while 循环。然后程序提示用户输入要插入的字符串。字符串必须以符号‘@’结束。由程序可得，当输入字符串后并不能跳出 while 循环，所以如果要结束插入数据操作，只要在程序提示用户输入将数据插入的行数时，输入的数值不在 1～100 之间就可以了。

同样如果调用删除行函数 DeleteLine()时，程序首先提示用户输入要删除的行数，即要删除第几行，然后通过语句 m=(Head[i].Len-1)/80+1 判断该行占用了几个链表节点。再通过 for 循环语句

```
for(j=0;j<m;j++)
{
    q=p->Next;
    free(p); p=q;
}
```

将各个节点 free 掉，最后给头节点的 Num 和 Len 赋值为零，表明改行为空。同样，如果要跳出删除行函数 DeleteLine()，只需在程序提示输入要删除的行数时，输入一个在 0～100 以外的数即可。

显示各行数据的函数 List()则是通过语句 Head[i].Len>0 判断各行是否有数据。如果有，则将该行的行数打印出来，然后通过一个 for 循环遍历本行，查找数据在哪个节点，再通过语句 printf("%c",p->Data[j%80])将字符数据依次打印出来。

实例76 树的基本操作

实例说明

前几个示例讨论了线性链表、堆栈、队列等线性数据结构，而许多问题是不能或难以用线性结构表示的，基于这一点，有必要研究一下非线性的数据结构。树（tree）就是一个应用十分广泛和重要的非线性数据结构。树中每一个数据元素至多有一个直接前趋，但是可以有多个直接后继，或者说，树是一种以分支关系定义的层次结构。

本例主要研究数据元素只有两个后继的二叉树（binary tree），在树的应用中，它起着特别重要的作用。处理树的许多问题用二叉树形式解决就变得非常简单，而任何的树跟二叉树之间可以通过简单规则的操作互相转换。本例通过一个相对比较简单的例子，构建一个二叉树，并按某种规则遍历树中的各个元素，最后将其打印出来。通过本例的学习，相信读者对二叉树实现方法会有一个比较清楚的了解，为以后的应用打下基础。

知识要点

树是n（n≥0）个节点的有限集合，当n=0时称为空树。在任一非空树中有且仅有一个称为该树之根的节点，如图1中的d便为根。除根节点以外，其余节点可以分为m（m≥0）个互不相交的集合，如图1中的b a c和f e g，而其中一个集合本身又是一颗树，称为根的子树。

树是一种极其重要的结构，它在许多领域都有着非常广泛的应用，例如它可以应用于分类、检索、网络、数据库、人工智能等方面。而一种特殊的树：二叉树（binary tree），也称作二分树或二元树，在树的应用中起着非常重要的作用，因为处理树的许多问题用二叉树形式解决就显得非常简单。二叉树也是节点的有限集合，它或者是空二叉树，或者是由一个根节点和两棵互不相交的被称为该根的左子树和右子树组成，且该根节点的左右子树也都是二叉树。图1便是一个典型的二叉树。

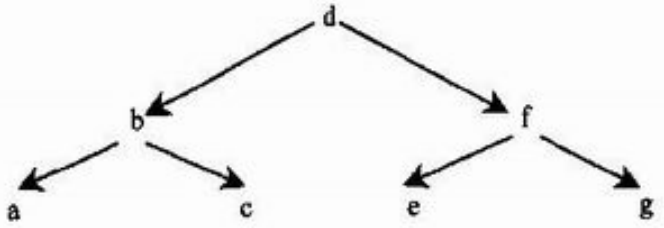
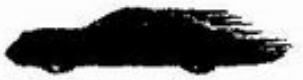


图1

二叉树可以采用顺序分配的存储方式，也可以采用链表的存储方式，但由于顺序分配存储方式可能会造成内存的极大浪费，因此较少使用。下面将简要介绍采用链表分配的存储方式。



当用链表表示二叉树时，每个节点需要几个域可以根据需要来设定，常设定三个域，即数据域、左子树域和右子树域（需要时还可以设立标志域）。本实例中就是采用这种链表的存储方式建立的树，建立一个如下的结构体：

```
struct tree
{   char info;
    struct tree *left;
    struct tree *right; };
```

其中字符型数据 info 是数据域，而 left 和 right 分别为指向该节点子树的指针。上述结构便是一个典型的双向链表。

访问树中各个节点的过程称为树的遍历。所谓的访问节点，其含义是很广的，可以理解为对节点的增、删、修改等各种运算的抽象。所以二叉树的遍历是最重要和最基本的操作，二叉树的许多操作都是以遍历为基础的。

遍历二叉树有三种方法：中序、前序和后序。以中序遍历时，先访问左子树，然后访问树根，最后访问右子树。以前序遍历时，先访问树根，然后访问左子树，最后访问右子树。以后序遍历时，先访问左子树，然后访问右子树，最后访问树根。使用三种遍历方法访问图 1 的顺序是：

中序：　　a b c d e f g

前序：　　d b a c f e g

后序：　　a c b e g f d

在后面程序分析中将结合下面实例详细介绍如何用程序实现不同的遍历。

程序源码

该应用程序的源代码如下：

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

struct tree
{
    char info;
    struct tree *left;
    struct tree *right;
};

struct tree *root;      //树的第一个节点
struct tree *construct(struct tree *root, struct tree *r, char info);
void print(struct tree *r, int l);

int main(void)
```

```
{
    char s[10];
    root = NULL;
    do
    {
        printf("请输入一个字符:");
        gets(s);
        root = construct(root,root,*s);
    }while(*s);
    print(root,0);
    return 0;
}

struct tree *construct(
    struct tree root,              //新建树的根
    struct tree *r,                //新插入的节点
    char info)                     //新插入节点的数据域
{
    if(!r)
    {
        r = (struct tree *)malloc(sizeof(struct tree));
        if(!r)
        {
            printf("内存分配失败! ");
            exit(0);
        }
//将该节点左右子树指针赋值为NULL，并使其数据域为参数info
        r->left = NULL;
        r->right = NULL;
        r->info = info;
        if(!root)
//如果root为NULL，则此插入值为该树的根
            return r;
        if(info < root->info)
        //如果当前插入的节点比此子树的根小
            root->left = r;
        //则将这个节点作为此子树的右子树
        else
            root->right = r;
        return root;
    }
//如果r为非空节点，则判断要插入的数据跟r数据域中的数据的大小
    if(info < r->info)
//如果插入的数据比r节点的数据域数据还小
```

```
        construct(r,r->left,info);
    //则以 r 节点为新插入节点的根，将新数据作为左子树节点
    递归调用 construct () 函数
        else
    //否则以 r 节点为新插入节点的根，将新数据作为右子树节点
    递归调用 construct () 函数
            construct(r,r->right,info);
        return root;
    }

    void print(struct tree *r, int l)
    {
        int i;
        if(!r)
            return;
        print(r->left,l+1);
        for(i = 0;i < l;++i)
            printf(" ");
        printf("%c\n",r->info);
        print(r->right,l+1);
    }
```

上例通过函数 construct()来构造一颗已排序的二叉树。其构造的树中任意根的左子树所包含的节点都小于或等于根，任意根的右子树所包含的节点都大于根。该函数是采用递归调用方式实现构造一棵已排序的二叉树。该函数首先判断其第二个参数 struct tree *r 是否为 NULL，如果是，则开辟一个新的节点的存储空间，并让其左右子树指针为 NULL，并将该新开辟的节点的数据域赋值为 construct()函数的第三个参数 info。如果此时 root 也为 NULL，说明此时为一个空树，则返回此时插入的节点的指针作为此树的根指针。如果 root 不是 NULL，则将插入的值 info 跟其子树的跟节点的值比较以决定插入节点为此子树跟节点的左子树还是右子树。如果参数 struct tree *r 不是 NULL，则通过判断当前的 info 值和 r 节点的 info 值的大小决定将新加节点作为 r 的左子树还是右子树递归调用函数 construct()，最终将 info 加到合适的位置递归调用完结，返回树根地址。

对树打印的过程就是对其遍历的过程，本例采用中序方式遍历。由以下函数完成：

```
void print(struct tree *root)
{
if(!root)return;
    scan(root->left);
    if(root->info)
        //执行相应的操作;
```

```
    scan(root ->right);
}
```

首先，如果当前树的根为空，即 if(!root)成立，则通过 return 语句返回。如果当前树的根不为空，则再以当前树的左子树为当前树调用函数，即 scan(root->left)。这样递归调用下去直到达到树的左终端节点，也就是直到 root->left = NULL ，此时调用 scan(root->left)时，满足 if(!root->left)，所以 return 语句使其返回本次递归调用，执行下一个语句 if(root->info)。也就是如果数据域有效，则执行相应的操作。然后再以 root->right 为参数递归调用 scan()函数。通过上面遍历树的函数可以看出，函数首先访问左子树：scan(root->left)，直到左终端节点，然后再访问该子树（也就是上面提到的左终端节点）的树根，最后再访问右子树，这是一个典型的中序遍历程序。

结合图 1 分析一下这种遍历实现的过程。首先 root 为节点 d 的指针，易得该节点指针不为 NULL，所以递归调用 scan(root->left)，也就是将 d 节点的左子树，即节点 b 作为树根，调用函数 scan()，同样该节点也不为空，再一次递归调用 scan(root->left)，此时将 a 节点作为树根，调用函数 scan()，该节点也不为空，但由于它为终端节点，也就是该节点的子树节点指针都为 NULL，所以当再次递归调用函数 scan(root->left)时，return 语句使本次递归结束，转而执行语句 if(root->info);也就是完成对 a 节点的访问，由于 a 节点的右子树也为 NULL，所以下面对 scan(root->right)的调用也没有任何操作。当完成对 a 节点的操作后，函数跳出上一次递归，执行函数 if(root->info);注意，此时的当前节点为 b，上面语句完成了对 b 的操作，然后执行语句 scan(root->right)完成对 c 的操作，依次类推。

如果读者有兴趣，可以仿照上面自己编写前序遍历和后序遍历函数。下面也将我们编写的这两种遍历函数给出：

```
void scan(struct tree root)      //前遍历程序
{
    if(!root)return;
    if(root->info);
    //执行相应的操作;
    scan(root->left);
    scan(root ->right);
}

void scan(struct tree root)      //后遍历程序
{
    if(!root)return;
    scan(root->left);
    scan(root ->right);
if(root->info);
    //执行相应的操作;
}
```

主函数 main()首先声明一个 char 型的字符串 s[10]，然后通过一个 do—while 循环语句利

用 gets()输入不同的字符型数据，再通过语句 root = construct(root,root,*s)完成对树的构建。当有空输入时结束 do－wihle 循环。转而执行打印函数 print()，按照中序遍历树中各个节点并将其数据域数据打印出来。

细心的读者可能发现，利用前面所提出的建树原则（任意左子树所包含的节点都小于等于根，任意根的右子树所包含的节点都大于等于根）建立的树，再通过中序方式遍历打印各个节点，就完成了一次简单的排序功能。例如本实例中，当分别输入 d a c b 后回车，则从上到下分别输出 a b c d，完成了一次排序工作。

实例77 冒泡排序法

实例说明

本例将向读者演示一种最著名的排序方法：冒泡排序法。

从基本原理讲，冒泡排序法属于一种交换排序的类型，它从数组的一端开始，依次对相邻两元素进行比较，当发现它们不合顺序时就进行一次交换。这样各个元素就像是水箱里面的气泡，每个气泡都在自身的平衡点。

本例排序主要是通过一个函数 bubble（）实现的，在主函数中通过 gets（）函数输入一定的字符串后调用该函数排序，然后再将其打印输出。

知识要点

所谓的排序就是指按增加或减少的顺序对一组类似的信息重新进行安排的过程。一般来说，当对信息进行排序时，只有信息中的某一小部分被用作排序的关键字，根据关键字对各元素进行比较。当必须要进行交换时，彼此交换的则是整个元素。

本例中冒泡排序法实现的思路如下：假定是对一个字符串按照升序方式排序。首先先将字符串中最后一个字符与倒数第二个字符进行比较。如果倒数第二个字符大于最后一个字符，则将它们两个交换一下位置，这样看起来就像是气泡向上浮了一层。如果倒数第二个字符比最后一个字符小，则保持两个字符顺便不变，然后再将倒数第二个字符跟倒数第三个字符相比较，依次比较交换下去，一轮完成以后，处于最顶端的必然是整个字符串中最小的字符。然后再重复上面的工作，进行第二轮比较，这一轮下来，必然将倒数第二小的字符排在了第二位，这样循环比较交换下去，最后实现使字符串按照要求的升序排列。

程序源码

该应用程序的源代码如下：

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

bubble(strings,count)          //冒泡排序函数
char *strings;                 //要排序的字符串
int count;                     //字符串的长度
{
    register int m,n;              //定义位寄存器型变量，加快访问速度
```

```
    register char s;
    for(m = 1;m<count;m++)
        for(n = count-1;n >= m;--n)
        {
            if(strings[n-1]>strings[n])
            {//借助 s 将 strings[n-1]和 strings[n]交换
                s = strings[n-1];
                strings[n-1] = strings[n];
                strings[n] = s;
            }
        }
}

int main(void)
{
    int count;
    char str[200];
    printf("请输入字符串: \n");
    gets(str);                    //输入要排序的字符串
    count = strlen(str);          //得到字符串的长度
    bubble(str,count);            //对字符串进行排序
    printf("排序之后的字符串是: \n");
    printf("%s.\n",str);          //打印排序好的字符串
    return 0;
}
```

程序的执行结果为:

```
请输入字符串:
fgertygfd                  //随机输入的字符
排序之后的字符串是:
deffggrty.
```

程序分析

本例是通过函数 bubble(strings,count)完成对字符串进行冒泡法排序的操作，其中 strings 是指向将要被排序的字符数组的指针，count 是这个数组中的元素数目。程序由两重循环组成的。因为数组元素的个数是 count，所以外层循环执行(count-1)次，即对数组搜索(count-1)次。因为外层循环至少使一个元素排到它应该在的位置上，所以，即使在最坏的情况下，循环(count-1)次以后会把数组按顺序排好。内层循环执行实际的比较和交换工作。数组中两个元素交换必须借助一个外部的字符型变量 s 来完成，其实现的语句为:

```
s = strings[n-1];
        strings[n-1] = strings[n];
        strings[n] = s;
```

在主函数 main()中，首先先用 gets()函数输入要排序的字符串，由 strlen()函数得到输入字符串的长度，然后调用排序函数 bubble(str,count)排序后将结果打印出来。

为了解释冒泡排序法的工作过程，下面给出一字符数组 dcab 在每遍外层循环的执行结果：

初始： d c a b;

第一遍： a d c b;

第二遍： a b d c;

第三遍： a b c d;

从这个过程可以看出，第一遍外层循环后，数组中最小的一个字符 a 上升到最高的位置。第二遍循环后，次小的字符 b 上升到次高的位置。重复下去，循环(count-1)遍后，所有元素就全部就序了。大家可以看到，这个过程很有点类似于水箱中水泡上升过程，冒泡排序的“冒泡”二字就是据此得来。

当分析一种排序的优劣时，必须确定在最好情况下、平均情况下、最坏情况下，这种算法所必须执行的比较资料和交换次数。对于冒泡法排序，不管排序数组的初始情况如何，比较次数总是一样的。这个次数是由两重循环的次数决定的。如果排序数组元素是 n 个，则外层循环是（n-1）次，内层循环是 n/2，总数是 n(n-1)/2，效率很低。

实例78 堆排序

实例说明

本例将介绍一种较为抽象的排序方法：堆排序。在堆排序中将要用到前面讲过的树的概念。本例首先将一些已经存在的整型数值用一个函数 adjust()调整成堆，然后利用堆排序的性质，将其顶端的极值取出，再将最后的节点放到顶端，进行堆排序，然后将顶端的极值取出，如此反复完成排序。

对于大量的记录的排序来说，堆排序是一种非常有效的方法。如果排序的记录数不大，则堆排序的优越性并不明显，而且还需要一个供交换时暂存记录的辅助空间。

知识要点

堆的定义为一组记录的关键字序列$\{k_1,k_2,\ldots,k_n\}$，当且仅当满足以下关系时，称之为堆。

$$\begin{cases} k_i \leqslant k_{2i} \\ k_i \leqslant k_{2i+1} \end{cases} \quad 或 \quad \begin{cases} k_i \geqslant k_{2i} \\ k_i \geqslant k_{2i+1} \end{cases} \qquad (i = 1,2,\ldots,[\frac{n}{2}])$$

若将和此序列对应的一维数组（即以一维数组做此序列的存储结构）看成是一个完全二叉树，则堆的含义表明，完全二叉树中所有非终端节点的值均不大于（或不小于）其左右子树节点的值。如下面这个堆序列：{60 47 58 25 18 10 36}可以用二叉树表示为：

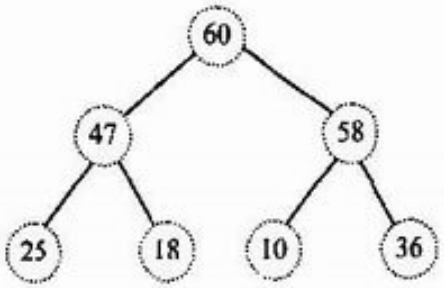


图 1

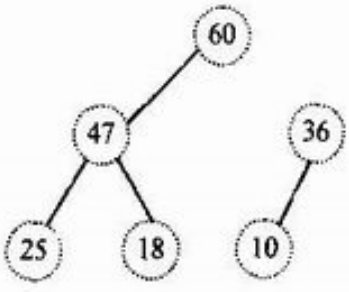


图 2

如图 1，如果在输出堆顶的最大值（60）后，使剩下的 n-1 个元素重新又生成一个堆，则得到 n 个元素的次大值（58），如图 2 所示。如此反复执行，则可得到一个有序序列，这个过程称为堆排序。

通过分析下面的例子来说明①如何由一个无序序列建成一个堆；②如何在输出堆顶元素后，调整剩余元素形成一个新堆。

程序源码

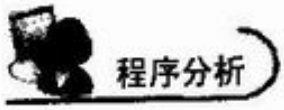
该应用程序的源代码如下：

```
#include <stdio.h>
#define MARK 0                          //占据a[0],不起实际作用

static a[11] = {MARK,25,4,36,1,60,10,58,14,47,18};
int count = 1;
void heap(int n);                       //建立堆的函数
void adjust(int i,int n);               //辅助函数

int main(void)
{
    int i;
    printf("源数据为：");
    for(i = 1;i<11;i++)
        printf("%5d",a[i]);
    heap(10);                                   //将数据建立为堆排序
    printf("\n排序后的数据为：");
    for(i = 1;i<11;i++)
        printf("%5d",a[i]);                     //将此堆排序打印出来
    printf("\n");
    return 0;
}

void heap(n)                                    //将一个长度为n的无序序列编程为堆序列
int n;                                  //序列长度
{
    int i,j,t;
    for(i =n/2;i>0;i--)
        adjust(i,n);                            //将的i个节点变为堆
    printf("\n初始化成堆===>   ");
    for(i = 1;i < 11;i++)
        printf("%5d",a[i]);
    for(i = n-1;i>0;i--)
    { //将堆序列中堆顶的值与未排序序列中的最后一个数对调
        t = a[i+1];
        a[i+1] = a[1];
        a[1] = t;
        adjust(1,i);                            //重新生成堆序列
        printf("\n第%2d步操作结果===>",count++);
        for(j = 1;j<11;j++)
            printf("%5d",a[j]);
```

```
    }
}

void adjust(i,n)
int i,n;
{
    int j,k,r;                  //k,r用来暂存a[i]的值
int done=0;
    k = r = a[i];
    j = 2*i;
    while((j<=n)&&(done==0))
    {
        if(j<n)                 //找出a[2i]和a[2i+1]中较大的值
        {
            if(a[j]<a[j+1])
                j++;
        }
        if(k>=a[j])             //如果满足a[i]大于等于a[2i]和a[2i+1]
            done = 1;           //设置此参数跳出while循环
        else
        {
            a[j/2] = a[j];
            j = 2*  j;
        }
    }
    a[j/2] = r;
}
```

程序运行后结果为：

```
源数据为:          25   4   36   1   60   10   58
初始化成堆===>      60   25   58   1   4   10   36
第 1 步操作结果===>  58   25   36   1   4   10   60
第 2 步操作结果===>  36   25   10   1   4   58   60
第 3 步操作结果===>  25   4   10   1   36   58   60
第 4 步操作结果===>  10   4   1   25   36   58   60
第 5 步操作结果===>   4   1   10   25   36   58   60
第 6 步操作结果===>   1   4   10   25   36   58   60
排序后的数据为:          1   4   10   25   36   58   60
```



在这里首先结合上面的例子回答知识要点中最后提出的第二个问题，如何在输出堆顶元素

后，调整剩余元素形成一个新堆。以图 1 为例，如果将堆顶元素 60 提出，而以堆中的最后一个元素 36 代替，则此时左右子树均为堆，所以仅需自上至下调整即可。首先以堆顶元素和其左右子树根节点比较，由于右子树节点值大于左子树根节点值，也大于根节点值 36，所以根节点值与右子树节点值交换，然后只考虑右子树情况，同样利用上述的办法，直到比较进行到终端节点（叶子）。在上面实例中，函数 heap(n)通过如下语句完成上述功能：

```
    for(i = n-1;i>0;i--)
  { //将堆序列中堆顶的值与未排序序列中的最后一个数对调
    t = a[i+1];
    a[i+1] = a[1];
    a[1] = t;
    adjust(1,i);                    //重新生成堆序列
  }
```

首先通过三个语句将堆序列顶的值与未排序序列中的最后一个值对调，然后通过 adjust()函数将剩余的序列重新生成堆序列，反复运行，直到 i 由 n—1 变到 1，完成全部排序。

把一个无序序列建堆的过程就是一个反复重复上面“筛选”的过程。如果将这个无序序列看成是一个完全二叉树，则最后一个非终端节点是第[n/2]个元素，由此筛选只要从第[n/2]个元素开始，直到筛选到堆顶。在 heap()函数中通过下面循环语句完成上述功能：

```
for(i =n/2;i>0;i--)adjust(i,n);
```

主函数将堆排序的每一步操作结果都清晰地打印出来，有助于读者理解这种排序的实现步骤。

实例79 归并排序

实例说明

本实例将介绍另外一种排序方法：归并排序。下面将要用到的是归并排序中最常用的方法：二路归并排序。二路归并的含义是把两个有序的序列合并成为一个有序的序列，其排序的基本思想是将有 n 个记录的原始序列看作 n 个有序的子序列，每个子序列的长度为 1，然后从第一个子序列开始，把相邻的子序列两两合并，得到[n/2]个长度为 2 或 1 的子序列（当子序列个数为奇数时，最后一组合并得到的序列长度为 1），把这一过程称为一次归并排序。对第一次归并排序后的[n/2]个子序列采用上述方法继续顺序成对归并，如此重复，当最后得到长度为 n 的一个子序列时，该子序列便是原始序列归并排序后的有序序列。本实例是通过函数 Msort（）完成归并排序，并将每一步排序结构清晰地打印出来，以便于读者观看并理解过程。

知识要点

这里将详细介绍二路归并排序的实现方法，以助于读者理解下面实例中的归并排序函数。在归并排序过程中，最基本的问题是如何把两个位置相邻的有序子序列合并为一个有序子序列。

设有两个有序子序列 L1 和 L2，它们顺序存放在 x[strat1],...x[end1]和 x[strat2],...x[end2]中，其中 strat2=end1+1。它们所对应的关键字序列为（$k_{start1},...k_{end1}$）和（$k_{start2},...k_{end1}$），若两序列归并结果放在数组 y 的同样的位置上，即 y[strat1},...y[end2]中，则归并方法可描述如下：

设置三个变量 i，j，m，其中 i 和 j 分别表示序列 L1 和 L2 中当前要比较的记录的位置号，初值为 i=starrtl,j=strat2,m 表示输入 y 中当前记录应放置的位置号，初值 m=1。

归并时反复比较 k_i 和 k_j 的值：

（1）若 $k_i \leqslant k_j$，则 x[i]→y[m],i=i+1,m=m+1；

（2）若 $k_i > k_j$，则 x[i]→y[m],j=j+1,m=m+1。

当序列 L1 和 L2 中的全部记录已经并到数组 y 中，即 i=end1+1 或者 j=end2+1 时，比较结束。然后将另一个序列中剩余的所有记录依次放到数组 y 中，这样就完成有序序列 L1 和 L2 的合并。

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>

void Mpass(int x[],int y[],int k,int n);        //声明函数
void Msort(int x[],int y[],int n);              //声明函数

int main(void)
{
    //要排序整型数据序列
    int a[] = {26,5,37,1,61,11,59,15,48,19};
    int y[10];                                  //用于暂时存储数据
    int i;
    printf("源数据为:       ");   //将源数据打印出来
    for(i=0;i<10;i++)
    printf("[%2d]",a[i]);
    Msort(a,y,10);                              //对源数据进行合并排序
    printf("\n排序后的数据为:  ");
    for(i = 0;i<10;i++)                         //将排序结果打印出来
    printf("%4d",a[i]);
    printf("\n");
    return 0;
}

void Mpass(x,y,k,n)
int x[];                    //要排序的数组
int y[];                    //用于存储临时数据的数组
int k;                  //表示当前序列中有若干长度为 k 的相邻有序子序
int n;                  //要排序序列的长度为 n

{
    int i,j;
    int strat1,end1;    //对应第一个有序子序列 L1 起始和终止位置号
    int strat2,end2;    //对应第二个有序子序列 L2 起始和终止位置号
    int m;              //表示输入 y 中当前记录应放置的位置号
    strat1 = 0;
    m = 0;
    while(strat1+k<=n-1)            //当第一个子序列没有占据整个 x 数组
    {
        strat2 = strat1+k;          //为两个有序子序列起始终止位置号赋值
        end1 = strat2-1;
        //如果第二个子序列长度不够 k，则其终止位置号为 n-1
        end2 = (strat2+k-1<=n-1)?strat2+k-1:n-1;
        for(i = strat1,j = strat2;i<=end1&&j<=end2;)
        {//若 ki≤kj，则 y[m]= x[i],i=i+1,m=m+1;
            if(x[i]<=x[j])
```

```
            {
                y[m] = x[i];
                i++;
                m++;
            }
            else
            //若 k_i > k_j，则 y[m]= x[i],j=j+1,m=m+1
            {
                y[m] = x[j];
                j++;
                m++
            }
        }
        while(i<= end1)
        {
            y[m] = x[i];
            m++;
            i++;
        }
        while(j<= end2)
        {
            y[m] = x[j];
            m++;
            j++;
        }
        strat1 = end2+1;
    }
    //将另一个序列中剩余的所有记录依次放到数组 y 中
    for(i=strat1;i<n;i++,m++)
        y[m] = x[i];
}

void Msort(x,y,n)
int x[];                        //要排序的数组
int y[];                        //用于存储临时数据的数组
int n;                          //数组长度
{
    int i,k,count;
    k = 1;
    count = 1;
    while(k<n)                  //当子序列比整个序列小时
    {
        Mpass(x,y,k,n);         //归并两有序子序列
        for(i= 0;i<n;i++)
```

```
            x[i] = y[i];          //返回数据
        printf("\n 第%2d 步后的结果==>  ",count++);
            for(i = 1;i<n+1;i++)
        {
            if((i ==n)&&((i%(2*k)!=0)))      //如果是最后一个元素
            //将该元素打印出来，并在其后面打印符号“]”
                printf("%4d]",x[i-1]);
            else
            {
                if((i%(2*k)==1))
    //如果是归并后生成的有序子序列的第一个元素
                    printf("[%2d",x[i-1]);
                //将其打印出同时在前打印符号“[”
                else if((i%(2*k))==0)
                //如果是归并后生成的有序子序列最后一个元素
                    printf("%4d)",x[i-1]);
                //将其打印出同时在前打印符号“]”
                else
                    printf("%4d",x[i-1]);
            }
        }
        k = 2*k;              //一次归并后新的有序子序列的长度
    }
}
```

函数执行结果为：

```
源数据为:          [26][ 5][37][ 1][61][11][59][15][48][19]
第 1 步后的结果==>  [ 5  26][ 1  37][11  61][15  59][19  48]
第 2 步后的结果==>  [ 1   5  26  37][11  15  59  61][19  48]
第 3 步后的结果==>  [ 1   5  11  15  26  37  59  61][19  48]
第 4 步后的结果==>  [ 1   5  11  15  19  26  37  48  59  61]
排序后的数据为:        1   5  11  15  19  26  37  48  59  61
```

程序分析

一次二路归并排序的目的是把若干个长度为 k 的相邻有序子序列，从前向后两两进行归并，得到若干个长度为 2k 的相邻有序子序列。这里有一个问题，即若记录的个数 n 为 2k 的整数倍时，序列的两两归并正好完成 n 的记录的一次归并；否则当归并到一定位置时，剩余的记录不足 2k 个，这时的处理方法是：

（1）剩余的记录个数大于 k 而小于 2k，把前 k 个记录作为一个子序列，把其他剩余的记录作为另一个子序列。根据假设可知，它们分别是有序的，采用前面知识要点中所介绍的方法对这两个子序列进行归并。

（2）剩余的记录的个数小于 k，根据假设可知，它们是有序排列的，可以直接把它们放到数组 y 中。

下面结合本例详细介绍如何用 c 语言实现一次二路归并排序。在程序中，要归并的各个有序子序列一次存放在 x[0],x[1],...,x[n-1]中，各个子序列的长度为 k，但最后一个序列的长度可以小于 k。

本例是通过函数 Mpass(x,y,k,n)完成一次二路归并排序的。其中 x 为要排序数组，其长度为 n，y 用于存储临时数据的数组，k 为当前序列中有若干长度为 k 的相邻有序子序列。程序中判断语句 while(strat1+k<=n-1)表示当有序子序列长度小于 n 即要排序数组 x 还没有排完。此时通过 for(i = strat1,j = strat2;i<=end1&&j<=end2;)循环语句完成如下的判断。

（1）若 $k_i \leqslant k_j$，则 x[i]→y[m],i=i+1,m=m+1；

（2）若 $k_i > k_j$，则 x[i]→y[m],j=j+1,m=m+1。

满足条件 while(i<=end1)时，说明数组 x[i]还有数据没有排入新的有序子序列 y[],而由于 x[i]本身为有序序列，所以通过赋值语句 y[m] = x[i]将剩余的 x[i]的序列赋值给 y。x[j]有剩余序列时情况相同。最后通过 for(i=strat1;i<n;i++,m++)循环语句将 x[]序列中剩余的所有记录依次放到数组 y 中。

在函数 void Msort(x,y,n)中，通过一个 while(k<n)循环反复调用一次二路归并排序函数 Mpass(x,y,k,n)，直到有序子序列长度等于 n，也就是整个要排序序列归并成一个有序子序列，完成排序。在每个 while(k<n)后，即每次完成一次二路归并后，通过语句 k = 2*k 通知程序归并后有序子序列长度为归并前的两倍。

为了便于观察归并排序的操作过程，Msort(x,y,n)函数每完成一次二路归并后，还将排序结果打印出来，根据当前有序子序列的长度，用符号“[”和“]”将各个有序子序列表示出来。如上面的程序结果所示：在第一次二路归并后，将原来无序序列两两按照从小到大排序。二次二路归并后，又将原来长度为 2 的有序序列两两归并成长度为 4 的有序序列，直到完成整个数组的排序。

对序列进行归并排序时，除采用二路归并排序外，还可以采用多路归并排序方法，这里不做进一步的叙述，有兴趣的读者可以参阅相关的书籍自己编程实现。

实例80 磁盘文件排序

实例说明

本实例中将向读者演示一下对随机访问的磁盘文件的排序。首先建立一个结构体 struct data，并初始化一个结构体数组 Sdata[NUM]，利用文件操作函数将该结构体数组写入一个文件，然后利用快速排序法，按照结构体中某个成员变量（name）对结构体数组进行排序，最后将结果打印出来。

本例不仅向读者介绍了一种新的排序方法—快速排序法，而且向读者演示了如何应用此种排序方法对字符串进行排序。另外由于字符串是在文件中提取的，也是对读者以前学过的输入输出流、文件操作的一个全面的复习。

知识要点

首先介绍一下快速排序法。快速排序（quick sort）是由 C.A.R.Hoare 发明并命名的，这种排序被认为是目前最好的一种排序算法。快速排序基于交换排序，与同样的基于交换排序的冒泡排序法相比，其效果是令人吃惊的。

快速排序的基本思想是分区。一般过程是先选一个称为比较数的值，把数组分成两段。大于或等于分区值的元素都放到一边，小于分区值的元素放到另一边，然后对数组的另一段重复上述过程，直到该数组完成排序。例如，假定有一数组 fedacb，用 d 值做比较数，第一遍快速排序后的数组如下所示：

初始：　f　e　d　a　c　b

第一遍：　b　c　a　d　e　f

对 b c a 和 d e f 这两段重复进行这一过程。显然，以上过程本质上是递归的，而实际上，快速排序的最清晰实现就是递归算法。

选择比较数值办法有两个，即可以随机选取也可以选择对数组中的一小组值求平均值得到。为了达到最优效果，所选的值尽可能使之达到值范围的中心值。然而，对多数数据集合而言，很难做到这一点。最坏的情况是被选值都在一端。即使在最后的情况下，快速排序的性能仍然不错。

本例快速排序是通过函数 quick_disk(FILE *fp,int count)中反复调用排序函数 qs_disk(FILE *fp,int left,int right)实现快速排序。在 qs_disk()中，通过函数 get_name(fp,(long)(i+j)/2)返回中间值作为比较数进行快速排序。

本实例是对结构体 struct data 中的成员变量 name 进行排序。name 是一个长度为 20 的字

符数组，对其排序也就是对字符串排序，因此需要用到字符串操作函数 strcmp()来比较两字符串的大小，用 strcpy()来完成字符串的复制。

由于本实例是将存于文件中的数据进行排序，所以需要用到对文件的基本操作，下面把在本实例中用到的对文件操作的一些内容和需要用到的函数做简要的介绍。

硬盘文件分为两类：随机访问文件和顺序访问文件。当两类文件都足够小时，可以将其读入内存，并用前述各种数组排序实用程序进行排序。然而，多数磁盘文件都较大，不能在内存中方便的排序，需要特殊的处理技术。多数数据应用程序都使用随机文件，因此本例讲述了对随机磁盘文件排序的一种办法。随机磁盘文件对于顺序磁盘文件有两个优点：第一，容易维护，更新时不需要复制整个列表；第二，可以作为盘上的大数组来处理，极大地简化了排序操作。

把随机文件当作大数组处理时，可以使用稍加修改的基本快速排序算法。快速排序的磁盘形式必须使用 fseek()函数寻找在盘上的记录，由此代替对数组的下标操作。其形式为：

int fseek(FILE *stream,long offset,int origin);函数 fseek()按照 offset 和 origin 的值设置与流 stream 相关联的文件位置指示器，用于支持随机访问 IO 操作。一般用 fseek()函数指定位置，接着用 fread（void *buf,size_t size,size_t count,FILE *stream）或者 fwrite（const void *buf,size_t size,size_t count,FILE *stream）执行相关的操作，完成对文件的随机访问。关于文件操作函数的具体用法，请参见 msdn。

程序源码

该应用程序的源代码如下：

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

#define NUM 4

struct data
{
    char name[20];
    char school[20];
    char city[20];
    char province[20];
}info;

struct data addrs[NUM]=
{
    "WenHai","BIT","JiLin","JiLin",
    "TongWei","BIT","ZhengJiang","JiangSu",
    "SunYou","BIT","WeiFang","ShangDong",
```

```
    "XiaoMing","PKU","TaiYuan","ShanXi"

};
//对后面要用到的函数进行声明
void quick_disk(FILE *fp,int count);
void qs_disk(FILE *fp,int left,int right);
void exchangedata(FILE *fp,long i, long j);
char *get_name(FILE *fp, long rec);
void print_data(struct data *p);
struct data *get_data(FILE *fp,long rec);

int main(void)
{
    int i;
    FILE *fp;                          //文件指针
    //以读写方式生成文本文件 fp
    if((fp = fopen("datalist.txt","w+")) == NULL)
    {
        printf("打开文件失败\n");
        exit(1);
    }
    printf("将未排序的数据写入文件\n");
    //将数组 Sdata[NUM]写入文件中
    fwrite(addrs,sizeof(addrs),1,fp);
    //将文件中的数组 Sdata[NUM]依次取出并打印
    for(i=0;i<NUM;i++)
    {
        struct data *p;
        p = get_data(fp,i);            //得到 Sdata[i]的指针
        print_data(p);                 //将结构体 Sdata[i]各个成员变量打印出
        printf("\n");
    }

    fclose(fp);                        //关闭文件指针
    //以二进制方式再次打开文件 datalist.txt
    if((fp=fopen("datalist.txt","rb+"))==NULL)
    {
        printf("不能以读写方式打开文件\n");
        exit(1);
    }

    printf("将文件数据排序\n");
    //调用字符串排序函数将文本中的字符串结构体排序
    quick_disk(fp,NUM);
```

```
    printf("排序结束\n");
    //将排序结束后的数组数据打印出来
    for(i=0;i<4;i++)
    {
        struct data *p;
        p = get_data(fp,i);
        print_data(p);
        printf("\n");
    }
    fclose(fp);

    return 0;
}
//应用快速排序方法对字符串进行排序
void quick_disk(FILE *fp,int count)
{
    qs_disk(fp,0,count-1);
}

//排序函数
void qs_disk(FILE *fp,int left,int right)
{
    long int i,j;
    char x[30];
    i = left;                          //要排序数组的最左端
    j = right;                         //要排序数组的最右端

    //比较字符串 x 为 Sdata 数组中间一个结构变量的 name 成员变量
    strcpy(x,get_name(fp,(long)(i+j)/2));
    do
    {   //比较当前结构变量的 name 成员变量于比较字符串 x 的大小
        while((strcmp(get_name(fp,i),x)<0)&&(i<right))
            i++;
        while((strcmp(get_name(fp,j),x)>0)&&(j>left))
            j--;
        if(i<=j)
        {
            exchangedata(fp,i,j);          //交换 i 和 j 对应的数据
            i++;
            j--;
        }
     }while(i<j);
```

```
    if(left<j)                          //反复调用此排序函数，直到 j 达到数据的最左端
        qs_disk(fp,left,(int)j);
    if(i<right)                         //反复调用此排序函数，直到 i 达到数据的最右端
        qs_disk(fp,(int)i,right);
}

//交换 i 和 j 在文件中对应的数据
void exchangedata(FILE *fp,long i,long j)
{
    char a[sizeof(info)],b[sizeof(info)];
    fseek(fp,sizeof(info)*i,SEEK_SET); //找到 i 在文件中对应的数据位置
    fread(a,sizeof(info),1,fp);        //将该位置数据读到字符串数组 a 中

    fseek(fp,sizeof(info)*j,SEEK_SET);     //找到 j 在文件中对应的数据位置
    fread(b,sizeof(info),1,fp);        //将该位置数据读到字符串数组 b 中
    fseek(fp,sizeof(info)*j,SEEK_SET);     //找到 j 在文件中对应的数据位置
    fwrite(a,sizeof(info),1,fp);       //将刚才读入 a 中的数据写入到该位置
    fseek(fp,sizeof(info)*i,SEEK_SET);     //找到 i 在文件中对应的数据位置
    fwrite(b,sizeof(info),1,fp);       //将刚才读入 b 中的数据写入到该位置
    //完成文件中 i 和 j 处对应数据的交换
}

//得到文件 fp 中变量 rec 对应的结构体变量的 name 成员变量
char *get_name(FILE *fp,long rec)
{
    struct data *p;
    p = &info;
    rewind(fp);
    //找到该结构体变量得的位置
    fseek(fp,rec*sizeof(struct data),SEEK_SET);
    //将其读到指针 p
    fread(p,sizeof(struct data),1L,fp);
    return p->name;                    //返回该结构体变量的 name 成员变量
}

//得到文件 fp 中变量 rec 对应的结构体变量的指针
struct data *get_data(FILE *fp,long rec)
{
    struct data *p;
    p = &info;
    rewind(fp);
    //找到该结构体变量的位置
    fseek(fp,rec*sizeof(info),SEEK_SET);
    //将其读到指针 p
```

```
    fread(p,sizeof(info),1,fp);
    return p;                          //返回该结构体指针
}
//将指针 p 对应的结构体的各个成员变量打印出来
void print_data(struct data *p)
{
    printf("姓名: %s\n",p->name);
    printf("学校: %s\n",p->school);
    printf("城市: %s\n",p->city);
    printf("省  : %s\n",p->province);
}
```

