

C 语言版

西南交通大学计算机基础教育课程

信息科学与技术学院

程序设计 综合实验教程

龚 勋

信息科学与技术学院-软件专业

程序设计综合实验教程

© 西南交通大学
四川成都市二环路北 1 段 • 111 号
电话 028.8646.6426 • 传真 028.8760.0743

前 言

程序设计是信息学科的一门重要专业基础课程，是计算机、软件工程、信息安全、网络等相关专业学生必需掌握的基本功，动手能力、编程能力在当前计算机学科教育中已经被提高到了十分重要的地位。在学习了高级语言程序设计的基础上，开展程序实验课程可以进一步提高学生的编程功底，让学生逐步掌握软件开发的基本思想，为将来走上工作岗位打下坚实的基础。本书正是作为《程序设计综合实验》课程的配套教材编写的，不论对计算机专业的学生，还是非计算机专业的学生，本书都非常适用。总体来讲，本实验教程的特色如下：

1. 本书的实验内容**均来自作者实际软件开发与教学实践**，学习本教程后，读者定能够解决实际应用中的许多难题；
2. 面对一个复杂的系统开发问题，本书通常将其分解为几个独立的模块，由几次实验分别完成，通过循序渐进的设计思路，让读者逐步掌握开发复杂程序的技巧与思路。前提条件是，让读者掌握了高级语言程序设计（如 C、C++）基本语法；
3. 编程基础好的读者能够得到进一步提高，掌握软件工作的初步思想和技巧，编程能力欠缺的读者可以得到较大的提高；
4. 相对传统的程序设计实验材料，本书涉及较多新知识点，如用户界面开发、信息管理系统、系统底层开发、键盘控制、图形绘制、动画编程等内容，能够充分激发读者学习兴趣。在每个实验的解析部分，对每个知识点都进行了详细的阐述，并附有足够的示例，有利于读者自学，提高自学能力；
5. 本书所有实验均提供源码。

本书内容组织：

本书由 14 个简单实验加上 3 个综合设计构成，能够作为本科生 1 学期的实验课程使用。其中每个简单实验要求完成 1 个独立的系统，建议时间为 1 周。为了降低读者在理解上的复杂度，本书特将每个实验分为 2 个紧密相关的实例，承前启后，各有侧重。最后 3 个综合实验基本概括了前面的全部知识，达到总结提高的作用，由学生选做 1 题，建议完成时间为 3 周。

目录

前言	I
实验 1	1
实例 1 ATM 机用户操作界面	1
实例 2 业务逻辑编写	1
实验 2	5
实例 3 进制转换器 1	5
实例 4 进制转换器 2	9
实验 3	12
实例 5 随机数生成	12
实例 6 中奖者	13
实验 4	16
实例 7 插入排序	16
实例 8 快速排序	17
实验 5	20
实例 9 程序调试工具：模块计时器	20
实例 10 常用排序算法性能测试	28
实验 6	30
实例 11 学生信息数据结构	30
实例 12 小型管理信息系统	31
实验 7	35
实例 13 文本文件格式的保存、读取	35
实例 14 二进制文件格式的保存、读取	36
实验 8	40
实例 15 系统信息读取之——区域信息	40
实例 16 系统操作命令	44
实验 9	47
实例 17 判断文件属性、创建文件夹	47
实例 18 密码输入	48
实验 10	51
实例 19 视窗程序编程基础	51
实例 20 重叠对话框	52
实验 11	61
实例 21 键盘消息	61
实例 22 绘制弹出式菜单	64
实验 12	67
实例 23 文本阅读器	67
实例 24 文本编辑器	68
实验 13	72
实例 25 图形绘制	72
实例 26 创建可独立运行的图形程序	72
实验 14	81
实例 27 图形模式中的文字显示	81
实例 28 动画	83
综合实验 1	89
可视化窗口编辑器	89
综合实验 2	91
图书管理系统设计	91
综合实验 3	93
五子棋游戏软件开发	93
附录 A 本书使用到的库函数说明	95
附录 B STRING 函数列表	97
附录 C C 语言文件操作函数大全	103
附录 D 标准 ASCII 码表及扩展字符集	116
附录 E 键盘扫描码	118

实验 1



实例 1

ATM 机用户操作界面

说明：根据 ATM 的工作流程编写用户界面，掌握文本用户菜单的编写方法。

要求：

- (1) 除提示用户输入的数字外，界面上不能响应、出现任何其它用户输入；
- (2) 每个菜单界面独立显示，不要出现多组菜单重叠显示的现象；

界面 语言选择

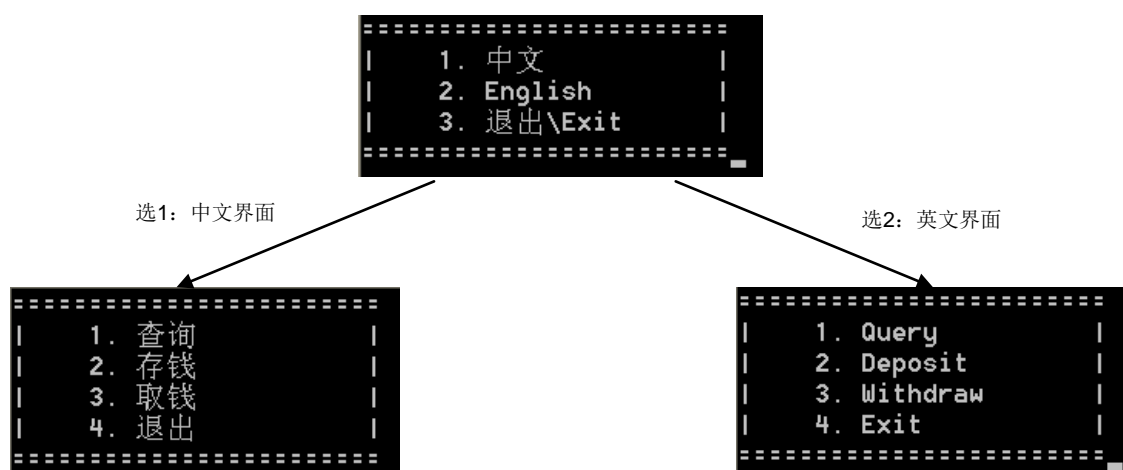


图 1-1 语言选择界面



实例 2

业务逻辑编写

说明：根据 ATM 的工作流程编写其业务逻辑，掌握搭建一个完整应用系统的方法及软件编程思想。

要求：每个业务逻辑可以多次执行，直到用户选择退出业务；

用户选择一种语言后，进入主业务界面。下面以中文界面为例介绍其它业务：

业务逻辑 1 查询界面

```
=====
此帐户有 ￥1000.00 元.
按任意键继续
=====
```

图 1-2 语言选择界面

默认设置用户账户上有 1000 元，界面提示用户按任意键返回主业务界面。

业务逻辑 2 存钱

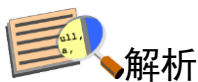
<p>(1) 提示用户输入要存钱的数目：</p> <pre>===== 输入您要存的数目： ￥ =====</pre>	<p>(2) 用户输入数据后，提示用户按任意键返回主业务界面：</p> <pre>===== 输入您要存的数目： ￥ 10 按任意键继续 =====</pre>
<p>(3) 存钱结束后，执行查询，则会显示新的账目信息。</p>	

业务逻辑 3 取款

<p>(1) 提示用户输入要取钱的数目：</p> <pre>===== 输入您要取钱的数目： ￥ =====</pre>	<p>(2) 用户输入数据后，提示用户按任意键返回主业务界面：</p> <pre>===== 输入您要取钱的数目： ￥ 200 按任意键继续 =====</pre>
<p>(3) 如果用户输入数目大于帐户余额，则提醒用户“余额不足，重新输入”</p>	

```
=====
输入您要取钱的数目： ￥ 2000
余额不足，按任意键后重新输入！
=====
```

(4) 取款成功后，执行查询，则会显示新的账目信息。



解析

1. 知识点

(1) 学习编写菜单，掌握人机交互界面的基本思想。

(2) 模块化编程的思想，将不同的菜单分别用不同的函数实现：

- void MainMenu(): 主菜单
- void LanguageMenu(): 语言选择菜单
- void Query(float): 查询菜单
- float Deposit(float): 存款菜单
- float Withdraw(float): 取款菜单

(3) 从键盘上获取一个字符有多种方法，它们各自的特点以及它们之间的区别：

- ch = getchar() 等价于 scanf(“%d”, ch), 等待用户输入，输入的字符会显示在屏幕上，输入字符后，**需要用户回车，才能结束输入**；
- int getche(void), 头文件” conio.h”, 输入的字符会显示在屏幕上，不等待用户输入回车，立刻结束输入；
- int getch(void), 头文件” conio.h”, 输入的字符不会显示在屏幕上，不等待用户输入回车，立刻结束输入。

(4) 调用系统命令用 system 函数，头文件 “stdlib.h”, 如需要清除屏幕，则调用语句

```
system(“cls”);
```

2. 系统主逻辑代码

建议采用 while 循环，反复执行所有逻辑操作，直到用户选择退出，伪码如下：

```
while (!bExit) {  
    MainMenu();           // 显示主菜单  
    in = getch();  
    switch(in) {  
        case '1':  
            查询;    break;  
        case '2':
```

```
        存款;    break;

    ...
    case '4':
        bExit = 1;    break;
    default:
        break;
    }
}
```



程序代码：参见“01-ATM”

实验 2



实例 3

进制转换器 1

说明：写一个“计算器”程序，实现十进制、二进制的相互转换，如下图所示（图 2-1 是一个 window 程序，本实验仅要求用 c 语言实现具有一个类似功能的控制台程序）。

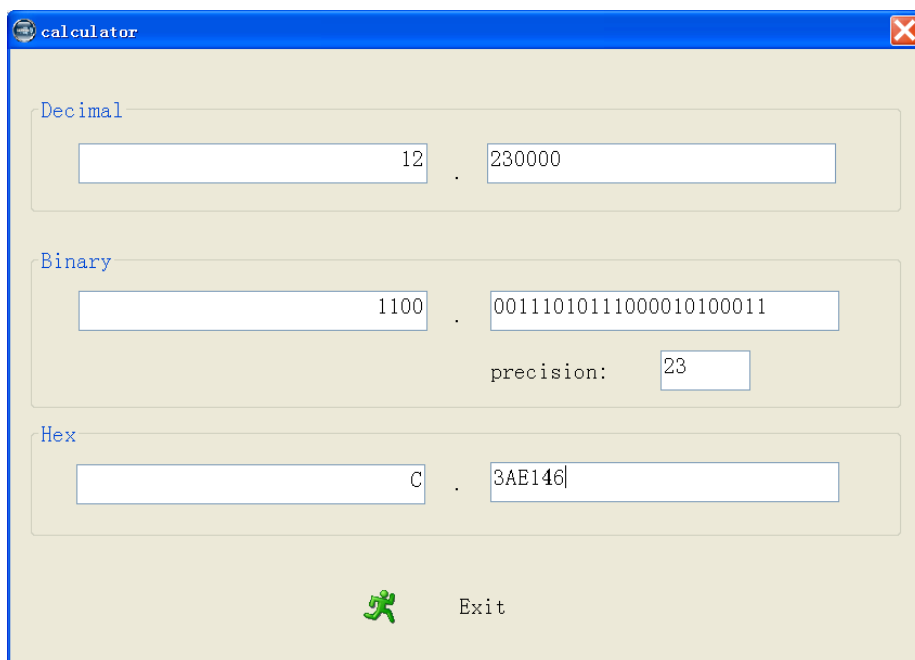


图 2-1 进制转换器

要求：

- (1) 转换要求包括整数和小数；
- (2) 提供清晰、友好的用户界面；
- (3) 各种转换可以反复多次执行，直到用户选择“退出”按钮。



解析

1. 知识点

- (1) 经典“十进制→二进制”转换的方法：

整数：采用余数法，除基数取余数、由下而上排列得到最终二进制排列。

示例：

2		75	1
2		37	1
2		18	0
2		9	1
2		4	0
2		2	0
2		1	1
		0	

结果为： $(75)_2 = 1001011$

小数：采用进位法，用十进制小数乘基数，当积为 0 或达到所要求的精度时，将整数部分由上而下排列。

示例：

0.625	
× 2	
1.250	整数为 1
× 2	
0.50	整数为 0
× 2	
1.0	整数为 1 小数值为 0

结果为： $(0.625)_2 = 0.101$

(2) 应用程序中通常需要大量用到各种类型的数字和字符串进行相互转换，系统通常会提供相应的函数（如 `atoi`, `itoa`, `atof` ...），请看下例：

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

void main( void )
{
    char *s; double x; int i; long l;

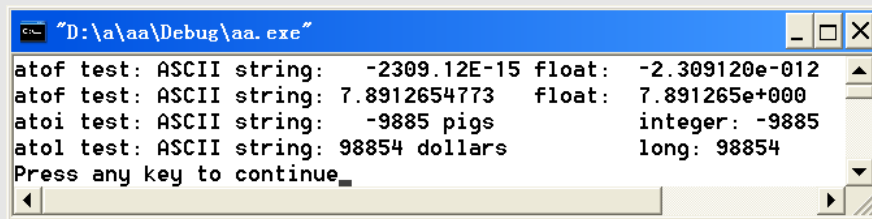
    s = " -2309.12E-15";    /* Test of atof */
    x = atof( s );
    printf( "atof test: ASCII string: %s\tfloat: %e\n", s, x );

    s = "7.8912654773";    /* Test of atof */
    x = atof( s );
    printf( "atof test: ASCII string: %s\tfloat: %e\n", s, x );

    s = " -9885 pigs";      /* Test of atoi */
    i = atoi( s );
    printf( "atoi test: ASCII string: %s\tinteger: %d\n", s, i );

    s = "98854 dollars";    /* Test of atol */
    l = atol( s );
    printf( "atol test: ASCII string: %s\tlong: %ld\n", s, l );
}
```

输出结果:



说明: 上例中 atof, atoi, atol 作用分别是将字符串转换为浮点数、整数、长整数。

如果需要将整数转换为字符串, 则可以相应地使用 itoa (具体函数说明参考“附录 A”), 例如:

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

void main( void )
{
    char s[20];    int i = 10;
    itoa(i, s, 10);
    printf("%s\n", s);
}
```

另外, 将数字转换为字符串最常用的函数是 sprintf (具体函数说明参考“附录 A”), 示例如下:

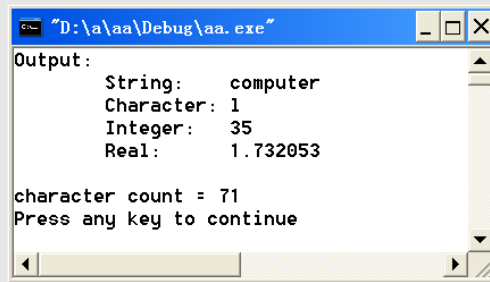
```
#include <stdio.h>

void main( void )
{
    char buffer[200], s[] = "computer", c = 'l';
    int i = 35, j;
    float fp = 1.7320534f;

    /* Format and print various data: */
    j = sprintf( buffer, "\tString: %s\n", s );
    j += sprintf( buffer + j, "\tCharacter: %c\n", c );
    j += sprintf( buffer + j, "\tInteger: %d\n", i );
    j += sprintf( buffer + j, "\tReal: %f\n", fp );

    printf( "Output:\n%s\n\ncharacter count = %d\n", buffer, j );
}
```

输出结果为：



(3) 为方便字符串操作，建议使用 C++ 中 string 类，详细使用方法参考“附录 B string 函数列表”。

2. 示例代码

(1) 十进制小数转换成二进制小数

// 十进制小数转换成二进制小数

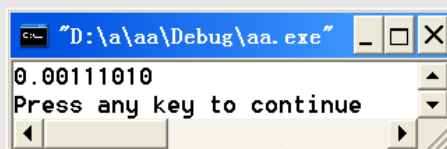
```
#include <string>
using namespace std;

string D2BDecimal(double in, int len)
{
    string ret;
    double product = in;

    while ( len > 0 ) {
        product *= 2;
        if (product >= 1) {
            ret += "1";
            product -= 1.0f;
        }else{
            ret += "0";
        }
        len --;
    }
    return ret;
}

void main(void)
{
    string s = D2BDecimal(0.23, 8);
    printf("0.%.8s\n", s.data());
}
```

输出结果：



(2) 十进制整数转换成二进制小数

// 十进制整数转换成二进制小数

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

// 十进制整数->二进制
string D2BInt(int in)
{
    if (in == 0) {
        return "0";
    }

    string ret = "";
    int quot = in;
    int remain;

    while (quot) {
        remain = quot % 2;
        quot = quot / 2;
        if (remain == 1) ret = "1" + ret;
        else ret = "0" + ret;
    }

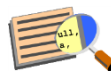
    return ret;
}

void main(void)
{
    string s = D2BInt(127);
    printf("%s\n", s.data());
}
```



实例 4 进制转换器 2

④ 说明：在前面“计算器”程序的基础上，实现二进制和十六进制的相互转换，进而实现十进制、二进制、十六进制之间的相互转换



解析

知识点

(1) 二进制 \longleftrightarrow 十六进制

Binary \rightarrow Hexadecimal：以小数点定位，四位并一

例： 1 1 0 1 0 0 1 1 **B** = D3 **H**

Hexadecimal → Binary：以小数点定位，一拆为四

例： 3 A 2 **H** = 0 0 1 1 1 0 1 0 0 0 1 0 **B**

表 2-1 常见二进制与十六进制数字之间的对应关系

二进制	0000	0001	0010	0011	0100	0101	0110	0111
十六进制	0	1	2	3	4	5	6	7
二进制	1000	1001	1010	1011	1100	1101	1110	1111
十六进制	8	9	a	b	c	d	e	f
二进制	10000	10001	10010	10011	10100	10101	10110	10111
十六进制	10	11	12	13	14	15	16	17
二进制	11000	11001	11010	11011	11100	11101	11110	11111
十六进制	18	19	1a	1b	1c	1d	1e	1f

2. 示例代码

二进制整数转换成十六进制代码