程序分析

该实例的基本存储单元是一个结构体：

```
struct data
{   char name[20];
    char school[20];
    char city[20];
    char province[20];
}info;
```

用于存储一些姓名、学校以及所在城市等信息。然后建立一个数组 struct data addrs[NUM] 并对其进行初始化。本实例主要是通过对这个数组的操作向读者演示如何进行文件排序以及排序后的效果。

在主函数 main()中，通过语句 fp = fopen("datalist.txt","w+")建立一个可以读写的文本文件 datalist.txt。本例是演示如何对磁盘文件数据进行排序，所以我们通过语句 fwrite(addrs,sizeof(addrs),1,fp)将上面建立的数组 addrs 写入到新建的文件 datalist.txt 中。具体如何将数据写入文件，前面的章节中已经介绍，这里不再赘述。为便于读者观察到排序前后的变化，将数组写入磁盘文件后，紧接着用循环语句：

```
for(i=0;i<NUM;i++)
{   struct data *p;
    p = get_data(fp,i);
    print_data(p);
    printf("\n");   }
```

将刚输入文件的数据打印出来。对数据的打印主要是通过两个函数完成：get_dat()和 print_data()。其中函数 get_dat（fp,i）返回在文件指针 fp 对应文件中第 i 个 struct data 类型的数据。其实现是通过语句 fseek(fp,rec*sizeof(info),SEEK_SET)，找到 rec 对应的结构体在指针 fp 指向的文件位置，然后通过语句 fread(p,sizeof(info),1,fp)将该结构体的指针赋值给一个指向 struct data 数据的指针 p。接着将这个指针返回，在函数 print_data(p)中调用该指针，并将该指针对应的结构体中各个数据打印出来。打印函数 print_data()比较简单，请读者参照程序说明自

已分析的工作过程。

将数据写入文件后，主函数将文件以二进制方式打开文件 datalist.txt，并调用函数 quick_disk(fp,NUM)完成对文件数据的排序，然后再利用上面所述的 for 循环语句将排序后的文件中的数据打印出来。下面将重点讲述一些排序函数 quick_disk()的实现方法。

结合知识要点中所述的快速排序的编程思想，在函数 qs_disk(FILE *fp,int left,int right)中，首先通过函数 strcpy(x,get_name(fp,(long)(i+j)/2))将参数 i 和 j 对应的中间数据的中间值作为快速排序选定的比较字符串，通过两个 while 循环语句

```
while((strcmp(get_name(fp,i),x)<0)&&(i<right)) i++;
while((strcmp(get_name(fp,j),x)>0)&&(j>left)) j--;
    if(i<=j)
        {   exchangedata(fp,i,j);        //交换 i 和 j 对应的数据
            i++;
            j--;}
```

找到左端比参考数值大的字符串和右端比参考数值小的字符串，然后通过函数 exchangedata(fp,i,j)调换它们两个的位置以保证在 x 左端的字符串都小于 x，在 x 右端的字符串都大于 x，完成一次快速排序。然后再反复递归调用函数 qs_disk(fp,left,(int)j)或者 qs_disk(fp,(int)i,right)对数组的剩余部分完成快速排序，从而实现对文件数据的排序过程。

交换数据函数 void exchangedata(FILE *fp,long i, long j)的实现方法也很简单，主要是利用函数 fseek()找到 i 和 j 对应的要交换的数据，并通过函数 fread()将其保存到字符型数组 a 和 b 中。然后再通过函数 fseek()重新定位 i 和 j 对应数据的位置，并将 b 中的数据通过函数 fwrite()写入 i 所定位的数据，把 a 中的数据写入到 j 所定位的数据，完成数据交换。

通过运行该程序可以清楚地看到排序前后的变化，另外，读者也可以到源程序目录下找到新建的文件 datalist.txt，将其打开观看其内部排序后的内容。

实例81 顺序查找

实例说明

本例演示的是一种最简单的查找方式：顺序查找。本例首先利用 creat()函数建立一个链表，将数组 Datas[NUM]存储的结构信息输入链表中，然后再调用函数 SequelSeach(head,name)将与用户输入的字符串 name 相匹配的信息调出，最后通过函数 print_data(point)将详细信息打印出来。

知识要点

查找又称为检索，就是从一个数据元素集合中找出某个特定的数据元素。查找是对数据进行处理时大量使用的一种操作，查找算法的优劣对整个软件系统的效率影响很大。

查找的过程是将给定值与记录集合中各个记录的关键字相比较，从而确定给定值在记录集合中是否存在，以及存在时它在记录集合中的位置。如果能够在记录集合中找到与给定值相等的关键字，则该关键字所属的记录就是所要查找的记录，此时称该查找是成功的；如果查遍整个记录集合也没找到与给定值相等的关键字，则称查找是失败的。

在查找过程中，要进行的操作分为两种类型：一种是只检查某个特定的记录是否存在于给定的记录集合，称这种查找为静态查找；另一种查找不但要检查记录中是否存在某个特定的记录，而且当该记录不存在时，要把它插入到记录集合中，或者当记录集合中存在该记录时，要将其内容进行修改或把它从记录集合中删去，这种查找称为动态查找。

衡量查找算法效率的标准是平均查找长度，也就是为确定某一记录在记录集合中的位置，给定值关键字与集合中的记录关键字所需要进行的比较次数的期望值。对于具有 n 个记录的记录集合，查找某记录成功的平均长度为：$ASL=\sum_{i=1}^{n}P_iC_i$。其中 P_i 为查找到第 i 个记录的概率；C_i 为查找第 i 个记录所进行的比较次数。

本例中主要讲述的顺序查找是一种最简单的查找方法，假设数据集合中有 n 个记录 R1,R2...Rn，其关键字分别为 K1,K2,...,Kn，它们顺序存放在某顺序表中。顺序查找的方法是，从顺序表中的一端开始，用给定值的关键字 Key 逐个顺序地与表中各记录的关键字相比较，若在表中找到某个记录的关键字与 Key 值相等，表明查找成功；若找遍了表中所有的记录也未找到与 Key 相等的关键字，表明查找失败。当得到查找成功或失败的结论时，查找结束。

程序源码

该应用程序的源代码如下：

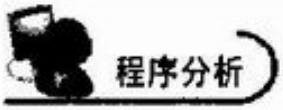
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NUM 4
//定义存储数据的基本单元——链表
struct chain
{
    char name[20];
    char city[20];
    char sex[10];
    char age[10];
    char job[10];
    struct chain *next;
};

struct chain *create();                          //生成链表函数
struct chain *SequelSeach(head,name);            //顺序查找函数
void print_data(point);
//初始数据
struct chain Datas[NUM]=
{
    "Sun","Weifang","Male","24","student",NULL,
    "Tom","Beijing","Male","31","doctor",NULL,
    "Marry","Shanghai","Female","19","techer",NULL,
    "Willing","Tianjing","Female","21","worker",NULL
};

int main(void)
{
    struct chain *head;                  //定义表头
    struct chain *p;
    char name[30];
    head = create();
    printf("请输入要查找的人名\n");
    scanf("%s",name);
    p = SequelSeach(head,name);
    print_data(p);
    return 0;
}
```

```
struct chain *create()
{
    struct chain *head, *tail, *p;
    int i;
    head = tail = NULL;
    printf("将名单数据输入到链表中:\n");
    for(i= 0;i < NUM; i++)
    {
        p = (struct chain *)malloc (sizeof (struct chain));
        strcpy(p->name, Datas[i].name);
        strcpy(p->city,Datas[i].city);
        strcpy(p->sex,Datas[i].sex);
        strcpy(p->age,Datas[i].age);
        strcpy(p->job,Datas[i].job);
        p->next = NULL;
        if(head == NULL)
            head = tail = p;
        else
            tail = tail ->next;
            tail ->next = p;
    }
    return head;
}

struct chain *SequelSeach(head,name)
struct chain *head;
char name[];
{
    struct chain *temp;
    temp = head;
    for(temp=head;temp!=NULL; )
    {
        if(strcmp(temp->name,name) == 0)
        break;
        else
            temp = temp->next;
    }
    if(temp == NULL)
        printf("没有查找到该人资料\n");
    return temp;
}

void print_data(point)
struct chain *point;
```

```
{
    if(point ==NULL)
        return;
    printf("查找结果: \n");
    printf("    姓名: %s\n",point->name);
    printf("    城市: %s\n",point->city);
    printf("    性别: %s\n",point->sex);
    printf("    年龄: %s\n",point->age);
    printf("    工作: %s\n",point->job);
}
```

执行结果如下所示:

```
将名单数据输入到链表中:
请输入要查找的人名
Tom
查找结果:
    姓名: Tom
    城市: Beijing
    性别: Male
    年龄: 31
    工作: doctor
```



本实例是将一些基本的信息存储到一个链表 chain 中，并通过 create（）函数利用事先已经存在的数据 Datas[NUM]建立一个链表。其建链表的过程及原理与前面介绍的链表的建立原理相同，这里就不再赘述。下面重点看看顺序查找函数 SequelSeach（）的实现方法。

在 main()函数中,把新建的链表的头指针 head 和要查找的人的姓名传给函数 SequelSeach（）。函数内部通过如下 for 循环逐个遍历链表:

```
for(temp=head;temp!=NULL;)
{
    if(strcmp(temp->name,name) == 0)
    break;
    else
    temp = temp->next;
}
```

如果当前指针 temp 不是最后一个即 temp!=NULL，则通过判断 strcmp(temp->name,name)是否为零来确定当前链表的 name 数据成员与要查找的 name 是不是相同，如果相同，则通过 break 跳出 for 循环，如果不同，则通过语句 temp = temp->next 查找下一个链表的 name 数据成员，直到找到要查找的单元或者链表结束。

for 循环结束后，如果 temp 为 NULL，说明链表查到最后没有发现与要查找的关键字匹配的单元,则通过打印语句 printf("没有查找到该人资料\n")做相应的提示。如果 temp 不为 NULL，

其必然为指向要查找单元的指针，则将该指针做为查找函数 SequelSeach（）的返回值。

最后主函数 main（）通过函数 print_data（）将函数 SequelSeach（）找到的单元的各个元素打印出来。

实例 82 二分法查找

实例说明

本实例将演示一种效率较高的查找方式：二分法查找。二分查找法要求被查找的数据是按照某种顺序排列的有序序列，因此在本例中，首先建立了一个结构体数组 SDatas[NUM]，并给其赋初值，然后按结构体数组中的一个成员变量 name 的大小对该结构体数组通过函数 qs_struct（）利用快速排序法进行排序，使其成为有序序列，最后通过函数 BinarySeach（）利用二分查找法根据用户输入的要查找的字符串，将其找到并利用函数 print_data（）把该结构体的详细信息打印出来。如果没有找到与输入关键字 name 匹配的信息，则程序作出相应的提示。

知识要点

二分法查找又称为折半查找，是一种效率高的有序顺序表上的查找方法，下面讨论一下二分查找的实现方法。

设顺序表存储在一维数组 s 中，各记录的关键字满足下列条件：

s[0].Key≤s[1].Key≤…≤s[n-1].Key

设置三个变量 low，high 和 mid，它们分别指向当前查找范围的下界、上界和中间位置。初始化时，令 low=0，high=n-1，设置待查找数据元素的关键字为 Key。

（1）$mid=[\frac{low+high}{2}]$。

（2）比较 Key 与 s[mid].Key 值的大小，若 s[mid].Key=Key，则查找成功，结束查找；若 s[mid].Key<Key，表明关键字为 Key 的记录可能存在于记录 s[mid]的后半部分，修改查找范围，令下界指示变量 low=mid+1，上界指示变量 high 的值保持不变。若 s[mid].Key>Key，表明关键字为 Key 的记录只可能存在于 s[mid]的前半部，修改查找范围，令 high=mid-1，变量 low 保持不变。

（3）当前变量 low 与 high 的值，若 low≤high，重复执行第（1）步和第（2）步，若 low>high，表明整个表已经查找完毕，表中不存在关键字为 Key 的记录。

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NUM 4
//定义存放基本单元的数据结构
struct Data
{
    char name[20];
    char city[20];
    char sex[10];
    char age[10];
    char job[10];
};

struct Data SDatas[NUM]=
{
    "Sun","Weifang","Male","24","student",
    "Tom","Beijing","Male","31","doctor",
    "Marry","Shanghai","Female","19","techer",
    "Willing","Tianjing","Female","21","worker"
};
//声明函数
void qs_struct(items,left,right);          //快速排序法函数
void quick_struct(items,count);
int BinarySeach(items,name,n);             //二分法查找函数
void print_data(point);                    //打印结果

int main(void)
{
    char name[30];
    int i;
    printf("将原始数据排序\n");
    quick_struct(SDatas,NUM);                    //快速排序法排序
    printf("请输入要查找人的名字: \n");
    scanf("%s",name);
    i = BinarySeach(SDatas,name,NUM);            //二分法查找
    if(i == -1)
    {
        printf("没有查找到该人信息\n");
        return 0;
    }
    printf("查找结果: \n");
    print_data(&SDatas[i]);
    return 1;
```

```
}

void quick_struct(items,count)
struct Data items[];                    //要排序的结构体数组头指针
int count;                              //元素的个数
{
    qs_struct(items,0,count-1);
}

void qs_struct(items,left,right)
struct Data items[];                    //要排序的结构体数组头指针
int left;                               //结构体数组的最左端
int right;                              //结构体数组的最右端
{
    register int i,j;                   //定义寄存器型变量，增快访问速度
    char *x;
    struct Data temp;                   //临时结构体，用于交换数据
    i = left;
    j = right;
    x = items[(left+right/2)].name;
    do
    {
        while((strcmp(items[i].name,x)<0)&&(i<right))
            i++;
        while((strcmp(items[j].name,x)>0)&&(j>left))
            j--;
        if(i<=j)
        {   //将 i 和 j 对应的结构体对象交换位置
            temp = items[i];
            items[i] = items[j];
            items[j] = temp;
            i++;
            j--;
        }
    }while(i<=j);
    if(left<j)
        qs_struct(items,left,j);
    if(i<right)
        qs_struct(items,i,right);
}

int BinarySeach(items,name,n)           //二分法查找
struct Data items[];                    //要查找的结构体数组的头指针
char name[];                            //要查找的关键字
```

```
int n;                                    //结构体数组的长度
{
    int low,high,mid;
    low = 0;
    high = n-1;
    while(low<=high)
    {
        mid = (low+high)/2;               //取中间数
        if((strcmp(items[mid].name,name)==0))
            return mid;
        else if((strcmp(items[mid].name,name)<0))
            low = mid+1;
        else high = mid-1;
    }
    return -1;
}

void print_data(point)
struct Data *point;
{
    if(point ==NULL)
        return;
    printf("    姓名: %s\n",point->name);
    printf("    城市: %s\n",point->city);
    printf("    性别: %s\n",point->sex);
    printf("    年龄: %s\n",point->age);
    printf("    工作: %s\n",point->job);
}
```

程序分析

该程序首先建立一个结构体数组 SDatas[NUM]并对其初始化，然后用函数quick_struct(items,count)对该结构体数组按照其中某成员变量name的顺序将其快速排序，使其成为有序序列。其实现方法在前面排序中已经介绍过了，这里不再赘述。下面重点讲述一下二分查找如何实现。二分查找函数BinarySeach()主要通过以下语句完成查找：

```
while(low<=high)
{
    mid = (low+high)/2;
    if((strcmp(items[mid].name,name)==0))
        return mid;
    else if((strcmp(items[mid].name,name)<0))
        low = mid+1;
    else high = mid-1;
```

}

首先给 mid 赋值为(low+high)/2，然后通过函数 strcmp（）判断当前结构对象的 name 成员变量与要查找的 name 的关系，如果相等，则将当前的 mid 值返回；如果当前的 name 小于要查找的关键字，则给 low 重新赋值为 mid+1，high 不变，重新二分法查找；反之，将 high 赋值为 mid-1，重新二分查找，直到查找到要查找的关键字或者查找完整个数组，即 low≥high。

实例83 树的动态查找

实例说明

前面的例子介绍的几种查找方法主要适用于顺序表结构，并且限定于对表中的记录只进行查找，而不作插入或删除操作，即只作静态查找。本例将向读者演示一种用树数据结构存储记录集合时的动态查找方法。首先程序通过 construct()函数，利用已经存在的结构体数组数据建立一个二叉树，建立树的过程中，要保证每个节点的值都大于它的左子树上节点的值而小于它右子树上所有节点的值，该函数返回建立树的根指针；然后通过函数 Search(root,name)查找，如果找到相应的数据，将其打印出来，如果没有找到，则用户可以选择是否将该数据插入到树中。

本例除了向读者演示了如何对二叉树进行查找以外，还对以前学过的树这种数据结构以及如何建树、如何遍历二叉树进行一个全面的复习。

知识要点

如果在一个二叉树中，每个节点的值都大于它的左子树上所有节点的值，而小于它的右子树上所有节点的值，这样的二叉树称为二叉排序树。对于一个记录集合，可以用二叉排序树来表示，树中的一个节点对应于集合中的一个记录，整棵树表示该记录集合。我们建立二叉树时，使每个二排序树中每个节点所记录的数据的关键字（本例为 name 成员变量）都大于它的左子树上所有节点存储的记录的关键字，而小于它的右子树上所有节点存储的记录的关键字。本实例中树的基本数据结构为下面的结构体：

```
struct tree
{char name[20];
    char city[20];
    char sex[10];
    char age[10];
    char job[10];
    struct tree *left;
    struct tree *right;}
```

对此二叉树的建立、排序、遍历和查找都是以其 name 成员变量为关键字进行的。具体的查找方法如下：

（1）当二叉树为空时，查找失败。

（2）如果二叉排序树根节点的关键字（name）等于要查找的关键字，则查找成功，返回

指向根节点的指针。

（3）如果二叉排序树根节点记录的关键字小于要查找的关键字，则用同样方法继续在根节点的左子树上查找。

（4）如果二叉排序树根节点记录的关键字大于要查找的关键字，则用同样方法继续在根节点的右子树上查找。

显然这是一个递归查找的过程。仔细分析会发现，其原理与前面讲述的二分查找基本相同。本例是用函数 Search(root,name)完成对以 root 为根节点的树的查找。

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NUM 4

struct tree
{
    char name[20];
    char city[20];
    char sex[10];
    char age[10];
    char job[10];
    struct tree *left;
    struct tree *right;
};

struct tree Datas[NUM]=
{
    "Willing","Tianjing","Female","21","worker",NULL,NULL,
    "Tom","Beijing","Male","31","doctor",NULL,NULL,
    "Sun","Weifang","Male","24","student",NULL,NULL,
    "Marry","Shanghai","Female","19","techer",NULL,NULL
};
//建立一个树
struct tree *construct(
    struct tree *root,
    struct tree *r,
    struct tree *Data)
{
    if(!r)
    {
```

```
        r = (struct tree *)malloc(sizeof(struct tree));
        if(!r)
        {
            printf("内存分配失败! ");
            exit(0);
        }
        r->left = NULL;
        r->right = NULL;
        strcpy(r->name,Data->name);
        strcpy(r->city,Data->city);
        strcpy(r->sex,Data->sex);
        strcpy(r->age,Data->age);
        strcpy(r->job,Data->job);
        if(!root)
            return r;
        if(strcmp(Data->name,root->name)<0)
            root->left = r;
        else
            root->right = r;
        return r;
    }
    if(strcmp(Data->name,r->name)<0)
        construct(r,r->left,Data);
    else
        construct(r,r->right,Data);

    return root;
}
//查找关键字为 name 的节点
struct tree *Search(root,name)
struct tree *root;
char name[];
{
    struct tree *p;
    if(root == NULL)
        printf("该树为空\n");
    p = root;
    while(strcmp(p->name,name)!=0)
    {
        if(strcmp(p->name,name)>0)
            p = p->left;
        else
            p = p->right;
        if(p == NULL)
```

```
            break;
        }
        return(p);
}
//中序遍历树的各个节点，并把各个节点的关键字 name 打印出来
void print(struct tree *r)
{
    if(!r)
        return;
    print(r->left);
    printf("%s\n",r->name);
    print(r->right);
}
//打印指针 point 指向的节点
void print_currentData(struct tree *point)
{
    if(point == NULL)
        return;
    printf("    姓名：%s\n",point->name);
    printf("    城市：%s\n",point->city);
    printf("    性别：%s\n",point->sex);
    printf("    年龄：%s\n",point->age);
    printf("    工作：%s\n",point->job);
}

int main(void)
{
    int i;  char c[10];
    char swap[20];                  //用于存储临时数据以初始化 temp
    char name[20];                  //用于存储要查找的关键字
    struct tree *root,*p;
    struct tree *temp;
    p = NULL;                       //初始化指针
    temp = NULL;
    root = NULL;
    for(i = 0;i<NUM;i++)
        root =construct(root,root,&Datas[i]);
    printf("现有人员资料：\n");
    print(root);                    //将各个节点的 name 关键字打印出来
    printf("请输入要查找的人的名字\n");
    scanf("%s",name);
    p = Search(root,name);
    if(p == NULL)
    {
```

```
        printf("没有该人资料\n");
        printf("是否要插入该人资料[y/n]\n");
        scanf("%s",c);
        if(strcmp(c,"y")==0)
        {
            temp = (struct tree *)malloc(sizeof(struct tree));
            if(!temp)
            {
                printf("内存分配失败! ");
                exit(0);
            }
            //利用用户输入的信息初始化结构体对象 temp
            printf("请输入该人姓名: \n");
            scanf("%s",swap);
            strcpy(temp->name,swap);
            printf("请输入该人所在城市: \n");
            scanf("%s",swap);
            strcpy(temp->city,swap);
            printf("请输入该人性别[Male/Female]: \n");
            scanf("%s",swap);
            strcpy(temp->sex,swap);
            printf("请输入该人年龄: \n");
            scanf("%s",swap);
            strcpy(temp->age,swap);
            printf("请输入该人工作: \n");
            scanf("%s",swap);
            strcpy(temp->job,swap);
            temp->left = NULL;
            temp->right = NULL;
            root =construct(root,root,temp);
            print_currentData(temp);          //将刚刚输入的信息打印出来
            printf("现有人员资料: \n");
            print(root);
        }
        else
            return 0;
    }
    print_currentData(p);
    return 1;
}
```

程序分析

该程序首先利用函数 construct()将已经存在的结构体数组 Datas[NUM]建成一个二叉排序

树，函数返回值为该树的根指针。该函数实现的基本原理跟前面讲的树的建立完全相同，读者可以参照前面树的基本操作的例子来理解函数 construct()，这里不再赘述。

下面重点看一下查找函数 Search(root,name)是如何实现的。主函数 main()调用 Search()函数时传递给其两个参数：一个为当前树的根指针，另一个为要查找的关键字 name。所有函数 Search()首先通过语句 if(root == NULL)判断当前树是否为空树，如果为空，则作出相应提示，函数返回；如果不为空，根据知识要点中介绍的步骤，令当前查找的节点指针为根指针，判断当前节点中存储的数据 name 与要查找的 name 的大小，如果二者恰好相等，则其便为要查找的数据，将根指针返回。如果前者大于后者，即 if(strcmp(p->name,name)>0)，由于该树为二叉排序树，也就是该树的左子树节点中的关键字比树根的关键字小，所以通过语句 p = p->left 使树根的左子树树根作为当前节点，判断该节点中的关键字与要查找关键字的关系。反之，则通过语句 p = p->right 使树根的右子树树根作为当前节点，判断该关键字与要查找关键字的关系。如此反复查找下去，直到查找到关键字为 name 的节点并将指向该节点指针返回，或者查找到叶子而未查找到相应数据，则函数返回 NULL 指针。

Main()函数首先判断查找函数 Search()的返回值是否为 NULL，如果是，则说明没有查找到关键字为 name 的数据，那么程序根据用户输入决定立即结束还是将该人信息插入到树中。如果用户希望将这个人的信息插入到树中，则首先用输入函数将该人的详细信息初始化一个 struct tree 型的临时变量 temp，然后调用 construct()函数将该 temp 变量插入到树中。若查找函数 Search()返回值不是 NULL，则表明查找到关键字为 name 的节点，那么就利用函数 print_currentData()将该节点的详细信息打印出来。

实例 84 二分法求解方程

实例说明

本例向大家演示用二分法求解方程 f(x)=sin(x)在（-3，7）这个范围内的解 C 语言实现方法。求解主要通过函数 BisectRoot()来完成。该函数首先根据二分法的需要扫描根的存在及根的大致位置，然后再用二分法使根进一步精确。

知识要点

二分法是最简单的求解一元非线性方程根的数值算法之一，是用二分区间的方法来逼近准确根的一种方法。二分法可以用来计算：f(x)=0　x∈[a,b]的所有实根。

确定是否有根，根的精确化共分两个步骤进行。二分法的使用是在第二个步骤，也就是在第一步确定有根的前提下进一步向准确根逼近的近似方法。下面讨论一下这两步具体的实现思路：

第一步：确定是否有根及根的大致位置。我们使用“逐步扫描法”确定是否有根及根的“初始近似值”。在方程的数值求解过程中首先确定方程是否有根，在探求是否有根的过程中，就意味着寻求根的大致位置，也就是寻找根的初始值范围。虽然二分法用于第二步，但是第一步却是不可少的，整个求根过程是两步的有机结合。

方程 f(x)=0 的根分布可能很复杂，假设法 f(x)在某个（a，b）区域内有且仅有一个实单根 x'。从该区间的左端点 x0=a 出发，按照预先设定的步长 h 一步一步向右前进，每前进一步就进行一次根的扫描，即检查每一步起点 x0 和终点 x0+h 的函数值是否同号。如果异号：

$$f(x0)\times f(x0+h)\leq 0 \qquad \exists\, x' \in (x0,x0-h)$$

说明该起点和终点间有根。因此可以取 x0 或者 x0+h 作为根的初始近似值。

第二步：用二分法使“初始近似值”精确化。设函数 f(x)在区间[a,b]上单调连续，且

$$f(a)\times f(b)<0$$

则方程 f(x)=0 在区间(a,b)内有且仅有一个实根 x'。在有限区间(a,b)内，二分法的基本思想就是将含根区间逐步二分，使含根区间逐步变小，即含初始近似根的范围随之逐步变小，检查函数值符号的变化，确定含根的小区间。

下面给出二分法求根公式及误差公式：

（1）二分法求根公式：

$$X_k=(a_k+b_k)(1/2)$$

（2）误差公式：

$|x'-X_k| \leqslant (1/2)(b_k-a_k) = b_{k+1}-a_{k+1} < \varepsilon$

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <stdlib.h>
// 声明函数
double Func(double);
int BisectRoot(double,double,double,double,double *,int,int *);

void main()
{
    int i,n,m;
    double a,b,h,eps,*x;
    n = 3;                          // 方程根的个数的预估值
    x = (double*)calloc(n,sizeof(double));      // 开辟内存空间
    if(x == NULL)
    {
        printf("内存分配失败\n");
        exit(1);
    }
    a = -3;                                     // 区间起始端点
    b = 7;                                      // 区间终止端点
    h = 0.1;                                    // 扫描步长
    eps = 1.e-8;                                // 要求达到的精度
    BisectRoot(a,b,h,eps,x,n,&m);               // 调用二分法函数
    printf("y=sin(x)在范围%2.0f和%2.0f之间的根有%d个根\n",a,b,m);
    printf("它们分别是：\n");
    for(i = 0;i<n;i++)
    printf("x[%d] = %e\n",i,x[i]);
    free(x);                        // 释放内存空间
}

double Func(double x)
{
    return(sin(x));
}

int BisectRoot(a,b,h,eps,x,n,m)
double a;           // 实型变量，输入参数，求根区间的起始端点
double b;           // 实型变量，输入参数，求根区间的终止端点
```

```
double h;           //利用逐步扫描法确定根位置时的步长
double eps;         //实型变量，输入参数，控制精度的参数
dcuble *x;          //实型一维数组，输出参数，存放计算得到的数组
int n;              // 输入参数，区间内方程根的个数的预估值
int *m;             // 输出参数，实际求得的根的个数
{
    double z,z0,z1,y,y0,y1;
    *m = 0;             // 当前根的个数为 0
    z = a;              // 从有根区间左端开始扫描
    y = Func(z);
    while(1)        // 无限循环，直到遇到 return 或者 break 语句
    {//如果逐步扫描到求根区间的右端点或者得到的根的个数达到预估根的个数
        if((z>b+h/2)||(*m==n))
            return(1);
        if(fabs(y)<eps)     // 如果当前根 z 对应的函数 f(z)满足精度要求
        {
            *m+=1;
            x[*m-1] = z;    // 将此时的 z 值赋值给 x 数组
            z+=h/2;
            y = Func(z);
            continue;       // 结束本次循环，即跳过循环体中下面尚未执行
                            // 的语句接着进行下一次是否执行循环的判定
        }

        z1 = z+h;           // 逐步扫描中小区间的右端点
        y1 = Func(z1);      // 小区间右端点对应的函数值
        if(fabs(y1)<eps)    // 如果右端点恰好满足根的精度要求
        {
            *m+=1;
            x[*m-1] = z1;
            z = z1+h/2;
            y = Func(z);
            continue;
        }
        if(y*y1>0)      // 如果对应根乘积大于零，说明该区间内没有根
        {
            y = y1;
            z = z1;
            continue;
        }
        while(1)    // 如果本 while 循环执行，说明逐步扫描小区间 z 和 z1 间有根
        {
            if(fabs(z1-z)<eps)      // 如果满足精度要求
            {
```

```
                *m+=1;
                x[*m-1]=(z1+z)/2;
                z = z1+h/2;
                y = Func(z);
                break;
            }
            z0 = (z1+z)/2;              // 二分法求根公式
            y0 = Func(z0);
            if(fabs(y0)<eps)
            {
                *m = *m+1;
                x[*m-1] = z0;
                z =z0+h/2;
                y = Func(z);
                break;
            }
            if(y*y0<0)                  // 如果乘积小于零，说明根在 z 和 z0 之间
            {
                z1 = z0;
                y1 = y0;
            }
            else                        // 否则根在 z0 和 z1 之间
            {
                z = z0;
                y = y0;
            }
        }
    }
}
```

程序的执行结果为：

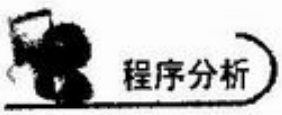
y=sin(x)在范围-3 和 7 之间的根有 3 个根

它们分别是：

x[0] = 1.526557e-015

x[1] = 3.141593e+000

x[2] = 6.283185e+000



本例二分法查找主要是通过函数 BisectRoot()实现的，下面具体介绍一下其实现的思路。如何用 C 语言实现，请读者参照例程以及其中的注释自己理解。

在进行二分法之前，根是否存在，确定其存在的位置，由逐步扫描法来判定。

设 f(x)是[a,b]上的连续函数。从左端点 x0=a 开始以步长 h 逐步向右扫描：

（1）若 $f(x_k)=0$，则 x_k 为一实根，且从 $x_k+h/2$ 开始再向右扫描；

（2）若 $f(x_k)\times f(x_{k+1})=0$，则 x_{k+1} 为一实根，且从 $x_{k+1}+h/2$ 开始再向右扫描；

（3）若 $f(x_k)\times f(x_{k+1})>0$，则说明当前小区间内无根，从 x_{k+1} 开始再向右扫描；

（4）若 $f(x_k)\times f(x_{k+1})<0$，则说明当前小区间内有一实根。此时反复将区间减半，直至区间长度小于给定的精确要求 ε 为止，区间的中点即为方程的一个实根。然后再从 x_{k+1} 开始向右扫描。

以上过程一直进行到区间的右端点 b 为止。在上述四种情况中，二分法仅用于第四种情况。在具体函数中，二分法由第二个 while 循环实现。

下面拿出一个单根区间分析一下二分法的实现：

假定方程 f(x)=0 在区间(a,b)内有且仅有一个根 x'。只需把区间一分为二，取其中点 x0=(1/2)(a+b)，再进行根的扫描，即查中点的函数 f(x0)与 f(a)是否同号，如果同号说明所要求的根 x'在 x0 的右侧，则令 a1=x0,b1=b；否则，x'必在 x0 的左侧，取 a1=a,b1=x0。无论 x'在 x0 的左侧还是右侧，都会得到新的有根区间(a1,b1)，其长度为原区间的一半。

再次把有根区间(a1,b1)一分为二，即中点 x1=(1/2)(a1+b1)将区间分为两半，通过根的扫描判定所求的根 x'在 x1 的哪一侧，依次确定出一个新的有根区间(a2,b2)，其长度为(a1,b1)的一半。如此反复得出一串有根区间：

(a,b),(a1,b1),(a2,b2),...(ak,bk),...

其中每个区间长度都是前一个区间长度的一半，区间（ak,bk）的长度为：

$bk-ak=(1/2^k)(b-a)$

持续这个过程，这些区间最终必然收敛于一点 x'，该点就是所求的根。

本程序中精度条件是 eps = 1.e-8，即当有根区间的长度小于这个值时，我们就认为该区间中点的值便是要求的根。

实例85 牛顿迭代法求解方程

实例说明

本例向大家演示用牛顿迭代法求解方程 $f(x)=x^3-x^2-1$ 的一个实根的 C 语言实现方法。通过求解该方程的根，向大家介绍另一种求解一元非线性方程的方法：牛顿迭代法。

当实际问题所建立的模型为数学物理方程时，其数值算法采用迭代法是一种简易的好方法。迭代法的基本思想就是构造一串收敛到解的序列，即建立一种从已有近似解来计算新的近似解的迭代式，然后选取方程的某个初始近似值 x_0 代入迭代式，反复这个过程使得到根逐渐逼近于真实根，直到得到满足精度要求的结果。

知识要点

首先对牛顿迭代法做一个简要的叙述：

设已知方程 $f(x)=0$ 的一个近似根 x_0，则 $f(x)$在 x_0 点附近可以用一阶泰勒多项式

$p_1(x)=f(x_0)+f'(x_0)(x-x_0)$

来近似，因此方程 $f(x)=0$ 在点 x0 附近可以近似地表示为

$f(x_0)+f'(x_0)(x-x_0)=0$

这个近似方程是个线性方程，设 $f'(x_0)\neq 0$，解之得：

$$x = x_0 - \frac{f(x_0)}{f'(x_k)}$$

取 x 作为原方程的新的近似根 x1。这种方法称为牛顿迭代法。牛顿迭代法的迭代公式是：

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

方程 $f(x)=0$ 的根 x'在几何上表示为曲线 $y=f(x)$与 x 轴的交点。设 x_k 是交点 x'附近的某个近似位置，过曲线 $y=f(x)$上的对应点 $p_k(x_k,f(x_k))$引切线，并将该切线与 x 轴交点 x_{k+1} 作为 x'新的近似位置。注意，到该切线的方程是：

$y=f(x_k)+f'(x_k)(x-x_k)$

这样得到的交点 x_{k+1} 必然满足牛顿迭代公式，因此，牛顿法也称为切线法。

使用牛顿迭代公式也是有一定条件的，如果 $f(x)$满足：

（1）$f(x)$在闭区间[a,b]上连续，$f'(x)f''(x)$均存在，$f'(x)$连续，且各自保持固定符号。

（2）$f(a)\times f(b)<0$；

（3）$f'(x0)\times f''(x)\geqslant 0;x,x0\in[a,b]$。

则方程 f(x)=0 在区间[a,b]上只有一个实根，取初始值 x0，由牛顿迭代公式

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

计算得到的序列：

$x_0, x_1, x_2, \ldots, x_k, \ldots$

收敛于方程 f(x)=0 的跟 x'。

而牛顿迭代法结束迭代的条件为：

设 ε 为预先给定的精度要求，则当满足下面表达式时：

$$\begin{cases} |f(x_{k+1})| < \varepsilon \\ |x_{k+1} - x_k| < \varepsilon \end{cases}$$

结束迭代。下面将具体讲述一下如何用 C 语言实现牛顿迭代。

程序源码

该应用程序的源代码如下：

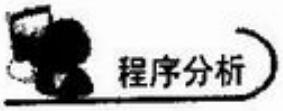
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int Function(double,double *,double *);          // 要求解的函数方程
int Newton(double *,double,int);                 // 牛顿迭代法求解

void main()
{
    double x,eps;
    int l;                                       // 迭代次数
    eps=1.e-6;                                   // 设定求解精度
    x=1.5;                                       // 初始值点
    l=60;
    if(!Newton(&x,eps,l))
// 调用迭代函数，如果返回值为 0，说明该方程不可以用牛顿迭代法求解
    {
        printf("该函数不可以用牛顿迭代法求根!\n");
    }
    printf("利用牛顿迭代法求的根为: \n");
    printf("x=%.10f\n",x);
}

int Function(x,f,dy)
double x;              // 输入参数，多项式 f(x)的自变量
```

```
double *f;              // 输出参数，返回值为给定 x 下 f(x)的值
double *dy;             // 输出参数，返回值为给定 x 下 f'(x)的值
{
    *f = x*x*(x-1)-1;
    *dy = 3*x*x-2*x;
    return(1);
}

int Newton(x,eps,l)     // 牛顿迭代法
double *x;      // 输入输出参数，输入时存放迭代初值，输出时存放方程根
double eps;     // 输入参数，存放精度要求
int l;          // 输入参数，存放迭代次数
{
    double f,dy,x1;
    Function(*x,&f,&dy);
A:  if(fabs(dy) == 0)
// 如果初值处函数一阶导数为零，则不可以用牛顿迭代法求根
    {
        l = 0;
        return (0);
    }
    x1=*x-f/dy;                     // 牛顿迭代公式
    Function(x1,&f,&dy);
    if(fabs(x1-*x)>=eps||fabs(f)>=eps)      // 如果不满足迭代结束条件
    {
        l-=1;                   // 相当于语句 I=I-1
        *x=x1;                  // 将本次迭代的结果赋值给 x 指针指向的存储区
        if(l==0)
            return(1);
        goto A;
    }

    *x = x1;
    return (1);
}
```

最终得到的结果是：

利用牛顿迭代法求的根为：

x=1.4655712319



本实例是通过函数 Newton(x,eps,l)来完成牛顿迭代的，其输入的三个参数分别为一次迭代的初始值、迭代要求的精度和最大迭代次数。首先调用 Function()函数，求得当前初始值 x 下

对应的 f(x)和其一阶导数 f'(x)的值，并通过语句 x1=*x-f/dy 调用牛顿迭代公式。然后函数通过判断语句 if(fabs(x1-*x)>=eps||fabs(f)>=eps)来判断此时得到的迭代结果是否满足迭代结束条件，即是否满足精度要求。如果不满足，那么将此时得到的 x1 赋值给 x，重新返回函数首部，再次调用迭代函数 Newton()进行第二次迭代，得到迭代结果 x2。同样，再次判断其是否满足迭代结束条件，如果不，则再次迭代，得到 x3。如此反复迭代下去，直到满足迭代精度要求或者达到事先设定的最大迭代次数，跳出循环，迭代结束。然后将此时的迭代结果 x1 赋值给 x 指针指向的存储单元，通过 x 指针将得到的结果传回主函数。

我们发现，用 C 语言实现牛顿迭代，语法上是非常简单的，关键是读者要能够彻底理解牛顿迭代法的实现原理，能够建立一个正确有效的数学模型。

实例86 弦截法求解方程

实例说明

前面向大家介绍了求解非线性方程数值解法中的二分法、牛顿迭代法。下面将继续介绍一种新的数值算法即弦截法。前面讲到的二分法虽然可以用来计算 f(x)=0 在[a,b]的所有实根，且方法简单，函数 f(x)要求连续就可以，但是二分法收敛速度较慢。而牛顿法虽然在单根附近收敛速度快，且具有二阶收敛速度，但牛顿法初值的选取要求比较苛刻，即要求初值选取充分靠近方程的根，否则不能收敛，而且它还有个明显的缺陷：需要计算函数 f(x)的导数，所以如果 f(x)比较复杂，那么使用牛顿迭代法就很不方便。

相对于前面讲述的两种求根方法，本例要介绍的弦截法有收敛速度快而且不需要求函数 f(x)的导数的特点。凡是可以建立成数学物理模型并且能够列成一元超越方程和代数方程的所有求解单实根的数值解问题都可以用弦截法来解决，应用范围较广。

本例将利用弦截法求解方程 $x^3-3x^2-6x+8=0$ 在[-3,5]之间的所有根。弦截法求根由函数 BowRoot()完成。

知识要点

首先介绍一下弦截法求根的原理和几何意义：

为了避免牛顿迭代法中的求导计算，采用差商 $\frac{f(x_k)-f(x_0)}{x_k-x_0}$ 作为导数的近似值，替换牛顿迭代公式中的 f' (x)就可以得到下面离散化的弦截公式：

$$x_{k+1}=x_k-\frac{f(x_k)}{f(x_k)-f(x_0)}(x_k-x_0)$$

若令 $g(x)=x-\frac{f(x)}{f(x)-f(x_0)}(x-x_0)$

所建立的迭代格式可以表述为：

$x_{k+1}=g(x_k)$

弦截法的几何意义如图 1 所示，曲线 y=f(x)上横坐标 x_k 的点记为 P_k，则弦线（P_0P_k）的方程为 $y=f(x_k)+\frac{f(x_k)-f(x_0)}{x_k-x_0}(x-x_k)$。上面弦截法计算出的 x_{k+1} 实际上就是弦线（P_0P_k）与 x 轴的交点。得到 x_{k+1} 以后，就可以得到它对应于函数 f(x)上的点 P_{k+1}，然后再由 P_0 点和 P_{k+1} 引弦线（P_0P_{k+1}）与 x 轴交点为 k+2，如此反复迭代下去，使得到的根逐渐逼近真实根 x'直到

达到事先设定的精度要求。

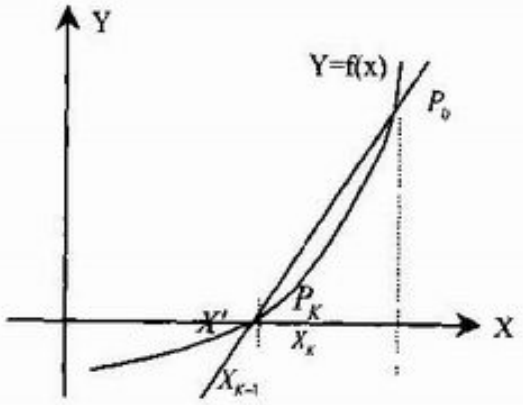


图 1

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <stdlib.h>

double Func(double);
int BowRoot(double,double,double,double,double *,int,int *);

void main()
{
    int i,n,m;
    double a,b,h,eps,*x;
    n = 3;                          // 方程根的个数的预估值
    x = (double*)calloc(n,sizeof(double));      // 开辟内存空间
    if(x == NULL)
    {
        printf("内存分配失败\n");
        exit(1);
    }
    a = -3;                                     // 区间起始端点
    b = 5;                                      // 区间终止端点
    h = 1;                                      // 步长
    eps = 1.e-8;                                // 要求达到的精度
    BowRoot(a,b,h,eps,x,n,&m);          // 调用二分法函数
    printf("函数 f(x)在范围%2.0f 和%2.0f 之间的根有%d 个根\n",a,b,m);
    printf("它们分别是: \n");
    for(i = 0;i<n;i++)
```

```
    printf("x[%d] = %e\n",i,x[i]);
    free(x);                        // 释放内存空间
}

double Func(double x)
{
    return (x*x*x-3*x*x-6*x+8);
}

int BowRoot(a,b,h,eps,x,n,m)
double a;           // 实型变量，输入参数，求根区间的起始端点
double b;           // 实型变量，输入参数，求根区间的终止端点
double h;           // 利用逐步扫描法确定根位置时的步长
double eps;      // 实型变量，输入参数，控制精度的参数
double *x;          // 实型一维数组，输出参数，存放计算得到的数组
int n;              // 输入参数，区间内方程根的个数的预估值
int *m;          // 输出参数，实际求得的根的个数
{
    double z,z1,z2,y,y1,y2;
    *m = 0;
    z = a;
    y = Func(z);
    while(1)        // 无限循环，直到遇到 return 或者 break 语句
    {// 如果逐步扫描到求根区间的右端点或者得到的根的个数达到预估根的个数
        if((z>b+h/2)||(*m==n))
            return(1);
        if(fabs(y)<eps)      // 如果当前根 z 对应的函数 f(z)满足精度要求
        {
            *m+=1;
            x[*m-1] = z;     // 将此时的 z 值赋值给 x 数组
            z+=h/2;
            y = Func(z);
            continue;        // 结束本次循环，即跳过循环体中下面尚未执行
                             // 的语句进行下一次是否执行循环的判定
        }

        z1 = z+h;            // 逐步扫描中小区间的右端点
        y1 = Func(z1);       // 小区间右端点对应的函数值
        if(fabs(y1)<eps)     // 如果右端点恰好满足根的精度要求
        {
            *m+=1;
            x[*m-1] = z1;
            z = z1+h/2;
            y = Func(z);
```

```
            continue;
        }
        if(y*y1>0)              // 如果对应根乘积大于零，说明该区间内没有根
        {
            y = y1;
            z = z1;
            continue;
        }
        while(1)    // 如果本 while 循环执行，说明逐步扫描的小区间[z,z1]间有根
        {
            if(fabs(z1-z)<eps)      // 如果满足精度要求
            {
                *m+=1;
                x[*m-1]=(z1+z)/2;
                z = z1+h/2;
                y = Func(z);
                break;
            }

            y1 = Func(z1);                  // 弦截法公式
            y = Func(z);
            z2=z1-(y1/(y1-y))*(z1-z);
            y2 = Func(z2);

            if(fabs(y2)<eps)
            {
                *m = *m+1;
                x[*m-1] = z2;
                z =z2+h/2;
                y = Func(z);
                break;
            }
            if(y*y2<0)              // 如果乘积小于零，说明根在 z 和 z2 之间
            {
                z1 = z2;
                y1 = y2;
            }
            else                    // 否则根在 z2 和 z1 之间
            {
                z = z2;
                y = y2;
            }
        }
    }
```

```
}
```

程序运行结果为：

函数 f(x)在范围-3 和 5 之间的根有 3 个根

它们分别是：

```
x[0] = -2.000000e+000
x[1] = 1.000000e+000
x[2] = 4.000000e+000
```

程序分析

下面结合实例讲述一下弦截法用 C 语言实现的方法。本例中，弦截法求解方程

$f(x)=0 \qquad x\in(a,b)$

根是由函数 BowRoot()完成的。由于方程 f(x)=0 在区域[a,b]内可能有多个根，因此首先要将区间[a,b]划分成几个小区间，确保每个区间内只有一个根，以便于用弦截法求解。

具体实现方法是：从端点 a 开始，按照给定的步长 h，取 z=a,z1=a+h，得到一个小区间[z,z1]，首先通过语句 if(fabs(y)<eps)和 if(fabs(y1)<eps)判断 f(z)和 f(z1)是否足够趋向于 0，如果是，那么 z 和 z1 就是方程的解。如果不是，判断此时小区间两端点的函数值 f(z)和 f(z1)是否异号。

```
if(y*y1>0)
    {   y = y1;
        z = z1;
        continue;
    }
```

如果同号，也就是 y*y1>0 成立，则该区间没有根。此时将 z1 值赋给 z，y1=f(z1)赋给变量 y，通过语句 continue 结束本次循环，跳到 while(1)处执行下一次循环，即给 z1 再次赋值为 z+h，也就是此时 z=a+h,z1=a+2h，再次判断 f(z)和 f(z1)是否同号，如此反复判断下去，直到 z>b 为止。反之如果 f(z)和 f(z1)异号，则在区间[z,z1]内必定有根，程序执行函数 BowRoot()中的第二个 while(1)语句。它按照弦截法公式 $x_{k+1}=x_k-\frac{f(x_k)}{f(x_k)-f(x_0)}(x_k-x_0)$，得到一个新的 x 值，然后判断 $f(x_{K+1})$和 $f(x_0)$的符号关系，如果同号，说明真实根 x'在 x_0 和 x_{K+1} 之间。如果异号，说明 x'在 x_{K+1} 和 x_K 之间。这里 x_{K+1} 对应函数中的 z2，x_K 对应函数中的 z1，而 x_0 对应函数中的 z。程序通过下面赋值语句实现一次弦截法求根：

```
z2=z1-(y1/(y1-y))*(z1-z);
```

BowRoot()函数通过一个数组 x 记录区间[a,b]内的所有实根，然后将该数组头指针传给主函数，最终由主函数将数组中的内容打印出来。

实例87 拉格朗日插值

实例说明

在生产实践中常常遇到这样的问题：给出一批离散的样点，要求作出一条通过这些样点的光滑曲线，以便满足设计要求或进行加工。反应在数学上，即已知函数在一些点上的值，寻求它的分析表达式，以便求得它在某个给定点的近似值。这个过程就称为插值。

插值的基本思想是，设法构造一个简单的函数 y=p(x)作为实际函数 f(x)的近似表达式，然后算出 p(x)的值，以此得到 f(x)的近似值。本实例演示的便是在已知六个样点值的情况下，用拉格朗日插值法计算出给定某 x 值处的近似函数值。

知识要点

用插值法求函数的近似值时，首先要选定近似的函数形式。可供选择的函数很多，常用的是多项式函数。因为多项式函数计算简便，只需要用加、减、乘等运算，其导数与积分仍为多项式。

用多项式作为研究插值的工具称为代数插值。基本思想是：已知函数 f(x)在区间[a,b]上 n+1 个不同点 $x_0,x_1,\ldots,x_n$ 处的函数值 $y_i=f(x_i)(i=0,1,\ldots,n)$,求一个至多 n 次的多项式

$$\phi_n(x_i)=a_0+a_1x+\ldots+a_nx^n$$

使其在给定点处与 f(x)同值，即满足插值条件：

$$\phi_n(x_i)=f(x_i)=y_i \qquad (i=0,1,\ldots,n)$$

那么 $\phi_n(x)$称为插值多项式，x_i (i=0,1,…,n)称为插值节点，简称节点，[a,b]称为插值区间。从几何上看，n 次多项式插值就是过 n+1 个点 $(x_i,f(x_i))$，(i=0,1,…,n),做一条多项式曲线 $y=\phi_n(x)$ 来近似曲线 y=f(x)。

先讨论最简单的情况：只有两个节点 x_0、x_1(n=1)的插值多项式。由前面所述，插值多项式为 $\phi_1(x)=a_0+a_1x$，且满足条件：

$$\phi_1(x_0)=a_0+a_1x_0=y_0=f(x_0)$$
$$\phi_1(x_1)=a_0+a_1x_1=y_1=f(x_1)$$

解此方程得：$a_0=\dfrac{(y_1x_0-y_0x_1)}{x_0-x_1}$，$a_1=\dfrac{(y_0-y_1)}{x_0-x_1}$

两节点的一次插值多项式为

$$\phi(x)=\frac{(y_1x_0-y_0x_1)}{x_0-x_1}+\frac{(y_0-y_1)}{x_0-x_1}x \qquad (1)$$

可以将上式改写成如下形式：

$\phi_1(x)=y_0[(x-x_1)/(x_0-x_1)]]+y_1[(x-x_1)/(x_0-x_1)]$

同样，若是 n 点插值，则有公式：$\phi_n(x_k)=\sum_{i=0}^{n} y_i l_i(x)$ （2）

其中 $l_i(x)=\dfrac{(x-x_0)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)}{(x_i-x_0)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_n)}=\prod_{\substack{j=0\\ j\neq i}}^{n}\dfrac{x-x_j}{x_i-x_j}$ （3）

式（2）就称为 n 次拉格朗日插值多项式。

程序源码

根据函数表：

X	0	0.1	0.195	0.3	0.401	0.5
y	0.39894	0.39695	0.39142	0.38138	0.36812	0.35206

求函数在 x=0.12 处的函数值。

该应用程序的源代码如下：

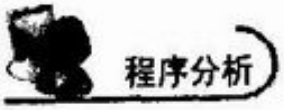
```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
double LAG(int,double *,double *,double);          // 拉格朗日插值函数
void main()
{
    int n;                                         // 已知样点的个数
    double *x,*y,t,lag;
    t = 0.15;                                      // 求 x 为 0.15 时函数的值
    n = 6;
    x = (double*)calloc(n,sizeof(double));         // 动态分配内存区
    if(x == NULL)
    {
        printf("内存分配失败\n");
        exit(1);
    }
    y = (double*)calloc(n,sizeof(double));         // 动态分配内存区
    if(y == NULL)
    {
        printf("内存分配失败\n");
        exit(1);
    }
    x[0] = 0;                                      // 输入各个样点的 x 值
```

```
    x[1] = 0.1;
    x[2] = 0.195;
    x[3] = 0.3;
    x[4] = 0.401;
    x[5] = 0.5;
    y[0] = 0.39894;                          // 输入各个样点的函数值 y
    y[1] = 0.39695;
    y[2] = 0.39142;
    y[3] = 0.38138;
    y[4] = 0.36812;
    y[5] = 0.35206;
    lag = LAG(n,x,y,t);                      // 调用拉格朗日插值函数
    printf("拉格朗日插值后得到的结果是: \n");
    printf("f(%.2f)=%e\n",t,lag);
    free(x);                                 // 释放内存
    free(y);                                 // 释放内存
}
double LAG(n,x,y,t)
int n;                          // 整型变量，给定插值点的个数
double *x;                      // 实型一维数组，存放给定的插值节点
double *y;                      // 实型一维数组，存放节点处的函数值
double t;                       // 实型变量，输入参数，插值点
{
    int i,j;
    double p,s;
    s = 0;
    for(i=0;i<n-1;i++)
    {
p = 1;
        for(j=0;j<n-1;j++)
            if(i!=j)
                p*=(t-x[j])/(x[i]-x[j]);
            s+=p*y[i];
    }
    return (s);
}
```

函数运行结果：

拉格朗日插值后得到的结果是：

```
f(0.15)=3.944728e-001
```



该程序中拉格朗日插值是由函数 LAG(n,x,y,t)完成的。其中参数 n 表示给定插值点的个数。

本例给定六个插值点，所以 n=6。参数 x 和 y 分别存储的是给定节点的 x 值和 y 值，参数 t 是插值点，即要求该点处对应的函数值。该函数主要由两个 for 循环组成，通过里面的 for 循环语句：

```
for(j=0;j<n-1;j++)
        {   if(i!=j)
            p*=(t-x[j])/(x[i]-x[j]);}
```

该循环语句根据前面得到的（3）式做了一个连乘操作，求得拉格朗日插值多项式的 $l_i(x)$。此时的 p 就是 $l_i(x)$，然后通过外层的 for 循环以及自加语句 s+=p*y[i]得到拉格朗日插值多项式 $\phi_n(x_k)=\sum_{i=0}^{n} y_i l_i(x)$ 在 x=t 点的函数值 s。最后函数利用 return(s)语句将该值返回给主函数。

实例88 最小二乘法拟合

实例说明

曲线拟合又称函数逼近，是求近似函数的另一种方法。它不要求近似曲线过已知点，只要求尽可能反应给定数据点的基本趋势，在某种意义下与函数最“逼近”。

最小二乘法曲线拟合是使拟合出来的曲线得到的各个观测点 yi 与真实点 y'之差的平方和最小，即 $\sum_i (y_i - y')^2 = \min$

本例首先给定函数 f(x)=x-e^{-x},然后从 x1=0 开始，取步长 h=0.1 的 20 个数据点，然后根据这些数据点求 n 次最小二乘拟合多项式 p(x)=a1+a2(x-$\overline{x}$)+…+a_{m-1}(x-$\overline{x}$)$^{m-1}$。

知识要点

人们在生产实践和科学研究中，对某些复杂的函数关系由于得不到精确的解析式，而人为的构造简单函数逼近，这个问题归结为由曲线 y=f(x)上已给出的 n 个点，求做该曲线的近似图形的问题。前面介绍的插值要求近似曲线严格通过给定的 n 个点，由于其得到的数据总是带有测试误差，这就使得近似曲线 y=p(x)保留了数据自身所带的全部测量误差。拟合又称为函数逼近，是求近似函数的另一种方法。它不要求近似曲线过已知点，只要求它尽可能反应给定数据点的基本趋势，在某种意义下与函数最“逼近”。

最小二乘拟合分为指数拟合和多项式拟合，这里介绍的是比较常用的多项式拟合。所谓多项式拟合就是指对于给定的数据组（x_i,y_i）(i=1,2,…,n)，求一个 m 次多项式（m<n）

$P_m(x)=a_0+a_1x+\ldots+a_mx^m$

使得 $\sum_{i=1}^{n}\delta_i^2 = \sum_{i=1}^{n}[y_i - P_m(x_i)]^2 = F(a_0, a_1, \cdots, a_m)$

为最小，即选取参数 a_i(I=0,1,2,…,m)，使得

$$F(a_0,a_1,\cdots,a_m) = \sum_{i=1}^{n}[y_i - P_m(x_i)]^2 = \min_{\phi \in H}\sum_{i=1}^{n}[y_1 - \varphi(x_i)]^2$$

其中 F 为至多 m 次多项式几何，这就是数据的多项式拟合，$P_m(x)$称为这组数据的最小 m 次拟合多项式。

最小二乘法拟合数据所用的方法简要介绍如下：

设给定 n+1 个点：$x_0<x_1<x_2<\ldots<x_n$ 及相应的函数值 $y_0,y_1,y_2\ldots y_n$，要求出 m-1(m≤n)次最小二乘拟合多项式：

$P_m(x)=a_0+a_1x+\ldots+a_{m-1}x^{m-1}$

设拟合多项式为正交多项式 $q_j(x)(j=0,1,2,\ldots,m)$的线性组合

$p_m(x)=c_0q_0(x)+c_1q_1(x)+\ldots+c_mq_m(x)$

而 $q_j(x)$可以由以下的递推公式得到

$q_0(x)=1$

$q_1(x)=(x-1a_2)$

$q_{j+1}=(x-a_{j+1})q_j(x)-B_jq_{j-1}(x)$　　j=1,2,...,m-1

若设 $d_j=\sum_{j=0}^{n}q_j^2(x_j)$　　j=0,1,2,...,m

则 $\begin{cases}a_j=\sum_{i=0}^{n}x_iq_j^2(x_i)/d_j\\B_j=d_j/d_{j-1}\end{cases}$　　j=0,1,2,...,m-1

上面得到的多项式{q_j}(j=0,1,2,...,m)是相互正交的。根据最小二乘法原理，可得

$c_j=\sum_{i=0}^{n}y_iq_j(x_i)/d_j$，　j=0,1,2,...,m

最后可以化成一般的 m-1 次多项式

$P_m(x)=a_0+a_1x+\ldots+a_{m-1}x^{m-1}$

在实际计算中，为了在实际运算中防止运算溢出，x_i用 $x'_i=x_i-\overline{x}$ 代替。此时拟合多项式的形式变为 $p(x)=a1+a2(x-\overline{x})+\cdots+a_{m-1}(x-\overline{x})^{m-1}$

程序源码

该应用程序的源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>

Smooth(double *,double *,double *,int,int,
       double *,double *,double *);
void main()
{
    int i,n,m;
    double *x,*y,*a,dt1,dt2,dt3,b;
    n = 20;
    m = 6;
    b = 0;
    // 分别为 x,y,a 分配存贮空间
    x = (double *)calloc(n,sizeof(double));
```

```
    if(x == NULL)
    {
        printf("内存分配失败\n");
        exit (0);
    }
    y = (double *)calloc(n,sizeof(double));
    if(y == NULL)
    {
        printf("内存分配失败\n");
        exit (0);
    }
    a = (double *)calloc(n,sizeof(double));
    if(a == NULL)
    {
        printf("内存分配失败\n");
        exit (0);
    }
    for(i=1;i<=n;i++)
    {
        x[i-1]=b+(i-1)*0.1;
        // 每隔0.1取一个点，这样连续取n个点
        y[i-1]=x[i-1]-exp(-x[i-1]);
        // 计算x[i-1]点对应的y值作为拟合已知值
    }
    Smooth(x,y,a,n,m,&dt1,&dt2,&dt3);          // 调用拟合函数
    for(i=1;i<=m;i++)
        printf("a[%d] = %.10f\n",(i-1),a[i-1]);
    printf("拟合多项式与数据点偏差的平方和为：\n");
    printf("%.10e\n",dt1);
    printf("拟合多项式与数据点偏差的绝对值之和为：\n");
    printf("%.10e\n",dt2);
    printf("拟合多项式与数据点偏差的绝对值最大值为：\n");
    printf("%.10e\n",dt3);
    free(x);                                   // 释放存储空间
    free(y);                                   // 释放存储空间
    free(a);                                   // 释放存储空间
}

Smooth(x,y,a,n,m,dt1,dt2,dt3 )
    double *x;      // 实型一维数组，输入参数，存放节点的x_i值
    double *y;      // 实型一维数组，输入参数，存放节点的y_i值
    // 双精度实型一维数组，长度为m。返回m-1次拟合多项式的m个系数
    double *a;
    int n;          // 整型变量，输入参数，给定数据点的个数
```

```
    int m;              // 整型变量，输入参数，拟合多项式的项数
    // 实型变量，输出参数，拟合多项式与数据点偏差的平方和
    double *dt1;
    // 实型变量，输出参数，拟合多项式与数据点偏差的绝对值之和
    double *dt2;
// 实型变量，输出参数，拟合多项式与数据点偏差的绝对值最大值
    double *dt3;
{
    int i,j,k;
    double *s,*t,*b,z,d1,p,c,d2,g,q,dt;
    // 分别为 s,t,b 分配存储空间
    s = (double *)calloc(n,sizeof(double));
    if(s == NULL)
    {
        printf("内存分配失败\n");
        exit (0);
    }
    t = (double *)calloc(n,sizeof(double));
    if(t == NULL)
    {
        printf("内存分配失败\n");
        exit (0);
    }
    b = (double *)calloc(n,sizeof(double));
    if(b == NULL)
    {
        printf("内存分配失败\n");
        exit (0);
    }
    z = 0;
    for(i=1;i<=n;i++)
        z=z+x[i-1]/n;          // z 为各个 x 的平均值
    b[0]=1;                    // 当 Q0(x)=1 时，可计算得 b0=1
    d1=n;                      // d0=n
    p=0;
    c=0;
    for(i=1;i<=n;i++)
    {
        p=p+x[i-1]-z;          //p=\sum_{i=0}^{n-1}(x_i-\overline{x})

        c=c+y[i-1];            //c=\sum_{i=0}^{n-1}yi
    }
```

```
c=c/d1;                          // c = (1/d0)*Σ(i=0..n-1) y_i = (1/n)*Σ(i=0..n-1) y_i

p=p/d1;                          //p = (1/d0)*Σ(i=0..n-1)(x_i - x̄) = (1/n)*Σ(i=0..n-1)(x_i - x̄)

a[0]=c*b[0];
if(m>1)                          // 当拟合多项式最高次数大于 1 时
{
    t[1]=1;                      // 赋值 t1=1
    t[0]=-p;                     // t0 = -a = -p
    d2=0;
    c=0;
    g=0;
for(i=1;i<=n;i++)                //注意此时 for 循环是从 1 开始的
{
        q=x[i-1]-z-p;            //q = (x_i - x̄) - (1/n)*Σ(i=0..n-1)(x_i - x̄)

        d2=d2+q*q;
        c=y[i-1]*q+c;
        g=(x[i-1]-z)*q*q+g;
    }
    c=c/d2;
    p=g/d2;
    q=d2/d1;                     // 对应程序说明中的 B = d_j / d_{j-1} */

    d1=d2;
    a[1]=c*t[1];
    a[0]=c*t[0]+a[0];
}
for(j=3;j<=m;j++)                // 对应程序说明中的 3 种情况
{
    s[j-1]=t[j-2];               //s_j=t_{j-1},注意循环是从 j=3 开始的
    s[j-2]=-p*t[j-2]+t[j-3];     //s_j-1=-αt_{j-1}+t_{j-2}
    if(j>=4)
        for(k=j-2;k>=2;k--)
        // s_k=-αt_i+t_{k-1}-Bb_k
            s[k-1]=-p*t[k-1]+t[k-2]-q*b[k-1];
        s[0]=-p*t[0]-q*b[0];
        d2=0;
        c=0;
        g=0;
        for(i=1;i<=n;i++)
        {
```

```
                q=s[j-1];
                for(k=j-1;k>=1;k--)
                    q=q*(x[i-1]-z)+s[k-1];
                d2=d2+q*q;
                c=y[i-1]*q+c;
                g=(x[i-1]-z)*q*q+g;
            }
            c=c/d2;                    // c_j=\frac{1}{d_j}\sum_{i=0}^{n-1}y_iQ_j(x_i)
            p=g/d2;
            q=d2/d1;                   // 对应程序说明中的B=\frac{d_j}{d_{j-1}} */
            d1=d2;
            a[j-1]=c*s[j-1];           // a_j=c_js_j
            t[j-1]=s[j-1];
            for(k=j-1;k>=1;k--)
            {
                // a_k=a_k+c_js_k,(k=j-1,…,1,0)
a[k-1]=c*s[k-1]+a[k-1];
                b[k-1]=t[k-1];
                t[k-1]=s[k-1];
            }
    }
    *dt1=0;
    *dt2=0;
    *dt3=0;
    for(i=1;i<=n;i++)
    {
        q=a[m-1];                  // 此时a[m-1]中为要求的系数a_1
        for(k=m-1;k>=1;k--)
            q=q*(x[i-1]-z)+a[k-1]; // q为拟合后x=x_i时得到的估计值
        dt=q-y[i-1];
        if(fabs(dt)>*dt3)          // 取dt3为最大偏差的绝对值
            *dt3=fabs(dt);
        *dt1=*dt1+dt*dt;           // 取dt1偏差的平方和
        *dt2=*dt2+fabs(dt);        // 取dt2偏差的绝对值之和
        }
    free(s);                       // 释放存储空间
    free(t);                       // 释放存储空间
    free(b);                       // 释放存储空间
    return(1);
    }
```

该程序执行结果为：

```
a[0] = 0.5632480493
a[1] = 1.3867467012
a[2] = -0.1931338887
a[3] = 0.0644035495
a[4] = -0.0168412204
a[5] = 0.0033442883
拟合多项式与数据点偏差的平方和为:
1.8017419171e-009
拟合多项式与数据点偏差的绝对值之和为:
1.6850492746e-004
拟合多项式与数据点偏差的绝对值最大值为:
1.5393962046e-005
```

程序分析

该程序是由函数 Smooth()实现最小二乘法曲线拟合的，该函数的具体实现方法和步骤如下：

（1）$Q_0(x)=1$，可得

$b_0=1$，　$d_0=n$，　$c_0=\frac{1}{d_0}\sum_{i=0}^{n-1}y_i$，　$a=\frac{1}{d_1}\sum_{i=0}^{n-1}x_i$

$a_0=c_0b_0$

（2）$Q_1(x)=(x-a)$，可得

$t_0=-a$，　$t_1=1$

$$d_1=\sum_{i=0}^{n-1}Q_1^2(x_i) \qquad c_1=\frac{1}{d_1}\sum_{i=0}^{n-1}y_iQ_1(x_i)$$

$$\alpha=\frac{1}{d_1}\sum_{i=0}^{n-1}x_iQ_1^2(x_i), \qquad B=\frac{d_1}{d_0}$$

$a_0=a_0+c_1t_0$，　$a_1=c_1t_1$

（3）对于 j=2,3,...,m-1 做以下各步：

$Q_j(x)=(x-\alpha)Q_{j-1}(x)-BQ_{j-2}(x)$

$=(x-\alpha)(t_{j-1}x^{j-1}+\ldots+t_1x+t_0)-B(b_{j-2}x^{j-2}+\ldots+b_1x+b_0)$

$\overset{令}{=}s_jx^j+s_{j-1}x^{j-1}+\ldots+s_1x+s_0$

其中 s_i（i=0,1,...j）由以下递推公式计算：

$$\begin{cases} s_j = t_{j-1} \\ s_{j-1} = -\alpha t_{j-1} + t_{j-2} \\ s_k = -\alpha t_k + t_{k-1} - Bb, (k = j-2,\ldots,1) \\ s_0 = -\alpha t_0 - Bt_0 \end{cases}$$

再计算 $d_j = \sum_{i=0}^{n-1} Q_j^{\,2}(x_i)$

$$c_j = \frac{1}{d_j} \sum_{i=0}^{n-1} y_i Q_j(x_i)$$

$$\alpha = \frac{1}{d_j} \sum_{i=0}^{n-1} x_i Q^2{}_j(x_i) \qquad B = \frac{d_j}{d_{j-1}}$$

由此可以计算相应的 a_i

$$\begin{cases} a_j = c_j s_j \\ a_k = a_k + c_j s_k, (k = j-1,\ldots,1,0) \end{cases}$$

在实际过程中，为了防止运算溢出，x_i 用

$x_i' = x_i - \bar{x}$，　　$I = 0,1,2,\ldots,n-1$

来代替，其中

$$\bar{x} = \sum_i^n x_i / n$$

此时拟合多项式为 $p(x) = a1 + a2(x-\bar{x}) + \cdots + a_{m-1}(x-\bar{x})^{m-1}$。

函数如何用 C 语言实现上述算法，请参照程序注释和上面的算法步骤自己理解，这里不再赘述。最后程序将拟合后的多项式的各个系数打印出来，并同时将拟合后的曲线与真实曲线参考点的偏差的平方和 dt1、偏差的绝对值之和 dt2 以及最大偏差的绝对值 dt3 分别打印出来。

实例89 辛普生数值积分

实例说明

在微积分中，积分值是通过找原函数的解析式求得的，然而寻找原函数往往非常困难，许多积分函数甚至找不到用初等函数表示的原函数。为此，研究积分的数值计算问题是非常必要的。

本实例通过求解定积分$\int_0^{0.8}\cos x dx$，介绍求解积分一种常用的方法：辛普生数值积分。辛普生数值积分有定步长和变步长两种方法，本实例分别通过两个函数SIMP1()和SIMP2()实现辛普生定步长和变步长积分。

知识要点

大家清楚，如果给出的函数f(x)比较复杂，研究和分析其性质就会十分困难，在这种情况下可以构造函数Pn(x)作为f(x)的近似表达式，运用Pn(x)得到f(x)的近似结果。在积分的实际计算中，常用的插值公式有两点公式（梯形公式），三点公式（辛普生公式）。如果积分区间比较大，直接使用这些求积公式精度难以保证。通常采取的办法是细分求积区间。譬如取步长h为区间(a,b)的n等份，分点为$x_k=a+kh,k=0,1,2,\ldots,n$，然后对每个分段$(x_{k-1},x_k)$，使用上述求积公式得到积分近似值$I_k$，并取其和值$I=\sum_{k=1}^{n}I_k$作为整个区间上的积分近似值，这种求积方案称作为复化求积法。辛普生积分主要从定步长和变步长两种方法来研究积分。

下面介绍一下辛普生公式的原理。如图1所示：如果用梯形面积代替函数f(x)在区间（a，b）的积分，即：

$$I(f)=\int_a^b f(x)dx\approx\frac{b-a}{2}(f(a)+f(b))$$

那么上式称为梯形公式。如果以过曲线上三点(a,f(a)),($\frac{a-b}{2}$，f$\frac{a+b}{2}$)),(b,f(b))的抛物线代替y=f(x)，求曲边梯形面积的近似值代替积分值．即：

$$I(f)=\int_a^b f(x)dx\approx\frac{b-a}{6}[f(a)+4f(\frac{a+b}{2})+f(b)]$$

上式称为抛物线求积公式，又称辛普生公式。

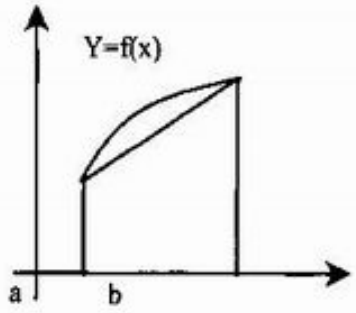


图 1

下面把定步长和变步长辛普生数值积分的实现思想简要介绍一下。

（1）定步长辛普生数值积分的基本思想是：把积分区间(a,b)分为 2n 等份，在每个小区间上利用二次多项式来近似 f(x)。计算定积分的辛普生公式如下：

$$\int_a^b f(x)dx \approx \frac{h}{3}\left\{f(a)+f(b)+2\sum_{i=1}^{n-1} f(x_{2i})+4\sum_{i=1}^{n} f(x_{2i-1})\right.$$

为了方便计算，把上式改写成

$$\int_a^b f(x)dx \approx \frac{b-a}{3n}\left\{0.5[f(x)-f(b)+\sum_{i=1}^{n}[2f(a+(2i-1)h+f(a+2ih)]\right\}$$

如果 n 固定，上式就是定步长辛普生求积分公式。使用此公式时，可以利用辛普生公式的截断误差确定 n，以保证结果的精度。它的截断误差是

$$R(f)=\frac{(b-a)^5}{2880n^4}f^{(4)}(\varepsilon) \qquad a<\varepsilon<b$$

（2）前面介绍的复化求积法对提高精度是行之有效的，但是使用复化求积公式之前必须给出合适的步长，步长取得太大精度难以保证，步长太小则导致计算量增加，给出一个恰当的步长是困难的。

实际计算时通常采用变步长求积方法，即在步长逐次折半（步长二分）的过程中，反复利用复化的求积分公式进行计算，直到二分前后的两次积分近似值之差符合精度要求为止。

程序源码

分别用定步长辛普生公式和变步长辛普生公式积分

$$I=\int_0^{0.8}\cos x dx$$

要求误差不超过 5×10^{-7}。

该应用程序的源代码如下：