```
// 二进制整数转换成十六进制
string B2HInt(string s)
{
    // 1 Make the string length is evenly divisible by 4
    int iStrlen = s.length();
    int iZeroNum = iStrlen % 4;

    iZeroNum = (4 - iZeroNum) % 4;
    while (iZeroNum) {
        s = "0" + s;
        iZeroNum--;
    }

    // 2 Convert four bits by four bits
    int num = s.length() / 4;
    string ret;
    for (int i = 0; i < num; i++) {
        string sPre = s.substr(0, 4);
        s = s.substr(4, s.length()-4);
        ret += B2H4bit(sPre);
    }
    return ret;
}

// 辅助函数，用于将 4 位二进制向十六进制字符进行转换
char B2H4bit(string s)
{
    char ret;
```



```

        if (s == "0000")      ret = '0';
        else if (s == "0001") ret = '1';
        else if (s == "0010") ret = '2';
        else if (s == "0011") ret = '3';
        else if (s == "0100") ret = '4';
        else if (s == "0101") ret = '5';
        else if (s == "0110") ret = '6';
        else if (s == "0111") ret = '7';
        else if (s == "1000") ret = '8';
        else if (s == "1001") ret = '9';
        else if (s == "1010") ret = 'A';
        else if (s == "1011") ret = 'B';
        else if (s == "1100") ret = 'C';
        else if (s == "1101") ret = 'D';
        else if (s == "1110") ret = 'E';
        else if (s == "1111") ret = 'F';
        else                  printf("input error!\n");
        return ret;
    }

void main(void)
{
    string s = "10111101001";
    string s1 = B2HInt(s);
    printf("%s\n", s1.data());
}

```



程序代码：参考 Windows 应用程序 “02-calculator”

实验 3



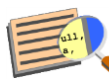
实例 5 随机数生成

说明：现实生活中存在很多随机事件：比如掷钱币、骰子、转轮、使用电子元件的噪音、核裂变等等，这样的随机数发生器叫做**物理性**随机数发生器。同时，为了模拟现实生活中的随机事件，计算机研究人员开发了许多随机数生成算法，其中最简单的是**均匀分布随机数**（算法见后文“解析”部分）。

任务：生成一组均匀分布的伪随机数。

要求：

- (1) 能够根据用户的输入，控制随机数生成的数目，随机数用动态数组保存；
- (2) 能够根据用户的输入，控制随机数的范围：比如能够生成一组【a, b】范围内的随机数。



解析

1. 知识点

随机数生成算法

需要明确：有限状态机不能产生真正的随机数的，所以在现在的计算机中并没有一个真正的随机数生成算法，现有的随机数生成算法生产的随机数只不过因为重复的周期比较大，可以做到使产生的数字重复率很低，这样看起来好象是真正的随机数，一般称作**伪随机数发生器**。首先，我们要搞清楚一个概念，生成单个的随机数是没有意义的，我们所说的随机数生成，其实都是生成一个随机序列。

均匀分布的随机数是指“所生成的数在[**min**, **max**]区间范围内出现的机率是均匀的”，通常方法是： $X_{n+1} = (aX_n + c) \bmod m$ ，其中 m 是模数，控制生成数的范围， $0 \leq a \leq m$ 是乘数， $0 \leq c \leq m$ 是增量， X_0 是种子，控制（影响）第一个元素，通常需要用户提供，如果用户每次给的种子相同，则每次都会生成相同一组随机数，因为采用的是相同的公式；如果每次用户提供的种子不同，则会生成不同的随机数序列。

2. 系统随机数生成代码

```
#include "time.h"
static unsigned holdrand = 1L; // holdrand 影响第一个随机数的生成
void myrand (unsigned seed )
{
    holdrand = seed;
}
int myrand ( void )
{
    return(((holdrand = holdrand * 214013L + 2531011L) >> 16) & 0x7fff);
}
void main( void )
{
    myrand((unsigned)time(NULL));
    for (int i = 0; i < 10; i++) printf("%d\n", myrand());
}
```

代码说明：

函数 time：返回从 1970 年 1 月 1 日 00:00:00 起距函数调用时相距的时间，以秒为单位

头文件：<time.h>

思考：mysrand 的作用是什么？



实例 6 中奖者

④ 说明：根据随机数编写一个抽奖程序。

④ 业务逻辑

(1) 用户输入参加抽奖者人数 N，如下图所示

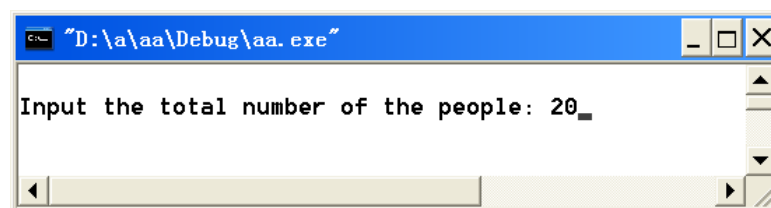
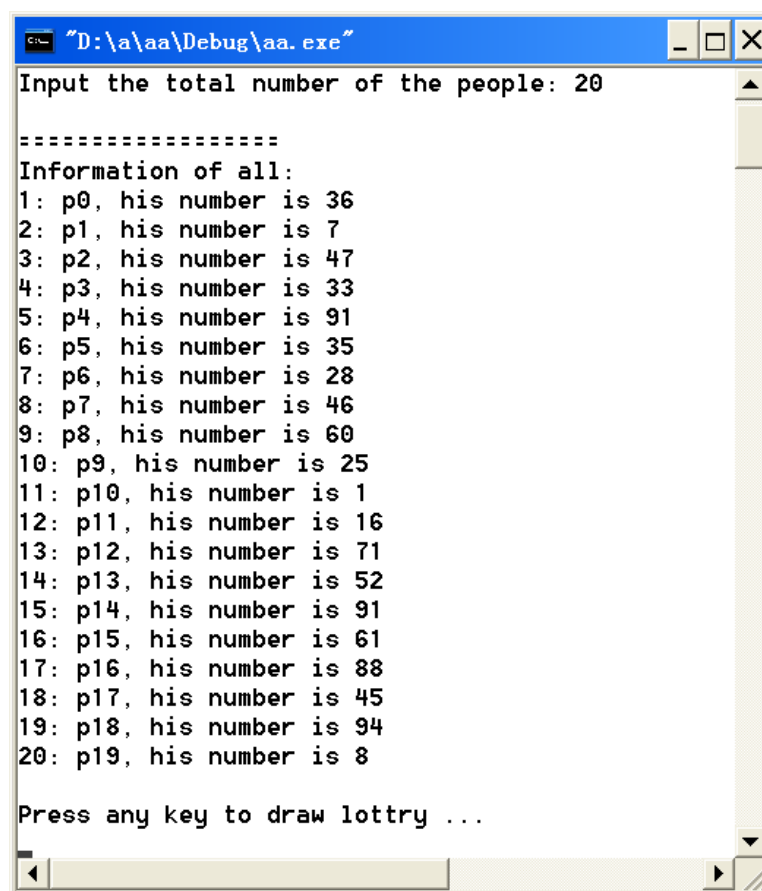


图 3-1 用户输入抽奖人数

(2) 为每一个人分配一个代号（类似于名字）和一个随机号码（即奖票号，建议控制在 1-N 之间），并将其打印出来：



```
"D:\a\aa\Debug\aa.exe"
Input the total number of the people: 20

=====
Information of all:
1: p0, his number is 36
2: p1, his number is 7
3: p2, his number is 47
4: p3, his number is 33
5: p4, his number is 91
6: p5, his number is 35
7: p6, his number is 28
8: p7, his number is 46
9: p8, his number is 60
10: p9, his number is 25
11: p10, his number is 1
12: p11, his number is 16
13: p12, his number is 71
14: p13, his number is 52
15: p14, his number is 91
16: p15, his number is 61
17: p16, his number is 88
18: p17, his number is 45
19: p18, his number is 94
20: p19, his number is 8

Press any key to draw lottery ...
```

图 3-2 打印每个人的信息

(3) 由机器摇号：生成一个在 1-N 之间的随机数作为中奖号码，查找中奖者，并公布中奖者信息（注：可以有多人同时中奖）：

```
The winner is: p5, his number is 35
The valid lottery number is: 35
```

图 3-3 中奖结果公布

(4) 如果没有对应的中奖者，需要将该号码公布出来，然后重新生成一个中奖号码，直到中奖者存在为止：

```
Press any key to draw lottry ...

The invalid lottery number is: 47
The invalid lottery number is: 72
The invalid lottery number is: 63
The invalid lottery number is: 34
The winner is: p6, his number is 66

The winner is: p7, his number is 66

The valid lottery number is: 66
```

图 3-4 无效号码输出



解析

- (1) 为抽奖者定义一个结构体，如

```
struct PEOPLE{
    char name[20];    // 姓名
    int number;       // 抽奖号码
};
```

- (2) 将数字 k 控制在 $[m, n]$ 范围之内可以通过如下表达式：

$$k = k \% (n - m) + m;$$

- (3) 使用 `malloc` 动态分配内存后，一定要调用 `free` 释放内存！



程序代码：见 “05-lottery”

实验 4



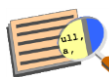
实例 7

插入排序

说明：生成一组随机整数，对该组数据进行排序，采用插入排序法。

要求：

- (1) 数组的长度由用户输入；
- (2) 随机数范围在[100, 10000]范围内；
- (3) 采用**直接插入排序法**进行排序；
- (4) 根据用户要求，可以实现增序、降序两种排序。



解析

1. 直接插入排序基本思路：

(1) 引例：写一个程序，将一个整数 x 插入到由升序数组 $a[0 \sim N-1]$ 中，使得插入后的数组 $a[0 \sim N]$ 保持升序。

该引例的解题思路分 2 步：首先将升序数组中元素值大于 x 的所有元素向后挪动一个位置，然后再将 x 插入，见下面示例：将 $x=8$ 插入一个整型数组中

下标：	0	1	2	3	4	5	6	7
$x=8$	-9	-5	5	6	9	13	34	空

图 4-1 示例，将 8 插入到一个有序数组中，最后一位为空

下标：	0	1	2	3	4	5	6	7
$x=8$	-9	-5	5	6		9	13	34

图 4-2 步骤 1：从后往前，将大于 8 的数据均向后挪一个位置，找到应该将 8 放入的位置

已知一个长度为 n 的整型数组 a ，步骤 1 的代码如下：

```

i = n - 1; // n为数组的长度，i 指向倒数第 2 个位置，也是最后一个元素
while( i >=0 && a[i] > x){
    a[i + 1] = a[i]; // 从后向前，将大于 x 的元素均向后挪动一个位置
    i --;
}

```

下标:	0	1	2	3	4	5	6	7
	-9	-5	5	6	x=8	9	13	34

图 4-3 步骤 2: 将 8 插入数组中，形成一个新的有序数组

(2) 直接插入排序思路

第 1 步: 将第 1 个元素 $a[0]$ 视为一个有序数组，将 $a[1]$ 作为待插入元素 x ，则 $a[1]$ 位置可以视为空闲，将 x 插入到有序数组 $a[0] \sim a[1]$ 中，使得 $a[0] \sim a[1]$ 有序；

第 2 步: 将第 2 个元素 $a[2]$ 作为待插入元素 x ，则 $a[2]$ 位置可以视为空闲，将 x 插入到有序数组 $a[0] \sim a[1]$ 中，使得 $a[0] \sim a[2]$ 有序；

.....

第 N-1 步: 将最后一个元素 $a[N-1]$ 作为待插入元素 x ，则 $a[N-1]$ 位置可以视为空闲，将 x 插入到有序数组 $a[0] \sim a[N-2]$ 中，使得 $a[0] \sim a[N-1]$ 有序。至此，整个数组有序。



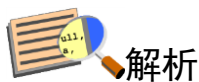
程序代码：参见“04-sort/插入排序.c”

实例 8 快速排序

④ **说明:** 生成一组随机整数，对该组数据进行排序，采用快速排序法。

④ **要求:**

- (1) 数组的长度由用户输入；
- (2) 采用 **普通快速排序法** 进行排序；
- (3) 根据用户要求，可以实现增序、降序两种排序。



解析

1. 快速排序基本思路：

快速排序 (Quick sort) 是所有内排序算法中最实用的排序算法之一，因此得到了广泛的应用。通过本例练习，望读者能够掌握分治法的基本思想：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

算法流程：

快速排序是对冒泡排序的一种改进。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按次方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。该方法的基本过程是：

1. 先从数列中取出一个数作为基准数。
2. 分区过程，将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。
3. 再对左右区间重复第二步，直到各区间只有一个数。

实例解析：

例如：待排序（从小到大）的数组 A 的值分别是：（初始关键数据 X=49）

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]:
49	38	65	97	76	13	27

快速排序就是递归调用此过程——在以第 1 个数为中点分割这个数据序列，分别对前面一部分和后面一部分进行类似的快速排序，从而完成全部数据序列的快速排序，最后把此数据序列变成一个有序的序列，根据这种思想对于上述数组 A 的快速排序的全过程如下所示：

初始状态	{49 38 65 97 76 13 27}
进行一次快速排序之后划分为	{27 38 13} 49 {76 97 65}
分别对前后两部分进行快速排序	{13} 27 {38} 结束
	{49 65} 76 {97} 结束
	49 {65} 结束
	结束

图 4.4 快速排序全过程

2. 快速排序主要代码

// 按照与第 1 个元素大小关系，将数组分成前后两部分
 // 实现了从大到小的排序

```
int Partition(int a[], int p, int r)
{
    int i = p;
    int j = r + 1;
    int x = a[p];

    while(1) {
        while (i <= r && a[++i] > x);
        while (j >= p && a[--j] < x);
        if (i >= j) break;
        Swap(&a[i], &a[j]);
    }

    if (i == j) {
        j--;
    }
    a[p] = a[j];
    a[j] = x;
    return j;
}
```



程序代码：参见“04-sort/快速排序.c”

实验 5



实例 9

程序调试工具：模块计时器

说明：在系统测试时，尤其在需要测试某算法（如排序）或者某些模块的运行时间时，往往需要调用一些时间函数库（如 VC 中的 `timeGetTime` 等可以获取毫秒级的时间），在待测试的模块前后分别测试时间，然后，计算前后两个时间的差值，就得到模块的运行时间，如下代码段所示：

//一个典型的模块计时方法

```
void functionA()
{
    DWORD start = timeGetTime();    // 开始计时
    functionB();
    DWORD end = timeGetTime();      // 结束计时
    DWORD total = end - start;      // 计算时间

    DWORD start = timeGetTime();    // 开始计时
    functionC();
    DWORD end = timeGetTime();      // 结束计时
    DWORD total = end - start;      // 计算时间
}
```

图 5-1 常用计时代码段

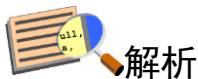
但是，使用原始的计时函数直接进行时间测试在很多复杂情况下不方便，如上段代码，当在一个模块中有多个子模块需要分别计时，所编写的计时代码甚至比原有的代码还多，这增加了程序维护和阅读的难度，容易出错。本实例要求设计一组计时函数，**封装所有计时函数**，仅在需要计时的模块前后分别添加如下两条指令即可：

```
BM_START(i)                // 测试开始标志
    for (i=0; i < N; i++){  // 待测试的模板
        ...
    }
BM_END(i)                   // 测试结束标志
```

其中, *i* 是计时器编号, 与前一段代码相比, 没有计时函数, 也没有运算操作, 编码界面相对简单。

要求:

- (1) 计时精确: 封装的高精度的计时 API 函数 `QueryPerformanceCounter()`, 可以达到微秒(us)级的精度;
- (2) 使用简单: 只用在待测试的模块前后加上两个宏 `BM_START` 和 `BM_END`, 不需要对结果进行计算, 也不需要考虑对各个模块测试结果数据的维护;
- (3) 多组测试: 最多可以同时实现 20 个模块的测试, 即可以保存 20 组数据;
- (4) 结果输出独立: 在系统运行结果时, 只需要调用一个函数就可以把计时结果保存在一个文本文件里, 如图 5-5 所示。



解析

1. 高精度计时函数

对于一般的实时控制, 使用 `GetTickCount()` 函数就可以满足精度要求, 但要进一步提高计时精度, 就要采用 `QueryPerformanceFrequency()` 函数和 `QueryPerformanceCounter()` 函数。这两个函数是 VC 提供的仅供 Windows 使用的高精度时间函数, 并要求计算机从硬件上支持高精度计时器。`QueryPerformanceFrequency()` 函数和 `QueryPerformanceCounter()` 函数的原型为:

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER *lpFrequency);
BOOL QueryPerformanceCounter(LARGE_INTEGER *lpCount);
```

数据类型 `LARGE_INTEGER` 既可以是一个作为 8 字节长的整型数, 也可以是作为两个 4 字节长的整型数的联合结构, 其具体用法根据编译器是否支持 64 位而定。该类型的定义如下:

```
#include <windows.h>
typedef union _LARGE_INTEGER
{
    Struct
    {
        DWORD LowPart;    // 4字节整型数
        LONG HighPart;    // 4字节整型数
    };
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

在进行计时之前，应该先调用 `QueryPerformanceFrequency()` 函数获得机器内部计时器的时钟频率。接着，在需要严格计时的事件发生之前和发生之后分别调用 `QueryPerformanceCounter()` 函数，利用两次获得的计数之差和时钟频率，就可以计算出事件经历的精确时间，如下代码所示：

```
LARGE_INTEGER litmp;
LONGLONG QPart1, QPart2;
double dfMinus, dfFreq, dfTime;
QueryPerformanceFrequency(&litmp);
dfFreq = (double)litmp.QuadPart;           // 获得计数器的时钟频率
QueryPerformanceCounter(&litmp);
QPart1 = litmp.QuadPart;                   // 开始计时

Block1();                                  // 工作模块

QueryPerformanceCounter(&litmp);
QPart2 = litmp.QuadPart;                   // 终止计时
dfMinus = (double)(QPart2 - Qpart1);       // 计算计数器差值
dfTime = dfMinus / dfFreq;                 // 获得对应的时间，单位为秒
```

图 5-2 精确计时代码段

2. 计时函数封装

(1) 数据结构

为了维护计时结果，可以定义如下几个数据：

```
#define      BENCHMARK_MAX_COUNT      20
double      gStarts[BENCHMARK_MAX_COUNT];
double      gEnds[BENCHMARK_MAX_COUNT];
double      gCounters[BENCHMARK_MAX_COUNT];
double      dfFreq = 1;
```

其中，`BENCHMARK_MAX_COUNT` 定义了需要计时的模块总数，20 表示最多可以定时 20 个模块，该值可以根据具体应用而定。`gStarts` 和 `gEnds` 分别用于保存开始计时和终止

计时的计数器的值，gCounters 用来保存计时结果。全局变量 dfFreq 用来保存上文介绍的时钟频率，如图 5-1 所示。

(2) 初始化 InitBenchmark()

初始化函数 InitBenchmark() 包括两部分内容：

- 对数组 gStarts, gEnds, gCounter 清零；
- 获得机器内部定时器时钟频率。

InitBenchmark() 代码如下所示：

```
void InitBenchmark()
{
    ResetBenchmarkCounters(); // 获得计时器的时钟频率
    GetClockFrequent();
}
```

该函数一般在程序运行最初调用。

(3) 开始计时 BMTimerStart()

开始计时函数 BMTimerStart() 放在计时模块的开始，函数定义如下：

```
void BMTimerStart(int iModel)
{
    LARGE_INTEGER litmp;
    QueryPerformanceCounter(&litmp);
    gStarts[iModel] = litmp.QuadPart;
}
```

其中参数 iModel 表示当前计时的模块序号， $0 \leq iModel \leq \text{BENCHMARK_MAX_COUNT}$ ；为了简化调用代码，给出一个宏定义如下，
#define BM_START(t) BMTimerStart(t);

(4) 终止计时 BMTimerEnd()

终止计时函数 BMTimerEnd() 放在计时模块的结束，函数定义如下：

```

void BMTimerEnd(int iModel)
{
    LARGE_INTEGER litmp;
    QueryPerformanceCounter(&litmp);
    gEnds[iModel] = litmp.QuadPart;
    gCounters[iModel] += (((gEnds[iModel] - gStarts[iModel]) / dfFreq) * 1000000);
}

```

参数 `iModel` 同 `BMTimerStart()`。本函数首先获取当前的时钟数，然后除以 `dfFreq` 得到运行时间。对于最后一条语句：

```
gCounters[iModel] += (((gEnds[iModel] - gStarts[iModel]) / dfFreq) * 1000000);
```

要注意两点：

- 用 “+=” 而不是 “=”，这个看似简单的代替，可以实现对同一个模块的重复计时；
- 乘以 1000000，表示计时单位为微秒（us）。

类似 `BMTimerStart()`，同样为 `BMTimerEnd()` 定义一个宏：

```
#define BM_END(t)      BMTimerEnd(t);
```

（5）结果输出 `WriteData()`

以一个文本文件（如图 5-5）把全局变量 `gCounters` 中的所有值输出，该函数一般在程序结束处调用，如图 5-4 中最后一行代码所示。

```

/*****

TotalCount: iteration count
sModel: name of current model in debug
path: path to write the log

*****/
void WriteData(int TotalCount, string sModel, string path) {
    //open the file and move pointer to the end of the file
    ofstream pFile(path.data(), ios::app | ios::out);

    // 1 Header
    string title, s2;
    string sperator = "\r\n\r\n===== \r\n";
    title = "Model --- " + sModel;

    char temp1[100], temp2[100];

    sprintf(temp1, "\r\nIteration Counter: %d\r\n", TotalCount);

```

```

        title += temp1;

        // 2 Times
        string s;
        double total = 0.0;

        //write the digits, five in a row
        for(int i = 0 ; i < BENCHMARK_MAX_COUNT; i ++)
        {
            if (gCounters [i] == 0) {
                continue;
            }
            sprintf(temp1, "Total %d", i);
            // * calculate the number of '\t'
            int len = strlen(temp1) / 8;
            s += temp1;
            for( ; len < 2; len ++){
                s += "\t";
            }
            sprintf(temp2, ": %d ms\r\n", (int)( gCounters [i]/1000));

            total += gCounters [i]; // Calculate the total ticks.
            s += temp2;
        }

        char temp[100];
        sprintf(temp, "** Total : %d ms **\r\n", (int)total/1000);
        total = 0;

        s += temp;

        //write entire string at one time
        s2 = sperator + title + s;

        pFile << s2;
        pFile.close();
    }

```

3. 使用举例

(1) 多个模块计时

图 5-3 展示了嵌套计时以及对一个函数中多个模块进行计时的代码，图中可以看到，利用输入参数我们对计时模块进行统一编号，测试代码相对图 5-1 更清晰、直观。

```

void functionA(void) {
    BM_START(0)    // 对functionB整体计时
    functionB();
    BM_END(0)
}

void functionB(void){
    BM_START(1)    // 嵌套计时, 模块1
    Block1();
    BM_END(1)

    BM_START(2)    // 嵌套计时, 模块2
    Block2();
    BM_END(2)
}

```

图 5-3 实现嵌套计时

```

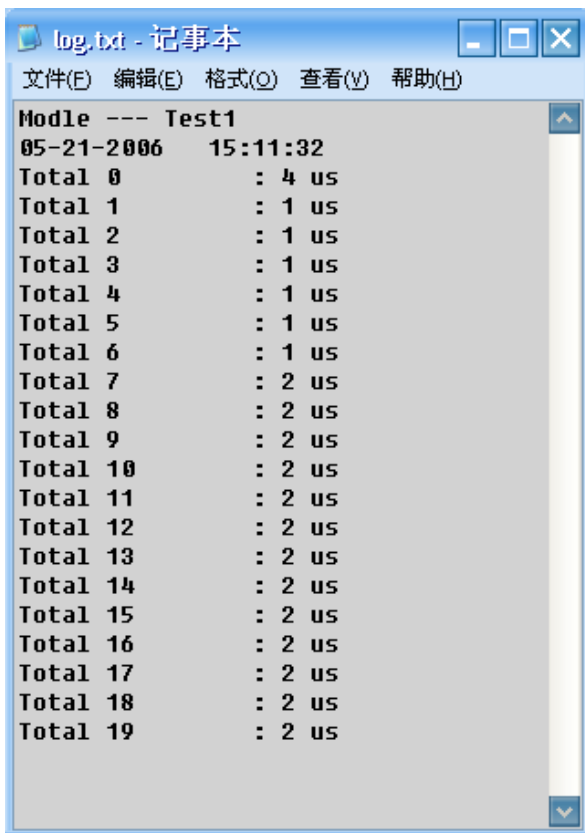
void CPerformanceDlg::OnTest1()
{
    int ret[20];
    for(int i = 0; i < 20; i++){
        BM_START(i)
        ret[i] = Calculator(i+1); // 计算阶乘
        BM_END(i)
    }
    WriteData("D:\\log.txt", "Test1");
}

```

图 5-4 对循环中每次运算的计时

(2) 循环内部计时

图 5-4 中的代码展示了我们对循环体内每次执行运算的计时，只需简单地给出参数 i 就可以得到从 1 到 20 的阶乘的每次计算计算时间，计时结果输出为文件 “D:\log.txt”，如图 5-5 所示。

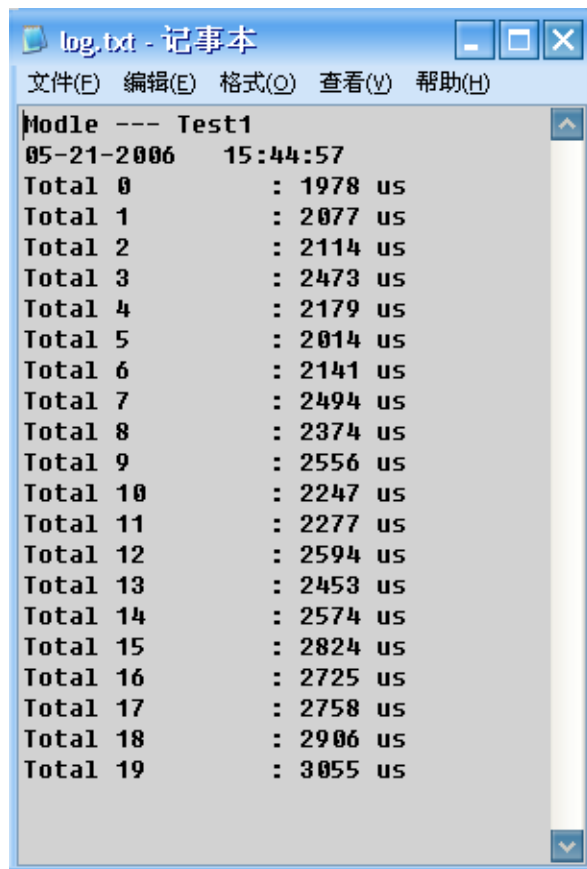


```

Modle --- Test1
05-21-2006 15:11:32
Total 0 : 4 us
Total 1 : 1 us
Total 2 : 1 us
Total 3 : 1 us
Total 4 : 1 us
Total 5 : 1 us
Total 6 : 1 us
Total 7 : 2 us
Total 8 : 2 us
Total 9 : 2 us
Total 10 : 2 us
Total 11 : 2 us
Total 12 : 2 us
Total 13 : 2 us
Total 14 : 2 us
Total 15 : 2 us
Total 16 : 2 us
Total 17 : 2 us
Total 18 : 2 us
Total 19 : 2 us

```

图 5-5 计时结果 1



```

Modle --- Test1
05-21-2006 15:44:57
Total 0 : 1978 us
Total 1 : 2077 us
Total 2 : 2114 us
Total 3 : 2473 us
Total 4 : 2179 us
Total 5 : 2014 us
Total 6 : 2141 us
Total 7 : 2494 us
Total 8 : 2374 us
Total 9 : 2556 us
Total 10 : 2247 us
Total 11 : 2277 us
Total 12 : 2594 us
Total 13 : 2453 us
Total 14 : 2574 us
Total 15 : 2824 us
Total 16 : 2725 us
Total 17 : 2758 us
Total 18 : 2906 us
Total 19 : 3055 us

```

图 5-6 计时结果 2

(3) 循环累加计时

从图 5-5 可以看到，由于现代计算机处理速度越来越快，一些简单运算的模块，微秒的计时单位几乎都不够精确，因此，一种常用的测试方法就是对同一模块进行 N (N 取 1000, 10000 等) 次重复执行。使用本实验的计时函数，可以采用两种方式对这种情况进行测试，代码分别如图 5-7 和图 5-8，请注意二者的区别，并请读者分析为何图 5-8 中的方法也是可行的。 N 次运算计时结果如图 5-6。

```
for(int i = 0; i < 20; i++){
BM_START(i)
    for (int k = 0; k < N; k++){
        ret[i] = Calculator(i+1); // 计算阶乘
    }
BM_END(i)
}
```

图 5-7 累加计时 1

```
for (int k = 0; k < N; k++){
    for(int i = 0; i < 20; i++){
BM_START(i)
        ret[i] = Calculator(i+1); // 计算阶乘
BM_END(i)
    }
}
```

图 5-8 累加计时 2



程序代码：参见“05-performance”文件夹下的“Benchmark.cpp”，“Benchmark.h”两个文件

说明：源码文件夹里“Benchmark.cpp”，“Benchmark.h”两个文件，封装了性能测试所需要函数，可以直接使用，读者也可以根据自己所学进行适当修改。

问：如何将这两个文件导入你自己的工程进行编译？

答：(以 VC6.0 英文版为例)

- (1) 在 VC 编译器中打开自己的工程；
- (2) 将这两个文件拷贝至工程文件夹中；
- (3) 点击菜单“project/Add to Project/Files ...”，如下图所示：
- (4) 在弹出的文件对话框中选择一个或多个文件（在这里，应选择工程文件夹中新拷贝的两个文件），点击“OK”按钮，完成源码文件的添加。在当前工程中已经增加了这两个文件中所包含的所有函数，如图所示。
- (5) 在需要使用性能测试函数的文件头添加引用：`#include "Benchmark.h"`

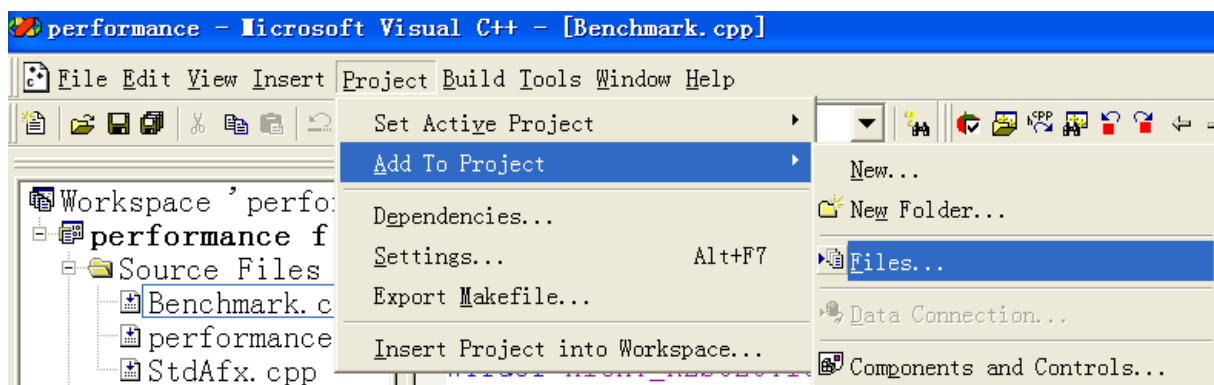


图 5-9 向工程中添加源码文件

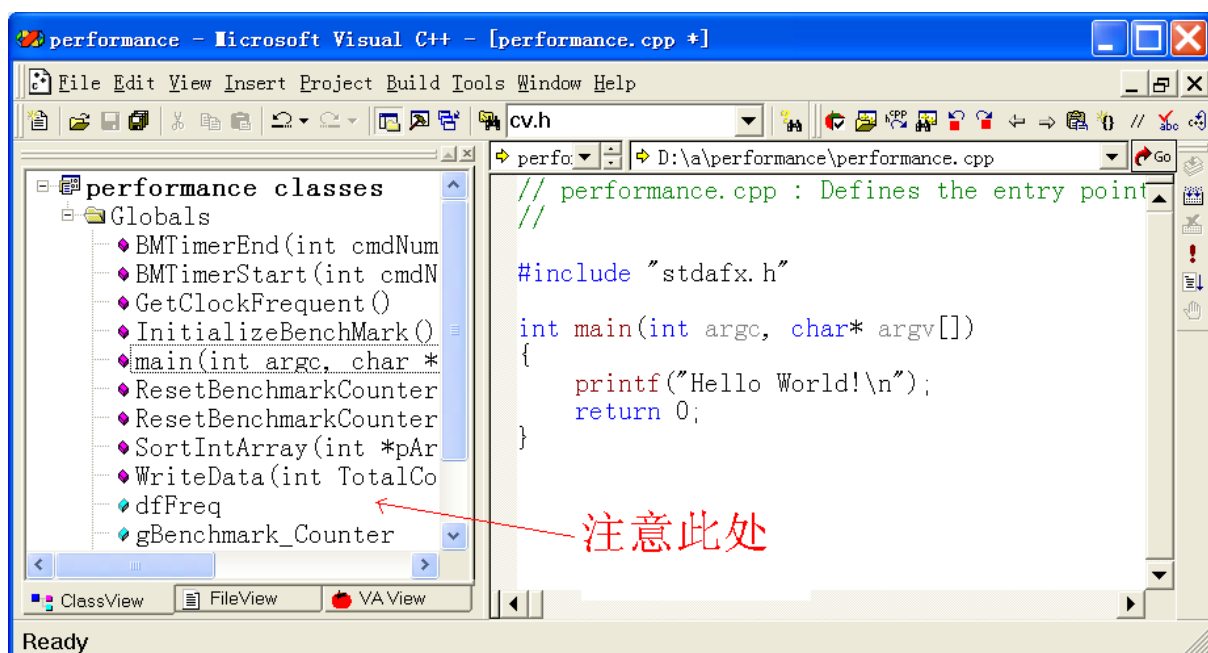


图 5-10 添加源码文件后新产生的类

实例 10 常用排序算法性能测试

④ 说明：要求采用实例 9 的性能测试函数对快速排序、插入排序、选择排序算法进行性能测试。

④ 要求：

- (1) 对一组整型数组进行排序；
- (2) 数组长度为 10000，数组应随机产生；
- (3) 为了保证测试数据的稳定，每种排序算法运行 10 次，计算总的运行时间；

(4) 以文本文档的形式对最终测试结果进行保存。



解析

主程序逻辑

```
// 1 初始化测试数据
定义两个具有相同长度的整型数组 a, b
为 a 的每个元素生成随机数，并将 a 的数据备份到 b 中

// 2 开始性能测试
InitializeBenchMark();           // 初始化

for (i = 0; i < 10; i ++) {
BM_START(0)
    对数组a进行排序（算法自定）
BM_END(0)
    memcpy(a, b, sizeof(int) * N); // 思考：这条语句的作用
}