```
#include <stdio.h>
#include <math.h>
// 对要用到的函数进行声明
double Function(double);
double SIMP1(double,double,int);
double SIMP2(double,double,double);
void main()
{
    double a1,b1,eps;
    int n1;
    double Result1;          // 存放定步长辛普生积分的积分结果
    double Result2;          // 存放变步长辛普生积分的积分结果
    a1 = 0.0;                // 积分区间的起始位置
    b1 = 0.8;                // 积分区间的终止位置
    n1 = 4;                  // 将积分区间(0,0.8)分成 2n 等份
    eps = 5e-7;              // 变步长积分要求达到的精度
    Result1 = SIMP1(a1,b1,n1);          // 调用定步长辛普生积分函数
    Result2 = SIMP2(a1,b1,eps);         // 调用变步长辛普生积分函数
    printf("利用定步长辛普生积分结果为: \n");
    printf("I1 = %.10f\n",Result1);
    printf("利用变步长辛普生积分结果为: \n");
    printf("I2 = %e\n",Result2);
}
double SIMP1(a,b,n)      // 定步长辛普生积分函数
double a;                // 积分区间的起始位置
double b;                // 积分区间的终止位置
int n;                   // 将积分区间分成 2n 等份
{
    int i;
    double h,s;
    h=(a-b)/(2*n);           // 根据 n 值确定步长的大小
    s=0.5*(Function(a)-Function(b));    //s=[f(a)-f(b)]/2
    for(i=1;i<=n;i++)
    // 调用辛普生积分公式
        s+=2*Function(a+(2*i-1)*h)+Function(a+2*i*h);
    return((b-a)*s/(3*n));
}
double SIMP2(a,b,eps)        // 变步长辛普生积分函数
double a;                    // 积分区间的起始位置
double b;                    // 积分区间的终止位置
double eps;                  // 变步长积分要求达到的精度
{
    int k,n;
    double h,t1,t2,s1,s2,p,x;
```

```
    n=1;
    h=b-a;                          // 初始步长设为b-a
    t1=h*(Function(a)+Function(b))/2;        // 求得t1
    s1 = t1;
    while(1)
    {
        p = 0;
        for(k=0;k<=n;k++)
        {
            x = a+(k+0.5)*h;
            p+=Function(x);
        }
        t2=(t1+h*p)/2;
        s2=(4*t2-t1)/3;
        if(fabs(s2-s1)>=eps)          // 如果不满足精度要求
        {
            t1=t2;
            n=n+n;                    // n变为原来的两倍
            h=h/2;                    // 步长变为原来的二分之一
            s1=s2;
            continue;                 // 返回while循环开始处
        }
        break;                        // 如果精度满足要求，则跳出while循环
    }
    return(s2);
}
double Function(double x)
{
    return(cos(x));
}
```

该程序的运行结果为：

利用定步长辛普生积分结果为：

I1 = 0.7173564899

利用变步长辛普生积分结果为：

I2 = 7.173566e-001

程序分析

该程序通过两个函数 SIMP1()和 SIMP2()分别完成定步长和变步长辛普生积分。在函数SIMP1()中，通过 for 循环语句：

```
for(i=1;i<=n;i++)
    s+=2*Function(a+(2*i-1)*h)+Function(a+2*i*h);
```

求得s等于$0.5[f(x)-f(b)]+\sum_{i=1}^{n}[2f(a+(2i-1)h+f(a+2ih)]$，然后通过语句return((b-a)*s/(3*n))将辛普生定步长求得积分值返回。

而 SIMP2()函数实现得步骤为：

第一步：用梯形公式计算：

$T_n=\frac{h}{2}(f(a)+f(b))$，其中 n=1,h=b-a。

第二步：用变步长梯形法计算：

$T_{2n}=\frac{1}{2}Tn+\frac{h}{2}\sum_{k=0}^{n-1}f(x_k+\frac{h}{2})$

第三步：用辛普生求积分公式：

$I_{2n}=\frac{4T_{2n}-T_n}{3}$

第四步：若$|I_{2n}-I_n|<\varepsilon$，则结束，I_{2n}即为所求积分的近似值。若$|I_{2n}-I_n|\geq\varepsilon$，则令 2n=>n,h/2=>h,重复第二、三步。具体如何用 C 语言实现，请结合上面所述算法步骤，参照源程序以及程序注释自己理解。

实例90 改进欧拉法

实例说明

许多实际问题的数学模型是微分方程或微分方程组的定解问题，如物体运动、电路震荡、化学反应及生物的群体变化等。能用解析方法求出精确解的微分方程为数不多，而且有的方程即使有解析解，也可能由于表达式非常复杂而不易计算，因此有必要研究微分方程的数值解法。

本实例演示的是用改进欧拉法求解一阶常微分方程组的 C 语言实现方法。程序中函数 Euler1()和 Euler2()分别用定步长和变步长方法实现改进欧拉法求解常微分方程组。

知识要点

这里主要讨论的是一阶常微分方程的初值问题，其一般形式是

$$\begin{cases} \dfrac{dy}{dx} = f(x,y) & (a \leqslant x \leqslant b) \\ y(a) = y_0 \end{cases} \qquad (1)$$

所谓的数值解法，就是求（1）式的解 y(x)在若干点 $a=x_0<x_1<x_2\ldots<x_N=b$ 处的近似值 y_n（n=1,2...,N）的方法，y_n（n=1,2...,N）称为上面方程的数值解，$h_n=x_{n+1}-x_n$ 称为由 x_n 到 x_{n+1} 的步长，在下面的讨论中取步长为常量 h。

建立数值解法，首先要将微分方程离散化，一般采用以下几种方法:

（1）用差商近似导数:

即用 $\dfrac{y(x_{n+1})-y(x_n)}{h}$ 代替 $y'(x_n)$代入（1）式中的微分方程，得

$$\frac{y(x_{n+1})-y(x_n)}{h} \approx f(x_n, y(x_n)) \qquad (n=0,1,\ldots)$$

化简得 $\begin{cases} y_{n+1} = y_n + hf(x_n, y_n) \\ y_0 = y(a) \end{cases}$ （n=0,1,...） （2）

如果用上式依次计算 $y(x_n)$的近似值就称为欧拉（Euler）法。

（2）用数值积分法:

将（1）式的解表示成积分形式，用数值积分方法离散化。对微分方程两端积分得

$$y(x_{n+1})-y(x_n) = \int_n^{n+1} f(x, y(x))dx \qquad (n=0,1\ldots)$$

用梯形公式计算上式右端得积分，即

$$\int_n^{n+1} f(x,y(x))dx \approx \frac{h}{2}[f(x_n,y(x_n))+f(x_{n+1},y(x_{n+1}))]$$

用 y_n 和 y_{n+1} 代替 $y(x_n)$ 和 $y(x_{n+1})$ 得到计算公式：

$$y_{n+1}=y_n+\frac{h}{2}[(f(x_n,y_n)+f(x_{n+1},\bar{y}_{n+1}))] \quad (3)$$

改进欧拉法是先用 Euler 公式（2）求得 y_{n+1} 的一个初步近似值 $\bar{y}_{n+1}$，称为预测值，然后用梯形公式（3）求得近似值 y_{n+1}，即：

$$\begin{cases}\bar{y}_{n+1}=y_n+hf(x_n,y_n) & \text{预测}\\ y_{n+1}=y_n+\dfrac{h}{2}[(f(x_n,y_n)+f(x_{n+1},\bar{y}_{n+1})] & \text{校正}\end{cases} \quad (4)$$

上式称为由欧拉(Euler)公式和梯形公式得到的预测-校正系统，也叫改进欧拉法。为了便于编制程序，上式常改写成

$$\begin{cases}y_p=y_n+hf(x_n,y_n)\\ y_q=y_n+hf(x_n+h,y_p)\\ y_{n+1}=(y_p+y_q)/2\end{cases} \quad (5)$$

程序源码

求下列微分方程组 $\begin{cases}y_1'=y_2, & y_1(0)=-1.0\\ y_2'=-y_1, & y_2(0)=0.0\\ y_3'=-y_3, & y_3(0)=1.0\end{cases}$

取 h=0.01,11 个积分点（包括初值点），$t_i=(i-1)h$ (i=1,2,3...,11)。

该应用程序的源代码如下：

```
#include "stdio.h"
#include "stdlib.h"
#include <math.h>
int Func(y,d)                    // 待求的常微分方程组
aouble y[],d[];
{   d[0]=y[1];                   // y0'=y1
    d[1]=-y[0];                  // y1'=y0
    d[2]=-y[2];                  // y2'=y2
    return(1);}
void Euler1(t,y,n,h,k,z)
int n;          // 微分方程组中方程的个数，也是未知函数的个数
int k;          // 积分步数(包括起始点这一步)
double t;       // 对微分方程进行积分的起始点 t0
```

```
double h;        // 积分步长
double y[]; // 长度为 n。存放 n 个未知函数 yi 在起始点 t0 处的函数值
double z[]; // 体积为 nxk。返回 k 个积分点(包括起始点)上的未知函数值
{
    extern int Func();
    int i,j;
    double *d;
    d=malloc(n*sizeof(double));          // 开辟存储空间
    if(d == NULL)
    {   printf("内存分配失败\n");
        exit(1);}
    // 将方程组的初值赋给数组 z[i*k]
    for (i=0; i<=n-1; i++) z[i*k]=y[i];
    for (j=1; j<=k-1; j++)
    {
        Func(y,d);                       // 求出 f(x)
        for(i=0; i<=n-1; i++)
            y[i]=z[i*k+j-1]+h*d[i];      // yp=yn+hf(xn,yn)
        Func(y,d);                       // 求得 f(xn+h,yp)
        for (i=0; i<=n-1; i++)
            d[i]=z[i*k+j-1]+h*d[i];      // yq=yn+h f(xn+h,yp)
        for (i=0; i<=n-1; i++)
        {
            y[i]=(y[i]+d[i])/2.0;        // yn+1=(yp+yn)/2
            z[i*k+j]=y[i];               // 将结果存到数组 z 中
        }
    }
    free(d);                             // 释放存储空间
     return;                             // 函数返回
  }
void Euler2(t,h,y,n,eps)
int n;           // 微分方程组中方程的个数，也是未知函数的个数
double t;        // 积分一步的起始点
double h;        // 积分步长
double eps; // 积分的精度要求
double y[]; // 长度为 n，存放起始点 t 处 n 个位置函数 yi(t)
返回 t+h 点处的 n 个未知函数值 yi(t+h)
{
     int i,j,m;
     double hh,p,q,*a,*b,*c,*d;
   a=malloc(n*sizeof(double));           // 开辟存储空间
   b=malloc(n*sizeof(double));           // 开辟存储空间
   c=malloc(n*sizeof(double));           // 开辟存储空间
   d=malloc(n*sizeof(double));           // 开辟存储空间
```

```
    hh=h;                                   // 初始步长
     m=1; p=1.0+eps;
     for (i=0; i<=n-1; i++) a[i]=y[i];      // 初始值暂存到数组 a 中
    while (p>=eps)                          // 如果精度不满足要求
    {
          for (i=0; i<=n-1; i++)
        { b[i]=y[i];
            y[i]=a[i]; }
        for (j=0; j<=m-1; j++)
        {
             for (i=0; i<=n-1; i++)
                c[i]=y[i];                  // 当前初值存储到数组 c 中
            Func(y,d);
            for (i=0; i<=n-1; i++)
               y[i]=c[i]+hh*d[i];           // yp=yn+hf(xn,yn)
            Func(y,d);                      // 求得 f(xn+h,yp)
            for (i=0; i<=n-1; i++)
               d[i]=c[i]+hh*d[i];           // yq=yn+h f(xn+h,yp)
            for (i=0; i<=n-1; i++)
               y[i]=(y[i]+d[i])/2.0;        // yn+1=(yp+yn)/2
        }
        p=0.0;
        for (i=0; i<=n-1; i++)
        {q=fabs(y[i]-b[i]);
           if (q>p) p=q;                    // 此处p为 max(y_ij^(h/2) - y_ij^(h))
        }
        hh=hh/2.0;                          // 取步长为原来的二分之一
 m=m+m;                                     // m 为原来的两倍
    }
   free(a); free(b); free(c); free(d);      // 释放存储空间
     return;                                // 函数返回
}
main()
{
    int i,j;
    double y[3],z[3][11],t,h,x,eps;
    y[0]=-1.0;                              // 初值 y0(0)=-1.0
    y[1]=0.0;                               // 初值 y1(0)=-1.0
    y[2]=1.0;                               // 初值 y2(0)=-1.0
    t=0.0;                                  // 起始点 t=0
    h=0.01;                                 // 步长为 0.01
    eps = 0.00001;                          // 精度为 0.00001
    Euler1(t,y,3,h,11,z);
    printf("定步长欧拉法结果: \n");
```

```
    for (i=0; i<=10; i++)
    { x=i*h;
      printf("t=%5.2f\t  ",x);   // 打印结果
      for(j=0; j<=2; j++)
        printf("y(%d)=%e  ",j,z[j][i]);
      printf("\n");
    }
    y[0]=-1.0;  y[1]=0.0; y[2]=1.0;     // 重新赋初值
    printf("变步长欧拉法结果: \n");
    printf("t=%5.2f\t  ",t);
    for (i=0; i<=2; i++)
      printf("y(%d)=%e  ",i,y[i]);
    printf("\n");
    for (j=1; j<=10; j++)
    {
      Euler2(t,h,y,3,eps);          // 调用变步长欧拉函数
      t=t+h;
      printf("t=%5.2f\t  ",t);   // 打印结果
      for (i=0; i<=2; i++) printf("y(%d)=%e  ",i,y[i]);
    }
}
```

程序执行结果:

定步长欧拉法结果:

```
t= 0.00   y(0)=-1.000000e+000  y(1)=0.000000e+000  y(2)=1.000000e+000
t= 0.01   y(0)=-9.999500e-001  y(1)=1.000000e-002  y(2)=9.900500e-001
t= 0.02   y(0)=-9.998000e-001  y(1)=1.999900e-002  y(2)=9.801990e-001
t= 0.03   y(0)=-9.995500e-001  y(1)=2.999600e-002  y(2)=9.704460e-001
t= 0.04   y(0)=-9.992001e-001  y(1)=3.999000e-002  y(2)=9.607901e-001
t= 0.05   y(0)=-9.987502e-001  y(1)=4.998000e-002  y(2)=9.512302e-001
t= 0.06   y(0)=-9.982005e-001  y(1)=5.996501e-002  y(2)=9.417655e-001
t= 0.07   y(0)=-9.975509e-001  y(1)=6.994401e-002  y(2)=9.323949e-001
t= 0.08   y(0)=-9.968016e-001  y(1)=7.991602e-002  y(2)=9.231176e-001
t= 0.09   y(0)=-9.959526e-001  y(1)=8.988004e-002  y(2)=9.139326e-001
t= 0.10   y(0)=-9.950040e-001  y(1)=9.983508e-002  y(2)=9.048389e-001
```

变步长欧拉法结果:

```
t= 0.00   y(0)=-9.950040e-001  y(1)=9.983508e-002  y(2)=9.048389e-001
t= 0.01   y(0)=-9.939559e-001  y(1)=1.097800e-001  y(2)=8.958356e-001
t= 0.02   y(0)=-9.928084e-001  y(1)=1.197140e-001  y(2)=8.869221e-001
t= 0.03   y(0)=-9.915616e-001  y(1)=1.296361e-001  y(2)=8.780972e-001
t= 0.04   y(0)=-9.902157e-001  y(1)=1.395453e-001  y(2)=8.693601e-001
t= 0.05   y(0)=-9.887707e-001  y(1)=1.494404e-001  y(2)=8.607100e-001
t= 0.06   y(0)=-9.872269e-001  y(1)=1.593207e-001  y(2)=8.521459e-001
t= 0.07   y(0)=-9.855843e-001  y(1)=1.691850e-001  y(2)=8.436671e-001
t= 0.08   y(0)=-9.838432e-001  y(1)=1.790324e-001  y(2)=8.352726e-001
```

```
t= 0.09    y(0)=-9.820037e-001  y(1)=1.888618e-001  y(2)=8.269616e-001
t= 0.10    y(0)=-9.800660e-001  y(1)=1.986724e-001  y(2)=8.187334e-001
```

程序分析

程序中函数 Euler1()实现定步长改进欧拉法求解微分方程。对于一阶微分方程组

$$\begin{cases} y_1' = f_1(t, y_1, y_2, \cdots y_m) & y_1(t_0) = y_{10} \\ y_2' = f_2(t, y_1, y_2, \cdots y_m) & y_2(t_0) = y_{20} \\ \cdots\cdots \\ y_m' = f_m(t, y_1, y_2, \cdots y_m) & y_m(t_0) = y_{m0} \end{cases}$$

已知 $t=t_{j-1}$ 点上的函数值 $y_{i,j-1}(i=1,2,\ldots,m)$，求 $t_j=t_{j-1}+h$ 点处的 $y_{ij}(i=1,2,\ldots,m)$。

函数通过三个 for 循环完成欧拉公式。具体如何用 C 语言实现者参见源程序和对应的注释。

$$\begin{cases} p_i = y_{i,j-1} + hf_i(t_{j-1}, y_{1,j-1}, \cdots y_{m,j-1}) \\ q_i = y_{i,j-1} + hf_i(t_{j-1}, p_1, \cdots, p_m) \\ yi_j = (1/2)(p_i + q_i) \end{cases}$$

函数 Euler2（）实现变步长改进欧拉法。基本方法跟定步长改进欧拉法大致相同，它根据改进欧拉公式以 h 为步长由 $y_{i,j-1}(i=1,2,\ldots,m)$ 计算 $y_{ij}^{(h)}$，再以 h/2 为步长，由 $y_{i,j-1}$ 跨两步计算 $y_{ij}^{(h/2)}$。

如果 $\max\limits_{1\le i\le m} | y_{ij}^{(h/2)} - yi_j^{(h)} | < \varepsilon$

则停止计算，取 $y_{ij}=y_{ij(h/2)}$，　　i=1,2,…,m

否则，将步长折半再进行计算。上述过程一直做到

$$\max_{1\le i\le m} | y_{ij}^{(h/2^k)} - yi_j^{(h/2^{k-1})} | < \varepsilon$$

为止，最后可以取 $y_{ij} \approx y_{ij}^{(h/2^k)}$，$i=1,2,\cdots,m$。

实例91 龙格－库塔法

实例说明

前面向读者介绍了用于解常微分方程的欧拉法和改进欧拉法，它们对应的局部截断误差为一阶 Taylor 余项 $O(h^2)$和二阶 Taylor 余项 $O(h^3)$。下面将向大家介绍一种精度更高的求解常微分方程的方法：龙格－库塔法。

本实例演示的是最常用四阶龙格－库塔法求解一阶常微分方程组的 C 语言实现方法。四阶龙格－库塔法的截断误差为四阶 Taylor 余项，显然比欧拉法和改进欧拉法精度高得多。

知识要点

回顾前面讲到的欧拉法，用一阶 Taylor 多项式近似函数得到欧拉公式，故其局部截断误差为一阶 Taylor 余项 $O(h^2)$。完全类似，用 p 阶 Taylor 公式展开，则局部截断误差应为 p 阶 Taylor 余项 $O(h^{p+1})$。由此可以得到启示：可以通过提高 Taylor 多项式的次数来提高算法的阶数以得到更高精度的数值算法。若直接对 y(x)用高次 Taylor 多项式近似，则因公式中出现 f 的各阶偏导数导致计算太复杂，计算量大而不实用。而龙格－库塔法（RK）很好地解决了这个问题。

一般的 RK 方法设近似公式为：

$$\begin{cases} y_{n+1} = y_n + h\sum_{i=1}^{p} c_i K_i \\ K_1 = f(x_n, y_n) \\ K_i = f(x_n + a_i h, y_n + h\sum_{j=1}^{i-1} b_{ij} K_j) \qquad (i = 2,3,...,p) \end{cases}$$

其中 a_i、b_{ij}、c_i 都是参数，确定它们的原则是使近似公式在（x_n,y_n）处的 Taylor 展开式与 y(x)在 x_n 处的 Taylor 展开式的前面的项尽可能多地重合，这样就使近似公式尽可能提高精度。

当 p=2 时，如果取 $c_1=c_2=1/2$,$a_2=b_{21}=1$，则近似公式为：

$$\begin{cases} y_{n+1} = y_n + h(K_1 + K_2)/2 \\ K_1 = f(x_n, y_n) \\ K_2 = f(x_n + h, y_n + hK_1) \end{cases}$$

这就是欧拉改进公式，它的局部截断误差为二阶 Taylor 余项 $O(h^3)$。

需要说明的是，当 p=1,2,3,4 时，RK 公式的最高阶数恰好是 p；当 p>4 时，RK 的最高阶数不是 p；如 p=5 时，RK 最高阶数仍为 4；p=6 时，RK 公式最高阶数为 5。对于大量实际问

题来说，一般四阶 RK 方法就可以达到精度要求。所有这里重点介绍四阶龙格－库塔公式。

程序源码

求下列微分方程组

$$\begin{cases} y_1' = y_2, & y_1(0) = -1.0 \\ y_2' = -y_1, & y_2(0) = 0.0 \\ y_3' = -y_3, & y_3(0) = 1.0 \end{cases}$$

取 h=0.01,11 个积分点（包括初值点）$t_i=(i-1)h$ (i=1,2,3...,11)。

该应用程序的源代码如下：

```
#include "stdio.h"
#include "stdlib.h"

void RKT(t,y,n,h,k,z)
int n;                  // 微分方程组中方程的个数，也是未知函数的个数
int k;                  // 积分的步数(包括起始点这一步)
double t;               // 积分的起始点 t0
double h;               // 积分的步长
double y[];      // 存放 n 个未知函数在起始点 t 处的函数值
double z[];      // 二维数组，体积为 n × k，返回 k 个积分点上的 n 个未知函数值
{
    extern void Func();                    // 声明要求解的微分方程组
    int i,j,l;
    double a[4],*b,*d;
    b=malloc(n*sizeof(double));      // 分配存储空间
    if(b == NULL)
    {
        printf("内存分配失败\n");
        exit(1);
    }
    d=malloc(n*sizeof(double));      // 分配存储空间
    if(d == NULL)
    {
        printf("内存分配失败\n");
        exit(1);
    }
     // 后面应用 RK4 公式中用到的系数 ai
    a[0]=h/2.0;
     a[1]=h/2.0;
    a[2]=h;
     a[3]=h;
    for(i=0; i<=n-1; i++)
```

```
        z[i*k]=y[i];                    // 将初值赋给数组 z 的相应位置
    for(l=1; l<=k-1; l++)
    {
        Func(y,d);
       for (i=0; i<=n-1; i++)
            b[i]=y[i];
       for (j=0; j<=2; j++)
       {
            for (i=0; i<=n-1; i++)
          {
               y[i]=z[i*k+l-1]+a[j]*d[i];
             b[i]=b[i]+a[j+1]*d[i]/3.0;
          }
          Func(y,d);
       }
       for(i=0; i<=n-1; i++)
         y[i]=b[i]+h*d[i]/6.0;
       for(i=0; i<=n-1; i++)
         z[i*k+l]=y[i];
       t=t+h;
    }
  free(b);              // 释放存储空间
    free(d);                // 释放存储空间
   return;
}
main()
{
    int i,j;
  double t,h,y[3],z[3][11];
  y[0]=-1.0;                            // 初始值 y1(0)=-1.0
    y[1]=0.0;                           // 初始值 y2(0)=0.0
    y[2]=1.0;                           // 初始值 y3(0)=1.0
  t=0.0;                                // 起始点为 t=0 处
    h=0.01;                             // 步长 h 为 0.01
  RKT(t,y,3,h,11,z);
  printf("\n");
  for (i=0; i<=10; i++)                 // 打印输出结果
  {
       t=i*h;
     printf("t=%5.2f\t   ",t);
     for (j=0; j<=2; j++)
       printf("y(%d)=%e  ",j,z[j][i]);
     printf("\n");
   }
}

void Func(y,d)              // 待求的常微分方程组
double y[],d[];
```

```
{
    d[0]=y[1];          // y0'=y1
    d[1]=-y[0];         // y1'=y0
    d[2]=-y[2];         // y2'=y2
    return;
}
```

程序的运行结果为：

```
t= 0.00   y(0)=-1.000000e+000  y(1)=0.000000e+000  y(2)=1.000000e+000
t= 0.01   y(0)=-9.999500e-001  y(1)=9.999833e-003  y(2)=9.900498e-001
t= 0.02   y(0)=-9.998000e-001  y(1)=1.999867e-002  y(2)=9.801987e-001
t= 0.03   y(0)=-9.995500e-001  y(1)=2.999550e-002  y(2)=9.704455e-001
t= 0.04   y(0)=-9.992001e-001  y(1)=3.998933e-002  y(2)=9.607894e-001
t= 0.05   y(0)=-9.987503e-001  y(1)=4.997917e-002  y(2)=9.512294e-001
t= 0.06   y(0)=-9.982005e-001  y(1)=5.996401e-002  y(2)=9.417645e-001
t= 0.07   y(0)=-9.975510e-001  y(1)=6.994285e-002  y(2)=9.323938e-001
t= 0.08   y(0)=-9.968017e-001  y(1)=7.991469e-002  y(2)=9.231163e-001
t= 0.09   y(0)=-9.959527e-001  y(1)=8.987855e-002  y(2)=9.139312e-001
t= 0.10   y(0)=-9.950042e-001  y(1)=9.983342e-002  y(2)=9.048374e-001
```

程序分析

该程序中，函数 RKT（）实现四阶龙格－库塔的算法，其实现的数学公式如下。设一阶微分方程组

$$\begin{cases} y_1' = f_1(t, y_1, y_2, \cdots, y_m), & y_1(t_0) = y_{10} \\ y_2' = f_2(t, y_1, y_2, \cdots, y_m), & y_2(t_0) = y_{20} \\ \cdots\cdots \\ y_m' = f_m(t, y_1, y_2, \cdots, y_m), & y_m(t_0) = y_{m0} \end{cases}$$

由 t_j 积分一步到 $t_{j+1}=t_j+h$ 的四阶龙格－库塔法的计算公式如下（式中 i=1,2,...,m）。

$$\begin{cases} K_{1i} = f_i(t_j, y_{1j}, y_{2j}, \cdots, y_{mj}) \\ K_{2i} = f_i(t_j + \frac{h}{2}, y_{1j} + \frac{h}{2}K_{11}, \cdots, y_{mj} + \frac{h}{2}K_{1m}) \\ K_{3i} = f_i(t_j + \frac{h}{2}, y_{1j} + \frac{h}{2}K_{21}, \cdots, y_{mj} + \frac{h}{2}K_{2m}) \\ K_{4i} = f_i(t_j + \frac{h}{2}, y_{1j} + \frac{h}{2}K_{31}, \cdots, y_{mj} + \frac{h}{2}K_{3m}) \\ y_{i,j+1} = yi_j + \frac{h}{6}(k_{1i} + 2K_{2i} + 2K_{3i} + k_{4i}) \end{cases}$$

具体如何用 C 语言实现上述算法，请参照源程序中的函数 RKT（）。

实例92 高斯消去法

实例说明

本实例将向大家介绍用高斯消去法求解线性方程组 AX=b 的 C 语言实现方法。

$$A=\begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & \cdots & a_{1n}^{(0)} \\ a_{21}^{(0)} & a_{22}^{(0)} & \cdots & a_{2n}^{(0)} \\ \cdots\cdots & & & \\ a_{n1}^{(0)} & a_{n2}^{(0)} & \cdots & a_{nn}^{(0)} \end{bmatrix} \qquad b=\begin{bmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \cdots \\ b_n^{(0)} \end{bmatrix}$$

知识要点

高斯消去法的基本思想是逐次消去一个未知数，把原来的方程化为等价的三角形方程组，这样，解就很容易求得。消去过程中，未知量是按其出现于方程式中的自然顺序消去的，用来消去其方程式中未知量的方程亦是按顺序选取的。

高斯消去过程分为两步，第一步是正消过程，目的是把 A 化为三角形矩阵。第二步是回代过程，目的是求方程的解。

正消的过程如下（要求 $a_{ii}\neq 0$）：

记 $A^{(0)}=A$，$b^{(0)}=b$，即

$a_{ij}^{(0)}=a_{ij}$，$b_i^{(0)}=b_i$，　　i,j=1,2,…,n

则高斯消去法的第一步是对增广矩阵$[A^{(0)}b^{(0)}]$，即

$$\begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & \cdots & a_{1n}^{(0)} & b_1^{(0)} \\ a_{21}^{(0)} & a_{22}^{(0)} & \cdots & a_{2n}^{(0)} & b_2^{(0)} \\ \cdots\cdots & & & & \\ a_{n1}^{(0)} & a_{n2}^{(0)} & \cdots & a_{nn}^{(0)} & b_n^{(0)} \end{bmatrix}$$

作如下变换

$a_{1j}^{(1)}=a_{1j}^{(0)}/a_{11}^{(0)}$　　(j=1,2,…,n)　　$b_1^{(1)}=b_1^{(0)}/a_{11}^{(0)}$

$a_{ij}^{(1)}=a_{ij}^{(0)}-a_{i1}^{(0)}a_{1j}^{(0)}$　　(i=2,…,n;j=1,2,…,n)

$b_i^{(0)}=b_i^{(0)}-a_{i1}^{(0)}b_1^{(0)}$　　(i=2,3,…,n)

即用第一行元素除以 $a_{11}^{(0)}$，然后用第 i(i=2,3,…,n)行元素减去第一行相应元素的 $a_{i1}^{(0)}$(i=2,…,n)倍，于是矩阵就化为

$$\begin{bmatrix} 1 & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ \cdots\cdots & & & & \\ 0 & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} & b_n^{(1)} \end{bmatrix}$$

同样对上面矩阵的第二行元素均除以 $a_{22}^{(1)}$后，再用第 i(i=3,4,…,n)行元素减去第二行相应元素的 $a_{i2}^{(1)}$倍(i=3,4,…,n)，即按下列公式计算

$a_{2j}^{(2)}=a_{2j}^{(1)}/a_{22}^{(1)}$　　(j=2,3,…,n)

$b_2^{(2)}=b_2^{(1)}/a_{22}^{(1)}$

$a_{ij}^{(2)}=a_{ij}^{(1)}-a_{i2}^{(1)}a_{2j}^{(2)}$　　(i=2,3,…,n; j=1,2,…,n)

$b_i^{(2)}=b_i^{(1)}-a_{i2}^{(1)}b_2^{(2)}$　　(i=3,…,n)

于是增广矩阵变为：

$$\begin{bmatrix} 1 & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ 0 & 1 & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ \cdots\cdots & & & & & \\ 0 & 0 & a_{n3}^{(2)} & \cdots & a_{nn}^{(2)} & b_n^{(2)} \end{bmatrix}$$

按上述消去过程继续下去，经过 n 次消去之后，矩阵便化为：

$$\begin{bmatrix} 1 & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1,n-1}^{(1)} & a_{1n}^{(1)} & b_1^{(1)} \\ 0 & 1 & a_{23}^{(2)} & \cdots & a_{2,n-1}^{(2)} & a_{2n}^{(2)} & b_2^{(2)} \\ 0 & 0 & 1 & \cdots & a_{3,n-1}^{(3)} & a_{2n}^{(2)} & b_2^{(2)} \\ \cdots\cdots & & & & & & \\ 0 & 0 & 0 & \cdots & 1 & a_{n-1,n}^{(n-1)} & b_n^{(2)} \\ 0 & 0 & 0 & \cdots & 0 & 1 & b_n^{(2)} \end{bmatrix}$$

于是，原矩阵最后化为一个单位上三角矩阵，即原方程组化为一个三角方程组，这样就可以通过回代过程求出方程组 AX=b 的解了。

程序源码

利用高斯消去法解方程组

$$\begin{bmatrix} 2 & 6 & -1 \\ 5 & -1 & 2 \\ -3 & -4 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -12 \\ 29 \\ 5 \end{bmatrix}$$

该应用程序的源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>

int GS(int,double**,double *,double);        // 高斯消去法求方程解
double **TwoArrayAlloc(int,int);             // 二维数组动态分配内存
void TwoArrayFree(double **);                // 释放二维数组分配的内存

void main()
{
    int i,n;
    double ep,**a,*b;
    n = 3;                    // 三个方程
    ep = 1e-4;                // 精度参数，如果数值比它小，便认为是零
    a = TwoArrayAlloc(n,n);                  // 动态分配一个 n×n 的二维数组
    b = (double *)calloc(n,sizeof(double));          // 动态分配一个一维数组
    if(b == NULL)
    {
        printf("内存分配失败\n");
        exit(1);
    }
// 按照给定命题初始化二维数组 a 和 b，并分别将其看作系数矩阵 A 和具体的 b
    a[0][0]= 2; a[0][1]= 6; a[0][2]=-1;
    a[1][0]= 5; a[1][1]=-1; a[1][2]= 2;
    a[2][0]=-3; a[2][1]=-4; a[2][2]= 1;
    b[0] = -12; b[1] = 29;  b[2] = 5;
    if(!GS(n,a,b,ep))
    {
        printf("不可以用高斯消去法求解\n");
        exit(0);
    }
    printf("该方程组的解为：\n");
    for(i=0;i<3;i++)                         // 打印结果
        printf("x%d = %.2f\n",i,b[i]);
    TwoArrayFree(a);              // 释放内存
    free(b);                      // 释放内存
}

int GS(n,a,b,ep)                  // 高斯消去法求解方程组
int n;              // 整型变量，输入参数，方程的阶数
double **a;     // 实型二维数组，输入参数，存放矩阵 A
double *b;      // 输入输出数组，输入时存放右端列向量 b，输出时存放解向量 x
```

```
double ep;              // 实型参数，输入参数，控制常数
{
    int i,j,k,l;
    double t;
    for(k=1;k<=n;k++)
    {
        for(l=k;l<=n;l++)                    // 判断上对角元素
            if(fabs(a[l-1][k-1])>ep)         // 判断 a_lk 是否为零
                break;
    // 如果可以执行 break 下面的语句，说明 a_lk 为矩阵 A 第 k 列第一个不为零的元素
            else if(l==n)          // 如果相等，说明为奇异矩阵，方程组无解
    return(0);
        if(l!=k)                   // 此时必定有 l>k，那么此时的 a_kk=0
        {//交换 a_k 和 a_l 行，保证对角元素不为零
            for(j=k;j<=n;j++)
            {
                t = a[k-1][j-1];                    //t=a_j_k
                a[k-1][j-1]=a[l-1][j-1];            //a_j_k=a_lj
                a[l-1][j-1]=t;                      //a_lj=t
            }
        // 交换系数矩阵的第 k 行和第 l 行的同时，交换矩阵 b 相应行的数据
            t=b[k-1];                       //t=b_k
            b[k-1]=b[l-1];                  //b_k=b_l
            b[l-1]=t;                       //b_l=t
        }
        t=1/a[k-1][k-1];                    //t=1/a_k_k

        // 下面的 for 循环将矩阵 A 和 b 的第 k 行都乘以 1/a_kk，即实现算法：
a_k_j^(k)=a_k_j^(k-1)/a_k_k^(k-1)                    (j=k,k+1,…,n)
                    b_k^(k)=b_k^(k-1)/a_k_k^(k-1)
        for(j=k+1;j<=n;j++)
            a[k-1][j-1]=t*a[k-1][j-1];
        b[k-1]*=t;

// 下面的 for 循环用第 i 行元素（i=k+1,…,n）减去第 k 行对应元素的 a_ik 倍（i=k+1,…,n）实现
//算法：
a_i_j^(k)=a_i_j^(k-1)-a_i_k^(k-1)a_k_j^(k)    (i=k+1,…,n;j=k,…,n)
            b_i^(k)=b_i^(k-1)-a_i_k^(k-1)b_k^(k)    (i=k+1,…,n)
        for(i=k+1;i<=n;i++)
        {
            for(j=k+1;j<=n;j++)
                a[i-1][j-1]-=a[i-1][k-1]*a[k-1][j-1];
            b[i-1]-=a[i-1][k-1]*b[k-1];
        }
```

```
    }
// 下面的 for 循环为回代过程求得 x_i 的值
    for(i=n-1;i>=1;i--)
        for(j=i+1;j<=n;j++)
            b[i-1]-=a[i-1][j-1]*b[j-1];      // 将求得的 x_i 值存放在 b_i 中
return(1);
}

double **TwoArrayAlloc(int r,int c)      // 创建二维动态数组
{
    double *x,**y;
    int n;
    x=(double *)calloc(r*c,sizeof(double));
    y=(double **)calloc(r,sizeof(double*));
    for(n=0;n<=r-1;++n)
        y[n]=&x[c*n];
    return (y);
}
void TwoArrayFree(double **x)          // 释放建立的二维动态数组
{
    free(x[0]); free(x);
}
```