// 3 输出测试结果
WriteData(N, "Sort test", "d:\\result.txt");
```



程序代码：参见“05-performance”

实验 6



实例 11 学生信息数据结构

说明：随着 IT 技术的普及，高校普遍采用计算机实现对学生信息的管理。本实例要求采用**单链表**实现对一组学生信息的组织。

要求：

(1) 参考实例 1-ATM，编写友好的用户界面，如下图所示：

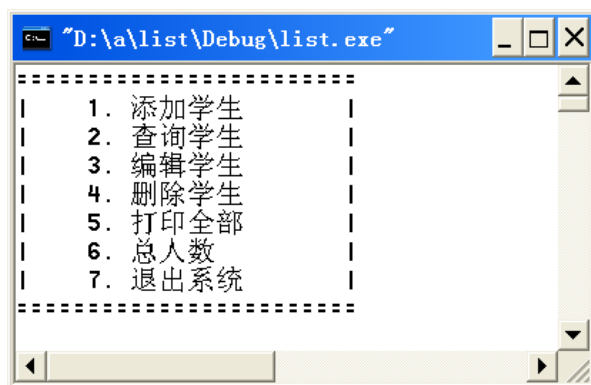


图 6-1 用户界面

(2) 采用**带头结点的单链表**实现对数据的组织，不允许使用数组；

(3) 学生信息至少包括如下三项条目：

- 学号（4 位编号，如 0101，0203 等，所有学生学号长度一致）
- 姓名
- 成绩

(4) 功能要求：

- 添加学生：提示用户输入信息，添加一个学生信息。

注：添加学号时，要求程序对学号长度进行检查，必须是 4 位，如下图所示：

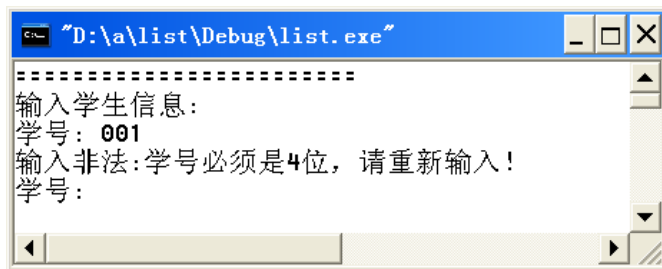


图 6-2 学号长度检查

➤ 总人数显示

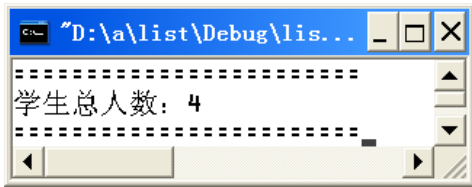


图 6-3 总人数信息

➤ 打印：按顺序在屏幕上打印所有学生信息，见图 6-4。

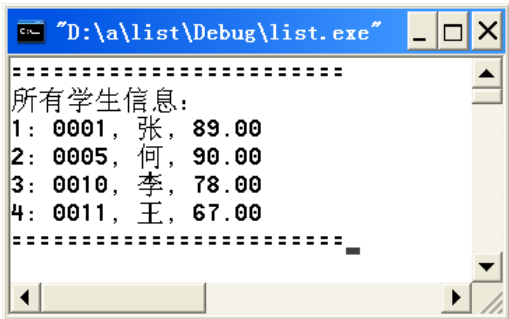


图 6-4 顺序输出所有学生信息

(5) 链表中的数据有序排列：要求根据学号升序排列。如图 6-4 打印结果所示。



实例 12

小型管理信息系统

④ 说明：本实例要求在实例 11 的基础上完成一个小型学生管理信息系统（MIS，Management Information System），实现除存储以外其它基本的信息管理功能。

④ 要求：

(1) 在实例 11 基础上，添加如下功能：

➤ 删除：a.根据学号删除任意学生信息；b.删除所有学生信息。要求删除成功与否都需要打印相应的提示信息，如“学号为****的学生成功删除”，“学号为****的学生不存在，删除失败”等，如图 6-6 所示。

小技巧：为删除操作提供辅助信息，可以在删除界面中打印所有学生信息，如图 6-5、图 6-6 所示：

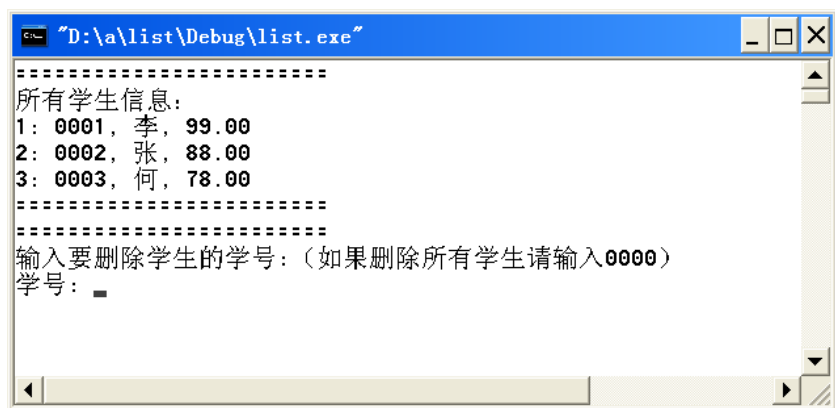


图 6-5 删除界面，包括打印全部学生，作为辅助信息

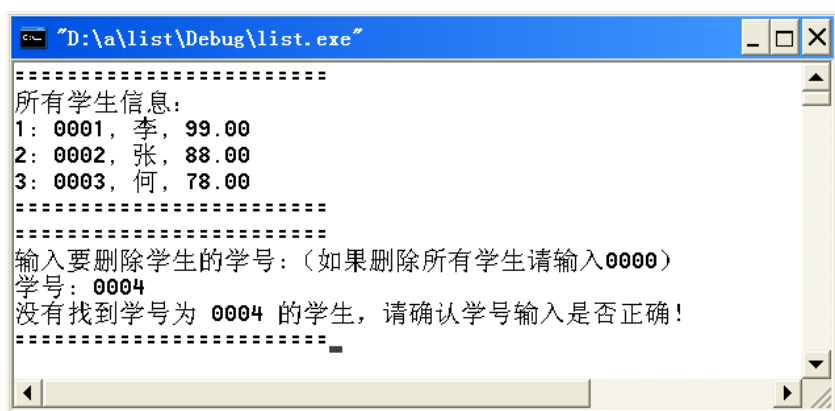


图 6-6 删除结束提示信息

- 查询：根据学号查询任意学生，并将信息打印在屏幕上，如下图所示



图 6-7 根据学号查询学生信息

- 编辑：修改任意一个学生除学号以外的信息，如姓名、成绩等。



图 6-8 根据学号查询学生信息



解析

1. 知识点

(1) 每个学生的信息可以采用如下结构体来保存：

```
typedef struct STD {  
    char ID[5];           // 学号采用字符串而非数值型，为保证所有学生的学号宽度一致  
    char name[20];  
    float score;  
    struct STD *pNext;    // 为创建链表所用，指向下一结点  
}STUDENT, *LIST;
```

(2) 学生总人数可以通过一个全局变量来保存，如 `int iTotal`;

(3) 带头结点的单链表优点：增加一个附加头结点，使很多算法得到简化。结构通常如下图所示：

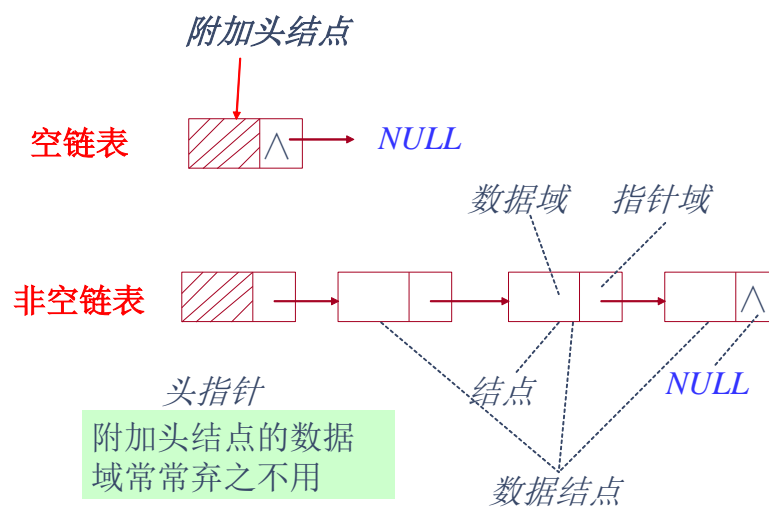


图 6-9 带头结点的单链表数据结构

2. 系统主逻辑代码

建议采用 while 循环，反复执行所有逻辑操作，直到用户选择退出，伪码如下：

```
while (!bExit) {  
    MainMenu();           // 显示主菜单  
    in = getch();  
    switch(in) {
```

```

        case '1':
            Add(list);        // 添加
            break;
        case '2':
            Search(list);     // 查询
            break;
        case '3':
            Edit(list);       // 编辑
            break;
        case '4':
            Delete(list);     // 删除
            break;
        case '5':
            PrintAll(list);   // 打印
            getch();
            break;
        case '6':
            PrintCnt();       // 打印总人数
            break;
        case '7':
            bExit = 1;       // 退出
            break;
        default:
            break;
    }
}

```



程序代码：参见“06-list”

实验 7



实例 13

文本文件格式的保存、读取

④ 说明：从严格意义上看，实例 12 中的小型学生管理信息系统并不是一个完整 MIS 系统，原因在于不能进行数据的保存与读取。本实例就是在实例 12 的基础上，进一步完善功能，使之实现数据的保存与读取，见图 7-1（注：一个实际应用中的 MIS 系统通常采用数据库来存储数据，本例仅采用文件进行数据存储，这是一种简单的操作方式）。

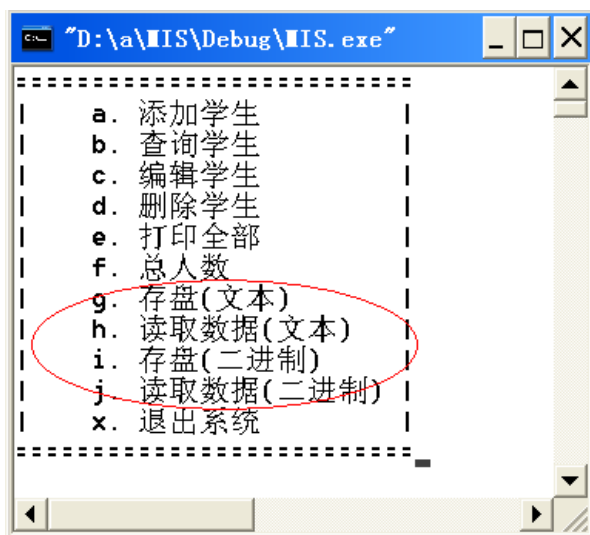


图 7-1 完善后的系统：添加了数据保存及读取功能

④ 要求：

- (1) 用文本（字符）文件格式进行数据的保存与读取；
- (2) 文件的存储路径，提示用户输入；
- (3) 存盘成功或者失败，输出提示信息，如图 7-2 所示；
- (4) 数据读入后能够修改，保存。

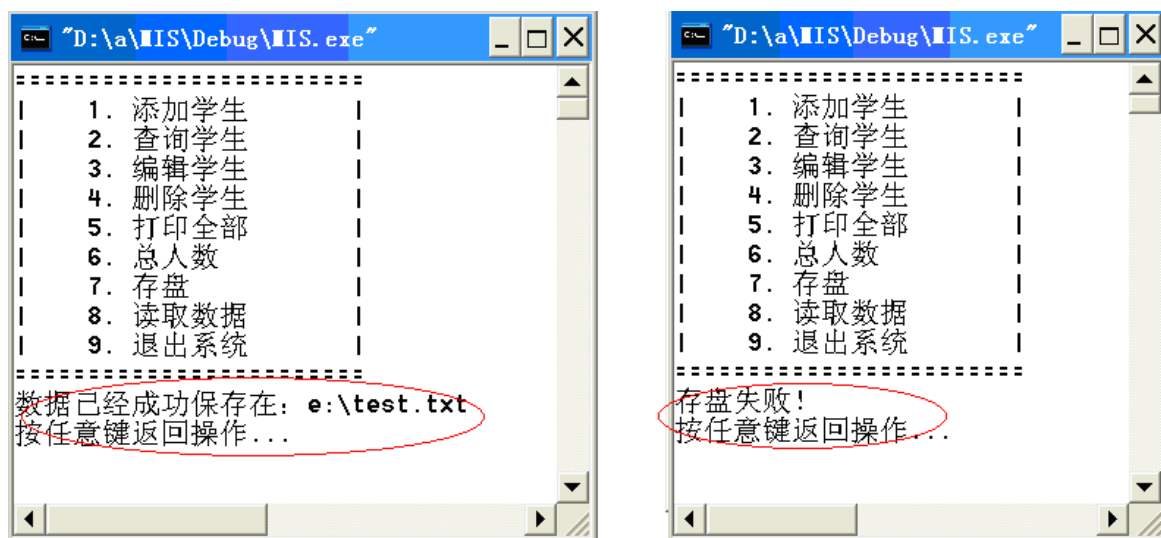


图 7-2 存盘提示信息

实例 14 二进制文件格式的保存、读取

④ 说明：实际上，将数据进行二进制保存和读取编程更方便。通常利用文件进行数据保存的软件系统都是采用二进制的文件格式，例如，常见 word 的*.doc 文档，excel 的*.xls 文件，甚至于常用的图片文件 jpg，视频文件 avi。文本文件仅用于保存一些数据量较小的系统配置参数，便于用记事本查看。相对文本文件，二进制文件格式的优点主要在于：

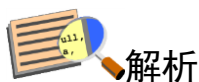
- (1) 读取、保存程序实现方便，可以不用考虑字符串和数值之间的转换；
- (2) 文件数据量较小；（请分析原因）

④ 要求：

(1) 进一步将例 13 进行改进，修改存盘及数据读取函数，添加用二进制文件实现数据的保存与读取的功能；

注：要求本系统既可以用二进制进行文件操作，亦保留文本文件操作的功能。

(2) 其它要求同实例 13。



解析

1. 知识点

(1) 在 C 语言中，通常采用一个**指针**指向一个文件，这个指针称为**文件指针**。通过文件指针就可对它所指的文件进行各种操作。定义说明文件指针的一般形式为：

FILE 指针变量标识符；*

其中 FILE 应为大写，它实际上是由系统定义的一个结构，该结构中含有文件名、文件状态和文件当前位置等信息。

在编写源程序时不必关心 FILE 结构的细节。例如：FILE *fp；然后，可通过 fp 实施对文件的操作。习惯上也笼统地把 fp 称为指向一个文件的指针。

(2) 文件的打开与关闭：文件在进行读写操作之前要先打开，使用完毕要关闭。所谓打开文件，实际上是建立文件的各种有关信息，并使文件指针指向该文件，以便进行其它操作。关闭文件则断开指针与文件之间的联系，也就禁止再对该文件进行操作。

(3) 文件打开采用 fopen 函数，它的调用方式为（详细说明见附录 C）：

文件指针名=fopen(文件名，文件操作方式)；

其中，C 语言的文件操作可以分为文本（字符）文件操作与二进制文件操作两类，文件操作方式字符串如下表所示：

表 7-1 文件操作字符串说明

字符串		操作说明
文本文件	二进制文件	
“r”或“rt”	“rb”	只读打开一个文件，只允许读数据。 要求文件存在，否则打开错误。
“w”或“wt”	“wb”	只写打开或建立一个文件，只允许写数据。 如果文件存在，删除原有数据；如果文件不存在，则建立新的文件。
“a”或“at”	“ab”	追加打开一个文件。 如果文件存在，并在文件末尾写数据；如果文件不存在，则建立新的文件。
“r+”或“rt+”	“rb+”	读写打开一个文本文件，允许读和写。 要求文件存在，否则打开错误。
“w+”或“wt+”	“wb+”	读写打开或建立一个文件，允许读和写。 如果文件存在，删除原有数据；如果文件不存在，则建立新的文件。
“a+”或“at+”	“ab+”	读写打开一个二进制文件，允许读。 如果文件存在，可在文件末尾写数据；如果文件不存在，则建立新的文件。

(4) 打开文件后，一定不要忘了关闭文件，fclose(fp)。

(5) 输入文件名，应当尽量包括扩展名。如果是文本文件，可以写为*.txt，其中*号代表文件名，以方便系统自动调用“记事本”程序打开；对于二进制文件，可以扩展名可以使用*.dat，或者自己定义的一个三位字符，如*.asf。

(6) 对于文件文件格式的输出，为了方便数值向字符的转换（而二进制文件读取则可以不用考虑这一问题，请思考原因），建议采用格式化输出函数 `fprintf`（函数说明详见附录 C），如下代码实现了对整个链表数据的输出：

```
STUDENT * p = list->pNext;
while(p != NULL) {
    fprintf(fp, "%s %s %.2f\n", p->ID, p->name, p->score);
    p = p->pNext;
}
```

(7) 对于文件文件格式的读取，为了方便字符向数值的转换，建议采用格式化输入函数 `fscanf`（函数说明详见附录 C），如下代码实现了对整个文档数据的读入：

```
STUDENT *newnode;
while(1) {
    if (feof(fp)) break;           // 检查文件是否结束
    newnode = (STUDENT *) malloc(sizeof(STUDENT)); // 动态分配内存
    fscanf(fp, "%s%s%f\n", newnode->ID, newnode->name, &(newnode->score));
    Insert(list, newnode);         // 将数据插入到链表
}
```

思考，与上面代码相比，下面代码错在哪里？

// 错误的方式：

```
STUDENT newnode;
while(1) {
    if (feof(fp)) break; // 检查文件是否结束
    fscanf(fp, "%s%s%f\n", newnode.ID, newnode.name, &(newnode.score));
    Insert(list, &node); // 将数据插入到链表
}
STUDENT *newnode;
```

(8) 对于二进制文件格式的输出，可以采用块输出函数 `fwrite`（函数说明详见附录 C）一次性将整个结构体输出，如下代码实现了对整个链表数据的输出，相对（6）中的代码，本方法的代码相对简单，不需要考虑数据类型的转换。

```

STUDENT * p = list->pNext;
while(p != NULL){
    fwrite(p, sizeof(STUDENT), 1, fp);
    p = p->pNext;
}

```

(9) 对于二进制文件格式的输入，可以采用块输出函数 `fread`（函数说明详见附录 C）一次性将整个结构体输入，如下代码实现了对整个文档数据的读入。

```

STUDENT *newnode;
while(1){ // 检查文件是否结束
    newnode = (STUDENT *) malloc(sizeof(STUDENT)); // 动态分配内存
    fread(newnode, sizeof(STUDENT), 1, fp);
    if (feof(fp)){
        free(newnode);
        break;
    }
    Insert(list, newnode); // 将数据插入到链表
}

```



程序代码：参见“07-MIS”

实验 8



实例 15

系统信息读取之——区域信息

说明：普通程序运行于操作系统之上，因此和系统交互、读取系统信息是常见的操作。本例要求读取保存在系统中的软、硬件信息，比如磁盘数目及大小、内存大小等。

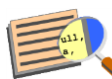
要求：

- (1) 输入操作系统版本号，并确定当前操作系统类型（XP? Win 7? Win 8?）；
- (2) 读取内存信息并输出
- (3) 读取硬盘信息并输出

系统信息结果如下图所示：

```
操作系统: 2 (6.1)  
总内存: 4006, 剩余内存: 1741  
硬盘c:\: 总空间75.22G, 可用空间:42.54G
```

图 8-1 系统信息显示结果



解析

知识点

(1) **操作系统信息**？在 windows 系统中，我们可以通过右键“我的电脑”，选择“属性”很方便地查看操作系统版本、内存容量、计算机名称，如图 8-1 所示。



图 8-1 通过 windows 系统查看本机操作系统信息

(2) 利用 Windwos 提供的 API 函数，我们同样可以得到这些信息：

- 获取 Windows 操作系统版本，采用 GetVersionEx 函数：

```
BOOL GetVersionEx(
    OSVERSIONINFO *lpVersionInfo
);
```

其中，lpVersionInfo 用于返回操作系统信息，结构体的定义为（windows.h 定义）：

```
typedef struct _OSVERSIONINFO {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;           // 主版本
    DWORD dwMinorVersion;          // 小版本
    DWORD dwBuildNumber;
    DWORD dwPlatformId;             // 平台号
    TCHAR szCSDVersion[128];
} OSVERSIONINFO;
```

说明：

- 在 Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, or Windows 2000 系统下，dwPlatformId 值是 2（相应的宏定义为 VER_PLATFORM_WIN32_NT）

- 具体操作系统版由 dwMajorVersion 和 dwMinorVersion 共同确定：

操作系统	版本	dwMajorVersion 值	dwMinorVersion 值	备注
Windows 8	6.2	6	2	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows Server 2012	6.2	6	2	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows 7	6.1	6	1	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows Server 2008 R2	6.1	6	1	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows Server 2008	6.0	6	0	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows Vista	6.0	6	0	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows Server 2003 R2	5.2	5	2	GetSystemMetrics(SM_SERVERR2) != 0
Windows Server 2003	5.2	5	2	GetSystemMetrics(SM_SERVERR2) == 0
Windows XP	5.1	5	1	Not applicable
Windows 2000	5.0	5	0	Not applicable

获取操作系统信息示例代码（读者可以尝试打印其它信息）：

```
OSVERSIONINFO os;
os.dwOSVersionInfoSize=sizeof(OSVERSIONINFO);
::GetVersionEx(&os);
printf("平台: %d", os.dwPlatformId);
printf(" (%d.%d)\n", os.dwMajorVersion, os.dwMinorVersion);
```

- 获取内存大小，采用 GlobalMemoryStatus 函数：


```
void WINAPI GlobalMemoryStatus(
    MEMORYSTATUS *lpBuffer
);
```

其中，lpBuffer 用于返回内存信息，结构体的定义为（windows.h 定义）：

```
typedef struct _MEMORYSTATUS {
    DWORD   dwLength;
    DWORD   dwMemoryLoad;    // 内存占用率
    SIZE_T  dwTotalPhys;     // 全部物理内存大小
    SIZE_T  dwAvailPhys;     // 剩余物理内存大小
    SIZE_T  dwTotalPageFile;
    SIZE_T  dwAvailPageFile;
    SIZE_T  dwTotalVirtual;  // 全部虚拟内存大小
    SIZE_T  dwAvailVirtual;  // 剩余虚拟内存大小
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

获取内存信息示例代码：

```
MEMORYSTATUS memoryStatus={0};
memoryStatus.dwLength=sizeof(memoryStatus);
GlobalMemoryStatus(&memoryStatus);
long total = memoryStatus.dwTotalPhys / (1024*1024);
long avail = memoryStatus.dwAvailPhys / (1024*1024);
printf("总内存: %d, 剩余内存: %d\n", total, avail);
```

● 获取磁盘信息，采用 GetDiskFreeSpaceEx 函数：

```
BOOL WINAPI GetDiskFreeSpaceEx(
    _In_opt_ LPCTSTR lpDirectoryName,
    _Out_opt_ PULARGE_INTEGER lpFreeBytesAvailable,
    _Out_opt_ PULARGE_INTEGER lpTotalNumberOfBytes,
    _Out_opt_ PULARGE_INTEGER lpTotalNumberOfFreeBytes
);
```

参数说明：

- lpDirectoryName，是磁盘名称，如“c:\\”
- lpFreeBytesAvailable，返回用户能使用的空余空间
- lpTotalNumberOfBytes，返回磁盘总空间

■ lpTotalNumberOfFreeBytes, 返回磁盘空余空间

获取磁盘信息的示例代码:

```
ULARGE_INTEGER lpuse;  
ULARGE_INTEGER lptotal;  
ULARGE_INTEGER lpfree;  
string szDisk = "c:\\";  
::GetDiskFreeSpaceEx(szDisk.c_str(), &lpuse, &lptotal, &lpfree);  
printf("硬盘%s: 总空间%.2fG, 可用空间: %.2fG\n\n", szDisk.c_str(),  
        lptotal.QuadPart/1024.0/1024.0/1024.0,  
        lpfree.QuadPart/1024.0/1024.0/1024.0);
```

(3) 注: 本示例由于采用了 windows 提供的 API 函数及数据结构, 需要引用 windows.h。

【思考】: 如何获得系统有哪些磁盘? 请查阅资料, 实现对系统所有磁盘信息的遍历输出。



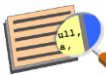
程序代码: 参见 “08-sysinfo”

实例 16 系统操作命令

④ 说明: 掌握程序休眠、获取当前时间的方法

④ 要求:

- (1) 使用 sleep 函数, 使程序休眠 5 秒钟
- (2) 获取当前的时间, 格式为: “星期 月 日 小时: 分钟: 秒 年份”



解析

知识点

- (1) sleep 函数在 dos.h 中定义, 仅有一个参数, 即休眠的秒数。

```
void    sleep (unsigned seconds);
```

(2) 获取当前时间的函数可以用 `time` 函数（头文件`<time.h>`）：

```
time_t current_time;
time(&current_time); // Get the time in seconds;
```

日历时间（Calendar Time）是通过 `time_t` 数据类型来表示的，用 `time_t` 表示的时间（日历时间）是从一个时间点（例如：1970 年 1 月 1 日 0 时 0 分 0 秒）到此时的秒数。在 `time.h` 中，我们也可以看到 `time_t` 是一个长整型数：

```
#ifndef _TIME_T_DEFINED
typedef long time_t;      /* 时间值 */
#define _TIME_T_DEFINED  /* 避免重复定义 time_t */
#endif
```

大家可能会产生疑问：既然 `time_t` 实际上是长整型，到未来的某一天，从一个时间点（一般是 1970 年 1 月 1 日 0 时 0 分 0 秒）到那时的秒数（即日历时间）超出了长整形所能表示的数的范围怎么办？对 `time_t` 数据类型的值来说，它所表示的时间不能晚于 2038 年 1 月 18 日 19 时 14 分 07 秒。为了能够表示更久远的时间，一些编译器厂商引入了 64 位甚至更长的整形数来保存日历时间。比如微软在 Visual C++ 中采用了 `__time64_t` 数据类型来保存日历时间，并通过 `__time64()` 函数来获得日历时间（而不是通过使用 32 位字的 `time()` 函数），这样就可以通过该数据类型保存 3001 年 1 月 1 日 0 时 0 分 0 秒（不包括该时间点）之前的时间。

(3) 固定的时间格式

我们可以通过 `ctime()` 函数将时间以固定的格式显示出来，返回值都是 `char*` 型的字符串。返回的时间格式为：

星期几 月份 日期 时:分:秒 年\n\0

例如：Wed Jan 02 02:03:55 1980\n\0

其中 `\n` 是一个换行符，`\0` 是一个空字符，表示字符串结束。下面是该函数的原型：

```
char * ctime(const time_t *timer);
```

其中 `ctime()` 是通过日历时间来生成时间字符串，先参照本地的时间设置，把日历时间转化为本地时间，然后再生成格式化后的字符串。



程序代码：参见 “08-cmds”

实验 9



实例 17

判断文件属性、创建文件夹

④ 说明：通过系统库函数判断文件的各种属性（存在性、只读等），学习创建文件夹的方法。

④ 要求：

- (1) 用户输入任意一个文件名（包括路径，如 d:\test.txt），写程序判断该文件是否存在；
- (2) 用户输入任意一个文件名，指出该文件的属性，例如是否只读；
- (3) 写程序判断“D:\test”文件夹是否存在，如果不存在则创建它；
- (4) 把以上 3 个功能集成在一个程序中，提供一个用户菜单，供用户选择测试，如下图所示：

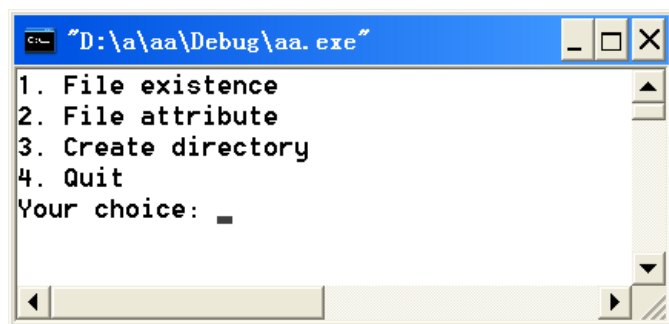
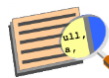


图 9-1 用户界面



解析

知识点

(1) 判断文件或者文件夹是否存在，可以采用库函数 `access`，头文件 `<io.h>`，`asscess` 具体的使用方法如下：

```
int access( const char *path, int mode );
```

参数说明：

path: 文件（或文件夹）名，包括完整路径

mode: 访问方式，可以设为如下几个值：

<i>mode Value</i>	含义
0	Existence only
2	Write permission
4	Read permission
6	Read and write permission

例：通过上表可知，如果仅需判断一个文件是否存在，只需要将 *access* 参数 *mode* 设置为 0 即可。

函数返回值：

返回 0：表示用户文件具有用户指定的属性；

返回-1：表示文件不存在或者不具有用户指定的属性。

（2）创建文件夹可以采用库函数 **mkdir**，头文件<direct.h>，定义如下

```
int mkdir( const char *dirname );
```

其中，参数 *dirname* 指出需要创建的文件夹路径，如”d: \test”。



程序代码：参见 “09-file, directory/1 file”



实例 18 密码输入

④ 说明：许多程序中要求输入用户名和密码，本实例通过库函数设计一个密码程序。如同其它 DOS 程序一样，要求输入密码并禁止回显，如图 9-2 所示。

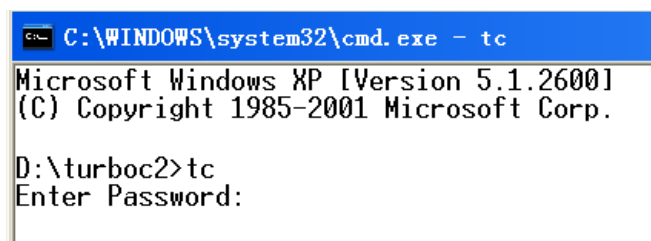


图 9-2 输入密码提醒

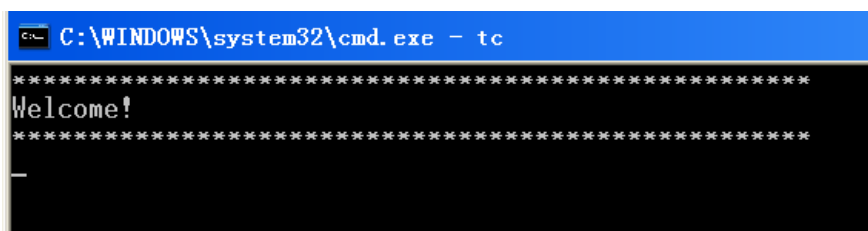
④ 要求:

- (1) 实现从控制台读入密码，并禁止回显（即不在屏幕上显示用户的输入）；
- (2) 在程序中预设一个密码（在 8 个字符以内），将用户输入与之进行匹配；
- (3) 用户只有 3 次机会，如果用户输入不正确，提醒输入错误，并提醒剩下的输入次数；

```
Enter Password:  
Password Incorrect, 2 times left  
Enter Password:  
Password Incorrect, 1 times left  
Enter Password:_
```

图 9-3 输入密码错误提醒

- (4) 如果密码正确，则刷屏，进入用户欢迎界面。



- (5) 如果三次输入错误，退出程序。



解析

知识点

- (1) 可以采用 `getpass` 函数实现本例要求的功能，头文件 `<dos.h>`，**请注意该函数仅能在 TC 环境下使用，VC 不支持。**

```
char * getpass( char *prompt );
```

参数说明:

prompt: 提示文字，类似用 `printf` 打印的打印效果，例如：`getpass(“请输入密码”)`。

返回值:

返回一个指向用户输入的字符串指针。



程序代码：参见 “09-file, directory/pass.c”

实验 10



实例 19

视窗程序编程基础

④ 说明：为了编写更良好的用户界面，Turbo C 提供了一系列屏幕操作函数，本实例将练习相关函数的用法

④ 要求：

(1) 掌握 window、clrscr、textbackground、textcolor 等函数功能及使用方法（参考“解析”部分），实现在屏幕中打开一个小窗口（宽：10 个字符，高：6 个字符），在窗口中显示两行文字：

Hello,
everybody!