程序分析

程序通过函数 GS()实现高斯消去法。高斯消去法的整个计算过程如下：

第一步：正消过程

依次按 k=1,2,…,n 进行下列计算

$a_{kj}^{(k)}=a_{kj}^{(k-1)}/a_{kk}^{(k-1)}$ (j=k,k+1,…,n)

$b_k^{(k)}=b_k^{(k-1)}/a_{kk}^{(k-1)}$

$a_{ij}^{(k)}=a_{ij}^{(k-1)}-a_{ik}^{(k-1)}a_{kj}^{(k)}$ (i=k+1,…,n;j=k,…,n)

$b_i^{(k)}=b_i^{(k-1)}-a_{ik}^{(k-1)}b_k^{(k)}$ (i=k+1,…,n)

第二步：回代过程

$x_n=b_n$

$x_i=b_i^{(i)}-\sum_{j=i+1}^{n}a_{ij}^{(i)}x_j$ (i=n-1,n-2,…,1)

函数 GS()如何实现上述算法，请参看源程序。需要说明的是，对于二维动态数组，C 语言并没有给出相应的函数，这里编写函数 TwoArrayAlloc(int,int)和 TwoArrayFree(double **)来实现二维动态数组的分配和释放。

实例93 正定矩阵求逆

实例说明

本实例演示的是用来求实对称正定矩阵逆矩阵的C语言实现方法。设A为n×n对称正定矩阵，其关系式y=Ax确定了R^n上的一个映像，如果能求出逆关系x=By，则得到A的逆矩阵$B=A^{-1}$。这就是下面要介绍的实对称矩阵求逆的基本思路。

该实例求逆过程通过函数GJ(int n,double **a)完成。

知识要点

下面将求实对称正定矩阵的逆矩阵的方法作一个简要的介绍。

将矩阵y=Ax详细写出，如下：

$$\begin{cases} y_1 = a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \\ y_2 = a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \\ \ldots\ldots\ldots\ldots \\ y_n = a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n \end{cases} \tag{1}$$

由于A正定，必有$a_{11}>0$，故可以从上式中第一个等式解出x_1（称为交换x_1和y_1的位置），再将x_1代入其他等式，得到新的关系式：

$$\begin{cases} x_1 = a'_{11}y_1 + a'_{12}x_2 + \ldots + a'_{1n}x_n \\ y_2 = a'_{21}x_1 + a'_{22}x_2 + \ldots + a''_{2n}x_n \\ \ldots\ldots\ldots\ldots \\ y_n = a'_{n1}x_1 + a'_{n2}x_2 + \ldots + a'_{nn}x_n \end{cases} \tag{2}$$

新系数的计算公式如下：

$$\begin{cases} a'_{11} = 1/a_{11} \\ a'_{1j} = -a_{1j}/a_{11} & j = 2,3,\ldots,n \\ a'_{j1} = a_{i1}/a_{11} & i = 2,3,\ldots,n \\ a'_{ij} = a_{ij} - a_{i1}a_{1j}/a_{11} & i,j = 2,3,\ldots,n \end{cases} \tag{3}$$

用同样的方法交换x2和y2的位置，如此继续下去，最后可得x=By。B就是所求矩阵A的逆矩阵A^{-1}。

程序源码

求下列矩阵的逆矩阵：

$$A=\begin{bmatrix}5 & 7 & 6 & 5\\ 7 & 10 & 8 & 7\\ 6 & 8 & 10 & 9\\ 5 & 7 & 9 & 10\end{bmatrix}$$

该应用程序的源代码如下：

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

int GJ(int,double **);                    // 对正定矩阵求逆函数
double **TwoArrayAlloc(int,int);          // 二维动态数组分配内存
void TwoArrayFree(double **);             // 二维动态数组释放内存

void main()
{
    int i,j,n;
    double **a;
    n=4;                                  // 矩阵的维数
    a=TwoArrayAlloc(n,n);                 // 二维动态数组 a 分配内存
// 矩阵存放到二维数组 a[n][n]中
    a[0][0]=5;  a[0][1]=7;  a[0][2]=6;  a[0][3]=5;
    a[1][0]=7;  a[1][1]=10; a[1][2]=8;  a[1][3]=7;
    a[2][0]=6;  a[2][1]=8;  a[2][2]=10; a[2][3]=9;
    a[3][0]=5;  a[3][1]=7;  a[3][2]=9;  a[3][3]=10;
    if(!GJ(n,a))
    {
        printf("矩阵求逆失败\n");
        exit(1);
    }
    printf("该矩阵的逆为: \n");
    for(i=0;i<n;i++)
    {// 打印数组 a[n][n],即要求矩阵的逆矩阵
        for(j=0;j<n;j++)
            printf("%.2f\t",a[i][j]);
        printf("\n");
    }
}

int GJ(int n,double **a)                  // 实对称正定矩阵求逆函数
```

```
{
    int i,j,k;
    double p,q,*h;
    h=(double *)calloc(n,sizeof(double));        // 动态分配内存
    if(h == NULL)
    {
        printf("内存分配失败\n");
        exit(1);
    }
    for(k=n;k>=1;k--)
    {
        p=a[0][0];
        if(p<=0)                                 // 如果不满足，则该矩阵不是正定矩阵
        {
            free(h);
            return (0);
        }
        for(i=2;i<=n;i++)
        {
            q=a[i-1][0];
            if(i>k)
                h[i-1]=q/p;
            else
                h[i-1]=-q/p;
            for(j=2;j<=i;j++)
                a[i-2][j-2]=a[i-1][j-1]+q*h[j-1];
        }
        a[n-1][n-1]=1/p;
        for(i=2;i<=n;i++)
            a[n-1][i-2]=h[i-1];
    }
    free(h);
    return(1);
}

double **TwoArrayAlloc(int r,int c)          // 二维动态数组分配内存
{
    double *x,**y;
    int n;
    x=(double *)calloc(r*c,sizeof(double));
    y=(double **)calloc(r,sizeof(double*));
    for(n=0;n<=r-1;++n)
        y[n]=&x[c*n];
    return (y);
}
```

```
void TwoArrayFree(double **x)                    // 二维动态数组释放内存
{
    free(x[0]);
    free(x);
}
```

程序执行结果为：

该矩阵的逆为：

```
68.00   7.00    6.00    5.00
-41.00  25.00   8.00    7.00
-17.00  10.00   5.00    9.00
10.00   -6.00   -3.00   2.00
```

程序分析

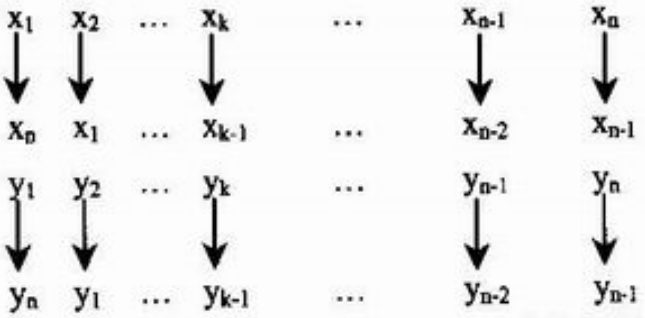
如果按照知识要点中介绍的方法编写程序，那么程序将非常冗长。为此，本实例的设计采用“变量循环重新编号法”，使每一步交换都在x1和y1的位置上进行。于是类似于式（3）的计算公式适用于每一个计算步骤，其计算公式如下：

$$\begin{cases} a'_{nn} = 1/a_{11} \\ a'_{n,j-1} = -a_{1j}/a_{11} & j = 2,3,...,n \\ a'_{i-1,1} = a_{i1}/a_{11} & i = 2,3,...,n \\ a'_{i-1,j-1} = a_{ij} - a_{i1}a_{1j}/a_{11} & i,j = 2,3,...,n \end{cases} \tag{4}$$

事实上，对（2）式稍加修改可得

$$\begin{cases} y_2 = a'_{22}x_2 + a'_{23}x_3 + ... + a'_{2n}x_n + a'_{21}y_1 \\ y_3 = a'_{32}x_2 + a'_{23}x_3 + ... + a'_{3n}x_n + a'_{31}y_1 \\ ..... \\ y_n = a'_{n2}x_2 + a'_{n3}x_3 + ... + a'_{nn}x_n + a'_{n1}y_1 \\ x_1 = a'_{12}x_2 + a'_{13}x_3 + ... + a'_{1n}x_n + a'_{11}y_1 \end{cases} \tag{5}$$

显然对（5）式中的变量按照下列规则重新编号：



可得到对于每一步均使用的变化公式（4），其中变量的顺序经过n次变化后恢复原状。具体如何用C语言实现上述算法，请结合注释参见程序源码。

第四篇
综合应用篇

相信经过前三篇的学习，读者已经掌握了C语言编程的基本技能，并拥有了一定的独立编程能力。作为提高，在这一篇中我们精心选择了几个规模较大的编程实例，它们分别是遗传算法、神经网络、聚类、快速傅利叶变换以及专家系统等算法的C语言实现。选择这几个实例的原因是它们基本涵盖了本书中除第三篇以外的全部知识点，例如结构、指针、链表、堆栈、队列、位操作等，并且在实例中补充了一些常用而又未在前几篇中介绍的函数。同时为进一步提高编程能力，在这一篇的几个实例中将进一步深入讲解动态内存分配、文件操作、数据结构的实现等知识点，并加入了矩阵操作、复杂逻辑结构的实现以及读程序流程图实现算法等内容。这些算法本身在工业控制、交通规划、人工智能、图像处理与信号系统以及经济等领域中得到了广泛的应用，有很强的实际意义。希望通过对上述实例的学习能够复习前面介绍过的基本概念以及算法，并且经过编程初步掌握这些在实际应用中较为常见的算法。在给出每个编程实例前我们都将对算法做较为详细的讲解，建议读者能够根据讲解利用前几篇所学知识尝试独立编程，再与作者所给实例加以比较，这样将更有利于提高编程技能。

另外需要指出的是，虽然这里给出的都是这些算法最原始的实现，在实际应用中人们常常根据需要加以改动，例如遗传算法的变异速率在所给出的原始算法中都是固定的，而在工程应用中常常会根据实际需要用专门的函数对其取值加以控制，因而会显得更加复杂，但是万变不离其宗，基本的思路并不会改变。在掌握本篇所介绍的内容后，读者可根据需要，结合在前几篇所学的知识实现自己的算法。

实例94 用C语言实现遗传算法

实例说明

本实例将通过对多项式求最小值来介绍遗传算法，并结合算法的实现介绍 srand( )、rand( ) 这两个较为常用的、与随机运算相关的函数，同时复习前几章学过的链表、排序、文件操作、位操作等内容，希望读者能认真体会。

先简要介绍用遗传算法求解的主要步骤，使读者掌握算法的基本思想；然后将算法实例化，借助一个典型例子讲解如何用 C 语言实现遗传算法，以及应用遗传算法求解的全过程，使读者具备独立实现简单遗传算法的能力；最后对实例的求解结果进行分析，令读者更好地掌握这种常用的算法。

知识要点

遗传算法（GA ,Genetic Algorithm）作为一种快速而有效的寻优算法，在工业控制、经济决策、交通规划等诸多领域得到了广泛的应用。它基于群体遗传演化机制，由美国学者 Holland 于 1975 年首先提出。遗传算法摒弃了传统的搜索寻优方式，模拟自然界生物进化过程，对目标空间（即解所在的空间）进行随机化搜索。它将目标空间中的每一个可能解看作是群体的一个染色体，并将每一个染色体编码成为符号串形式（即由一组基因组成），模拟达尔文的遗传选择和自然淘汰的生物进化过程，对群体内的染色体以基因为单位反复进行基于遗传学的操作（包括遗传，交叉和变异）。根据预定的适应度函数对染色体进行评价，依据适者生存，优胜劣汰的进化规则，不断得到更优的群体，以求得满足要求的最优解。在这个过程中染色体的优劣直接由组成它的基因串决定，那些能使染色体“优胜”的基因由于染色体得到了更大的生存机会而保留下来，而那些不好的基因也会因为染色体的“劣汰”而消失，从而在一定的进化过程后整个群体的适应度将得到提升，而个别染色体也将在此过程中达到最优，当满足求解的要求时这个最优染色体就成为了问题的解。

- 遗传算法中的基本概念：

基因：组成染色体的单元，可以表示为一个二进制位，一个整数或一个字符等。

染色体：表示待求解问题的一个可能解，由若干基因组成，是 GA 操作的基本对象。

个体：在本节中个体等同于染色体，从基因角度而言称为染色体，从适应度函数角度称为个体。

群体：一定数量的染色体组成了群体，表示 GA 的遗传搜索空间。

适应度：代表一个染色体所对应解的优劣，通常由某一适应度函数表示。

➢ 遗传算法中的基本操作：

选择：根据个体的适应度，在群体中按照一定的概率选择可以作为父本的个体，选择依据是适应度大的个体被选中的概率高。选择操作体现了适者生存、优胜劣汰的进化规则。

交叉：将父本染色体的基因按照一定概率随机地交换，形成新的染色体。

变异：按一定概率改变染色体内的某个基因值。

遗传算法包含以下的主要处理步骤：首先对优化问题的解进行编码，即用染色体表示优化问题的可能解，组成染色体的元素称为基因。编码的目的主要是用于问题解的表现和利于遗传算法中的其他操作；然后是适应度函数的构造和应用。适应度函数因优化问题的目标函数而定，当适应度函数确定以后，选择定律将以各染色体所对应的适应度函数值的大小决定哪些染色体适应生存，哪些被淘汰。生存下来的染色体组成种群，形成一个可以繁衍下一代的群体。此后是染色体的结合，即交叉。双亲遗传的结合通过编码之间的交叉（crossover）实现下一代的产生。新的染色体产生后将以一定的概率发生基因变异，变异使解有更大的遍历性，防止过早收敛于局部极小值。

在下面的程序中将要运用遗传算法对一个多项式求最小值。

程序解析

需要求解的多项式的表达式如下：

$y = x^6 - 10x^5 - 26x^4 + 344x^3 + 193x^2 - 1846x - 1680$

要求在（ -8 , +8 ）间寻找使表达式达到最小值的 x，误差为 0.001。

➢ 问题分析：

编码：采用常规码，即二进制码编码。采用这种编码方式的优点是算法构造比较简单，交叉、变异的实现非常容易，同时解的表示也很简洁、直观。可以每隔 0.001 取一个点，这样理论误差将小于 0.0005，可以满足题目中的误差要求。此时总的求解空间为：

N = （ 8 - （ - 8 ）） * 1000 = 160000

可以用 n = 14 位二进制数来表示。

群体规模 m：群体的规模 m 可选为 n 和 2n 之间的一个确定数，这里选择 m = 20。

初始种群的选取：大多数学者认为初始群体应该随机选取，只有随机选取才能达到所有状态的遍历，保证所求得的解为全局最优解。在这里初始种群将在值域范围内随机选取。

终止规则：本例中进化过程将在两种情况下终止：①最优解在连续的 20 次循环中改变量小于 0.01，此时认为这个最优解为满足题目要求的最优解，求解成功，退出程序；②总的循环次数大于 1200 次时循环也将结束，这种情况按求解失败处理。

交叉规则：遗传算法中，交叉规则较多，这里只采用最常用的双亲双子法。这种方法是在双亲确定后产生一个随机位，将双亲在随机位后的所有基因对换，形成两个后代。双亲双子法简单的示例图如下：

```
          交叉位                  交叉位
            |                       |
父代A   100|100          子代A   100|010
                  ——→
父代B   010|010          子代B   010|100
```

选择：在进行交叉、变异后，种群中的个体个数达到2m个，将这2m个染色体按其适应度进行排序，保留最优的m个，淘汰其他的，使种群在整体上得到进化。

➢ 源程序及讲解

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#define SUM 20                  // 定义群体的染色体数量
#define MAXloop 1200            // 最大循环次数
#define error 0.01              // 若两次最优值之差小于此数则认为结果没有改变
#define crossp 0.7              // 交叉概率，所选中的双亲按此概率进行交叉
#define mp 0.04                 // 变异概率
struct gen                      // 定义染色体结构
{
    int info ;                  // 染色体结构，用一整型数的后14位作为染色体编码
    float suitability ;         // 此染色体所对应的适应度函数值，即表达式的值
} ;
struct gen gen_group[ SUM ] ; // 定义含有20个染色体的种群
struct gen gen_new[ SUM ] ;  // 定义含有20个染色体的种群，记录交叉产生的子代染色体
struct gen gen_result ;       // 记录上一轮循环中得到的最优的染色体
int result_unchange_time ;    // 当相邻两轮得到的最优值对应的适应度之差小于error时
//其值增1，反之清零
struct log                  // 形成链表，记录每次循环所产生的最优的适应度
{
    float suitability ;
    struct log * next ;
}llog , * head , * end ;
int log_num ;                 // 链表长度
////////////////////// 下面是函数声明
void initiate ( ) ;             // 初始化函数，主要负责产生初始种群
void evaluation ( int flag ) ;  // 评估种群中各染色体的适应度，并据此进行排序
void cross ( ) ;                // 交叉函数
void selection ( ) ;            // 选择函数
int record ( ) ;                // 记录每次循环所产生的最优解并判断循环是否终止
void mutation ( ) ;             // 变异函数
void showresult ( int ) ;       // 显示结果
```

```
//////////////////////// 以上函数由主函数直接调用
int  randsign ( float p ) ;        // 按概率 p 产生随机数 0、1，其值为 1 的概率为 p
int  randbit ( int i , int j ) ;      // 随机产生一个在 i、j 两个数之间的整数
int  randnum ( ) ;              // 随机产生一个由 14 个基因组成的染色体
int  createmask ( int a ) ;        // 用于交叉操作
int  convertionD2B ( float x ) ; // 对现实解空间的可能解 x 进行二进制编码（染色体形式）
float  convertionB2D ( int x ) ;  // 将二进制编码 x 转化为现实解空间的值
void  main ( )
{
    int  i , flag ;
    flag = 0 ;
    initiate ( ) ;                // 产生初始化种群
   evaluation ( 0 ) ;             // 对初始化种群进行评估、排序
for ( i = 0 ; i < MAXloop ; i++ )
    {// 进入进化循环，当循环次数超过 MAXLoop 所规定的值时终止循环，flag 值保持为 0
        cross ( ) ;               // 进行交叉操作
        evaluation ( 1 ) ;        // 对子种群进行评估、排序
        selection ( ) ;           // 从父、子种群中选择最优的 NUM 个作为新的父种群
        if ( record ( ) == 1 )    // 如果满足终止规则 1 时将 flag 置 1，停止循环
        {
            flag = 1 ;
            break ;
        }
        mutation ( ) ;            // 进行变异操作
    }
    showresult ( flag ) ;         // 按 flag 值显示寻优结果
}
void  initiate ( )
{
    int  i , stime ;
    long  ltime ;
    ltime = time ( NULL ) ;
    stime = ( unsigned ) ltime / 2 ;
    srand ( stime ) ;                              // 这一部分将在 randnum 中讲解
    for ( i = 0 ; i < SUM ; i++ )
        gen_group[ i ].info = randnum ( ) ;        // 调用 randnum ( ) 函数建立初
始种群
    gen_result.suitability = 1000 ;
    result_unchange_time = 0 ;
    head = end = ( struct log * ) malloc ( sizeof ( llog ) ) ;// 初始化 log 链
表
    if ( head == NULL )
    {
        printf ( "\n 内存不够! \n" ) ;
```

```
        exit ( 0 ) ;
    }
    end->next = NULL ;
    log_num = 1 ;
}
```

本函数实现三个功能：①初始化随机序列；②建立规模为 SUM 的初始种群；③建立 log 链表头，其中指针 head 记录链表头的位置，end 始终指向链表的尾部，具体的链表操作请参考第二篇的讲解。建立这个链表的目的是使读者能够更加清晰地观察搜索的收敛过程，并不是遗传算法的必须步骤。

```
void evaluation ( int flag )
{// 按照 flag 的指示分别对父种群、子种群进行评估、排序
    int i , j ;
    struct gen * genp ;
    int gentinfo ;
    float gentsuitability ;
    float x ;
    if ( flag == 0 )                          // 当 flag 值为 0 时对父种群进行操作
        genp = gen_group ;
    else genp = gen_new ;                     // 反之对子种群进行操作
    for ( i = 0 ; i < SUM ; i++ )             // 计算各染色体对应的适应度
    {
        x = convertionB2D ( genp[ i ].info ) ;
    genp[ i ].suitability = x * ( x * ( x * ( x * ( x * ( x - 10 ) - 26 ) +
344 ) + 193 ) - 1846 ) - 1680 ;
    }
    for ( i = 0 ; i < SUM - 1 ; i++ )  // 按照适应度值的大小进行排序
    {
    for ( j = i + 1 ; j < SUM ; j++ )
       {
    if ( genp[ i ].suitability > genp[ j ].suitability )
            {
                gentinfo = genp[ i ].info ;
                genp[ i ].info = genp[ j ].info ;
                genp[ j ].info = gentinfo ;
                gentsuitability = genp[ i ].suitability ;
                genp[ i ].suitability = genp[ j ].suitability ;
                genp[ j ].suitability = gentsuitability ;
            }
}
}
}
```

需要注意的是，在计算适应度的值时并没有调用原始的多项式，而是将其改写为：

$$x*(x*(x*(x*(x*(x-10)-26)+344)+193)-1846)-1680$$

这是在数值计算中为了加快计算速度而经常采用的方法。对于计算机而言进行一次乘法操作所占用的时间要远大于进行一次加、减法操作，因此乘、除法的操作次数对算法的运行时间有很大的影响。当采用原始的表达式时需要对 x^6、x^5、x^4、x^3、x^2 分别进行计算，再加上与参数进行的乘法计算，总共需要进行 20 次乘法操作，而改写后的多项式只需进行 5 次乘法操作，对于遗传算法这样重复计算量非常大的程序而言将节省大量的运行时间。

```
void cross ( )
{                              // 对父种群按概率 crossp 做交叉操作
    int i , j , k ;
    int mask1 , mask2 ;
    int a[ SUM ] ;
    for ( i = 0 ; i < SUM ; i++ ) a[ i ] = 0 ;
    k = 0 ;
    for ( i = 0 ; i < SUM ; i++ )
    {
        if ( a[ i ] == 0 )
        {
            for ( ; ; )              // 随机找到一组未进行过交叉的染色体与 a[i]交叉
            {
                j = randbit ( i + 1 , SUM - 1 ) ;
                if ( a[ j ] == 0 ) break ;
            }
            if ( randsign ( crossp ) == 1 )
            {
                mask1 = createmask ( randbit ( 0 , 14 ) ) ;
                mask2 = ~ mask1 ;
                gen_new[ k ].info = ( gen_group[ i ].info ) & mask1 +
(gen_group[j].info) & mask2 ;
                gen_new[ k + 1 ].info = ( gen_group[ i ].info) & mask2 +
(gen_group[ j ].info) & mask1 ;
                k = k + 2 ;
            }
            else
            {
                gen_new[ k ].info = gen_group[ i ].info ;
                gen_new[ k + 1 ].info = gen_group[ j ].info ;
                k = k + 2 ;
            }
            a[ i ] = a[ j ] = 1 ;
        }
    }
;
```

cross ( )函数主要由三部分组成，①随机选择将要进行交叉的染色体对，保证种群中的每一个染色体都有唯一一次交叉的机会；②按 crossp 的概率对选择的染色体对进行交叉操作；③未进行交叉的染色体直接复制作为子染色体。在第二部分首先由 ranbit()选择交叉位，

createmask()返回一个交叉位以右皆为 1 的二进制数，赋给 mask1；mask2 为交叉位以左皆为 1 的二进制数，它们分别与父染色体 gen_group[i].info、gen_group[j].info 进行与操作，所得的结果的和即为交叉的结果。

```
void selection( )
{                                   // 从父代种群和子代种群中选择最优的染色体组成新的种群
    int i , j , k ;
    j = 0 ;
    i = SUM / 2 - 1 ;
    if ( gen_group[ i ].suitability < gen_new[ i ].suitability )
    {
        for ( j = 1 ; j < SUM / 2 ; j++ )
        {
            if ( gen_group[ i + j ].suitability > gen_new[ i - j ].suitability)
                break ;
        }
    }
    else
        if ( gen_group[ i ].suitability > gen_new[ i ].suitability )
        {
            for ( j = -1 ; j > -SUM / 2 ; j-- )
            {
                if(gen_group[i+j].suitability<=gen_new[i-j].suitability)
                    break ;
            }
        }
    for ( k = j ; k < SUM / 2 + 1 ; k++ )
    {
        gen_group[ i + k ].info = gen_new[ i - k ].info ;
        gen_group[ i + k ].suitability = gen_new[ i - k ].suitability ;
    }
}
```

selection()函数从父代种群（gen_group）和子代种群（new_group）中选择最优的染色体组成新的父代种群。由于两个种群都已进行了排序，因此只需找到子代种群中的第 t 个个体使其适应度函数值满足下列条件：大于父代中群中的第 NUM-t 个个体，并小于父代种群中的第 NUM-t+1 个个体，然后用子代种群中的前 t 个个体代替父代种群中的后 t 个个体即可。为了加快对 t 的寻找速度，从两个种群的中部开始比较，这种方法类似于二分法。

```
int record ( )
{
    float x ;
    struct log * r ;
    r = ( struct log * ) malloc ( sizeof ( llog ) ) ;
    if ( r == NULL )
```

```
    {
        printf ( "\n 内存不够! \n" ) ;
        exit ( 0 ) ;
    }
    r->next = NULL ;
    end->suitability = gen_group[ 0 ].suitability ;// 记录各轮循环中的最优值
    end->next = r ;
    end = r ;
    log_num++ ;
    //// 下面部分判断是否满足循环停止条件 1
    x = gen_result.suitability - gen_group[ 0 ].suitability ;
    if ( x < 0 )  x = -x ;
    if ( x < error )
    {
        result_unchange_time++ ;
        if ( result_unchange_time >= 20 )  return 1 ;
    }
    else
    {
        gen_result.info = gen_group[ 0 ].info ;
        gen_result.suitability = gen_group[ 0 ].suitability ;
        result_unchange_time = 0 ;
    }
    return  0 ;
}
```

record()函数主要完成两个任务，一是用链表记录各轮循环中的最优值，二是判断是否满足循环停止条件，当两次循环的适应度值的差小于 error 时 result_unchange_time 的值加 1，若 result_unchange_time 达到 20 时表示已经搜索到最优值，返回 1；若适应度的差大于、等于 error 则将 result_unchange_time 清零，并记录本轮最优值，函数返回 0。

```
void  mutation ( )
{
    int  i , j , m ;
    int  gentinfo
    float  x ;
    float  cmp ;
    float  gentsuitability ;
    cmp = 1 - pow ( 1 - mp , 11 ) ; // 基因变异概率为 mp 时染色体中有基因发生变异的
                                    // 概率
    for ( i = 0 ; i < SUM ; i++ )
    {
        if ( randsign ( cmp ) == 1 )
        {
            j = randbit ( 0 , 14 ) ;
```

```
                m = 1 << j ;
                gen_group[ i ].info = gen_group[ i ].info ^ m ;
                x = convertionB2D ( gen_group[ i ].info ) ;
    x = x * ( x * ( x * ( x * ( x * ( x - 10 ) - 26 ) + 344 ) + 193 ) - 1846 ) -
1680 ;
    gen_group[ i ].suitability = x ;
            }
        }
        for ( i = 0 ; i < SUM - 1 ; i++ )
        {
            for ( j = i + 1 ; j < SUM ; j++ )
            {
                if ( gen_group[ i ].suitability > gen_group[ j ].suitability )
                {
                    gentinfo = gen_group[ i ].info ;
                    gen_group[ i ].info = gen_group[ j ].info ;
                    gen_group[ j ].info = gentinfo ;
                    gentsuitability = gen_group[ i ].suitability ;
                    gen_group[ i ].suitability = gen_group[ j ].suitability ;
                    gen_group[ j ].suitability = gentsuitability ;
                }
            }
        }
    }
```

为了提高执行速度，在进行变异操作时并没有直接确定需要进行变异的位，而是先以 cmp 概率确定将要发生变异的染色体，再从这个染色体中随机选取一个基因进行变异。另外，由于进行选择和变异后父代种群的次序已被打乱，因此在进行变异后对种群进行了一次排序。

```
    void  showresult ( int flag )
    {// 显示搜索结果并释放内存
        int  i , j ;
        struct  log  * logfree ;
        FILE  * logf ;
        if ( flag == 0 )
            printf ( "已到最大搜索次数，搜索失败！" ) ;
        else
        {
            printf ( "当取值%f 时表达式达到最小值为%f\n" ,
    convertionB2D ( gen_result.info ) , gen_result.suitability ) ;
            printf ( "收敛过程记录于文件 log.txt" ) ;
            if ( ( logf = fopen ( "log.txt" , "w+" ) ) == NULL )
            {
                printf ( "Cannot create/open file" ) ;
                exit ( 1 ) ;
            }
```

```
            for ( i = 0 ; i < log_num ; i = i + 5 ) // 对收敛过程进行显示
            {
                for ( j = 0 ; ( j < 5 ) & ( ( i + j ) < log_num-1 ) ; j++ )
                {
                    fprintf ( logf , "%20f" , logprint->suitability ) ;
                    logprint = logprint->next ;
                }
                fprintf ( logf , "\n \n" ) ;
            }
        }
        for ( i = 0 ; i< log_num ; i++ )          //释放内存
        {
            logfree = head ;
            head = head->next ;
            free ( logfree ) ;
            fclose ( logf ) ;
        }
    getchar ( ) ;
    }
```

若搜索成功则显示搜索的结果和对应的最小值，并将搜索过程各轮循环中的最优解对应的最小值写入文本文件 log.txt。若搜索失败则显示失败信息，最后释放链表所占用的内存。

```
int  randsign ( float p )
{                          // 按概率 p 返回 1
    if ( rand ( ) > ( p * 32768 ) )
        return  0 ;
    else  return  1 ;
}
```

rand()函数将按伪随机序列返回一个 0～32767 的整型数，在 initiate()中调用了 void srand (unsigned int seed)函数，目的是为 rand()生成的伪随机数序列设置起点，srand()一般用于多道程序运行时通过制定不同的起点而使用不同的伪随机数序列。在 initiate()中采用系统时间作为种子（seed），以便在每次程序执行时都会使用不同的伪随机序列，可尝试将这一部分去掉，多次运行程序，对比观察各次产生的 log.txt 记录，就会发现每次运行的结果都是一样的。

```
int  randbit (int i , int j )     //产生在 i 与 j 之间的一个随机数
{
    int a , l ;
    l = j - i + 1 ;
    a = i + rand ( ) * l / 32768 ;
    return  a ;
}
int  randnum ( )
{
    int  x ;
```

```
    x = rand ( ) / 2 ;
    return x ;
}
float convertionB2D ( int x )
{                       // 对二进制的染色体编码进行译码，取值范围为 (-8.192, 8.191)
    float y ;
    y = x ;
    y = ( y - 8192 ) / 1000 ;
    return y ;
}
int convertionD2B ( float x )
{                       // 将可能解编码成为二进制序列
    int g ;
    g = ( x * 1000 ) + 8192 ;
    return g ;
}
int createmask ( int a )
{
    int mask ;
    mask = ( 1 << ( a + 1 ) ) - 1 ;
    return mask ;
}
```

总结分析

观察 log.txt 就会发现，函数每次运行所得到的收敛路径都是不同的，而且有时并未收敛到最优解，或者搜索失败，这是因为遗传算法本身是一种随机化搜索算法，它的收敛路径受到初始群体的选择、交叉位的选择以及变异等随机化因素的影响。现在已经证明遗传算法可以从理论上收敛到全局最优解，同时可以通过适当的选择参数，例如变异概率、初始种群的个数等加快遗传算法的收敛速度，读者可以尝试改变 SUM、crossp、mp 的值，观察算法的收敛过程。

本例所实现的是最基本的遗传算法，在实际应用中，往往要事先建立问题的数学模型，选择合适的适应度函数，并根据问题的特点确定求解空间及染色体形式。本例事先用 matlab 绘出函数的曲线图，确认函数的最小值处于±8 之间，然后在此区间内求解。交叉、变异、选择的实现方式常常由选定的适应度函数以及染色体形式决定，可供选择的方式有多种，这里不再一一列举，有兴趣的读者可查看相关资料，但是遗传算法的基本思想都已在本实例中体现出来，读者根据自己的需要进行改进，便可实现属于自己的遗传算法了。

实例 95 人工神经网络的 C 语言实现

实例说明

人工神经网络目前已经在模式识别、知识工程、最优化问题求解、系统辨识、自适应控制、图像处理、计算机视觉、语音合成、数据压缩、函数拟和等领域得到了广泛的应用，在这一节里将摒弃大量的概念、公式，从一个简单的神经网络的应用实例出发，使读者对神经网络有所管窥。我们所采用的神经网络为最常用的 BP 算法（Back-Propagation Algorithm），即反向传播算法。

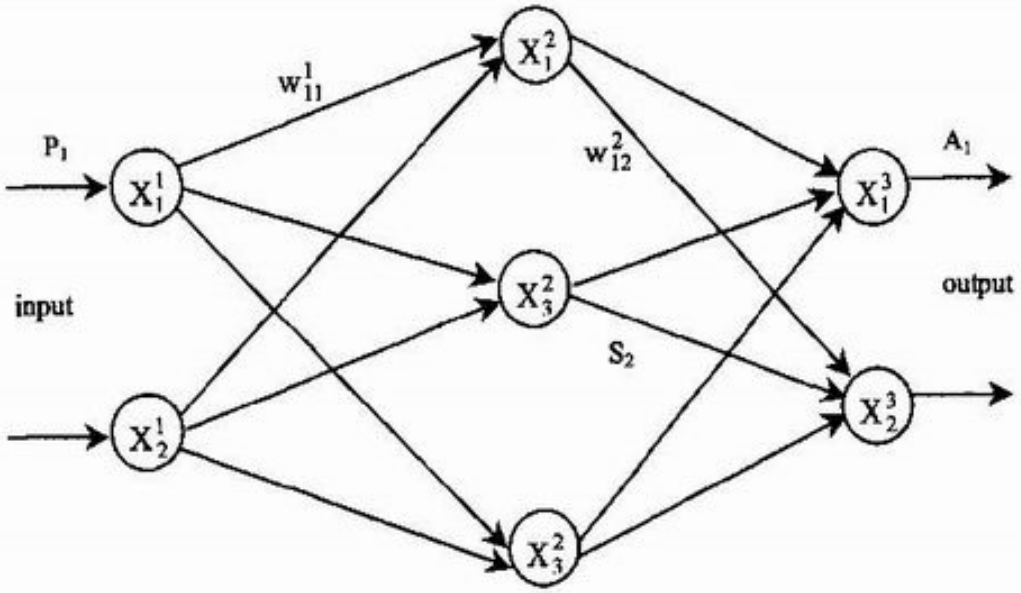
程序将涉及到梯度下降法、函数求逆及偏导数的实现等数学问题，并且在实例中我们还将使用一个较为常用的函数：toupper()。

知识要点

人工神经网络是在现代神经学、生物学、心理学等科学研究成果的基础上产生的，反映了生物神经系统的基本特征，是对生物神经系统的某种抽象、简化与模拟。具体的人工神经网络由许多并行互联的相同神经元模型组成，网络的信号处理由神经元之间的相互作用实现。一个神经网络主要由组成它的神经单元（也称为处理单元），神经单元之间的连接方式，以及网络的学习方式来确定。其中神经单元可以用来表达特殊的概念对象，它有这样几个特点：①每个处理单元可以接受任意个扇入连接(输入连接)；②每个处理单元的每个扇出连接上的信号都必须相同；③处理单元可以有局部记忆；④每个处理单元具有一个传递函数；神经网络单元间的连接模式决定了该网络对任意输入如何反应，以及信息在各神经单元间的流动方式，神经网络中所有的长期知识都由连接或其权值来编码，通常用权 W 来表示两个单元之间的连接；神经网络模型中信息处理或知识结构的变化是由处理单元之间联系模式的变化体现的，学习就是改变权或添加、移去连接，学习方式则是指采用怎样的规则实现这种权值的改变。

人工神经网络主要有两种模型：前向（feed-forward）和反馈（feed-back）型网络。前向型人工神经网络的特点是将神经元分若干层，每一层的神经元之间没有信息交流，并且计算是一层一层且同步地进行的；反馈型神经网络则将整个网络看成一个整体，神经元相互作用，计算是整体性的。对前向型人工神经网络，一般将网络的计算分为两个阶段：学习阶段和应用阶段。学习阶段的主要工作是确定权数 W。应用阶段是在权数确定的基础上用带有确定权数的神经网络解决实际问题。这一节将要实现的 BP 网络就属于前向型神经网络。在人工神经网络的实际应用中，80%～90%的人工神经网络模型采用 BP 网络或它的变化形式，它也是前向网络的核心，是人工神经网络最精华的部分。

BP 网络的产生归功于 BP 算法的获得。BP 算法是一种监督式的学习算法。它的主要思想是：已知 q 个输入学习样本 P_1，P_2，…，P_q，和与其对应的输出样本 T_1，T_2，…，T_q，用实际输出 A_1，A_2，…，A_q与目标矢量 T_1，T_2，…，T_q之间的误差来修改网络权值，使 A_ℓ（ℓ=1，2，…，q）与期望的 T_ℓ尽可能接近，即使网络输出层的误差平方和达到最小。它采用梯度下降法，即通过在误差函数斜率下降的方向上计算网络权值和偏差的变化而逐渐逼近目标。每一次权值和偏差的变化都与网络误差的影响成正比，并以反向传播的方式传递到每一层。



三层 BP 网络示意图

上图为由两个输入层节点、三个隐层节点、两个输出层节点组成的 BP 网络。其中 P_1 为神经网络的一个输入，S_2 为隐层第二个单元的输出，A_1 为神经网络的输出之一。X_1^1 为第一层即输入层的第一个神经单元，X_1^2 为第二层即隐层的第一个神经单元，X_1^3 为第三层即输出层的第一个神经单元，W_{11}^1 为连接 X_1^1、X_1^2 的连接权值，W_{12}^2 为连接 X_1^2、X_2^3 的连接权值，图中的箭头代表信息正向传递时信号的流向。

BP 算法由两部分组成：信息的正向传递与误差的反向传播。在正向传播过程中，输入信息从输入层经隐层逐层计算，传向输出层，每一层神经元的状态只影响下一层神经元。如果在输出层没有得到期望的输出，则计算输出层的误差，然后转向反向传播，通过网络将误差信号沿原来的连接通路反传回来修改各层神经元的权值直至达到期望目标。

程序解析

- 问题描述：在本实例中用 C 语言编写一个简单的 BP 神经网络，并用这个网络对正弦函数 sin(x)在（ 0，1 ）段的部分进行拟合。
- 程序源代码及讲解

```
#include "math.h"
```

```
#include "time.h"
#include "stdio.h"
#include "stdlib.h"
#define Ni 1 //定义输入层神经单元数量，由于待拟合的函数只有一个自变量，因此定义Ni为1
#define Nm 4 //定义隐层神经单元数量，这是一个可调节值，可以根据不同的问题进行改变
#define No 1 //定义输出神经单元数量，由于只有一个因变量，因此为1
#define L 100 //确定待学习的样本数量
#define Enom 0.02 //确定误差上限，当神经单元对所有待学习样本拟合的误差之和小于此值
//时认为训练可以结束
#define loopmax 100000// 确定最大迭代次数，当达到loopmax时误差之和扔大于Enom则认
// 为训练失败
#define e 2.71828
double E;       // 训练时每轮迭代的误差之和
double a , u , n ;   // BP网络中的参数
double W1[ Ni ] [ Nm ] , W2[ Nm ] [ No ] ;     // 网络连接权值
double D1[ Ni ] [ Nm ] , D2[ Nm ] [ No ] ;     // 每一轮对连接权值的修正量
double D22[ Nm ] [ No ] , D11[ Ni ] [ No ] ;   // 上一轮对连接权值的修正量，第
                                               // 一轮时定义为0
double a1[ Ni ] [ Nm ] , a2[ Nm ] [ No ] ;     // 神经网络的学习速率
double Pi[ L ] [ Ni ] , Pm[ L ] [ Nm ] , Po[ L ] [ No ] ; // 三层神经元的输入
double T[ L ] [ No ] ;                         // 神经网络的期望输出
double Xm[ L ] [ Nm ] , Xo[ L ] [ No ] ;       // 隐层及输出层神经元的输出
double Qm[ L ] [ Nm ] , Qo[ L ] [ No ] ;       // 对应于各连接的δ值
void initiate ( ) ;          // 训练时调用的函数，对神经网络进行初始化
void proceed ( ) ;           // 训练时调用的函数，信号的正向传播
void forQ ( ) ;// 训练时调用的函数，计算各神经元的δ值，将训练误差进行反向传播
vcid amend ( ) ;   // 训练时调用的函数，依据误差反向传播的结果对网络权值进行修改
void proceedR ( ) ;  // 应用训练好的BP网络拟合原函数
double newa (double a , double D ) ;       // 修改网络的学习速率
double cal ( double d ) ;                  // 计算Signoid函数值
double vcal ( double d ) ;                 // 计算下降梯度
main ( )
{
   long int i ;
    int flag ;                             // 表示训练是否成功
    char choice ;
   for ( ; ; )
    {
        flag = 0 ;
        initiate ( ) ;          // 初始化神经网络
        for ( i = 0 ; i < loopmax ; i++ )
        {
            proceed ( ) ;
            if ( E < Enom )
```

```
                {                    // 当误差和小于设定值时表示训练成功，结束循环
                    flag = 1 ;
                    break ;
                }
    if ( i % 2500 == 0 )// 每进行 2500 次循环显示一次训练情况，帮助用户确定是否继
    // 续训练或是否需要修改参数
                    printf ( "第%10d 轮误差：%20f，学习速率：%10f\n" , i , E , a1[ 0 ]
[ 0 ] ) ;
                forQ ( ) ;           // 计算各连接的 δ 值，将训练误差进行反向传播
                amend ( ) ;          // 依据误差反向传播的结果对网络权值进行修改
            }
            if (flag > 0)  proceedR ( ) ;// 如果训练成功，将神经网络投入使用
            else  printf ( "训练失败! \n" ) ;
            for ( ; ; )                  // 如果训练失败则询问是否重新训练
            {
                printf ( "是否继续? (Y / N)\n" ) ;
                scanf ( "%s" , & choice ) ;
                choice = toupper ( choice ) ;
                if ( choice == 'Y' )  break ;
                if ( choice == 'N' )  exit ( 0 ) ;
            }
        }
    }
```

作为 BP 算法的主干，函数分为两部分，第一部分从 initiate()函数开始到第一个循环结束，是 BP 网络的训练阶段，第二部分为训练成功后的应用阶段。训练阶段又分为三个子过程，即信息的前向传递（proceed()），终止条件判断，误差的反向传播和权值的修改。在两种条件下神经网络训练停止，一是训练成功，对学习样本拟合的误差和小于设定值；二是训练次数达到设定的上限，这是因为虽然从理论上而言，三层前向网络可对任意函数进行任意精度的拟合，但是由于初值的选择，计算机最大精度的限制，在实际应用中仍会出现网络收敛于局部最小值而达不到要求的情况，同时实际应用中常常对算法实时性有严格的要求而不允许过多的迭代次数，因此，当达到一定迭代次数后训练必须停止，出现这种情况时可以考虑修改训练参数或是修改网络结构（例如隐层节点数等）。

在询问是否重新训练时还调用了 toupper()函数，其标准表达式为：

```
#include  <ctype.h>
int  toupper ( int ch );
```

函数 toupper (int ch)在 ch 为字母时，返回等价的大写字母，否则返回 ch 值不变。与之对应的是：

```
#include  <ctype.h>
int  tolower ( int ch );
```

它将 ch 返回等价的小写字母，这两个函数常被用来对输入的 Dos 指令进行规范化处理。

```
void  initiate ( )
```

```
{
    int i , j , random ;
    double x , step ;
    int stime ;
    long ltime ;
    ltime = time ( NULL ) ;
    stime = ( unsigned ) ltime / 2 ;
    srand ( stime ) ;                    // 初始化随机序列
    u = 1 ;
n = 1 ;                                  // 设定参数
    printf ( "本程序将用BP神经网络拟合函数：Y = sin(X)  \n \n" ) ;
    for ( i = 0 ; i < Nm ; i++ )
    {
        for ( j = 0 ; j < Ni ; j++ )
        {
            random = rand ( ) % 100 - 50 ;
            x = random ;
            x = x / 100 ;
            W1[ j ] [ i ] = x ;          // 设定网络连接初值取值范围为 ( 0.5 , 0.5 )
            D11[ j ] [ i ] = 0 ;
            D1[ j ] [ i ] = 0 ;
            a1[ j ] [ i ] = 0.01 ;       // 设定初始学习速率
        }
        for ( j = 0 ; j < No ; j++ )
        {
            random = rand ( ) % 100 - 50 ;
            x = random ;
            x = x / 100 ;
            W2[ i ] [ j ] = x ;
            D22[ i ] [ j ] = 0 ;
            D2[ i ] [ j ] = 0 ;
            a2[ i ] [ j ] = 0.01 ;
        }
    }
    step = 1.0 / L ;                     // 将待训练的区间等分，步长为 step
    for ( i = 0 ; i < L ; i++ )
    {
        x = i ;
        Pi[ i ] [ 0 ] = x * step ;
        T[ i ] [ 0 ] = sin ( Pi[ i ] [ 0 ] ) ;
    }
    printf ( "初始化成功! \n \n 下面将对神经网络进行训练请稍候。\n" ) ;
}
void proceed ( )
```

```
{
    int i , j , k ;
    E = 0 ;
    for ( i = 0 ; i< L ; i++ )
    {
        for ( j = 0 ; j < Nm ; j++ )
        {
            Pm[ i ] [ j ] = 0 ;
            for ( k = 0 ; k < Ni ; k++ )
                Pm[ i ] [ j ] = Pi[ i ][ k ] * Wl[ k ] [ j ] + Pm[ i ] [ j ] ;
            Xm[ i ] [ j ] = cal ( Pm[ i ] [ j ] ) ;
        }
        for ( j = 0 ; j < No ; j++ )
        {
            Po[ i ] [ j ] = 0 ;
            for ( k = 0 ; k < Nm ; k++ )
                Po[ i ] [ j ] = Xm[ i ] [ k ] * W2[ k ] [ j ] + Po[ i ] [ j ] ;
            Xo[ i ] [ j ] = cal ( Po[ i ] [ j ] ) ;
            E = E + ( Xo[ i ] [ j ] - T[ i ] [ j ] ) * ( Xo[ i ] [ j ] - T[ i ]
[ j ] ) / 2 ;
        }
    }
}
```

在信号的前向传递过程中第 i+1 层第 l 个神经单元的输入为：

$$P_l^{i+1} = \sum_{k=0}^{Qi} X_k^i \times W_{kl}^i$$

其中 X_k^i 为第 i 层第 k 个神经单元的输出，其输出为：

$$X_k^{i+1} = g_{i+1}(P^{i+1}{}_l)'$$

其中 g_{i+1} ()为第 i+1 层神经单元的激活函数，在程序中由 cal ()函数实现。BP 网络要求其激活函数必须连续可微，常用的是 S 型的对数或正切激活函数以及线性函数。S 型函数具有非线性放大系数功能，它可以把输入从负无穷大到正无穷大的信号变换成-1 到 1 之间输出，对较大的输入信号，放大系数较小，而对较小的输入信号，放大系数则较大，所以采用 S 型激活函数可以处理和逼近非线性的输入、输出关系。如果在输出层采用 S 型函数，输出则被限制到一个很小的范围，若采用线性激活函数，则可使网络输出任何值。所以当网络的输出没有限制时在隐含层采用 S 型激活函数，而输出层采用线性激活函数。本实例中由于待拟合的是 sin() 函数，因此隐层及输出层都采用对数激活函数。

```
void forQ ( )
{
```

```
        int i , j , k ;
        for ( i = 0 ; i < L ; i++ )
        {
            for ( j = 0 ; j < No ; j++ )
                Qo[ i ] [ j ] = ( T[ i ] [ j ] - Xo[ i ] [ j ] ) * vcal ( Xo[ i ]
[ j ] ) ;
            for ( j = 0 ; j < Nm ; j++ )
            {
                Qm[ i ] [ j ] = 0 ;
                for ( k = 0 ; k < No ; k++ )
                    Qm[ i ] [ j ] = Qo[ i ] [ k ] * W2[ j ] [ k ] + Qm[ i ] [ j ] ;
                Qm[ i ] [ j ] = Qm[ i ] [ j ] * vcal ( Xm[ i ] [ j ] ) ;
            }
        }
    }
```

forQ ()函数计算神经元的 δ 值，并将训练误差进行反向传播，其中 Qo[i][j]、Qm[i][j] 分别代表输出层第 j 个神经单元、隐层第 j 个神经单元相对于第 i 个训练样本的 δ 值。在采用梯度下降法训练神经网络时， δ 可视为由第 i 个输入样本产生的神经元实际输入与神经网络完全拟合于目标函数时的理想输入的差，它可由输出误差与激励函数导数的乘积近似计算。在本程序中激励函数导数由 vcal (double d)实现。

```
    void amend ( )
    {
        int i , j , k ;
        double D ;
        for ( i = 0 ; i < Nm ; i++ )
        {
            for ( j = 0 ; j < Ni ; j++ )
                D1[ j ] [ i ] = 0 ;
            for ( j = 0 ; j < No ; j++ )
            D2[ i ] [ j ] = 0 ;
        }
        for ( i = 0 ; i < Ni ; i++ )
        {
            for ( j = 0 ; j < Nm ; j++ )
            {
                for ( k = 0 ; k < L ; k++ )
                    D1[ i ] [ j ] = Qm[ k ] [ j ] * Pi[ k ] [ i ] + D1[ i ] [ j ] ;
                D = D1[ i ] [ j ] * D11[ i ] [ j ] ;        // 为 D11 赋初值
                a1[ i ] [ j ] = newa ( a1[ i ] [ j ] , D ) ;  // a 赋初值
                W1[ i ] [ j ] = W1[ i ] [ j ] + a1[ i ] [ j ] * ( n * D1[ i ] [ j ]
+ ( 1 - n ) * D11[ i ] [ j ] ) ;
                D11[ i ] [ j ] = D1[ i ] [ j ] ;
            }
```

```
        }
        for ( i = 0 ; i < Nm ; i++ )
        {
            for ( j = 0 ; j < No ; j++ )
            {
                for ( k = 0 ; k < L ; k++ )
                    D2[ i ] [ j ] = Qo[ k ] [ j ] * Xm[ k ] [ i ] + D2[ i ] [ j ] ;
                D = D2[ i ] [ j ] * D22[ i ] [ j ] ;          //为 D11 付初值
              a2[ i ] [ j ] = newa ( a2[ i ] [ j ] , D ) ;
                W2[ i ] [ j ] = W2[ i ] [ j ] + a2[ i ] [ j ] * ( n * D2[ i ] [ j ]
+ ( 1 - n ) * D22[ i ] [ j ] ) ;
                D22[ i ] [ j ] = D2[ i ] [ j ] ;
            }
        }
    }
```

amend()函数根据反向传播的误差对网络连接权值进行修改，在神经网络训练时可采取两种方法，一种为模式学习，即每一次对一个样本进行学习，修改连接权值，直到最后一个样本，然后再进行下一轮循环；另一种方法是批学习，即每次同时对所有的样本进行学习，计算误差时计算全体样本产生的误差之和，在误差反向传播时分别计算各样本产生的误差，而当修改权值时同时考虑所有的样本的影响，这样不断循环直到误差之和达到可以接受的程度。本实例中采用的就是批学习的方式，两种学习方法各有优点，从实时性考察模式学习需要更小的存储空间，而且由于样本以随机的顺序输入，影响权值，使得权值在权值空间中以随机的方式逼近最优值，这样 BP 算法将不会陷入局部最小值。但相比而言批学习有更快的收敛速度，并且批学习比模式学习更容易通过并行算法来实现。在 amend()函数中还涉及到两个参数：学习速率与动量项，其中学习速率决定了每次循环中网络权值的改动幅度；动量项使网络在修正其权值时不但考虑当前误差对权值的修改，而且考虑上次循环时的误差对权值的影响，其作用如同一个低通滤波器，它允许网络忽略网络上的微小变化特性。在没有附加动量的作用下，网络可能陷入浅的局部极小值，利用附加动量的作用则有可能滑过这些极小值。

```
    void  proceedR ( )
    {
        int  i , j ;
        float  x ;
        double  input , output ;
        char  choice ;
        for ( ; ; )
        {
            for ( ; ; )
            {
                printf ( "在此输入需要计算的值(0 , 1): \n" ) ;
                scanf ( " % f " , & x ) ;
                input = ( double ) x ;
```

```
            if ( ( input >= 0 ) & ( input <= 1 ) ) break ;
            printf ( " 注意输入值应介于 0、1 之间! \n " ) ;
            for( ; ; )
            {
                printf ( "是否继续? (Y / N ) \n " ) ;
            scanf ( "%s" , & choice ) ;
                choice = toupper ( choice ) ;
                if ( choice == 'Y' ) break ;
                if ( choice == 'N' ) exit( 0 ) ;
            }
        }
        for ( i = 0 ; i < Nm ; i++ )
        {
            Pm[ 0 ] [ i ] = 0 ;
            For ( j = 0 ; j < Ni ; j++ )
                Pm[ 0 ] [ i ] = input * W1[ j ] [ i ] + Pm[ 0 ] [ i ] ;
            Xm[ 0 ] [ i ] = cal ( Pm[ 0 ] [ i ] ) ;
        }
        for ( i = 0 ; i < No ; i++ )
        {
            Po[ 0 ] [ i ] = 0 ;
            for ( j = 0 ; j < Nm ; j++ )
                Po[ 0 ] [ i ] = Xm[ 0 ] [ j ] * W2[ j ] [ i ] + Po[ 0 ] [ i ] ;
        }
        output = cal ( Po[ 0 ] [ 0 ] ) ;
        printf ( " 输入值为 %20f 对应的结果为 %f \n " , input , output ) ;
        printf ( " 输入值为 %20f 对应的正常结果为 % f \n" , input , sin ( input ) ) ;
    }
}
```

神经网络训练成功后调用 proceedR ( )函数，proceedR ( ) 与 proceed ( ) 相近，只是增添了接收输入、输出显示的功能。

```
dcuble  newa ( double a , double D )
{
    if ( D > 0 )
    {
        if ( a <= 0.04 )
            a = a * 2 ;
        else  a = 0.08 ;
    }
    else
        if ( D < 0 )
        {
            if ( a >= 0.02 )
                a = a / 2 ;
```

```
            else  a = 0.01 ;
        }
    return  a ;
}
```

对于一个特定的问题，要选择适当的学习速率不是一件容易的事情。通常是凭经验或实验获取。但即使这样，对训练开始初期功效较好的学习速率，不见得对后来的训练合适。为了解决这一问题，人们自然会想到在训练过程中，自动调整学习速率。通常调节学习速率的准则是检查权值的修正值是否真正降低了误差函数，如果确实如此，说明所选取的学习速率值小了，可以对其增加一个量。若不是这样，产生了过调，那么就应该减小学习速率的值。在本例中用newa() 函数实现学习速率自动调节的功能，它有两个参数，参数 a 为上一轮对应于当前连接的学习速率，参数 D 起判断作用。当最近两次对连接权值调整的方向相同：都是增强或都是减弱时 D 为一正值，此时表示修正值降低了误差参数，a 增大；反之则为负，a 减小。为了避免学习速率过大造成神经网络学习收敛的不稳定，设定了学习速率的上限：0.08，同时为了避免学习速率过小而造成收敛过慢设定了下限：0.01。

```
double  cal ( double d )
{
    d = - ( d * u ) ;
    d = exp ( d ) ;
    d = 1 / ( 1 + d ) ;
    return d ;
}
```

cal (double d)用于实现 S 型激励函数：

$$X_k^{i+1} = \frac{1}{1+\exp(-u \times P^{i+1})} \quad \text{其中} u > 0$$

u 为一可调参数。

```
double  vcal ( double d )
{
    return  u * d * ( 1 - d ) ;
}
```

vcal （double d ）函数用于返回 S 型激励函数的导数，有兴趣的读者可尝试自行推导。

总结分析

与遗传算法类似，BP 神经网络的学习过程也表现出一定随机性，表现在相邻两次学习所用的时间不同甚至有时会不收敛，而且两次学习成功后对同一输入的输出也会有所不同。这是因为在对神经网络初始化时为了保证网络的遍历性、防止出现局部最小，对连接权值赋随机值。需要注意的是由于系统是非线性的，连接权值的初始值对于学习是否达到局部最小、是否能够收敛以及训练时间的长短有很大影响。如果初始权值太大，使得加权后的输入落在了 S 型激励函数的饱和区，导致其导数非常小，而在计算权值修正公式中，因为 δ 与导数

成正比，当导数趋于0时，δ 亦趋于0，这使得连接权值的改变量非常小，从而使得调节过程几乎停顿下来，所以一般总是希望经过初始加权后的每个神经元的输出值都接近于零，这样可以保证每个神经元的权值都能够在它们的S型激活函数变化最大之处进行调节。一般取初始权值在（-1，1）之间的随机数，本例中连接权值的初始值取为（-0.5，0.5）。

为了使读者更好地了解BP网络的收敛过程，本实例在训练过程中每经过2500轮就显示一次当前的误差及连接 W_{11}^{1} 所对应的学习速率，读者可以自行改变网络的参数，如动量项、连接权值的初始值范围等，或改变网络的结构，如隐层单元个数等，观察这些变化对学习效果的影响。

需要注意的是，虽然BP算法当前得到了广泛的应用，但仍有一定的不足，例如收敛速度慢，可能会陷入局部极小值等，人们已经提出了很多对其加以改进的方法，例如在训练初期采用上一节所学到的遗传算法对网络进行训练，当达到一定误差范围后再改为BP算法进行进一步拟合，这时神经网络的连接权值作为遗传算法的待求解，而神经网络的误差则成为相应染色体的适应度。

实例96 K_均值算法

实例说明

在本节及下一节中将实现一种动态聚类法——迭代自组织数据分析算法（ISODATA 算法）。选择这个算法的原因是因为它有较为复杂的逻辑结构，包括了多次条件判断、循环、中断等程序控制操作，包含了插入排序算法，并需要运用动态内存分配以满足参数动态配置的需要。我们希望这两个实例能够帮助读者提高实现复杂逻辑结构程序的能力，同时也希望大家能够掌握这种在数据图像及模式识别、经济、社会学等需要对大量数据进行分类、分析的领域中得到广泛应用的算法。

由于算法较为复杂，同时考虑到大部分读者对动态聚类算法缺乏了解，将分两步进行。在本节中先介绍聚类分析的概念，实现一种较为简单的 K_均值算法，并复习、总结前面已经学过的内存的三种分配方法及文件操作等内容。

知识要点

聚类分析是指将一批没有标出类别的模式样本集按照样本之间的相似程度进行分类，相似的归为一类，不相似的归为另一类，划分的结果应使某种表示聚类质量的准则函数为最大。当样本由 n 维向量表示，并用距离表示两个样本间的相似度时，n 维特征空间被划分成若干个区域，每个区域相当于一个类别。一些常用的距离度量都可以作为这种相似度量。采用距离表示样本间的相似度的原因是因为从经验上看，凡是同一类样本，其特征向量应该是互相靠近的，而不同类的样本其特征向量之间的距离要大得多。在聚类分析中这种用来衡量样本之间相似性程度的距离被称为相似性的测度，常用的相似性测度有欧氏距离、马氏距离、明氏距离、角度相似性函数等，其中最常用的为欧氏距离，简称为距离，定义为 D=||X-Z||，即为两项间的范数，通常范数取 2，即：

$$D=\sqrt{(X-Z)^2}$$

在本例中也取用 2 范数作为相似性测度。本节的 K_均值算法及下一节要介绍的 ISODATA 算法都属于聚类分析中的动态聚类法。这种方法先行选择若干样本点作为聚类中心，再按某种聚类准则（通常采用最小距离原则）使各样本点向各个中心聚集，从而得到初始分类。然后，判断初始分类是否合理，如果不合理，就修改分类，如此反复进行修改聚类的迭代运算，直到合理为止。

K_均值算法步骤：

（1）选 K 个初始聚类中心：$Z_1(1)$、$Z_2(1)$、……，$Z_K(1)$。括号内的序号为寻找聚类中心的迭代运算的次序号。聚类中心的向量值可任意设定，例如可用开头 K 个模式样本的向量值作为初始聚类中心；

（2）逐个将需分类的模式样本{X}按最小距离原则分配给 K 个聚类中心中的某一个 $Z_i(1)$；

（3）计算各个聚类中心的新的向量值：

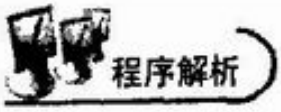
$$Z_j(k+1)=\frac{1}{N_j}\sum_{x\in S_j(k)}X \quad ，j=1，2，\cdots，K$$

式中 N_j 为第 j 个聚类域 S_j 中所包含的样本个数。以均值向量作为新的聚类中心，可使聚类准则函数

$$J_j=\sum_{x\in S_j(k)}\left\|X-Z_j(k+1)\right\|^2$$

最小。其中 j=1，2，…，K

（4）如果 $Z_j(k+1)\neq Z_j(k)$，j=1，2，…，K，则回到第 2 步，将模式样本逐个重新分类，重复迭代计算，反之算法收敛，计算完毕。




```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
int vectornum ;              // 样本点的数量
int vectorsize ;             // 代表样本点的向量的维数
float * datafield ;          //指向记录样本点相关信息的内存块，所指内存在程序运行时动态分配
float * tempcenter ;         // 在程序运行时暂时记录聚类中心向量
struct GROUP                 // 定义聚类域结构
{
    float * center ;         // 聚类中心坐标
    int groupsize ;          // 聚类包含的样本个数
} g , * group ;              // * group 指向记录聚类域相关信息的内存块，此内存在程序运
                             // 行时动态分配
int K ;                      //聚类中心个数
void initiate ( ) ;          // 读入数据，初始化聚类中心，参数设定默认值
int allocate ( ) ;           // 将模式样本分配给最近的聚类，当返回值为 1 时表示算法已收敛
void showresult ( ) ;        // 显示分类结果
float distance ( float * x , float * y ) ;// 计算两个向量间的欧氏距离
float data ( int i , int j ) ; // 从 datafield 中读取指定位置的值
```

```
float  * vector ( int i ) ;  // 从datafield中读取指定的样本向量
void  write ( int i , int j , float data ) ;// 向datafield中指定位置写入值
void  main ( )
{
    int  i ;
    initiate ( ) ;
    for ( i = 1 ; i < 50 ; i++ )
    {
        showresult ( ) ;
        if ( allocate ( ) == 1 )  break ;
    }
    showresult ( ) ;
    free ( datafield ) ;
    free ( tempcenter ) ;
    for ( i = 0 ; i < K ; i++ )
        free ( group[ i ].center ) ;
}
void  initiate ( )
{
    int  i , j , size ;
    float  d ;
    FILE  * df ;
    K = 2 ;        // 设定聚类数默认为2，可在后面进行修改
    if ( ( df = fopen ( "data.txt" , "r" ) ) == NULL )
    {
        printf ( "Cannot open file\n" ) ;
        exit ( 1 ) ;
    }
    fscanf ( df , "%5d" , & vectornum ) ;
    fscanf ( df , "%5d" , & vectorsize ) ;
    size = vectornum * ( vectorsize + 1 ) ;
    datafield = ( float * ) calloc ( size , sizeof ( float ) ) ;
    tempcenter = ( float * ) calloc ( vectorsize , sizeof ( float ) ) ;
    for ( i = 0 ; i < vectornum ; i++ )
    {
        for ( j = 0 ; j < vectorsize ; j++ )
        {
            fscanf ( df , "%10f" , & d ) ;
            write ( i , j + 1 , d ) ;
        }
        write ( i , 0 , -1 ) ;
    }
    if ( feof ( df ) )  printf ( " File read error! " ) ;
    fclose ( df ) ;
```

```
    printf ( "请输入聚类数:\n" ) ;
    scanf ( "%d" , & K ) ;
    group = ( GROUP * ) calloc ( K , sizeof ( g ) ) ;
    for ( i = 0 ; i < K ; i++ )
    {
        group[ i ].center = ( float * ) calloc ( ( vectorsize ) , sizeof ( float ) ) ;
        group[ i ].groupsize = 0 ;
    }
    for ( i = 0 ; i < K ; i++ )
    {
        for ( j = 0 ; j < vectorsize ; j++ )
        {
            * ( group[ i ].center + j ) = data ( i , j + 1 ) ;
        }
    }
}
```

initiate ( )函数从 data.txt 文件中读入待处理的样本，并用样本的前 K 个作为聚类的初始中心。函数首先读入样本的个数 vectornum，维数 vectorsize，并根据这两个量分配适当的内存。所分配的内存的第一个指针赋给 float * datafield，然后从文件中读入样本向量，保存到 datafield 所指的数据块中。内存的分配方式为：

第 i 个样本	所属的聚类域	表示样本的向量
1	0	vectorsize
2	vectorsize+1	vectorsize*2
...	...	...
vectornum	vectorsize*（vectornum-1）	Vectorsize*vectornum

其中 0 为 datafield 所指的内存地址。分配内存的操作由函数 calloc( )完成。calloc( )函数定义如下：

```
#include <stdlib.h>
void  * calloc ( size_t  num , size_t  size ) ;
```

可见函数 calloc( )的使用方法与 malloc( )相似，分配的内存量等于 num*size，即 calloc( )为 num 个大小为 size 的数组分配足够内存。分配内存中的所有位被初始化为零。它返回一个指针，指向分得的区域的第一字节。内存不足以满足请求时返回空指针。

为聚类域信息（* group）分配内存的方式与 datafield 的相同，只不过将 float 数据类型改为了 GROUP 结构。同时由于在不同的样本条件下，聚类中心的向量维数也不同，因此也应采用动态分配内存的方式，所分配的内存首地址赋给 group[i].center。

```
int  allocate( )
{
    int  i , j , k , flag , index ;
```

```
    float  D , D1 , sum ;
    for ( i = 0 ; i < K ; i++ )
    {
        group[ i ].groupsize = 0 ;
    }
    for ( i = 0 ; i < vectornum ; i++ )
    {                              // 将各样本点按最小距离分配到各聚类域
        D = distance ( group[ 0 ].center , vector ( i ) ) ;
        k = 0 ;
        for ( j = 1 ; j < K ; j++ )
        {
            D1 = distance ( group[ j ].center , vector ( i ) ) ;
            if ( D > D1 )
            {
                k = j ;
                D = D1 ;
            }
        }
        write ( i , 0 , ( float ) k ) ;        // 记录到样本点距离最近的聚类中心
        group[ k ].groupsize++ ;
    }
    flag = 1 ;
    for ( index = 0 ; index < K ; index++ )     // 计算新的聚类中心
    {
        for ( j = 0 ; j < vectorsize ; j++ )
            tempcenter[ j ] = 0.0 ;
        sum = ( float ) group[ index ].groupsize ;
        for ( i = 0 ; i < vectornum ; i++ )
        {
            if ( index == ( int ) data ( i , 0 ) )
                for ( j = 0 ; j < vectorsize ; j++ )
                    tempcenter[ j ] += data ( i , j + 1 ) / sum ;
        }
        for ( j = 0 ; j < vectorsize ; j++ )
        {
            if ( tempcenter[ j ] != group[ index ].center[ j ] )
            {
                group[ index ].center[ j ] = tempcenter[ j ] ;
                flag = 0 ;
            }
        }
    }
    return  flag ;
}
```

allocate( )函数首先按照距离最近原则将各样本点分配到已有的聚类域中，当第 i 个样本点被分配到第 j 个域中时，它在 datafield 中对应内存区域的第一个单元记录下这个域的代号 i；在所有的样本点都分配到相应的域中之后，函数将各个域样本点坐标的平均值作为这个域的新的中心。将新的中心与原有的中心坐标进行比较，如果相同则 flag 保持不变，否则赋零，并记录下新的中心坐标。如果所有的中心坐标都没有变，则 flag 值为 1，表示迭代已经收敛，可以结束，反之为 0。

```
void showresult ( )
{
    int i , j , k ;
    for ( i = 0 ; i < K ; i++ )
    {
        printf ( "第%3d组聚类中心坐标为:" , i + 1 ) ;
        for ( j = 0 ; j < vectorsize ; j++ )
            printf ( " %10f " , group[ i ].center[ j ] ) ;
        printf ( " \n聚类包含的样本点的坐标为:\n " ) ;
        for ( j = 0 ; j < vectornum ; j++ )
        {
            if ( data ( j , 0 ) == i )
            {
                for ( k = 0 ; k < vectorsize ; k++ )
                {
                    printf ( " %10f " , data ( j , k + 1 ) ) ;
                }
                printf ( " \n " ) ;
            }
        }
    }
}
float data ( int i , int j )
{
    return * ( datafield + i * ( vectorsize + 1 ) + j ) ;
}
```

data (int i , int j)函数返回指定的第 i 个样本点的第 j-1 个坐标分量的值（ j 不为 0 时 ），当 j = 0 时返回的是这个样本所属的聚类域的编号。由前面的图表可知，在所分配的内存空间中，相邻两个样本点的同一个坐标分量之间的距离为 vectorsize+1，因此第 i 个样本点的第 j 个信息到内存块的起始地址的距离为 i × (vectoesize + 1) + j , write (int i , int j , float data)函数的原理相同，只是将读操作改为了写操作。

```
void write ( int i , int j , float data )
{
    * ( datafield + i * ( vectorsize + 1 ) + j ) = data ;
}
```

```
float * vector ( int i )
{
    return datafield + i * ( vectorsize + 1 ) + 1 ;
}
float distance ( float * x , float * y )
{
    int i ;
    float z ;
    for ( i = 0 , z = 0 ; i < vectorsize ; i++ )
        z = z + ( ( * ( x + i ) ) - ( * ( y + i ) ) ) * ( ( * ( x + i ) ) - ( *
( y + i ) ) ) ;
    return ( float ) sqrt ( z ) ;
}
```

总结分析

到目前为止，我们已经接触到了三种最常用的内存分配方式，下面将对数组、链表、以及现在所采用的这种动态分配内存方式的优点及缺点进行分析，希望读者能从中得到一点启示。

数组所占用的内存是在程序进行编译时分配的，这种分配方式的优点在于，由于事先已经分配好内存，因此在程序运行时不会因为内存不足而导致操作失败，此外由于为数组分配的内存是连续的，因此可对数组元素按序号直接进行检索，省时省力。采用数组进行内存分配的缺点是当程序运行之前不知道具体对内存的需求时，往往要采用折衷的办法，要么考虑最坏的可能性，分配大于实际需要的存储空间，造成存储空间的浪费；要么对部分操作进行限制，当所需空间可能大于已分配空间时禁止这些操作。

链表正好与数组相反，它的元素可以在程序运行时进行动态的增减，往往每次只申请一个元素的空间，链表的长度可以根据需要逐渐增加或减小，因此不存在采用数组时所遇到的尴尬，所以链表对于那些需要存储的信息量变化较大的问题来说十分有效，但是链表的缺点也正在于它所采用的动态分配内存的方式，在程序运行中可能会遇到返回值为 NULL 的情况，即分配失败，尽管产生这种情况的可能性很小，但仍为程序的正常运行增加了变数。另外当需要存储的元素本身所占空间较小时，记录下一节点的指针也形成了存储空间的浪费，而且由于各元素分配的空间是不连续的，对链表的操作也不如对数组操作那样简单、迅速。

本文中所采用的动态内存分配方式是一种介于数组与链表之间的方法，它的存储空间在程序运行时进行分配，而同时又由于存储空间是连续的，所以可以进行类似数组的操作。当存储的元素呈一维排列时可直接作为数组进行操作，如本例中对 group 的操作。当呈二维或更高维数排列时需要自定义读、写函数，如本例中的 datafield。当然这种方法也有缺点，虽然分配的存储空间大小可用 realloc( )函数进行修改，但是还是希望在分配内存时已经知道对空间的需求量，而且由于期望所有的元素存储在一块连续的内存空间，因此对内存的要求也要比链表方式苛刻许多，特别是在对空间需求量较大的情况下。