要求将屏幕背景设置为白色，窗口背景设置为蓝色，文字颜色为黄色，如图 10-1 所示。



图 10-1 窗口示例

(2) 为使窗口更美观，为其绘制边框（用 gotoxy 函数设置光标位置，在窗口中输出，需要采用专门的窗口输出函数——参考“解析”部分的相关函数介绍；为使边框绘制美观，绘制的符号采用“ASCII 码的扩展集字符”，见附录 D：“标准 ASCII 码表及扩展字符集”），效果如图 10-2 所示。



图 10-2 具有边框的窗口



程序代码：参见“10-text window/window.c”



实例 20 重叠对话框

说明：用户界面友好（User Interface Friendly）的应用程序通常都采用弹出式（Pop-up）对话框输出信息，以提醒用户进行各种响应。如图 10-3 所示，当编译成功后，TC 将弹出一个对话框。本实例要求在实例 19 的基础上，实现美观、通用的弹出式对话框进行消息显示。

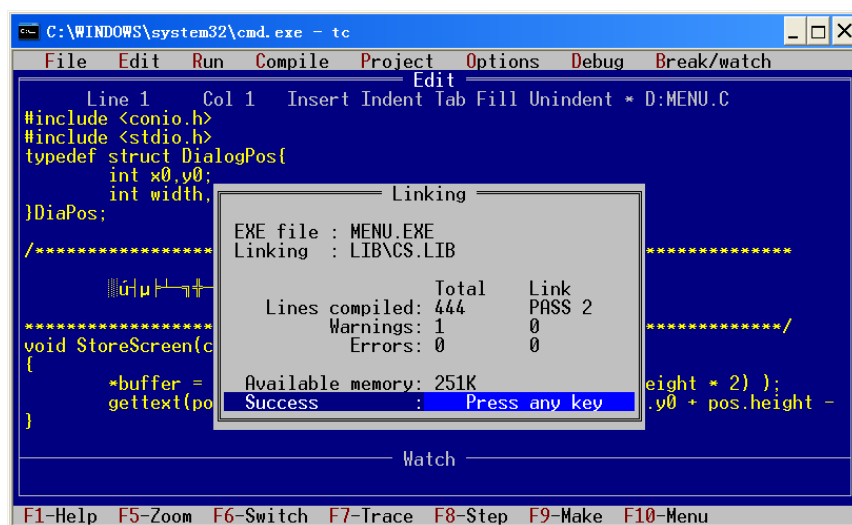


图 10-3 TC 编译成功后的弹出式对话框

要求：

- 实现图 10-4 所示的对话框，为体现对话框的立体感，要求绘制其阴影。将对话框分为上、下两部分：上部分为标题栏，居中显示对话框标题；下部分为信息栏，显示相关消息，可以有多行。要求写一个通用对话框函数，根据用户的输入的参数，可以显示不同的对话框，定义如下：

```
void Dialog( int x0, int y0, int iWidth, int iRow, char *sTitle, char *information[ ])
```

参数说明：

x0, y0: 对话框左上角的位置

iWidth: 对话框的宽度，以字符为单位（在本例中，调用时 iWidth 设置为 50）

iRow: 文字的行数（在本例中，iRow 为 2）

sTitle: 标题名（在本例中，sTitle 为字符串：” ***** NOTICE *****”）

information: 字符串指针，指向需要显示的文字信息，在本例中，参数值设置如下：

```
char *information[]={ "This is the first text window program",  
                      "press any key to begin your exploration ....."};)
```

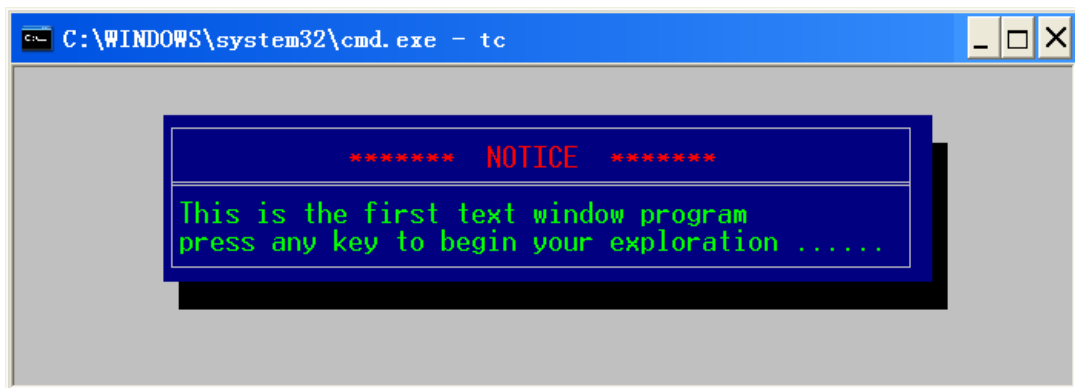


图 10-4 带阴影的弹出式对话框

- (2) 绘制对话框之前要求保存对话框所在区域的屏幕数据，以便在对话框关闭后，正确恢复信息，例如：在对话显示之前，屏幕为图 10-5 所示，当显示对话框后，屏幕为图 10-6，关闭对话框后，要求能够将屏幕恢复成图 10-5，而不是在对话框区域留下空白。参考 `gettext()` 及 `puttext()` 函数。

```
C:\WINDOWS\system32\cmd.exe - tc
17951895199520952195229523952495259526952795289529953095319532953395349535953695
37953895399540954195429543954495459546954795489549955095519552955395549555955695
57955895599560956195629563956495659566956795689569957095719572957395749575957695
77957895799580958195829583958495859586958795889589959095919592959395949595959695
97959895999600960196029603960496059606960796089609961096119612961396149615961696
17961896199620962196229623962496259626962796289629963096319632963396349635963696
37963896399640964196429643964496459646964796489649965096519652965396549655965696
57965896599660966196629663966496659666966796689669967096719672967396749675967696
77967896799680968196829683968496859686968796889689969096919692969396949695969696
97969896999700970197029703970497059706970797089709971097119712971397149715971697
17971897199720972197229723972497259726972797289729973097319732973397349735973697
37973897399740974197429743974497459746974797489749975097519752975397549755975697
57975897599760976197629763976497659766976797689769977097719772977397749775977697
77977897799780978197829783978497859786978797889789979097919792979397949795979697
97979897999800980198029803980498059806980798089809981098119812981398149815981698
17981898199820982198229823982498259826982798289829983098319832983398349835983698
37983898399840984198429843984498459846984798489849985098519852985398549855985698
57985898599860986198629863986498659866986798689869987098719872987398749875987698
77987898799880988198829883988498859886988798889889989098919892989398949895989698
97989898999900990199029903990499059906990799089909991099119912991399149915991699
17991899199920992199229923992499259926992799289929993099319932993399349935993699
37993899399940994199429943994499459946994799489949995099519952995399549955995699
57995899599960996199629963996499659966996799689969997099719972997399749975997699
77997899799980998199829983998499859986998799889989999099919992999399949995999699
9799989999_
```

图 10-5 对话框显示之前以关闭之后的屏幕

```
C:\WINDOWS\system32\cmd.exe - tc
17951895199520952195229523952495259526952795289529953095319532953395349535953695
37953895399540954195429543954495459546954795489549955095519552955395549555955695
57955895599560956195629563956495659566956795689569957095719572957395749575957695
77957895799580958195829583958495859586958795889589959095919592959395949595959695
97959895999600960196029603960496059606960796089609961096119612961396149615961696
17961896199620962196229623962496259626962796289629963096319632963396349635963696
37963896399640964196429643964496459646964796489649965096519652965396549655965696
57965896599660966196629663966496659666966796689669967096719672967396749675967696
77967896799680968196829683968496859686968796889689969096919692969396949695969696
97969896999700970197029703970497059706970797089709971097119712971397149715971697
17971897199720972197229723972497259726972797289729973097319732973397349735973697
37973897399740974197429743974497459746974797489749975097519752975397549755975697
57975897599760976197629763976497659766976797689769977097719772977397749775977697
77977897799780978197829783978497859786978797889789979097919792979397949795979697
97979897999800980198029803980498059806980798089809981098119812981398149815981698
17981898199820982198229823982498259826982798289829983098319832983398349835983698
37983898399840984198429843984498459846984798489849985098519852985398549855985698
57985898599860986198629863986498659866986798689869987098719872987398749875987698
77987898799880988198829883988498859886988798889889989098919892989398949895989698
97989898999900990199029903990499059906990799089909991099119912991399149915991699
17991899199920992199229923992499259926992799289929993099319932993399349935993699
37993899399940994199429943994499459946994799489949995099519952995399549955995699
57995899599960996199629963996499659966996799689969997099719972997399749975997699
77997899799980998199829983998499859986998799889989999099919992999399949995999699
9799989999_

***** NOTICE *****
This is the first text window program
press any key to begin your exploration .....
```

图 10-6 对话框显示状态



解析

1 知识点

1.1 字符窗口基本概念

Turbo C 中提供了字符屏幕和图形模式¹。字符屏幕的核心是窗口 (Window)，它是屏幕的活动部分，字符输出或显示在活动窗口中进行。窗口在缺省时，就是整个屏幕。窗口可以根据需要指定其大小。

窗口是在字符屏幕或者图形模式下的概念，字符只能在窗口中显示出来，这时可以访问的最小单位为一个字符。字符状态下，屏幕上的位置都是由它们的行与列所决定的。有一点须指出：字符状态左上角坐标为 (1, 1)。

了解字符屏幕和图形函数与窗口的关系是很重要的。例如，字符屏幕中，用光标位置函数 `gotoxy()` 将光标移到窗口的 (x, y) 位置上，这未必是相对于整个屏幕。下文将介绍常用的几类字符屏幕函数的功能用途、操作方法及其例行程序。

1.2 屏幕操作函数

编写程序绘图经常要用到对字符屏幕进行操作。例如，在往屏幕上写字符之前，首先要将屏幕清除干净。又如，有时需要在屏幕上多处写字符内容，这时最好用屏幕拷贝来高效率地完成这一任务。对这些操作，Turbo C 提供了一系列字符屏幕操作函数来实现。

(1) `clrscr()`清除字符窗口函数

功能：函数 `clrscr()` 清除整个当前字符窗口，并且把光标定位于左上角 (1, 1) 处。

用法：此函数调用方式为 `void clrscr(void);`

说明：括号中 `void` 表示无参数。

该函数相应的头文件为 `conio.h`

返回值：无

(2) `window()`字符窗口函数

功能：函数 `window()` 用于在指定位置建立一个字符窗口。

用法：此函数调用方式为 `void window(int left,int top,int right,int bottom);`

说明：函数中参数 `left,top` 为窗口左上角坐标;`right,bottom` 为其右下角坐标。

若有一个坐标是无效的，则 `window()` 函数不起作用。一旦该函数调用成功，那么所有定位坐标都是相对于窗口的，而不是相对于整个屏幕。但是建立窗口所用的坐标总是相对整个屏幕的绝对坐标，而不是相对当前窗口的相对坐标。这样用户就可以根据各种需要建立多个互不嵌套的窗口。

此函数的头文件为 `conio.h`。

返回值：无

¹ 本实验以及后续的图形编程实验涉及的相关函数除特殊说明外，均指仅在 Turbo C2.0 环境下可运行，而 VC6.0 编译环境不支持。VC6.0 中有另外的图形处理函数，不在本书范围内。

(3) gotoxy()光标定位函数

功能：函数 `gotoxy()` 将字屏幕上的光标移到当前窗口指定的位置上。

用法：这个函数调用方式为 `void gotoxy(int x,int y);`

说明：括号里 `x,y` 是光标定位的坐标，如果其中一个坐标值无效(如坐标超界)，那么光标不会移动。

此函数在字符状态（有时称为文本状态）下经常用到，其相应的头文件为 `conio.h`

返回值：无

7.gettext() 拷进文字函数

功能：函数 `gettext()` 用于文本状态下将屏幕上矩形域内的文字拷进内存。

用法：该函数调用方式为 `int gettext(int left, int top, int right, int bottom, void *buffer);`

说明：函数中参数 `left,top` 为矩形区域的左上角坐标,`right, bottom` 为其右下角坐标，这些坐标是屏幕的绝对坐标，不是窗口的相对坐标。`buffer` 指针必须指向一个足够保存该矩形域内文字的内存。所用内存大小按下式计算：

头用字节数=矩形内的行数×矩形域的列数×2

这里将行数乘以列数再乘以 2 的原因是保存屏幕上每个字符要用两个字节存储单元，一个字节存储单元存放字符本身，而另一个存放其属性。

此函数相应的头文件是 `conio.h`

返回值：若函数调用成功则返回 1，否则返回 0。

例：把屏幕左上角点(1,1)和右下角点(10,10)的区域拷贝到 `buf` 指向的内存中去。

```
buf=(char *)malloc(10*10*2);
if(!buf)gettext(1,1,10,10,buf);
```

8. puttext() 拷出文字函数

功能：函数 `puttext()` 把先前由 `gettext()` 保存到 `buffer` 指向的内存中的文字拷出到屏幕上一个矩形区域中。

用法：此函数调用方式为 `int puttext(int left, int top, int right, int bottom, void *buffer);`

说明：函数里 `left, top` 为给出的屏幕上矩形区域的左上角点，`right, bottom` 为其右下角点，其坐标是用屏幕的绝对坐标，而不是用窗口的相对坐标。

该函数相应的头文件为 `conio.h`

返回值：函数调用成功返回值为 1，否则返回 0。

例：屏幕上某个区域内容拷进 `buf` 指向的内存中，然后又将这些文字拷出到屏幕上新位置。

```
buf=(char *)malloc(10*10*2);
gettext(1,1,10,10,buf);
puttext(16,16,30,30,buf);
```

1.3 窗口内文本的输出函数

(1) `int cprintf("<格式化字符串>", <变量表>)`

功能： `cprintf()` 函数输出一个格式化的字符串或数值到窗口中。它与 `printf()` 函数的用法完全一样，区别在于 `cprintf()` 函数的输出受窗口限制，而 `printf()` 函数的输出为整个屏幕。

(2) `int cputs(char *string); int putch(int ch)`

功能： `cputs()` 函数输出一个字符串到屏幕上，它与 `puts()` 函数用法完全一样，只是受窗口大小的限制。

(3) `int putch(int ch)`

`putch()` 函数输出一个字符到窗口内。

注：①使用以上几种函数，当输出超出窗口的右边界时会自动转到下一行的开始处继续输出。②当窗口内填满内容仍没有结束输出时，窗口屏幕将会自动逐行上卷直到输出结束为止。

1.4 字符属性函数

(1) `textcolor()` 文本颜色函数

功能： 函数 `textcolor()` 设置字符屏幕下文本颜色(或字符颜色)，它也可以用于使字符闪烁。

用法： 这个函数调用方式为 `void textcolor(int color);`

说明： 函数中参数 `color` 的有效值可取表 10-1 中的颜色名(即宏名)或等价值。

例如： `textcolor (BLACK)` 等价于 `textcolor (0)`

注： `textcolor()` 函数执行后，只影响其后输出探险符颜色，而不改变已经在当前屏幕上的其它字符颜色。显然，如果需要输出的字符闪烁，只要将函数中参数 `color` 取为 `BLINK` 即可，如果要使字符带颜色闪烁，就必须将所选的颜色值与 128 作“或”运算。

此函数相应的头文件是 `conio.h`

返回值： 无

例： 下面程序中第一条语句使输出的字符为蓝色，第三条语句使后续字符输出为红色

```
textcolor(BLUE);
printf("hello");
textcolor(RED);
```

表 10-1 颜色名与等价值

名	等价值	含义
BLACK	0	黑
BLUE	1	蓝
GREEN	2	绿
CYAN	3	青
RED	4	红
MAGENTA	5	洋红

BROWN	6	棕
LIGHTGRAY	7	淡灰
DRAKGRAY	8	深灰
LIGHTBLUE	9	淡蓝
LIGHTGREEN	10	淡绿
LIGHTCYAN	11	淡青
LIGHTRED	12	淡红
LIGHTMAGENTA	13	淡洋红
YELLOW	14	黄
WHITE	15	白
BLINK	128	闪烁

(2) textbackground() 文本背景函数

功能：函数 `textbackground()` 设置字符屏幕下文本背景颜色(或字符背景颜色)。

用法：此函数调用方式为 `void textbackground(int bcolor);`

说明：参数 `bcolor` 的有效值取表 10-2 背景颜色(即宏名)或等价值。

表 10-2 背景颜色与等价值

背景颜色	等价值	含义
BLACK	0	黑
BLUE	1	蓝
GREEN	2	绿
CYAN	3	青
RED	4	红
MAGENTA	5	洋红
BROWN	6	棕

2 代码解析

2.1 开辟窗口代码

作为示例，实例 19 “要求 1” 的实现代码如下：

```
textbackground(WHITE);
clrscr();
window(20, 10, 40, 15); /* 窗口起始坐标：(20,10)，结束坐标(40,15) */
textbackground(BLUE);
clrscr();

textcolor(YELLOW);
cprintf("\r\n    Hello    ");
cprintf("\r\n    everybody!");
```

代码解析：

- 未开辟小窗口时（即未调用 `window` 函数），调用 `textbackground`、`clrscr` 等函数，都是针对整个屏幕进行操作，如代码中的前两行，设置了屏幕的背景颜色并清屏；开辟小窗口（即调用 `window` 函数后）后，操作均是针对当前窗口；
- `textbackground` 函数仅仅设置了屏幕（或者窗口）的颜色，如果不调用 `clrscr` 并不会改变屏幕的颜色，这两个函数**通常联合使用**：首先通过 `textbackground` 设置背景色，然后通过 `clrscr` 用背景色对屏幕进行刷屏。示例见上面代码第 1, 2 行及第 3、5 行；
- 在窗口用输出文字注意要使用 `cprintf` 或者 `cputs` 等，见前文知识点部分内容；
- 使用 `cprintf` 换行注意要使用 `'\r\n'`，仅使用 `'\n'` 只能换行，不会回车，请读者测试。

2.2 绘制边框代码

实例 19 “要求 2”的绘制边框可以参考如下代码：

```
/* 边框绘制 */
/*1 横线 */
for (i = 2; i < 20; i ++){
    gotoxy(i, 1); /* 设置光标位置：第一行，从第2个字符位开始*/
    putchar(205); /* 输出双横线字符，注：205 > 127，是一个ASCII扩展字符*/
    gotoxy(i, 6); /* 设置光标位置：最后一行，从第2个字符位开始*/
    putchar(205);
}

/*2 竖线 */
for (i = 2; i < 6; i ++){
    gotoxy(1, i);
    putchar(186);
    gotoxy(20, i);
    putchar(186);
}

/*3 四个角 */
gotoxy(1, 1); /*窗口(1,1)位置，也是屏幕的坐标(20,10) */
putchar(201);

gotoxy(20, 1);
putchar(187);

gotoxy(1, 6);
putchar(200);

gotoxy(20, 6);
putchar(188);
```

代码解析：

为了边框美观，绘制的符号采用“ASCII 码的扩展集”，见附录 D：“标准 ASCII 码表及扩展字符集”



程序代码：参见“10-text window/dialog.c”

实验 11



实例 21 键盘消息

④ 说明：如何判断用户的按键，是程序设计中通常遇到的问题。本实例要求能够判断用户的按键信息，并打印出来。

④ 要求：

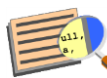
(1) 实现主要按键信息判断，并打印出来（见图 11-1），要求**至少**包括如下按键：

- F1~F12
- 4 个方向键
- Page Up, Page Down
- Home, End
- 字母 A~Z, a~z
- 数字 0~9
- 其它常用字符，如*, (,), -, + 等
- Backspace（退格）

(2) 程序循环执行，显示用户的所有按键信息，直到用户选择 Esc 退出，如下图所示：

```
C:\WINDOWS\system32\cmd.exe - tc
Up
Down
Right
Left
F1
Backspace
Backspace
Up
```

图 11-1 程序执行效果图



解析

1 知识点

1.1 键盘上的按键

键盘提供了三种基本类型的键：

- (1) 字符键，如字母 A—Z，数字 0—9，%，+等；
- (2) 扩展功能键：如 Home, end, PageUp, PageDown, Up, Down, Left, Right 等程序功能键；
- (3) 和其它键组合使用的控制键：如 Alt, Ctrl 和 Shift

如何判断用户的按键信息，是程序设计中通常遇到的问题。对于第（1）类按键，可以通过与其标准 ASCII 码进行匹配来判断，而第（2）（3）类键则通常没有与之对应的 ASCII 码值，故需要采用另外的策略进行判断。

1.2 键盘扫描码

事实上，当按下键盘上某个键（如 A）时，将产生两个数据：

- (1) 扫描码（Scan Code）；
- (2) ASCII 码。

扫描码是唯一指定给键盘上每一个键的编码（包括上面三类按键），它与 ASCII 码无关。多数情况下（通常在进行游戏编程设计时），你只想了解用户是否按下了 A 键，而并不关心到底是大写（A）还是小写（a），即将该键当作一种瞬时开关来使用，通过扫描码就可以很好地完成任务。IBM 键盘扫描码表详见附录 E。

1.3 获取扫描码函数 bioskey

函数原型：int bioskey (int cmd)说明：bioskey()接收的是扫描码。

头文件：bios.h

功 能：直接使用 BIOS 服务的键盘接口（也可以理解为：读取键盘值）

输入参数：

cmd=0 返回一个键盘值，如无键盘按下，一直等待。当 cmd 是 0，bioskey()返回下一个在键盘键入的值（它将等待到按下一个键）。它返回一个 16 位的二进制数，包括两个不同的值。当按下一个普通键时，它的低 8 位数存放该字符的 ASCII 码；对于特殊键（如方向键、F1~F12 等等），低 8 位为 0，高 8 位字节存放该键的扫描码。

cmd=1 查询键盘是否按下，返回 0 表示无键按下，返回非 0 表示有键按下。

cmd=2 返回控制键状态，即返回 Shift、Ctrl、Alt、ScrollLock、NumLock、CapsLock、Insert 键的状态。各键状态存放在返回值的低 8 位字节中，如下表所示：

表 11-1 返回字节位含义

字节位	含义
0	右边 Shift 键状态
1	左边 Shift 键状态
3	Ctrl 键状态
4	Alt 键状态
5	ScrollLock 键状态
6	NumLock 键状态
7	CapsLock 键状态
8	Insert 键状态

注：字节位为 1 表示该键被按下，为 0 表示松开。

2 代码解析

(1) 写一个独立的按键信息捕获函数 `int get_key()`，返回用户所按键的编码，如下所示

```
typedef union
{
    int word;
    char byte;
}keycode;

int get_key()
{
    keycode key;
    key.word=bioskey(0);
    return key.byte ? key.byte: key.word;
}
```

(2) 用宏定义来定义常用键用 `get_key` 函数调用后的返回值，类似下面代码

```
#define Esc    27
#define Enter  13
#define Up     0x4800
#define Down   0x5000
```

注：Esc, Enter（回车）有对应的 ASCII 码，故根据 `bioskey` 函数的特征，我们直接采用其 ASCII 码。请思考：对于字符键，应该如何处理？

(3) 主程序采用 **while** 循环加 **swith** 的方式，参考以下代码：

```
key=get_key();
while(key!=Esc)
{
    switch(key)
    {
        case Up:      .....
        case Down:    .....
        .....
        default: break;
    }
    key=get_key();
}
```

注意：以上代码没有处理字符键。可以想像，如果按照以上的方式，当需要判断所有字符时，需要添加 128 条 **case** 语句（ASCII 表中的字符数目）。请思考：当用户按任意字符键时，如何在代码中不进行一一匹配，就可以直接输出用户的按键字符？



程序代码：参见 “11-menu/key.c”



实例 22

绘制弹出式菜单

说明：菜单是与用户交互的一个重要接口，本例要求根据自己应用程序的功能绘制一个弹出式菜单，并包括多个菜单子项，用户可以通过键盘上、下方向键选择相应的子键，按下“回车”键程序执行相应的子功能。如图 11-2 所示。

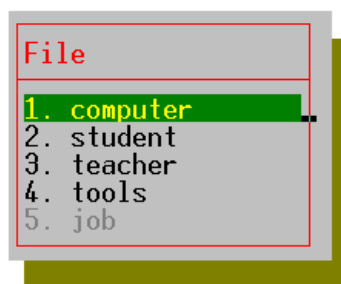


图 11-2 弹出式菜单

要求：

(1) 显示一个带阴影的弹出式菜单，要求绘制菜单函数具有较好的封装性，能够根据需
要移植到不同程序；

(2) 参考图 11-2 的样式，对于用户选中的子项字体颜色及背景要与其它菜单项有区别；

(3) 由于本实例重点练习菜单的绘制，对菜单项所涉及的具体功能并不要求，故只需要能够判断用户当前的选择即可。如图 11-3 所示，另开一个窗口，专门显示用户当前选择的菜单子项。

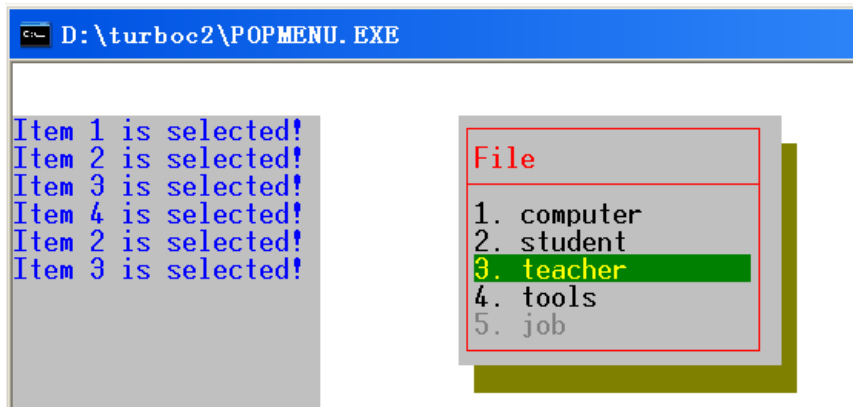


图 11-3 对用户选择的响应。左边的窗口专门开辟为显示当前用户选择的菜单项

(4) 能够设置一些用户无法选择的菜单子项（这通常是应用程序根据具体需要的一种保护措施，比如该功能执行的条件尚未成熟，或者该功能根本就还未开发），比如图 11-2 的“job”项，用户回车后不响应，并要求对这类菜单子项用特殊的颜色（如灰色）显示；

(5) 要求程序可以循环执行，直到用户按 ESC 键退出程序。类似实例 20，要求能够恢复菜单显示之前屏幕的文字信息。

解析

(1) 本实例实际上是对前面实例 20、21 的综合练习，可以参考实例 20 绘制主菜单框架。但注意，由于菜单子项都是独立的，如选中时和非选中时显示效果不同，故可以考虑封装一个函数专门用以绘制一个菜单子项，绘制效果可以根据参数来设置。如下面代码所示：

```
/*----用于显示一个菜单条----*/
void DipItem(int row,int text_color,int back_color,char *menu_item)
/* 参数说明: row,该菜单子项的行位置; text_color,菜单子项文字颜色;
   back_color, 菜单子项背景颜色; menu_item, 菜单项文字*/
{
    gotoxy(2,row);
    textbackground(back_color);
```

```

        textcolor(text_color);
        cputs(menu_item);
    }

```

(2) 如图 11-3 所示，要实现在两个窗口之间进行切换，并正确显示相应的文字信息，要求在一个窗口中要保存另一个窗口的当前光标的位置，如下所示：

```

void output(int out, int x, int y, int width, int height)
/* 参数说明： out, 输出的信息； x, y, width, height, 是当前光标所在窗口位置及大小 */
{
    struct text_info *r;
    static int cury=1;
    gettextinfo(r); /*保存当前窗口中光标位置*/
    /*开新窗口*/
    window(1, 5, 20, 15);
    textbackground(WHITE);
    if(cury == 1) clrscr();
    textcolor(LIGHTBLUE);
    gotoxy(1, cury++);
    cprintf("Item %d is selected!\n", out);

    /* 恢复光标 */
    window(x, y, x+width, y+height);
    gotoxy(r->curx, r->cury);
}

```



程序代码：参见 “11-menu/popmenu.c”

实验 12



实例 23 文本阅读器

④ 说明：本实例要求实现文本模式下的一个简单文本编辑器，界面类似 Turbo C 编辑环境，可以实现文本文件的读入和显示。

④ 要求：

(1) 友好的用户界面，首先用一个弹出式窗口（可以使用实例 20 的 dialog 函数）提示用户输入文件名，如图 12-1 所示，其中黄色部分是用户输入的文件名。

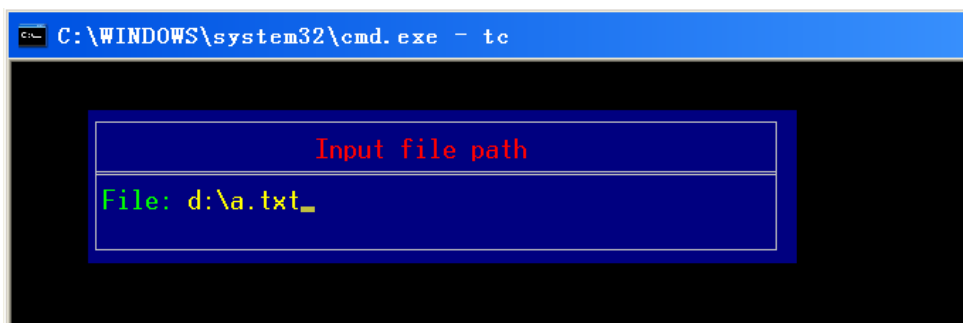


图 12-1 弹出式菜单用于用户输入文件名

(2) 在用户输入文件名回车后，关闭上面的对话框，并打开相应的文本文件，然后将其内容显示在一个文本窗口中，如图 12-2 所示（读者可以根据实例 20 的内容，进一步美化界面，如仿照 TC 编辑窗口，添加边框等）。要求：当文本文件内容行数大于窗口行数时，能够通过上、下方向键进行翻页，定位相应部分的文本内容。

(3) 能够响应上、下、左、右四个方向键，修改光标的位置。

(4) 能够响应 Esc 键，退出应用程序。

(a) 文本文件 d:\a.txt

(b) 自编的文本阅读器

图 12-2 文本文件及文本阅读器

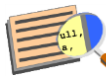


实例 24 文本编辑器

说明：本实例在实例 23 的基础上，添加文本编辑、保存功能。

要求：

- (1) 能够响应退格 (BackSpace) 键，删除相应的文字；
- (2) 能够添加文字，修改文字；
- (3) 能够响应 Enter 键，添加行数；
- (4) 具备存盘功能，根据用户的输入，保存为新的文件。



解析

(1) 获取用户在窗口输入的文字可以使用“窗口内文本的输入函数”int getche(void); 该函数需要说明的是，getche()函数从键盘上获得一个字符，在屏幕上显示的时候，如果字符超过了窗口右边界，则会被自动转移到下一行的开始位置。由于 getche 获取的是用户输入的每一个字符，无法像 gets 函数一样可以返回一组字符串，然后用回车结束输入，因此需要自己判断用户是否输入回车键，如下代码所示：

```

i = 0;
gotoxy(8, 4);
key=getche();
while(key != Enter) {           /* #define Enter 13 */
    filename[i++] = key;
    key=getche();
}

```

(2) 采用 `getche` 函数获取用户输入（如上面的代码）最大的问题是，无法直接响应功能键，如退格 `BackSpace` 键、回车键 `Enter` 等，用户需要自己添加处理代码，相对麻烦。因此，建议使用 `char * gets(char *s)` 函数直接获取用户输入的字符串，直到回车为止。但值得指出的是：`textcolor` 函数对 `gets` 函数无效，即不能自行设置文字颜色。可以用如下代码代替上面的代码段，更容易操作：

```

gotoxy(8, 4);
gets(filename);

```

(3) 要将文本文件的内容进行显示，需要将文本文件的字符全部读入到内存中（当文件较大时，由于内存限制，一个实用的编辑通常一次只读取部分数据，本实例所显示的文件通常较小，故不考虑分块读取数据问题）。故首先需要明确文件字符数目，以方便开辟内存空间，如下代码所示：

```

/*—获得文件长度length—*/
while(!feof(fp))
{
    fgetc(fp);
    length++;
}
rewind(fp); /*—fp回到头部—*/
/*—将文章读到内存当中text数组中—*/
text=(char *)malloc(length*sizeof(char));
fread(text, sizeof(char), length, fp);

```

(4) 文本文件的内容需要逐行显示，故另一个重要问题是获取文件的行数，可以通过读取文件中的字符 `'\n'` 的数目来确定行数，如下代码所示：

```

int RowNum(char *text, int len)
{
    int number=0;

```

```

int i;
for(i=0;i<len;i++)
{
    if(text[i]=='\n')
        number++;
}
return(++number);
}

```

(5) 当文本行数大于窗口行数时，本实验的关键是要随时掌握当前窗口显示内容的所在文本文件中的位置，建议设计一个结构体，专门保存当前窗口在文件中的位置：

```

typedef struct
{int top;
  int bottom;
}screen;

```

(6) 程序主框架可以参考如下伪码：

```

begin:
    key=get_key();
    switch(key)
    {
        case Down:
            1. 根据当前操作设置dips_text函数（其中dip_text函数需要自行编写，
               主要用于在窗口中显示格式化字符串）的相关参数，如显示字符串，
               长度、当前首行位置等。
            2. 调用dip_text函数
            3. goto begin;
        case Up :
            1. 根据当前操作设置dips_text函数（其中dip_text函数需要自行编写，
               主要用于在窗口中显示格式化字符串）的相关参数，如显示字符串，
               长度、当前首行位置等。
            2. 调用dip_text函数
            3. goto begin;
        case Left:
            1. 根据当前操作设置dips_text函数（其中dip_text函数需要自行编写，
               主要用于在窗口中显示格式化字符串）的相关参数，如显示字符串，
               长度、当前首行位置等。
            2. 调用dip_text函数
            3. goto begin;
        case Right:
            1. 根据当前操作设置dips_text函数（其中dip_text函数需要自行编写，
               主要用于在窗口中显示格式化字符串）的相关参数，如显示字符串，
               长度、当前首行位置等。
    }

```

```
        2. 调用dip_text函数
        3. goto begin;
    case Esc: goto over;
    default : .....; goto begin;
}
over: cprintf("over");
```

注意：从上面的代码可以看到，使用了 goto 语句。通常结构化的程序设计并不建议使用 goto 语句，因为随意跳转会增加程序的复杂度，容易出错，增加 BUG 潜在的机率，也会给阅读者带来困扰。当然，goto 语句自然也有其存在的理由，实际上，只要使用合理，goto 语句也会提高编码效率，可以很快地从多重循环中跳出，直达目标代码段。总的来说，goto 语句的使用不要过度，也不应过于避讳。



程序代码：参见“12-editor.c”

实验 13



实例 25

图形绘制

④ **说明：**绘制图形是可视化编程的重要手段。Turbo C 提供了非常丰富的图形函数。本实例主要学习图形模式的基本概念，掌握图形模式的初始化、基本图形绘制等，要求能够灵活运用相关函数进行复杂图形的绘制。

④ **要求：**

(1) 掌握图形模式的初始化函数及显卡检测函数，能够自动得出当前显卡驱动兼容的模式（何谓显卡驱动模式，参考后文“解析”部分）并将其信息打印出来，例如：

```
The graphics driver is 9, mode is 2.
```

(2) 掌握画线及设置线型函数，绘制一个立方体，要求被遮挡的三条边线为虚线，颜色也与可见边相区别，参考图 13-1：

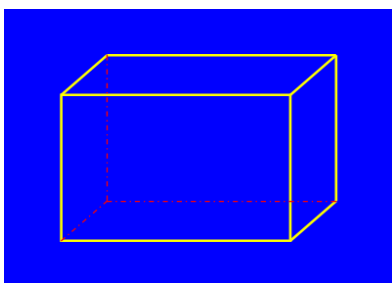


图 13-1 绘制立方体

(3) 掌握背景、前景颜色设置函数 `setbkcolor()` 及 `setcolor()`，能够灵活设置背景及图形颜色。



实例 26

创建可独立运行的图形程序

④ **说明：**由于 Turbo C 编写的图形程序在运行时调用了相应的显卡驱动程序，如果在发布程序时没有将相应的驱动程序一起发布，则该图形程序在别的机

器上不能正常运行。本实例要求对实例 25 进行改进，生成能够独立运行的图形程序。

④ 方法

Turbo C 对于用 `initgraph()`函数直接进行的图形初始化程序，编译和链接时并没有将相应的驱动程序 (*.BGI) 装入到执行程序，程序进行到 `initgraph()` 语句时，从该函数中第三个形式参数 `char *path` 中所规定的路径中去查找相应的驱动程序。若没有驱动程序，则在 C:\TC 中去找，如 C:\TC 中仍没有或 TC 不存在，将会出现错误，表现为程序无法正常运行。

因此，为了使用方便，应该建立一个不需要驱动程序就能独立运行的可执行图形程序，可以采用下述步骤（这里以 EGA、VGA 显示器为例，假设 Turbo C 编译环境的路径为 “C:\TC”）：