实例97 ISODATA 算法

实例说明

上一节介绍了动态聚类分析及 k_均值算法。本节将继续实现动态聚类中的 ISODATA 算法，这是一种逻辑结构较为复杂的算法，希望读者在学习这种算法的同时不要忘记选择这个算法的目的。正如在本篇开头所提到的，希望读者能够根据给出的程序步骤独立完成，以检验自己完成复杂逻辑结构的能力。

知识要点

ISODATA 算法与 K_均值算法有相似之处，即聚类中心同样是通过样本均值的迭代运算决定的。但 ISODATA 还加入一些试探步骤，并且组合成人机交互的结构，使之能利用通过间接结果所取得的经验。

ISODATA 算法的基本步骤为：

➢ 分配样本点

（1）将 N 个模式样本{X_i，i=1，2，…，N}读入，预选 Nc 个初始聚类中心{Z_1，Z_2，…，Z_{Nc}}，它不必等于所要求的聚类中心的数目，初始位置也可从样本中任选一些带入。

预选：

K = 预期的聚类中心数目；

θ_N = 每一聚类域中最少的样本数目；

θ_S = 一个聚类域中样本距离分布的标准差；

θ_c = 两聚类中心间的最小距离；

L = 在一次迭代运算中可以合并的聚类中心的最多对数；

I = 迭代运算的次数序号。

（2）将 N 个模式样本分给最近的聚类 S_j，与在 K_均值算法中的操作相同；

（3）如果 Sj 总的样本数目 $N_j<\theta_N$，取消该样本子集，这时 Nc 减去 1；

➢ 修正聚类中心参数

（4）修正各聚类中心值：

$$Z_j=\frac{1}{N_j}\sum_{x\in S_j}X \quad ，j=1，2，\cdots，N_c$$

（5）计算各聚类域 S_j 中诸聚类中心间的平均距离：

$$\overline{D}_j = \frac{1}{N_j}\sum_{x\in S_j}\left\|X - Z_j\right\|，j = 1，2，\cdots，N_c$$

（6）计算全部模式样本对其相应聚类中心的总平均距离：

$$\overline{D} = \frac{1}{N}\sum_{j=1}^{Nc} N_j \overline{D}_j$$

（7）判断分裂、合并及迭代运算等步骤：

- ✧ 如果迭代运算次数已达 I 次，即最后一次迭代，置 $\theta_c = 0$，跳到第 11 步，运算结束。
- ✧ 如果 $N_c \leqslant K/2$，即聚类中心的数目等于或不到规定值的一半，进入第 8 步，将已有的聚类分裂。
- ✧ 如迭代运算的次数是偶次，或 $N_c \geqslant 2K$，不进行分裂处理，跳到第十一步；如不符合以上两个条件，则进入第 8 步，进行分裂处理。

➢ 分裂处理

（8）计算每聚类中样本距离的标准差向量：$\sigma_j = (\sigma_{1j}\sigma_{2j}\ldots\sigma n_j)^t$；

（9）求每一标准差向量$\{\sigma_j，j=1，2，\ldots，Nc\}$中的最大分量，以$\{\sigma_{jmax,}，j=1，2，\ldots，Nc\}$代表；

（10）在任意最大分量集$\{\sigma_{jmax,}j=1，2，\ldots，Nc\}$中，如果有 $\sigma_{jmax} > \theta_S$（该值给定），同时又满足以下两个条件中之一：

- ✧ $\overline{D}_j > \overline{D}$ 和 $N_j > 2(\theta_N + 1)$
- ✧ $Nc \leqslant K/2$

则将 z_j 分裂为两个新的聚类中心 z_j^+ 和 Z_j^-，且 Nc 加 1。z_j^+ 对应于 σ_{jmax} 的分量，可加上 σ_{jmax}，其中 $0<k\leqslant 1$；Z_j^- 中对应于 σ_{jmax} 的分量，可减去 σ_{jmax}。如果本部完成了分离运算，则跳回第 2 步，否则继续。

➢ 合并处理：

（11）计算全部聚类中心的距离：

$$\overline{D}_{ij} = \left\|Z_i - Z_j\right\|，i=1，2，\ldots，N_c\text{-}1$$

$$j=1+1，\ldots，N_c$$

（12）比较 D_{ij} 与 θ_c 值，将 $D_{ij}<\theta_c$ 的值按最小距离次序递增排序，记录其中的前 L 个；

（13）如将距离为 D_{ij} 的两个聚类中心 Z_i 和 Z_j 合并，得新中心为：

$$Z^*_l = \frac{1}{N_i + N_j}\left[N_i Z_i + N_j Z_j\right] \qquad l=1，2，\ldots，L$$

式中，被合并的两个聚类中心向量，分别以其聚类域内的样本数加权；

（14）如果是最后一次迭代运算（即第I次），算法结束，否则返回第1步。如果操作者需要改变输入参数，回到第2步，迭代运算的次数加1。


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
float * datafield;
int vectornum;
int vectorsize;
struct GROUP
{
    float * center;
    float D;
    float variance;
    int groupsize;
    int flag ;    // 代表这个数组单元是否被占用，为0时表示为空聚类域
};
struct GROUP group[100] ;
int maxindex ;  // 最大聚类号
int groupnum ;
struct CCD    // 记录各聚类中心的距离
{
    float dst ;
    int g1 , g2 ;      // 聚类编号
}ccd , * ccdhead ;
float Dst ;
//此处省略函数声明，请读者编程时自行添加
int K ;       // 预期的聚类中心数目
int Nmin ;    // 每一聚类中最少的样本数目，如少于此数就不作为一个孤立的聚类
float Ds ;     // 一个聚类域中样本距离分布的标准差
float Dm ;    // 两个聚类中心之间的最小距离，如小于此数，两个聚类进行合并
int L ;       // 在于此迭代运算中可以合并的聚类中心的最多对数
int I ;       // 迭代运算的次数序号
void main ()
{
    int i ;
    initiate ( ) ;              // 读入数据，初始化聚类中心，参数设定默认值
    for ( i = 1 ; ; i++ )
    {
        input ( ) ;             // 显示、修改参数
        allocate ( ) ;          // 完成第2～6步
```

```
        if ( i == I )          // 如果迭代运算次数已达 I 次，即最后一次迭代，
        {                  // 置 0，=0，跳到第 11 步，运算结束。
            Dm = 0 ;
            converge ( ) ;
            break ;
        }
        if ( groupnum <= K / 2 )  diverge ( ) ;
        else
        {
            if ( ( groupnum >= K * 2 ) | ( i % 2 == 0 ) )
                converge ( ) ;
            else
                diverge ( ) ;
        }
        renum ( ) ;            // 重新排列聚类域的次序
    }
    showresult ( ) ;
    free ( datafield ) ;
    free ( ccdhead ) ;
    for ( i = 0 ; i < 100 ; i++ )
        free ( group[ i ].center ) ;
}
void initiate ( )
{
    int i , j , size ;
    FILE * df ;
    char s[ 10 ] ;
    float d ;
    K = 2 ;
    Nmin = 1 ;                 // 设置默认的聚类参数
    Ds = 1 ;
    Dm = 4 ;
    L = 1 ;
    I = 5 ;
    maxindex = groupnum = 1 ;
    if ( ( df = fopen( "data.txt" , "r" ) ) == NULL )
    {
        printf ( "Cannot open file \n" ) ;
        exit ( 1 ) ;
    }
    fscanf ( df , "%5d" , & vectornum ) ;
    fscanf ( df , "%5d" , & vectorsize ) ;
    size = vectornum * ( vectorsize - 1 ) ;
    datafield = ( float * ) malloc ( size * sizeof ( float ) ) ;
```

```
    for ( i = 0 ; i < vectornum ; i++ )
    {
        for ( j = 0 ; j < vectorsize ; j++ )
        {
            fscanf ( df , "%10f" , & d ) ;
            write ( i , j + 1 , d ) ;
        }
        write ( i , 0 , -1 ) ;
    }
    fscanf ( df , "%10s" , s ) ;
    if ( feof ( df ) ) printf ( " File read error! " ) ;
    fclose ( df ) ;
    for ( i = 0 ; i < 100 ; i++ )
    {
        group[ i ].flag = 0 ;
        group[ i ].center = ( float * ) malloc ( ( vectorsize ) * sizeof ( float ) ) ;
        group[ i ].groupsize = 0 ;
        group[ i ].variance = 0 ;
        group[ i ].D = 0 ;
    }
    for ( i = 0 ; i < groupnum ; i++ )
    {
        for ( j = 0 ; j < vectorsize ; j++ )
            * ( group[ i ].center + j ) = data( i , j + 1 ) ;
        group[ i ].flag = 1 ;
    }
}
```

对聚类操作进行初始化，与K_均值中的相应函数作用相似。需要注意的是，由于ISODATA算法中存在取消聚类的操作，所以在某些时候样本点可能不属于任何一个聚类域，此时样本的聚类标示（记录样本信息的第一个元素）被置为-1。另外，由于存在聚类域的和并与分裂，所以group数组的元素并不是在任何时候都被顺序占用，即有可能出现第i个元素未被占用，而第i+1个元素被占用的情况。因此程序中专门设定了maxindex指向最后一个被占用的元素，而groupnum代表总的聚类个数，即被占用的元素数量。

```
void  input ( )
{
    char  choice ;
    printf ( "\n\n现用的参数取值:\n" ) ;
    printf ( "K = %d\n,Nmin=%d\n,Ds=%f\n,Dm=%f\n,L=%d\n,I=%d\n",K,Nmin,Ds,
Dm,L,I ) ;
    showresult ( ) ;
    printf ( "是否需要进行修改? (Y/N) \n" ) ;
    scanf ( "%s" , & choice ) ;
    choice = toupper ( choice ) ;
```

```
    if ( choice == 'Y' )
    {
        printf ( "\n请输入预期的聚类中心数目:" ) ;
        scanf ( "%d" , & K ) ;
        printf ( "\n请输入每一聚类域中最少样本数:" ) ;
        scanf ( "%d" , & Nmin ) ;
        printf ( "\n请输入一个聚类域中样本距离分布的标准差:" ) ;
        scanf ( "%f" , & Ds ) ;
        printf ( "\n请输入两聚类中心之间的最小距离:" ) ;
        scanf ( "%f" , & Dm ) ;
        printf ( "\n请输入一次迭代运算中可以合并的聚类中心的最多数目:" ) ;
        scanf ( "%d" , & L ) ;
        printf ( "\n请输入最大迭代次数:" ) ;
        scanf ( "%d" , & I ) ;
    }
}
```

每进行一次聚类操作，调用一次 input()函数，根据当前聚类情况，判断是否需要更改聚类参数。

```
void allocate ( )
{
    int i , j , k , num , index ;
    float D , D1 , sum ;
    for ( i = 0 ; i < 100 ; i++ )        // 为各聚类中心值清零
    {
        group[ i ].D = 0 ;
        group[ i ].groupsize = 0 ;
        group[ i ].variance = 0 ;
    }
    for ( i = 0 ; i < vectornum ; i++ ) // 按距离分配到各聚类域
    {
        D = distance ( group[ 0 ].center , vector ( i ) ) ;
        k = 0 ;
        for ( j = 1 ; j < groupnum ; j++ )
        {
            D1 = distance ( group[ j ].center , vector ( i ) ) ;
            if ( D > D1 )
            {
                k = j ;
                D = D1 ;
            }
        }
        write ( i , 0 , ( float ) k ) ;
        group[ k ].groupsize++ ;
    }
```

```
num = groupnum ;
for ( i = 0 ; i < num ; i++ )          // 当聚类中样本个数小于规定值时取消聚类
{
    if ( group[ i ].groupsize < Nmin )
    {
        group[ i ].flag = 0 ;
        group[ i ].groupsize = 0 ;
        for ( j = 0 ; j < vectornum ; j++ )
        {
            if ( data ( j , 0 ) == i )
            {
                write ( j , 0 , -1.0 ) ;
            }
        }
        groupnum-- ;
    }
}
for ( i = 0 ; i < 100 ; i++ )          // 为各聚类中心值清零
{
    for ( j = 0 ; j < vectorsize ; j++ )
        * ( group[ i ].center + j ) = 0 ;
}
for ( i = 0 ; i < vectornum ; i++ )     // 计算新的聚类中心
{
    index = ( int ) data ( i , 0 ) ;
    if ( index != -1 )
    {
        sum = ( float ) group[ index ].groupsize ;
        for ( j = 0 ; j < vectorsize ; j++ )
        {
            * ( group[ index ].center + j ) = * ( group[ index ].center + j )
                                + data( i , j + 1 ) / sum;
        }
    }
}
for ( i = 0 ; i < vectornum ; i++ )       // 计算各样点到聚类中心距离
{
    index = ( int ) data ( i , 0 ) ;
    if ( index != -1 )
    {
        sum = ( float ) group[ index ].groupsize ;
        D = distance ( group[ index ].center , vector ( i ) ) ;
        group[ index ].D = group[ index ].D + D / sum ;
    }
```

```
        }
        Dst = 0 ;
        for ( i = 0 ; i < maxindex ; i++ )
        {
            if ( group[ i ].flag != 0 )
                Dst = Dst + group[ i ].D ;
        }
        Dst = Dst / groupnum ;
    }
```

allocate( )函数与 K_均值中的 allocate( )函数作用类似，都进行样本点的分配、聚类中心的更新操作。除此之外 ISODATA 算法中的 allocate( )函数还要根据聚类中的样本个数判断聚类是否符合要求，当样本个数小于规定值时要执行取消聚类操作，为了后面操作的需要，allocate( )函数还对各聚类域中样本到聚类中心的平均距离、总的平均距离进行计算。

```
    void  diverge ( )
    {
        float  newvar , oldvar , center ;
        int  i , j , k , l , flag ;
        flag = 0 ;
        for ( i = 0 ; i < maxindex ; i++ )
        {
            if ( group[ i ].flag != 0 )
            {
                oldvar = 0 ;      // 标准差
                for ( j = 0 , l = 0 ; j < vectorsize ; j++ )
                {              // 计算同一聚类域中各分量对应的标准差的最大值
                    newvar = 0 ;
                    center = * ( group[ i ].center + j ) ;
                    for ( k = 0 ; k < vectornum ; k++ )
                    {
                        if ( data ( k , 0 ) == i )
                    newvar = newvar + ( center - data ( k , j +1 ) ) * ( center -
data ( k , j +1 ) ) ;
                    }
                    if ( newvar > oldvar )
                    {
                        oldvar = newvar ;
                        l = j ;
                    }
                }
                group[ i ].variance = ( float ) sqrt ( oldvar / group[ i ].groupsize ) ;
                if ( group[ i ].variance > Ds )
                    if ( ( groupnum <= K / 2) | ( ( group[ i ].D > Dst )
    & ( group[ i ].groupsize > 2 * ( Nmin + 1) ) ) )
```

```
                {
                    split ( i , 1 ) ;// 当满足条件时进行分裂操作
                    flag = 1 ;
                }
        }
    }
    if ( flag == 0 )                    // 如果未进行分裂操作则进行合并操作
        converge ( ) ;
}
void  split ( int  i , int  j )
{
    int  k , 1 ;
    k = maxindex ;
    group[ k ].flag = 1 ;
    for ( 1 = 0 ; 1 < vectorsize ; 1++ )
    {
        * ( group[ k ].center + 1 ) = ( * ( group[ i ].center + 1 ) ) ;
    }
    *(group[k].center+j)=(*(group[k].center+j)+group[i].variance);
    *(group[i].center+j)=(*(group[i].center+j)-group[i].variance);

    maxindex ++ ;
    groupnum ++ ;
}
void  converge ( )
{
    int  i , j , k , h , 1 ;
    float  D , c1 , c2 , n1 , n2 ;
    ccdhead = (struct CCD *) malloc (sizeof ( ccd ) * L ) ;
    for ( i = 0 , k = 0 ; i < maxindex - 1 ; i++ )
    {
        if ( group[ i ].flag != 0 )
            for ( j = i + 1 ; j < maxindex ; j++ )
                if ( group[ j ].flag != 0 )
                {
                    D = distance ( group[ i ].center , group[ j ].center ) ;
                    if ( D < Dm )
                    {// 比较Dij与θc值，将Dij<θc的值按最小距离次序递增排序，
                    //记录其中的前L个
                        if ( k < L )
                        {
                            ( ccdhead + k )->dst = D ;
                            ( ccdhead + k )->g1 = i ;
                            ( ccdhead + k )->g2 = j ;
```

```
                        k++ ;
                    }
                    else
                        insert ( i , j , D ) ;
                    break ;
                }
            }
        }
        for ( h = 0 ; h < k ; h++ )
        {               //合并选中的k对聚类域
            i = ( ccdhead + h )->g1 ;
            j = ( ccdhead + h )->g2 ;
            n1 = ( float ) group[ i ].groupsize ;
            n2 = ( float ) group[ j ].groupsize ;
            for ( l = 0 ; l < vectorsize ; l++ )
            {
                c1 = ( * ( group[ i ].center + l ) ) ;
                c2 = ( * ( group[ j ].center + l ) ) ;
                * ( group[ i ].center + l ) = ( n1 * c1 + n2 * c2 ) / ( n1 + n2 ) ;
            }
            group[ i ].groupsize = ( int ) ( n1 + n2 ) ;
            for ( l = 0 ; l < vectornum ; l++ )
                if (data ( l , 0 ) == j )
                    write ( l , 0 , ( float ) i ) ;
            group[ j ].flag = 0 ;          // 取消被合并的一对聚类域中的一个
            group[ j ].groupsize = 0 ;
            groupnum-- ;
        }
    }
    void  insert ( int i , int j , float D )
    {
        int  h , l ;
        for ( h = 0 ; h < L ; h++ )
        {
            if ( D < ( ccdhead + h )->dst )
            {
                for ( l = L - 1 ; l > h ; l-- )
                {
                    ( ccdhead + l )->dst = ( ccdhead + l - 1 )->dst ;
                    ( ccdhead + l )->g1 = ( ccdhead + l - 1 )->g1 ;
                    ( ccdhead + l )->g2 = ( ccdhead + l - 1 )->g2 ;
                }
                ( ccdhead + h )->dst = D ;
                ( ccdhead + h )->g1 = i ;
```

```
                ( ccdhead + h )->g2 = j ;
            }
        }
    }
```

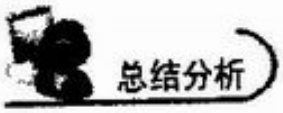
当 Dij<θc 的聚类对数大于 L 时，insert( )函数选择其中最小的 L 个，按照由小到大的顺序排列存储到 ccdhead 指向的内存空间。实现 insert( )函数功能的方法有多种，本例只采用了其中的一种，读者也可利用其他方法实现相同的功能。

```
void renum ( )
{
    int i , j , k ;
    for ( i = 0 ; i < maxindex - 1 ; i++ )
    {
        if ( group[ i ].flag == 0 )
        {
            for ( j = maxindex ; j > i ; j-- )
            {
                if ( group[ j ].flag == 1 )
                {
                    group[ i ].flag = 1 ;
                    group[ j ].flag = 0 ;
                    for ( k = 0 ; k < vectorsize ; k++ )
                    {
                        * ( group[ i ].center + k ) = ( * ( group[ j ].center
+ k ) ) ;
                        * ( group[ j ].center + k ) = 0 ;
                    }
                }
            }
        }
    }
    maxindex = groupnum ;
}
```

前面已经提到过，由于 ISODATA 算法中包含聚类域的取消、合并、以及分裂，所以当进行完一轮聚类运算后，group 数组中的元素已不再是按顺序被占用的了，例如聚类域 group[i]可能由于合并操作或所含样本数小于规定值而被取消，但是 group[i+1]所代表的聚类域仍然存在，这为下一轮的运算带来了不便，降低了运算效率，因此在每进行完一轮聚类运算后都要调用 renum( )函数，使 group 数组中的元素按顺序排列。

```
void showresult ( )
{
    int i , j , k , l ;
    for ( j = 1 , i = 0 ; i < maxindex ; i++ )
    {
```

```
        if ( group[ i ].flag != 0 )
        {
            printf ( "第%3d组聚类中心坐标为:" , j ++ ) ;
            for ( k = 0 ; k < vectorsize ; k++ )
                printf ( " %10f " , * ( group[ i ].center + k ) ) ;
            printf ( " \n聚类包含的样本点的坐标为:\n " ) ;
            for ( k = 0 ; k < vectornum ; k++ )
            {
                if ( data ( k , 0 ) == i )
                {
                    for ( l = 0 ; l < vectorsize ; l++ )
                    {
                        printf ( " %10f " , data ( k , l + 1 ) ) ;
                    }
                    printf ( " \n " ) ;
                }
            }
        }
    }
}
```



本节和上一节分别讲了动态聚类算法中的K_均值算法和ISODATA算法，经过比较可知，二者都是通过事先假定若干聚类中心，然后将样本分配到由这些聚类中心确定的聚类域中，再根据各聚类域中的样本确定新的聚类中心，往复循环直到达到设定的目标。与K_均值算法不同的是ISODATA算法除了设定期望的聚类中心数之外还通过统计条件对聚类操作加以限制，由于这些条件之间相关不独立，因此有可能会出现实际聚类域的数量和期望值不同，或有样本点未被分配到任何聚类域中的情况。当出现这些情况时可通过在线改变某些参数来达到目的。

实例 98 快速傅立叶变换

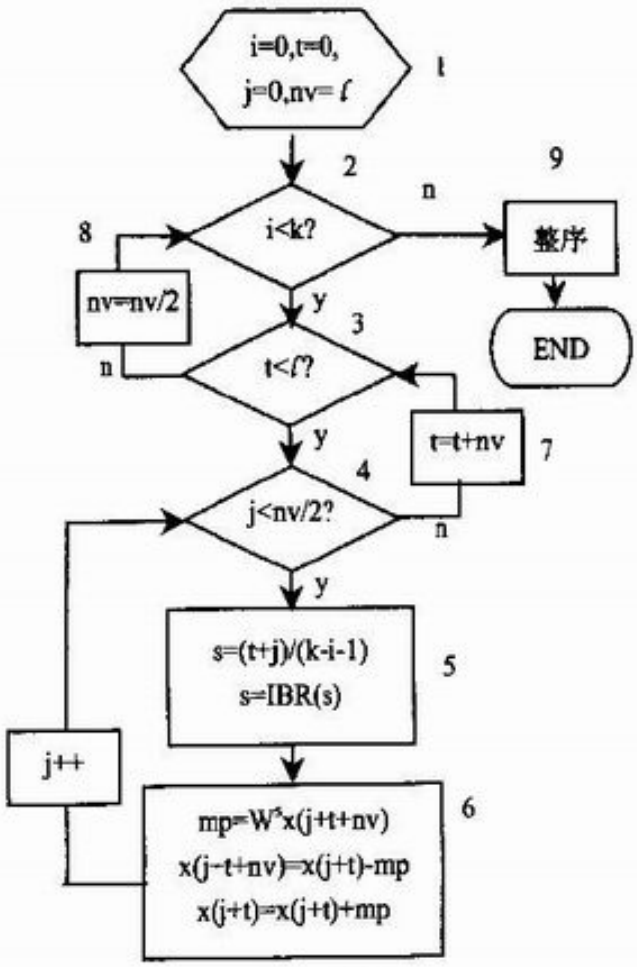
实例说明

本实例将实现二维快速傅立叶变换，这是第一次通过读程序流程图来设计程序，同时也将借此实例学习用 C 语言实现矩阵的基本操作、复数的基本操作，复习前面所学过的动态内存分配、文件操作、结构指针的函数调用等内容。

知识要点

很久以来，傅立叶变换一直是许多领域，如线性系统、光学、概率论、量子物理、天线、数字图像处理和信号分析等的一个基本分析工具，但是即便使用计算速度惊人的计算机计算离散傅立叶变换所花费的时间也常常是难以接受的，因此导致了快速傅立叶变换（FFT）的产生。下面将简要介绍快速傅立叶变换的实现方法。普通离散傅立叶变换方程为：

$$X(m)=\sum_{n=0}^{N-1}x(n)W^{mn} \quad W=e^{-j2\pi/l}$$



所谓快速傅立叶变换算法就是在研究离散傅立叶变换计算的基础上，节省计算量。理论基础是当 n 为 2 的倍数时，上面的方程可写为：

$$X(m) = X_1(m) + W^m{}_N X_2(m)$$

其中 $X_1(m)$ 为 $x_1(n) = x(2n)$ 的傅立叶变换，$X_2(m)$ 为 $x_2(n) = x(2n+1)$ 的傅立叶变换，可见一个 N 点的傅立叶变换可以由两个 N/2 点的傅立叶变换得到。这样分解后乘法次数大大减少，由原来的 N^2 次乘法减少到 $N^2/2 + N$ 次。当 $N=2^r$ 时，$X_1(m)$，$X_2(m)$又可以进行分解，此时直接算法的计算时间与 FFT 算法的计算时间之比有下列近似关系：$2N/r$。当 $N=1024=2^{10}$ 时可使计算量减少到原来的 1/200。FFT 的程序流程图如右上所示。各步骤说明如下：

（1）读入初始化数据，并对参数进行初始化；

（2）判断是否继续进行变换，对维数为 $l=2^k$ 的一组数据作 FFT 需要进行 k 轮变换操作；

（3）判断是否已经完成一轮变换；

（4）判断是否已完成一个区间的变换，nv 为一个区间的长度，在同一个区间内相隔 nv/2 的两个量称为对偶量，计算时一对对偶量只需进行一次乘法计算，见第 6 步；

（5）计算 W^s 中的 s，此式的推导较为复杂，请有兴趣的读者参考相关专著，其中 IBR(s) 是将 s 作为二进制数进行翻转，例如将 0010 变为 0100；

（6）计算一个对偶对的值，其中 x(j + t)和 x(j + t + nv)是对偶对；

（7）当一个区间已经完成变换后转移到下一区间，两个区间的第一个分量间相距 nv；

（8）每进行完一轮变换后，区间宽度减为原来的 1 / 2；

（9）快速傅立叶变换后并不是按照正序进行输出的，必须进行整序操作，既需要进行 IBR() 操作。

程序解析

本实例将对一个二维数组进行正、反快速傅立叶变换。正傅立叶变换时 dfft()函数先调用 fft ( )按行对数组进行变换，再对结果调用 fft()按列进行变换，此时完成了快速傅立叶变换，再调用 rdfft ( )函数进行傅立叶逆变换。如果程序设计正确的话，变换的结果应与原数组相同。

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#define  PI  3.14159265358979323846
struct  COMPLEX                    // 定义复数结构
{
    float  re ;
    float  im ;
} cplx , * Hfield , * S , * R, * w ;    // Hfield 指向将要被进行二维傅立叶变换操作
                                        //的数据块
                                  //S 指向将要做 FFT 的一维向量
//R 保存 FFT 后经过整序的结果
```

```
// w保存W^s表，进行第6步时只需直接从中调用W^s即可
int  n , m ;                    // 二维数组的长、宽
int  ln , lm ;                   // n=2^ln, m=2^lm
void  initiate ( ) ;
void  dfft ( ) ;                 // 进行二维傅立叶变换
void  rdfft ( ) ;                // 进行二维傅立叶逆变换
void  showresult ( ) ;
void  fft ( int l , int k ) ;          // 对S所指的一维向量进行FFT
int  reverse ( int t , int k ) ;       // 将t作为k位二进制数进行翻转
void  W ( int l ) ;              // 计算W^s
int  loop ( int l ) ;              // 通过n、m求ln、lm
void  conjugate ( ) ;            // 求复数矩阵的共轭矩阵
void  add (struct COMPLEX  * x , struct COMPLEX  * y , struct COMPLEX  * z);
void  sub (struct COMPLEX  * x , struct COMPLEX  * y , struct COMPLEX  * z);
void  mul (struct COMPLEX  * x , struct COMPLEX  * y , struct COMPLEX  * z);
struct COMPLEX *Hread ( int i , int j ) ;       // 读Hfield指向数据块指定位置，返回
                                                // 其指针
void  Hwrite ( int i , int j , struct COMPLEX x) ;// 向指定位置写
void  main ( )
{
    initiate ( ) ;
    printf ( "\n原始数据:\n" ) ;
    showresult ( ) ;
    getchar ( ) ;
    dfft ( ) ;                   // 进行二维离散傅立叶变换
    printf ( "\n快速傅立叶变换后的结果:\n" ) ;
    showresult ( ) ;
    getchar ( ) ;
    rdfft ( ) ;                  // 进行傅立叶逆变换
    printf ( "\n快速复利叶逆变换后的结果:\n" ) ;
    showresult ( ) ;
    getchar ( ) ;
    free ( Hfield ) ;
}
void  initiate ( )
{// 程序初始化操作，包括分配内存、读入要处理的数据、进行显示等
    FILE  * df ;
    df = fopen ( "data.txt" , "r" ) ;
    fscanf ( df , "%5d" , & n ) ;
    fscanf ( df , "%5d" , & m ) ;
    if ( ( ln = loop ( n ) ) == -1 )
    {
        printf ( " 列数不是2的整数次幂 " ) ;
        exit ( 1 ) ;
```

```
    }
    if ( ( lm = loop ( m ) ) == -1 )
    {
        printf ( " 行数不是 2 的整数次幂 " ) ;
        exit ( 1 ) ;
    }
    Hfield = ( struct COMPLEX * ) malloc ( n * m * sizeof ( cplx ) ) ;
    if ( fread ( Hfield , sizeof ( cplx ) , m * n , df ) != ( unsigned ) ( m
* n ) )
    {
        if ( feof ( df ) ) printf ( " Premature end of file " ) ;
        else printf ( " File read error " ) ;
    }
    fclose ( df ) ;
}
```

本实例所处理的数据是存储在子目录下的 data.txt 文件中，因为大多数情况下，C 程序的操作对象是存储在磁盘上的数据，例如在进行图像分析时就需要从硬盘上读取待分析的 BMP 文件，然后进行傅立叶变换，分析图像的频谱特性。

```
void dfft ( )
{           // 进行二维快速傅立叶变换
    int i , j ;
    int l , k ;
    l = n ;
    k = ln ;
    w = ( struct COMPLEX * ) calloc ( l , sizeof ( cplx ) ) ;
    R = ( struct COMPLEX * ) calloc ( l , sizeof ( cplx ) ) ;
    S = ( struct COMPLEX * ) calloc ( l , sizeof ( cplx ) ) ;
    W ( l ) ;
    for ( i = 0 ; i < m ; i++ )
    {       // 按行进行快速傅立叶变换
        for ( j = 0 ; j < n ; j++ )
        {
            S[ j ].re = Hread ( i , j )->re ;
            S[ j ].im = Hread ( i , j )->im ;
        }
        fft ( l , k ) ;
        for ( j = 0 ; j < n ; j++ )
            Hwrite ( i , j , R[ j ] ) ;
    }
    free ( R ) ;
    free ( S ) ;
    free ( w ) ;
    l = m ;
    k = lm ;
```

```
    w = ( struct COMPLEX * ) calloc ( l , sizeof ( cplx ) ) ;
    R = ( struct COMPLEX * ) calloc ( l , sizeof ( cplx ) ) ;
    S = ( struct COMPLEX * ) calloc ( l , sizeof ( cplx ) ) ;
    W ( l ) ;
    for ( i = 0 ; i < n ; i++ )
    {// 按列进行快速傅立叶变换
        for ( j = 0 ; j < m ; j++ )
        {
            S[ j ].re = Hread ( j , i )->re ;
            S[ j ].im = Hread ( j , i )->im ;
        }
        fft ( l , k ) ;
        for ( j = 0 ; j < m ; j++ )
            Hwrite ( j , i , R[ j ] ) ;
    }
    free ( R ) ;
    free ( S ) ;
    free ( w ) ;
}
```

根据傅立叶变换的可分离性，进行二维傅立叶变换时可先对 X(i)进行变换（即行变换），再对变换的结果进行 Y(i)变换（即列变换）。由于进行行、列变换时操作的对象都是一维向量，因此可用同一个 fft()函数实现，不同的是 fft()函数所处理的对象——S [ ]。将行向量付给 S [ ] 时进行的就是行变换，反之为列变换，经 fft()变换的结果存储在 R [ ]中。需要注意的是进行列变换时向量读写的次序。

```
void  rdfft ( )
{
    conjugate ( ) ;
    dfft ( ) ;
    conjugate ( ) ;
}
```

进行傅立叶逆变换。首先调用 conjugate()函数对数据矩阵进行求共轭操作，然后对结果进行正向傅立叶变换，再求一次共轭就是逆变换的结果。读者只需对比一下正反傅立叶变换的公式就可理解这样做的原因了。

```
void  showresult ( )
{
    int  i , j ;
    for ( i = 0 ; i < m ; i++ )
    {
        printf ( " \n第%d行\n " , i ) ;
        for ( j = 0 ; j < n ; j++ )
        {
            if ( j % 4 == 0 )  printf ( " \n " ) ;
```

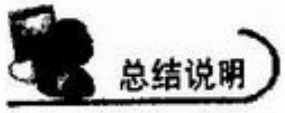
```
                printf ( " ( %5.2f,%5.2fi ) " , Hread ( i , j )->re , Hread ( i ,
j )->im ) ;
            }
        }
    }
    void  fft ( int l , int k )
    {
        int  i , j , s , nv , t ;
        float  c ;
        struct COMPLEX  mp , r ;
        nv = l ;
        c = ( float ) l ;
        c = pow ( c , 0.5 ) ;
        for ( i = 0 ; i < k ; i++ )
        {
            for ( t = 0 ; t < l ; t += nv )
            {
                for ( j = 0 ; j < nv / 2 ; j++ )
                {
                    s = ( t + j ) >> ( k - i -1 ) ;
                    s = reverse ( s , k ) ;
                    r.re = S[ t + j ].re ;
                    r.im = S[ t + j ].im ;
                    mul ( & w[ s ] , & S[ t + j + nv / 2 ] , & mp) ;
                    add (& r , & mp , & S[ t + j ] ) ;
                    sub (& r , & mp , & S[ t + j + nv / 2 ] ) ;
                }
            }
            nv = nv >> 1 ;
        }
        for ( i = 0 ; i < l ; i++ )
        {
            j = reverse ( i , k ) ;
            R[ j ].re = S[ i ].re / c ;
            R[ j ].im = S[ i ].im / c ;
        }
    }
```

对S[]进行快速傅立叶变换，具体的过程请参见前面的流程图。

```
    int  reverse ( int t , int k )
    {
        int  i , x , y ;
        y = 0 ;
        for ( i = 0 ; i < k ; i++ )
        {
```

```
        x = t & 1 ;
        t = t >> i ;
        y = ( y << 1 ) + x ;
    }
    return y ;
}
void W ( int l )
{
    int i ;
    float c , a ;
    c = ( float ) l ;
    c = 2 * PI / c ;
    for ( i = 0 ; i < l ; i++ )
    {
        a = ( float ) i ;
        w[ i ].re = ( float ) cos ( a * c ) ;
        w[ i ].im = -( float ) sin ( a * c ) ;
    }
}
int loop ( int l )
{   // 检验输入数据是否为 2 的整数次幂，如果是，返回用二进制表示的位数
    int i , m ;
    if ( l != 0 )
    {
        for ( i = 1 ; i < 32 ; i++ )
        {
            m = l >> i ;
            if ( m == 0 )
                break ;
        }
        if ( l == ( 1 << (i - 1 ) ) )
            return i - 1 ;
    }
    return -1 ;
}
void conjugate ( )
{   // 求复数矩阵的共轭矩阵
    int i , j ;
    for ( i = 0 ; i < m ; i++ )
        for ( j = 0 ; j < n ; j++ )
            Hread ( i , j )->im * = -1 ;
}
struct COMPLEX * Hread ( int i , int j )
{   // 按读矩阵方式返回 Hfield 中指定位置的指针
```

```
    return ( Hfield + i * n + j ) ;
}
void  Hwrite ( int i , int j , struct COMPLEX  x )
{    // 按写矩阵方式将复数结构 x 写到指定的 Hfield 位置上
    ( Hfield + i * n + j )->re = x.re ;
    ( Hfield + i * n + j )->im = x.im ;
}
void  add ( struct COMPLEX  * x , struct COMPLEX  * y , struct COMPLEX  * z )
{    // 定义复数加法
    z->re = x->re + y->re ;
    z->im = x->im + y->im ;
}
void  sub ( struct COMPLEX  * x , struct COMPLEX  * y , struct COMPLEX  * z )
{    // 定义复数减法
    z->re = x->re - y->re ;
    z->im = x->im - y->im ;
}
void  mul ( struct COMPLEX  * x , struct COMPLEX  * y , struct COMPLEX  * z )
{    // 定义复数乘法
   z->re = ( x->re ) * ( y->re ) - ( x->im ) * ( y->im ) ;
    z->im = ( x->im ) * ( y->re ) + ( x->re ) * ( y->im ) ;
}
```



总结说明

本实例实现了矩阵、复数及复数矩阵的基本操作，并复习了前面所学过的文件流操作、动态内存分配操作等知识点。需要说明的是，在 data.txt 文件中所存储的数据为二维复数矩阵，这与大多数实际情况是不同的。在应用中，傅立叶变换的对象大多为实数矩阵，我们选择复数矩阵的原因是为了使读者更好地观察变换过程，以及傅立叶逆变换的原理。细心的读者还会发现进行傅立叶变换后矩阵中大多数位置上的元素值为 0，我们称之为稀疏矩阵。产生这种现象的原因是 data.txt 文件中的数据呈现出一定的规律，即有周期性，当矩阵从时域变换到频域后，大部分的能量集中在一定的周期上，而其他周期上没有能量，因此对应的元素值为 0。进行逆变换后矩阵与原矩阵相同，表示程序运行正常。

实例 99 求解野人与传教士问题

实例说明

前面的实例大多把重点放在 C 语言在科学计算方面的应用。实际上，C 语言作为一种功能强大的计算机语言其应用范围也是非常广泛的，在本节及下一节中将用 C 语言实现简单的人工智能，使程序具备一定的推理能力。

本节先简单介绍人工智能中的搜索原理，再利用 C 语言实现简单的宽度搜索算法，实现自动对一个智力测验题求解，同时还将复习前面所学过的链表、堆栈的相关操作。

知识要点

搜索可以用图的概念进行解释：如果已知图中的两个点，以及图的连接方式，寻找从起始节点到目标节点的过程即为搜索。根据对节点的不同扩展方式和不同扩展顺序，有多种搜索算法可供选择，这里将要使用的是宽度优先搜索算法。

如果搜索是以接近起始节点的程度依次扩展节点的，那么这种搜索就叫做宽度优先搜索，也称为广度优先搜索，这种搜索是逐层进行的，同一层节点距离起始节点的距离相同；在对下一层的任意节点进行搜索之前，必须搜索完本层的所有节点。

宽度优先搜索算法如下：

- 把起始节点放到链表 UPOPENED 中（如果该起始节点为一目标节点，则求得一个解答）；
- 如果 UNOPENED 表示一个空表，则没有解，失败退出，否则继续；
- 把 UNOPENED 表的第一个节点 ntx 从 UNOPENED 表中移出，并把它放入 OPENED 的扩展节点表中。
- 扩展 ntx，如果没有后继节点则转向第 2 步。
- 把 ntx 的所有后继节点放到 UNOPENED 表的末端，并提供从这些后继节点回到 ntx 的指针。
- 如果 ntx 的任意个后继节点是目标节点，则找到一个解答，成功退出，否则转向第 2 步。

程序解析

题目：设有 n 个传教士和 m 个野人来到河边，打算乘一只船从右岸到左岸去。该船的负

载能力为两人。在任何时候，如果野人人数超过传教士人数，野人就会把传教士吃掉。他们怎样才能用这条船安全地把所有人都渡过河去?

➢ 题目分析：

✧ 定义节点的结构：以取船的一个来回作为一步搜索，这样节点可由下面几个量进行描述：两岸的传教士人数和野人人数、本节点距离起始节点的距离，即由初始节点搜索几步后到达本节点。需要注意的是并不是所有节点都是可达的，题目中对可达节点作出了限制，只有两岸上传教士人数都不少于野人人数（无传教士除外）的节点才是可达的，另外岸上的人数必须不能为负。

✧ 定义连接：两个节点间的连接可由船从右到左岸时船上的传教士人数、野人人数、船返回时船上的传教士人数、野人人数进行描述。由于船上最多只能载两个人，因此不存在限制问题。

✧ 有4种连接可供选择：两个人乘船到左岸，一个人返回；两个人到左岸，两个人返回；一个人到左岸，两个人返回；一个人到左岸，一个人返回。考虑到本实例的重点在于讲述搜索算法，因此程序中只考虑了第一种连接，在后面对 stretch()函数的讲解中将会给出考虑4种连接的程序的修改方法。

程序源代码及讲解：

```
#include <stdio.h>
#include <stdlib.h>
#define maxloop 100     // 最大搜索距离
#define pristnum 3      // 传教士的默认初始人数
#define slavenum 3      // 野人的默认初始人数
struct SPQ
{
    int sr,pr;          // 船运行一个来回后河右岸的野人、传教士的人数
    int sl,pl;          // 船运行一个来回后河左岸的野人、传教士的人数
    int ssr,spr;        // 回来（由左向右时）船上的人数
    int sst,spt;        // 去时（由右向左时）船上的人数
    int loop;           // 本结点所在的层数，及到初始节点的距离
    struc SPQ *upnode ,*nextnode;// 本结点的父结点和同表中的下一个结点的地址
}spq;
// 为了使程序更加简明，这里将节点及达到这个节点的连接合并，由结构体 SPQ 表示。
int loopnum;            // 记录总的扩展次数
int openednum;          // 记录已扩展节点个数
int unopenednum;        // 记录待扩展节点个数
int resultnum;          // 记录结果链表中节点的个数
struc SPQ *opened;          // 已扩展链表头
struc SPQ *oend;            // 已扩展链表尾
struc SPQ *unopened;        // 待扩展链表头
struc SPQ *uend;            // 待扩展链表尾
struc SPQ *result;          // 结果链表头
```

```
void initiate();          // 初始化搜索程序
int search();             // 进行搜索
int stretch(struc SPQ* ntx);          // 扩展节点
void recorder();          // 记录搜索到的解
void releasemem();        // 释放占用内存
void showresult();        // 显示解
void addtoopened(struc SPQ *ntx);      // 将节点 ntx 从 UNOPENED 链表移至 OPRNENED
                                       // 链表中
void goon();              // 判断是否继续搜索
void main()
{
    int flag;             // 标记扩展是否成功
    for(;;)
    {
        initiate ();
        flag = search ();
        if ( flag == 1 )
        {
            recorder ();
            releasemem ();
            showresult ();
            goon ();
        }
        else
        {
            printf("无法找到符合条件的解");
            releasemem ();
            goon ();
        }
    }
}
void initiate()
{
    int x;
char choice;
    uend = unopened = (struc SPQ*)malloc(sizeof(spq));
    if(uend==NULL)
    {
        printf("\n 内存不够! \n");
        exit(0);
    }
    unopenednum=1;
    openednum=0;
    unopened -> upnode = unopened;        // 保存父结点的地址以成链表
```

```
    unopened -> nextnode = unopened;
    unopened -> sr = slavenum;
    unopened -> pr = pristnum;
    unopened -> sl = 0;
    unopened -> pl = 0;
    unopened -> sst = 0;
    unopened -> spt = 0;
    unopened -> ssr = 0;
    unopened -> spr = 0;
    unopened -> loop = 0;
    printf("题目：设有 n 个传教士和 m 个野人来到河边，打算乘一只船从右岸到\n",
"左岸去。该船的负载能力为两人。在任何时候，如果野人人数\n",
"超过传教士人数，野人就会把传教士吃掉。他们怎样才能用\n",
"这条船安全地把所有人都渡过河去?\n");
    printf("默认的 n、m 值皆为 3");
  for(;;)
{
        printf("是否修改? (Y/N)");
        scanf ("%s",& choice);
        choice = toupper (choice);
        if(choice == 'Y')
        {
            printf("请输入传教士人数");
            for( ; ; )
            {
                scanf(" %d " , &x);
                if(x > 0)
                {
                    unopened -> pr = x;
                    break;
                }
                else printf("\n 输入值应大于 0! \n 请重新输入");
            }
            printf("请输入野人人数");
            for(;;)
            {
                scanf("%d",&x);
                if(x>0)
                {
                    unopened -> sr = x;
                    break;
                }
                else printf("\n 输入值应大于 0! \n 请重新输入");
            }
```

```
            break;
        }
        if(choice=='N')break;
    }
}
int search()
{
    int flag;
    struc SPQ *ntx;              // 提供将要扩展的结点的指针
    for( ; ; )
    {
        ntx = unopened;          // 从待扩展链表中提取最前面的一个
        if(ntx->loop == maxloop)
            return 0;
        addtoopened(ntx);        // 将加入已扩展链表，并从待扩展链表中去掉
        flag = stretch(ntx);     // 对ntx进行扩展
        if(flag == 1)
            return 1;
    }
}
```

search()函数实现对问题解的搜索过程，它首先从待扩展链表中取出链表的第一个节点，将其放入已扩展链表的尾部，然后调用stretch()函数对这个节点进行扩展。当扩展的节点距初始节点距离已经达到事先规定的最大搜索距离时判定搜索失败。

```
int stretch(struc SPQ *ntx)
{
    int fsr , fpr ;  // 在右岸上的人数
    int fsl , fpl ;  // 在左岸上的人数
    int sst , spt ;  // 出发时在船上的人数
    int ssr , spr ;  // 返回时船上的人数
    struc SPQ *newnode;
    for (sst = 0 ; sst <=  2 ; sst++)
{   // 循环中的上限“2”代表从左岸到右岸船上的总人数，当要考虑4种情况
    //时可用一变量代替，取值范围为1、2。
        fsr = ntx -> sr;
        fpr = ntx -> pr;
        fsl = ntx -> sl;
        fpl = ntx -> pl;
        if ((sst <=  fsr) && (( 2 - sst) <=  fpr))//满足人数限制
        {
            spt = 2 - sst;
            fsr = fsr - sst;
            fpr = fpr - spt;
            if((fpr ==  0) && (fsr ==  0))
            {                                    // 右岸人数为0，搜索成功
```

```
        newnode = (struc SPQ*) malloc (sizeof(spq));
        if(newnode==NULL)
        {
            printf("\n 内存不够! \n");
            exit(0);
        }
        newnode -> upnode = ntx;        // 保存父结点的地址以成链表
        newnode -> nextnode = NULL;
        newnode -> sr = 0;
        newnode -> pr = 0;
        newnode -> sl = opened -> sr;
        newnode -> pl = opened -> pr;
        newnode -> sst = sst;
        newnode -> spt = spt;
        newnode -> ssr = 0;
        newnode -> spr = 0;
        newnode -> loop = ntx -> loop + 1;
        oend -> nextnode = newnode;
        oend = newnode;
        openednum++;
        return 1;
    }
    else if ((fpr - fsr) * fpr >= 0)
    {  // 判断是否满足传教士人数大于或等于野人人数的要求
        fsl = fsl + sst;
        fpl = fpl + spt;
        for (ssr = 0 ; ssr <= 1 ; ssr++)
        {
            int ffsl , ffpl;
            if ((ssr <= fsl) && ((1 - ssr) <= fpl))
            {
                spr = 1 - ssr;
                ffsl = fsl - ssr;
                ffpl = fpl - spr;
                if ((ffpl - ffsl) * ffpl >= 0)
                {   // 若符合条件则分配内存并赋值
                    int  ffsr , ffpr;
                    ffsr = fsr + ssr;
                    ffpr = fpr + spr;
                    newnode = (struc SPQ*) malloc (sizeof(spq));
                    if(newnode==NULL)
                    {
                        printf("\n 内存不够! \n");
                        exit(0);
```

```
                    }
                    newnode -> upnode = ntx;
                  newnode -> sr = ffsr;
                    newnode -> pr = ffpr;
                    newnode -> sl = ffsl;
                    newnode -> pl = ffpl;
                newnode -> sst = sst;
                newnode -> spt = spt;
                    newnode -> ssr = ssr;
                    newnode -> spr = spr;
                    newnode -> loop = ntx -> loop + 1;
                    uend -> nextnode = newnode;
                    uend = newnode;
                unopenednum++;
                }
            }
        }
    }
}
    return 0;
}
```

stretch( )函数负责对具体的节点进行展开，展开的过程必须满足对节点的限制条件，除目标节点之外其他新展开的节点被依次存入 UNOPENED 链表的尾部，目标节点由于已没有必要再进行扩展，因此直接存入 OPENED 链表中。

```
void addtoopened(struc SPQ *ntx)
{
    unopened = unopened -> nextnode;
    unopenednum--;
    if (openednum == 0 )
        oend = opened = ntx;
    oend -> nextnode = ntx;
    oend = ntx;
    openednum++;
}
```

addtoopened( )函数将节点 ntx 从 UNOPENED 链表中去除，将其后继节点作为 UNOPENED 链表的头节点，并将 ntx 存入 OPENED 链表的尾部，表示此节点已被展开。

```
void recorder()
{
    int i , loop;
    struc SPQ * newnode , *ntx;
    loop = oend -> loop;
    ntx = oend;
```

```
    resultnum = 0;
    for( i = 0 ; i <= loop ; i++ )
    {
        newnode = (struc SPQ*) malloc (sizeof(spq));
        if(newnode==NULL)
        {
            printf("\n内存不够! \n");
            exit(0);
        }
        newnode -> sr = ntx -> sr;
        newnode -> pr = ntx -> pr;
        newnode -> sl = ntx -> sl;
        newnode -> pl = ntx -> pl;
        newnode -> sst = ntx -> sst;
        newnode -> spt = ntx -> spt;
        newnode -> ssr = ntx -> ssr;
        newnode -> spr = ntx -> spr;
        newnode -> nextnode = NULL;
        ntx = ntx -> upnode;
        if(i == 0)
            result = newnode;
        newnode -> nextnode = result;
        result = newnode;
        resultnum++;
    }
}
```

recorder( )函数在搜索成功后负责从目标节点延已扩展出的树状结构反向寻找到解路径，并以压堆栈的形式存入以 result 节点为顶节点的链表中。

```
void releasemem()
{
    int i;
    struc SPQ* nodefree;
    for ( i = 1 ; i < openednum ; i++ )
    {
        nodefree = opened;
        opened = opened -> nextnode;
        free(nodefree);
    }
    for ( i = 0 ; i < unopenednum ; i++ )
    {
        nodefree = unopened;
        unopened = unopened -> nextnode;
        free(nodefree);
    }
```

```
}
```

releasemem()释放 OPENED 和 UNOPRENED 两个链表所占用的内存。

```
void showresult()
{
    int i;
int fsr , fpr ;     // 在右岸上的人数
    int fsl , fpl ; // 在左岸上的人数
    int sst , spt ; // 出发时船上的人数
    int ssr , spr ; // 返回时船上的人数
    struc SPQ * nodefree;
    printf("河的右岸有%d个传教士" , result -> pr);
    printf("%d个野人  \n" , result -> sr);
printf("河的左岸有%d个传教士" ,result -> pl);
printf("%d个野人  \n" , result -> sl);
    for ( i = 1 ; i < resultnum ; i++ )
    {
        nodefree = result;
        result = result -> nextnode;
        free(nodefree);
        printf("\n\t 左岸人数 船上人数及方向 右岸人数\n");
        printf("第%d轮\n",i);
        fpl = result -> pl - result -> spt + result -> spr;
        fpr = result -> pr - result -> spr;
        fsl = result -> sl - result -> sst + result -> ssr;
  fsr = result -> sr - result -> ssr;
  spt = result -> spt;
  sst = result -> sst;
        printf("传教士 %8d %8d \t <- \t %8d \n" , fpl, spt , fpr);
        printf("野  人 %8d %8d \t <- \t %8d \n" , fsl, sst , fsr);
  fpl = result -> pl;
        fpr = result -> pr - result -> spr;
        fsl = result -> sl;
  fsr = result -> sr - result -> ssr;
  spr = result -> spr;
  ssr = result -> ssr;
        printf("传教士 %8d %8d \t -> \t %8d \n" , fpl , spr , fpr);
        printf("野  人 %8d %8d \t -> \t %8d \n" , fsl , ssr , fsr);
    }
    printf("\n全体传教士和野人全部到达对岸");
    free(result);
}
```

showresult()函数负责显示搜索的结果，并同步释放 RESULT 链表所占用的内存。

```
void goon()
{
```

```
    char choice;
    for(;;)
    {
        printf("是否继续? (Y/N)\n");
        scanf ("%s",&choice);
        choice=toupper(choice);
        if(choice=='Y')break;
        if(choice=='N')exit(0);
    }
}
```

总结分析

在实现宽度优先搜索算法的过程中一共使用了四个链表，即 OPENED 链表、UNOPENED 链表、RESULT 链表及一个隐含的以展开关系形成的链表，这里称为父子链表。父子链表记录了搜索的全过程，当搜索成功时目标节点即为这个链表的最下层节点，由此链表可倒推至起始节点。从数据输入输出角度来看，RESULT 链表和父子链表同为堆栈结构，当进行搜索时展开的节点被压入父子链表形成的堆栈；当搜索成功后其中以目标节点为顶节点的堆栈开始弹出，而同时压入 RESULT 堆栈；在显示时节点信息又从 RESULT 堆栈弹出，这样经过两次堆栈操作，搜索结果被以正序显示出来。UNOPENED 链表则以队列形式出现，在 stretch()函数中新展开的节点被加到 UNOPENED 队列的尾端，在 search()函数中 UNOPENED 队列的首元素被用于下一次展开。

实例100 简单专家系统

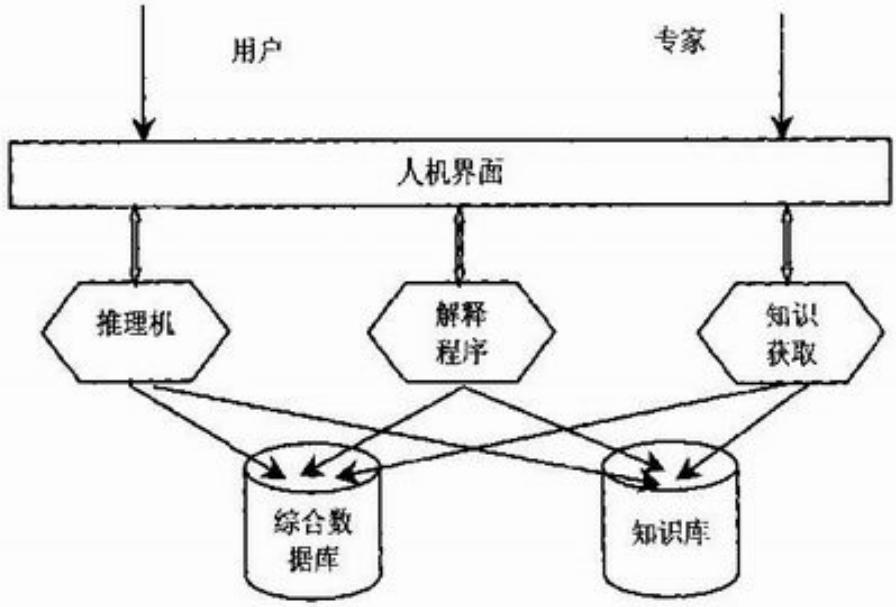
实例说明

本例将要介绍一个简单的专家系统。通过这个实例一方面复习前面所学过的结构、链表、指针数组、文件的标准化读写、函数递归调用等知识，其中重点介绍专家系统数据结构的设计；另一方面将介绍怎样构造一个简单专家系统，并实现一个动物自动识别专家系统，实现知识的表示、学习、与运用。

知识要点

专家系统（Expert System，简记 ES），也称为基于知识的系统，是人工智能的一个最为重要的应用领域，它是一种利用人类专家经验解决实际问题的智能化程序。专家系统产生于 20 世纪 60 后代中期，经过 20 多年的科学研究，理论和技术日臻成熟，其应用得到了飞速发展。至今，世界各国已经在医疗诊断、化学工程、语音识别、图像处理、金融决策、信号解释、地质勘探、石油、军事等领域研制出了大量的实用专家系统，其中不少系统在性能上已达到甚至超过了同领域人类专家的水平，已经产生或正在产生巨大的经济效益和社会影响。

虽然由于应用方向不同，各领域的专家系统表现出极大的不同，但其基本结构是相似的可以由下图表示。



专家系统各组成部分的功能可以归纳如下：

- 知识库：用于存储某领域专家系统的专门知识，包括实事、可执行操作、规则等。
- 综合数据库：用于存储领域或问题的初始数据和推理过程中得到的中间数据。
- 推理机：用于记忆所采用的规则和控制策略的程序，使整个专家系统能够以逻辑方式协调地工作。
- 解释程序：向用户解释专家系统的行为，包括解释推理结论的正确性及系统推理根据等。
- 知识获取：专家系统的学习过程。
- 人机接口：将专家或用户的输入信息翻译为系统可按受的内部形式，把系统向专家或用户输出的信息转换成人类易于理解的外部形式。

程序解析

- 题目要求：

实现一个动物识别专家系统，这个系统应具有一定的有关动物特点的知识，当用户输入对某动物的描述时，能够根据已有的知识，自动判断出所描述的是什么动物；同时应具有学习功能，允许用户在线扩充专家系统的知识库。系统原本具有的知识如下：①有毛发的动物是哺乳类；②有奶的动物是哺乳类；③有羽毛的动物是鸟类；④若动物会飞且生蛋则是鸟类；⑤吃肉的哺乳类称为食肉动物；⑥反刍食物的哺乳类是偶蹄类；⑦有蹄的哺乳类是有蹄类；⑧黄褐色有暗斑点的食肉类是金钱豹；⑨黄褐色有黑色条纹的食肉类是老虎；⑩尖牙利爪且眼睛向前的是食肉动物；⑪脖有黄褐色暗斑点的有蹄类是长颈鹿；⑫纹的有蹄类是斑马；⑬长脖的鸟类是鸵鸟；⑭泳的鸟类是企鹅。要求系统实现后对下列知识进行学习：①不会飞是某一动物为鸵鸟的必要条件之一，也是企鹅的必要条件之一；②善飞的鸟类是信天翁。

- 题目分析：

- 系统的知识有继承关系，例如：有毛发的动物是哺乳类，而吃肉的哺乳类称为食肉动物，可推出有毛发且吃肉的动物为食肉动物。为了叙述方便，将因果关系中的条件称为父节点，结论称为子节点，一个因果关系中的父节点可能成为另一关系中的子节点，这样，知识系统就可视为一个树状结构；
- 同一子节点的多个父节点间有两种关系，即“与”关系和“或”关系，当某一子节点的父节点间同时具有两种关系时，为了新建一个“与”节点，将“与”关系的父节点的输出作为新的节点的输入，节点的输出作为原子节点的输入。例如，如果一个动物“尖牙利爪并且眼睛向前”则是一个食肉动物，同时如果一个“哺乳动物吃肉”亦可判定为食肉动物，这时可以引入两个“与”节点，“尖牙利爪并且眼睛向前”和“吃肉的哺乳动物”分别作为两个节点的输入，而两个节点的输出则作为“食肉”节点的输入，成“或”关系。
- 建立节点的数据结构是系统实现中一个非常关键的问题，是本节的重点之一，也是专

家系统成功的关键。选择一个合理的数据结构将有利于推理机的实现和知识库的扩充。节点的数据结构应该包含节点所代表的意义，同时应包含节点的激活信息。激活信息表明节点所代表的意义是否有效，例如，“有蹄类”的必要条件：“哺乳类”和“有蹄”都得到满足时，这一节点有效，即被激活，此时若“黑色条纹”已有效，则“斑马”节点被激活。节点的激活信息由 fullfill 描述，当取值为 1 时表示此节点被激活，同时节点被激活的信息将被传送到其各个子节点。当子节点为“或”节点时，子节点被激活，若为“与”节点则应记录下这一信息，当子节点的所有父节点都被激活时子节点被激活。用 state 记录这一信息，节点的一个父节点被激活，tate 的值增 1，当 state 的值与父节点的个数 upnum 相等时表明所有的父节点都已被激活，子节点被激活。根据前面的分析可以看出，节点数据结构还应该能有效地记录与之相连的父节点信息、子节点信息、节点类型（是“或”节点还是“与”节点）。父节点信息由 upnum 和链表 upnode 来进行描述，其中 upnum 代表父节点的个数；链表 upnode 则记录了父节点的地址信息，它由结构 SUBLINK 定义，结构中元素 index 指向数组 nodelink[] 中记录父节点地址的位置。当节点作为专家系统的知识从硬盘上加载到系统中时，每个节点的地址都被按固定的顺序记录在 nodelink 数组中，因此只要知道节点地址在数组中的位置就可以找到节点本身。子节点信息的保存与父节点一致，这里不再详述。

➢ 源代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct SUBLINK
{
    int index;
    struct SUBLINK *next;
}sublink;
struct NODE
{
    char feature[20];
    int upnnum;
    struct SUBLINK *upnode;
    int state;
    int type;
    int fullfill;
    int sonnum;
    struct SUBLINK *sonnode;
}node;
#define MAXNUM 1000
struct NODE *nodelink[MAXNUM];
int nodenum;
// 略函数预定义
```

```
void main()
{
int flag;
initiate(); // 入知识库
for(;;)
{
    printf("请选择操作：\n1、查询;\n2、添加新知识;\n3、退出程序.");
    scanf("%d" , &flag);
    switch(flag)    // 根据用户输入选择需要进行的操作
    {
    case 1:
        quarry();   // 进行查询
        break;
    case 2:
        modify();   // 增添知识库
        break;
    case 3:
        store();    // 保存对知识库的修改并退出
    default:
        printf("\n 输入错误，请重新输入\n");
    }
}
}
void initiate()     // 初始化操作
{
    int i , j;
    char s[10];
    FILE *kf;
    struct NODE *newnode;
    struct SUBLINK *newlink , *oldlink;
    if((kf = fopen("knowledgestore.txt" , "r")) == NULL)
    {
        printf("Cannot create/open file");
        exit(1);
    }
    fscanf(kf , "%5d" , &nodenum);      // 读入节点数
    for(i = 0 ; i < nodenum ; i++)      // 顺序读入 nodenum 个节点
    {
        newnode = (struct NODE*)malloc (sizeof(node));
        if(newnode == NULL)
        {
            printf("\n 内存不够! \n");
            exit(0);
        }
```

```
        fscanf(kf , "%20s" , &newnode->feature);
        fscanf(kf , "%5d" , &newnode->upnnum);
        for(j = 0 ; j < newnode->upnnum ; j++)
        {
            newlink = (struct SUBLINK*)malloc(sizeof(sublink));
            if(newlink == NULL)
            {
                printf("\n 内存不够! \n");
                exit(0);
            }
            fscanf(kf , "%5d" , &newlink->index);
            if(j == 0)
                newnode->upnode = oldlink = newlink;
            newlink->next = NULL;
            oldlink->next = newlink;
            oldlink = newlink;
        }
        newnode->state = 0;          // 初始条件下父节点激活状态为 0
        newnode->fullfill = 0;       // 初始条件下节点激活状态为 0
        fscanf(kf , "%5d" , &newnode->type);
        fscanf(kf , "%5d" , &newnode->sonnum);
        for(j = 0 ; j < newnode->sonnum ; j++)
        {
            newlink = (struct SUBLINK*)malloc(sizeof(sublink));
            if(newlink == NULL)
            {
                printf("\n 内存不够! \n");
                exit(0);
            }
            fscanf(kf , "%5d" , &newlink->index);
            if(j == 0)
                newnode->sonnode = oldlink = newlink;
            newlink->next = NULL;
            oldlink->next = newlink;
            oldlink = newlink;
        }
        nodelink[i] = newnode;
    }
    fscanf(kf , "%10s" , s);
    if(strcmp(s , "end") != 0)       // 判断载入是否成功，若不成功则退出程序
    {
        printf("\n 程序初始化失败! ");
        exit(0);
    }
```

```
}
void quarry()
{
    struct NODE *ntx;
    char feature[100];
    int i , flag;
    for(;;)
    {
        flag = 0;
        printf("\n请输入动物的特征:");
        scanf("%s" , feature);
        for(i = 0 ; i < nodenum ; i++)
        {
            ntx = nodelink[i];
            if(strstr(feature , ntx->feature) != NULL)
            {   // 寻找与输入相匹配的特征，若相匹配则将相应节点激活
                ntx->fullfill = 1;
                flag = extend(ntx);     // 将激活信息向下传播
            }
        }
        if(flag >= 1)       // 代表已经搜索到叶节点，即不再有子节点的节点，搜索成功
        {
            for(i = 0 ; i < nodenum ; i++)
            {   // 将各节点的激活状态清零，去除记忆效应
                nodelink[i]->state = 0;
                nodelink[i]->fullfill = 0;
            }
            break;
        }
        if(flag == 0)
            if(showfault() == 0)break;//若继续进行搜索则保留原有激活纪录
    }
}
```

quarry()函数并不直接进行解的搜索，它的主要功能是用已知的知识同用户的输入相匹配，从而将用户的自然语言转化为专家系统中使用的内部语言，即本实例中节点的激活状态，完成一个输入接口的功能。当用户输入的信息不完整时，例如只输入“尖牙利爪”，系统将无法做出判断，或者输入“有毛发”，系统只能推出“哺乳类”的结论，此时将保存原节点的激活状态，要求用户输入更多的信息，以做出进一步的判断。

```
int extend(struct NODE *ntx)
{
    int i , index;
    int flag;
    struct NODE *nextone;
```

```
    struct SUBLINK *son;
    if(ntx->sonnum == 0)
    {   // 当节点没有子节点时认为已经搜索到最终的解，停止进一步搜索
        printf("\n 结果为%20s\n" , ntx->feature);
        return 1;
    }
    son = ntx->sonnode;
    flag = 0;
    for(i = 0 ; i < (ntx->sonnum) ; i++)
    {
        index = son->index;
        nextone = nodelink[index];
        if(nextone->type == 0)
        {   // 当节点为“或”节点时，节点被激活，并向其子节点传递激活信息
            if(nextone->fullfill!= 1)
            {
                nextone->fullfill = 1;
                printf("\n 表明具有%20s 特征" , nextone->feature);
                flag += extend(nextone);
            }
        }
        else
        {   // 当节点为“与”节点时，节点的父节点激活记录(state)加 1
            nextone->state++;
            if(nextone->state == nextone->upnnum)
            {   // state 与父节点数相等时表示所有的父节点都已被激活，子节点被激活
                nextone->fullfill = 1;
                printf("\n 表明具有%20s 特征" , nextone->feature);
                flag += extend(nextone);
            }
        }
        son = son->next;
    }
    return flag;
}
```

extend()函数是实现查询功能的核心，它采用递归调用方式，将节点的激活信息向下传播，具体的实现原理见题目分析中的第 3 项。

```
void modify()
{
    int i ;
    char choice , feature[100];
    struct NODE *ntx , *newnode;
    struct SUBLINK *end1 , *new1;
    newnode = (struct NODE*)malloc(sizeof(node));
```

```
    if(newnode == NULL)
    {
        printf("\n 内存不够! \n");
        exit(0);
    }
    newnode->sonnum = 0;    // 初始化新建节点
    newnode->upnnum = 0;
    newnode->state = 0;
    printf("\n 请输入新特征\n");
    scanf("%s" , newnode->feature);
    printf("新特征类型: \n 与节点(1) , 或节点(0)");
    scanf("%d" , &newnode->type);
    newnode->fullfill = 0;
    newnode->state = 0;
// 使新建节点与知识库中的其他节点建立连接
    for(;;)     // 建立与子节点的连接
    {
        printf("\n 是否为叶节点? (Y/N)\n");
        scanf("%s" , &choice);
        choice = toupper(choice);
        if(choice == 'N')
        {   // 如果新建节点不是叶节点则与子节点建立连接
            printf("\n 请输入新特征描述的对象\n");
            scanf("%s" , feature);
            for(i = 0 ; i < nodenum ; i++)
            {   // 寻找匹配的子节点
                ntx = nodelink[i];
                if(strstr(feature , ntx->feature) != NULL)
                {   // 记录子节点信息
                    newl = (struct SUBLINK*)malloc(sizeof(sublink));
                    if(newl == NULL)
                    {
                        printf("\n 内存不够! \n");
                        exit(0);
                    }
                    if(newnode->sonnum == 0)
                        newnode->sonnode = endl = newl;
                    newl->index = i;
                    endl->next = newl;
                    endl = newl;
                    newl->next = NULL;
                    newnode->sonnum++;
                  // 将信息写入子节点
                    newl = (struct SUBLINK*)malloc(sizeof(sublink));
```

```
                    if(new1 == NULL)
                    {
                        printf("\n 内存不够! \n");
                        exit(0);
                    }
                    if(ntx->upnnum == 0)
                        ntx->upnode = end1 = new1;
                    new1->index = nodenum;
                    new1->next = ntx->upnode;
                    ntx->upnode = new1;
                    ntx->upnnum++;
                }
            }
            break;
        }
        if(choice == 'Y')break;
    }
    for(;;)          // 建立与父节点的连接
    {
        printf("\n 是否为顶点? (Y/N)\n");
        scanf("%s" , &choice);
        choice = toupper(choice);
        if(choice == 'N')
        {
            printf("\n 请输入对新对象的描述\n");
            scanf("%s" , feature);
            for(i = 0 ; i < nodenum ; i++)
            {
                ntx = nodelink[i];
                if(strstr(feature , ntx->feature)!=NULL)
                {
                    new1 = (struct SUBLINK*)malloc(sizeof(sublink));
                    if(new1 == NULL)
                    {
                        printf("\n 内存不够! \n");
                        exit(0);
                    }
                    if(newnode->upnnum == 0)
                        newnode->upnode = end1 = new1;
                    new1->index = i;
                    end1->next = new1;
                    end1 = new1;
                    new1->next = NULL;
                    newnode->upnnum++;
```

```
        //将信息写入父节点
                    newl = (struct SUBLINK*)malloc(sizeof(sublink));
                    if(newl == NULL)
                    {
                        printf("\n 内存不够! \n");
                        exit(0);
                    }
                    if(ntx->sonnum == 0)
                        ntx->sonnode = endl = newl;
                    newl->index = nodenum;
                    newl->next = ntx->sonnode;
                    ntx->sonnode = newl;
                    ntx->sonnum++;
                }
            }
            break;
        }
        if(choice == 'Y')break;
    }
    nodelink[nodenum] = newnode;          // 学习成功后将新的节点信息记录到 nodelink
                                          //数组中
    nodenum++;        // 以供其他节点调用
}
```

modify()函数完成对新知识的学习过程，其中的关键是成功的建立与其他节点的联系，包括与父节点的联系和与子节点的联系，在建立联系的过程中不但要记录其他节点的信息，还要将自身的信息写入到与之相连的节点中，这样才能保证激活信息的正确传播。最后不要忘记将节点地址进行“登记”：添加到 nodelink 数组的相应位置，因为各节点间的联系是通过使用相应的 index 到 nodelink 中去查找地址来实现的。

```
void store()
{
    int i , j;
    char s[10];
    FILE *kf;
    struct NODE *writenode;
    struct SUBLINK *newlink , *oldlink;
    if((kf = fopen("knowledgestore.txt" , "w")) == NULL)
    {
        printf("Cannot create/open file");
        exit(1);
    }
    fprintf(kf , "%5d" , nodenum);
    for(i = 0 ; i < nodenum ; i++)
    {
```

```
            writenode = nodelink[i];
            fprintf(kf , "%20s" , writenode->feature);
            fprintf(kf , "%5d" , writenode->upnnum);
            newlink = writenode->upnode;
            for(j = 0 ; j < writenode->upnnum ; j++)
            {
                fprintf(kf , "%5d" , newlink->index);
                oldlink = newlink;
                newlink = newlink->next;
                free(oldlink);
            }
            fprintf(kf , "%5d" , writenode->type);
            fprintf(kf , "%5d" , writenode->sonnum);
            newlink = writenode->sonnode;
            for(j = 0 ; j < writenode->sonnum ; j++)
            {
                fprintf(kf , "%5d" , newlink->index);
                oldlink = newlink;
                newlink = newlink->next;
                free(oldlink);
            }
            free(writenode);
        }
        strcpy(s , "end");
        fprintf(kf , "%10s" , s);
        fclose(kf);
        exit(0);
    }
```

store()函数将现有的知识库保存到硬盘上，以便下次运行程序时能够使用最新的知识库，为了顺利完成程序初始化，store()函数保存数据的格式和 initiate()读取数据的格式必须一致，并且 store()函数在文件头要写入文件数据的信息即 nodenum，当 initiate()函数初始化专家系统时就按 nodenum 读取节点信息、判断文件结尾及文件完整性。最后写入的“end”保证所读取文件没有遭到破坏，这个功能也可以通过前面学过的文件操作中的 feof()函数实现。

```
    showfault()
    {
        char choice;
        for(;;)
        {
            printf("是否继续? (Y/N)\n");
            scanf("%s" , &choice);
            while(choice == '10');
            choice = toupper(choice);
            if(choice == 'Y')return 1;
```

```
            if(choice == 'N')exit(0);
        }
    }
```

总结分析

恭喜大家！到此为止本书中最复杂的程序已经完成了，为了验证它的正确性可以作以下试验：

- 验证查询功能：进入系统后按提示键入 1，并输入：有毛发而且吃肉并黄褐色有暗斑点

此时系统显示：

表明具有	哺乳类特征
表明具有	and 特征
表明具有	食肉特征
表明具有	金钱豹特征
结果为	金钱豹

说明系统能够正确执行查询功能，读者还可以将上述特征分三次输入，看系统的反应。

- 验证系统的学习能力：使系统对下列知识进行学习。不会飞是某一动物为鸵鸟的必要条件之一，也是企鹅的必要条件之一；善飞的鸟类是信天翁。分析可知“不会飞”是一个新的节点，它是节点“鸵鸟”、“企鹅”的父节点之一，因此进入 modify 功能，按提示输入新节点为“不会飞”，type 设定为 0，是否为叶节点输入“n”，输入“鸵鸟和企鹅”，第一个节点添加完毕。对于第二个知识点，善飞和信天翁都应成为新的节点，二者的输入次序任意，只须注意“信天翁”是鸟类的子节点即可，节点性质为“与”节点。此后可进入 quarry 功能，按提示输入“有羽毛且善飞”，验证系统的学习能力。
- 验证系统的保存与初始化功能：验证过系统的学习能力后，按系统提示键入“3”退出程序，重新运行程序，输入“有羽毛但不会飞而且长腿长脖”，系统输出“鸵鸟”证明系统的保存与初始化功能运行正确。