- (1) 点击“开始”，选择“运行…”，输入“cmd”，进入命令窗口，如图 13-2 所示：

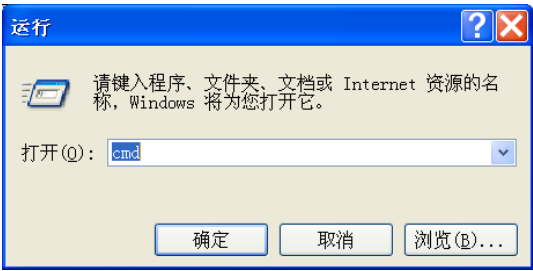


图 13-2 运行 CMD

- (2) 在 DOS 命令行中，进入 “C:\TC” 子目录；
- (3) 输入命令：

BGIOBJ EGAVGA

download

English

Install

Java

movie

msdos71f

MySQL InnoDB Data

Program Files

RECYCLER

System Volume Inf

AA.OBJ

B.OBJ

C.OBJ

CMDS.OBJ

COUNTRY.OBJ

DTA.OBJ

EGAVGA.OBJ

FILE.OBJ

MENU.OBJ

NONAME.OBJ

PASS.OBJ

POPMENU.OBJ

TEST.OBJ

2 KB Intermediate file 2010-5-3 21:08

4 KB Intermediate file 2010-5-17 22:25

6 KB Intermediate file 2010-5-17 23:01

2 KB Intermediate file 2010-4-19 23:11

2 KB Intermediate file 2010-4-19 22:37

1 KB Intermediate file 2010-4-19 21:53

6 KB Intermediate file 2010-5-26 21:09

1 KB Intermediate file 2009-12-20 21:50

Intermediate file 2010-5-6 18:29

Intermediate file 2010-5-6 17:40

Intermediate file 2010-4-26 23:13

Intermediate file 2010-5-12 22:37

1 KB Intermediate file 2010-4-19 22:51

类型: Intermediate file
修改日期: 2010-5-26 21:09
大小: 5.33 KB

图 13-3 在 TC 目录中生成了 EGAVGA.OBJ 文件

此命令将“C:\TC”目录下的驱动程序“EGAVGA.BGI”转换成“EGAVGA.OBJ”的目标文件，可以从资源管理器进入“C:\TC”目录进入查看，如图 13-3 所示：

(4) 继续在命令行中输入命令：

TLIB LIB\GRAPHICS.LIB+EGAVGA

此命令是将 EGAVGA.OBJ 的目标模块装载到 GRAPHICS.LIB 库文件中。可以从资源管理器进入“C:\TC\LIB”目录，如图 13-4 所示，在 TC\LIB 目录中更新了 GRAPHICS.LIB 文件（查看文件的修改日期），并将原始文件进行了备份。

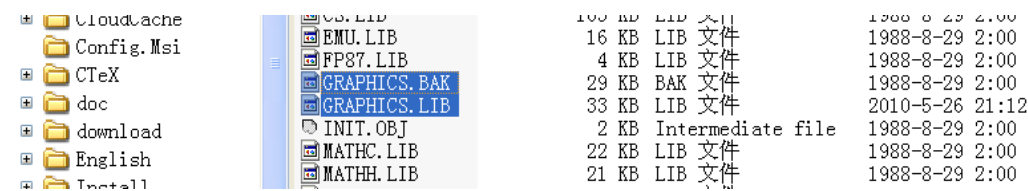


图 13-4 在 TC\LIB 目录中更新的 GRAPHICS.LIB 文件及其备份

(5) 在程序中 initgraph()函数调用之前加上一条语句：

registerbgidriver(EGAVGA_driver);

该函数告诉连接程序在连接时把 EGAVGA 的驱动程序装入到用户的执行程序中。

至此，编译链接后的执行程序可在任何目录或其它兼容机上运行，假设已作了前 4 个步骤，一个可以独立运行的简单绘图程序如下：

```
#include<stdio.h>
#include<graphics.h>
int main()
{
    int gdriver=DETECT,gmode;
    registerbgidriver(EGAVGA_driver); /*建立独立图形运行程序 */
    initgraph( gdriver, gmode,"c:\\tc");
    /*
    此处自行添加绘图代码
    */
    getch();
    closegraph();
    return 0;
}
```


注：如不初始化成 EGA 或 CGA 分辨率，而想初始化为 CGA 分辨率，则只需要将上述步骤中有 EGA/VGA 的地方用 CGA 代替即可。



1. 确定显卡，加载驱动

计算机系统显示部分由显示器和显（示）卡（adapter）两部分组成。显示器是独立于主机的一种外部设备；显卡或称显示适配器，也有的称图形卡，是图形显示的核心部件。不同的显卡有不同的图形分辨率。即是同一显卡，在不同模式下也有不同分辨率。因此，在屏幕绘图之前，必须根据显卡种类将屏幕设置成为某种图形模式，在未设置图形模式之前，系统默认屏幕为文本模式（80 列，25 行字符模式），此时所有图形函数均不能工作。设置屏幕为图形模式可用下面的图形初始化函数：

```
void far initgraph(int far *gdriver, int far *gmode, char *path);
```

其中，gdriver 和 gmode 分别表示图形驱动器和模式，path 是指图形驱动程序所在的目录路径。表 13-1 列出了 Turbo C 图形库支持的显卡类型，指出了图形驱动器、图形模式的符号常数及对应的分辨率。

表 13-1 图形驱动器、模式的符号常数及数值

图形驱动器(gdriver)		图形模式(gmode)		色调	分辨率
符号常数	数值	符号常数	数值		
CGA	1	CGAC0	0	C0	320*200
		CGAC1	1	C1	320*200
		CGAC2	2	C2	320*200
		CGAC3	3	C3	320*200
		CGAHI	4	2 色	640*200
MCGA	2	MCGAC0	0	C0	320*200
		MCGAC1	1	C1	320*200
		MCGAC2	2	C2	320*200
		MCGAC3	3	C3	320*200
		MCGAMED	4	2 色	640*200
		MCGAHI	5	2 色	640*480
EGA	3	EGALO	0	16 色	640*200
		EGAHI	1	16 色	640*350
EGA64	4	EGA64LO	0	16 色	640*200
		EGA64HI	1	4 色	640*350
EGAMON	5	EGAMONHI	0	2 色	640*350
IBM8514	6	IBM8514LO	0	256 色	640*480
		IBM8514HI	1	256 色	1024*768
HERC	7	HERCMONHI	0	2 色	720*348
ATT400	8	ATT400C0	0	C0	320*200
		ATT400C1	1	C1	320*200
		ATT400C2	2	C2	320*200
		ATT400C3	3	C3	320*200

		ATT400MED ATT400HI	4 5	2 色 2 色	320*200 320*200
VGA	9	VGA LO VGAMED VGAHI	0 1 2	16 色 16 色 16 色	640*200 640*350 640*480
PC3270	10	PC3270HI	0	2 色	720*350
DETECT	0	用于硬件测试			

示例 1：使用图形初始化函数设置 VGA 高分辨率图形模式

```
#include <graphics.h>
int main()
{
    int gdriver, gmode;
    gdriver=VGA;
    gmode=VGAHI;
    initgraph(&gdriver, &gmode, "c:\\tc");
    /*
    此处添加图形绘制操作
    */
    getch();
    closegraph(); /* 退出绘图状态 */
    return 0;
}
```

当然，一个兼容性好的程序通常能够在不同的显卡下工作，即能够检测当前运行时的显卡种类，Turbo C 提供了一个自动检测显示器硬件的函数，其调用格式为：

```
void far detectgraph(int *gdriver, *gmode);
```

其中，gdriver 和 gmode 的意义与 initgraph 函数相同。

示例 2：自动进行硬件测试后进行图形初始化

```
#include <graphics.h>
int main()
{
    int gdriver, gmode;
    detectgraph(&gdriver, &gmode); /*自动测试硬件*/
    initgraph(&gdriver, &gmode, "c:\\tc"); /* 根据测试结果初始化图形*/
    /*
    此处添加图形绘制操作
    */
}
```

```

    */
    closegraph();/* 退出绘图状态 */
    return 0;
}

```

上例程序中先对图形显示器自动检测，然后再用图形初始化函数进行初始化设置。但 Turbo C 提供了一种更简单的方法，即将参数 `gdriver` 设置为 `DETECT` 语句后，再调用 `initgraph()` 函数。因而，上例可改为：

```

#include <graphics.h>
int main()
{
    int gdriver=DETECT, gmode;
    initgraph(&gdriver, &gmode, "c:\\tc");

    /*
    此处添加图形绘制操作
    */

    closegraph();/* 退出绘图状态 */
    return 0;
}

```

注意：结束绘图模式后，需要调用退出图形状态函数 `closegraph()`，从而退出图形状态而进入文本方式（Turbo C 默认方式），并释放用于保存图形驱动程序和字体的系统内存。其调用格式为：

```
void far closegraph(void);
```

2. 屏幕及图形颜色的设置

对于图形模式的屏幕颜色设置，同样分为背景色的设置和前景色的设置。在 Turbo C 中分别用下面两个函数。

设置背景色： `void far setbkcolor(int color);`

设置作图色： `void far setcolor(int color);`

其中，`color` 为图形方式下颜色的规定数值，对 EGA，VGA 显示器适配器，有关颜色的符号常数及数值见下表所示。

表 13-2 有关屏幕及图形颜色的符号常数表

符号常数	数值	含义	符号常数	数值	含义
BLACK	0	黑色	DARKGRAY	8	深灰

BLUE	1	兰色	LIGHTBLUE	9	深兰
GREEN	2	绿色	LIGHTGREEN	10	淡绿
CYAN	3	青色	LIGHTCYAN	11	淡青
RED	4	红色	LIGHTRED	12	淡红
MAGENTA	5	洋红	LIGHTMAGENTA	13	淡洋红
BROWN	6	棕色	YELLOW	14	黄色
LIGHTGRAY	7	淡灰	WHITE	15	白色

清除图形屏幕内容使用清屏函数，其调用格式如下：

```
void far cleardevice(void);
```

3. 基本图形函数

基本图形函数包括画点，线以及其它一些基本图形的函数，下面对一些主要函数进行介绍。

(1) 画点函数

```
void far putpixel(int x, int y, int color);
```

该函数表示在指定的位置画一个点，点的颜色由参数 `color` 确定，对于颜色 `color` 的取值可从参考表 2。

在图形模式下，是按像素来定义坐标的。对 VGA 适配器，它的最高分辨率为 640x480，其中 640 为整个屏幕从左到右所有像素的个数，480 为整个屏幕从上到下所有像素的个数。屏幕的左上角坐标为(0, 0)，右下角坐标为(639, 479)，水平方向从左到右为 x 轴正向，垂直方向从上到下为 y 轴正向。

(2) 画线函数

TURBO C 提供了一系列画线函数，下面介绍两个常用的：

```
void far line(int x0, int y0, int x1, int y1);
```

该函数画一条从点 (x0, y0) 到 (x1, y1) 的直线。

```
void far lineto(int x, int y);
```

该函数画一条从当前坐标到点 (x, y) 的直线。

在使用 `lineto` 函数之前，需要通过调用 `moveto()` 函数设置起始点位置，该函数的原型为：

```
void far moveto(int x, int y);
```

(3) 其它函数

● 圆

```
void far circle(int x, int y, int radius);
```

以 (x, y) 为圆心, radius 为半径, 画一个圆。

● 圆弧

```
void far arc(int x, int y, int stangle, int endangle, int radius);
```

以(x, y)为圆心, radius 为半径, 从 stangle 开始到 endangle 结束(用度表示)画一段圆弧线。在 TURBO C 中规定 x 轴正向为 0 度, 逆时针方向旋转一周, 依次为 90, 180, 270 和 360 度。

● 椭圆

```
void ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);
```

以(x, y)为中心, xradius, yradius 为 x 轴和 y 轴半径, 从角 stangle 开始到 endangle 结束画一段椭圆线, 当 stangle=0, endangle=360 时, 画出一个完整的椭圆。

● 矩阵

```
void far rectangle(int x1, int y1, int x2, int y2);
```

以(x1, y1)为左上角, (x2, y2)为右下角画一个矩形框。

● 多边形

```
void far drawpoly(int numpoints, int far *polypoints);
```

画一个顶点数为 numpoints, 各顶点坐标由 polypoints 给出的多边形。polypoints 整型数组必须至少有 2 倍顶点数个元素。每一个顶点的坐标都定义为 x, y, 并且 x 在前。注意: 当画一个封闭的多边形时, numpoints 的值取实际多边形的顶点数加一, 并且数组 polypoints 中第一个和最后一个点的坐标相同。

(4) 设定线型函数

在没有对线的特性进行设定之前, 采用其默认值, 即宽度为 1 像素的实线。同时, 绘图库也提供了可以改变线型的函数。线型包括: 宽度和形状。其中宽度只有两种选择: 一像素宽和三像素宽。而线的形状则有五种。下面介绍有关线型的设置函数:

```
void far setlinestyle(int linestyle, unsigned upattern, int thickness);
```

该函数用来设置线的有关信息, 其中 linestyle 是线形状的规定, 见表 13-3。thickness 是线的宽度, 见表 13-4。

参数 `pattern` 用于自定义线图样，它是 16 位（bit）字，只有当 `style=USERBIT_LINE`（值为 1）时，`pattern` 的值才有意义，使用用户自定义线图样，与图样中“1”位对应的像素显示，因此，`pattern=0xFFFF`，则画实线；`pattern=0x9999`，则画每隔两个像素交替显示的虚线，如果要画长虚线，那么 `pattern` 的值可为 `0xFF00` 和 `0xF00F`，当 `style` 不为 `USERBIT_LINE` 值时，虽然 `pattern` 的值不起作用，但仍须为它提供一个值，一般取为 0。

表 13-3 有关屏幕 有关线的形状(linestyle)

符号常数	数值	含义
<code>SOLID_LINE</code>	0	实线
<code>DOTTED_LINE</code>	1	点线
<code>CENTER_LINE</code>	2	中心线
<code>DASHED_LINE</code>	3	点画线
<code>USERBIT_LINE</code>	4	用户定义线

表 13-4 有关线宽(thickness)

符号常数	数值	含义
<code>NORM_WIDTH</code>	1	一像素宽
<code>THIC_WIDTH</code>	3	三像素宽



程序代码：参见“13-initGraph.C”

实验 14



实例 27

图形模式中的文字显示

④ **说明：**图形系统中文字显示是必不可少的，Turbo C2.0 图形库也提供了专门用于在图形显示模式下的文字输出函数。本实例通过学习相关函数的使用方法，实现样式丰富的文字显示。

④ **要求：**

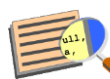
(1) 将屏幕设置为白色，在屏幕上显示如下文字，其中第一行起始位置为 (350, 50)，其它行根据情况自行设置；第一行文字颜色为蓝色，字体大小为“16*16”的点阵，其它行为红色，字体大小为“8*8”的点阵。

*** Tool seletion ***

1. cubic
2. Tiangle
3. line
4. triangle
5. exit

(2) 掌握设置文字属性函数，实现纵向输出文字（文字大小为 16*16 点阵，颜色为蓝色），并将字体设置为“笔划式三倍字型”，如下所示：

Hello World!



解析

Turbo C2.0 图形库提供了一些专门用于在图形显示模式下的文本输出及设置相关函数。下面分别进行介绍：

1. 文字输出函数

```
void far outtext(char far *textstring);
```

该函数在当前位置输出字符串 `textstring`。注：如果要在屏幕上任意位置上显示文字，该函数必须要和函数 `moveto()` 函数结合使用，`moveto` 函数用来设置当前画笔的位置，其原型为：

```
void far moveto(int x, int y);
```

当然，图形库也提供了一个函数 `outtextxy()`，用于在坐标 (x, y) 处开始顺序输出字符串 `textstring`，该函数的功能是 `moveto()` 和 `outtext()` 的整合体，读者可以认为函数 `outtextxy()` 在绘图前，调用了 `moveto()` 函数。`outtextxy()` 函数的原型为：

```
void far outtextxy(int x, int y, char far *textstring);
```

该函数以坐标 (x, y) 为输出起始位置，顺序输出字符串 `textstring`。其中， x, y 位置必须在屏幕像素范围内（参考实验 13 的解析第 1 部分内容“1. 确定显卡，加载驱动”），才能正确地输出文字。

2. 文字属性设置

(1) 文字颜色设置，采用 `setcolor` 函数，函数原型为：

```
void far setcolor(int color);
```

其中，`color` 为图形方式下颜色的规定数值，对 EGA, VGA 显示器适配器，有关颜色的符号常数及数值见如表 14-1 所示：

表 14-1 文字颜色的符号常数表

符号常数	数值	含义	符号常数	数值	含义
BLACK	0	黑色	DARKGRAY	8	深灰
BLUE	1	蓝色	LIGHTBLUE	9	深兰
GREEN	2	绿色	LIGHTGREEN	10	淡绿
CYAN	3	青色	LIGHTCYAN	11	淡青
RED	4	红色	LIGHTRED	12	淡红
MAGENTA	5	洋红	LIGHTMAGENTA	13	淡洋红
BROWN	6	棕色	YELLOW	14	黄色
LIGHTGRAY	7	淡灰	WHITE	15	白色

(2) 文字形状、大小、方向

函数 `settextstyle()` 用来设置输出字符的字形(由参数 `font` 确定)、输出方向(由参数 `direction` 确定)和字符大小(由参数 `charsize` 确定)等特性。其原型为：


```
void far settextstyle(int font, int direction, int charsize);
```

该函数中各个参数的取值参考表 14-2~表 14.4。

表 14-2 文字 font 的取值

符号常数	数值	含义
DEFAULT_FONT	0	8*8 点阵字(缺省值)
TRIPLEX_FONT	1	笔划式三倍字型 (三倍字体)
SMALL_FONT	2	笔划式小字型(小字体)
SANSSERIF_FONT	3	笔划式字(Sanserif 字体)
GOTHIC_FONT	4	笔划黑体字型(哥特体)

表 14-3 文字 direction 的取值

符号常数	数值	含义
HORIZ_DIR	0	从左到右
VERT_DIR	1	从底到顶

表 14-4 文字 charsize 的取值

符号常数或数值	含义
1	8*8 点阵
2	16*16 点阵
3	24*24 点阵
4	32*32 点阵
5	40*40 点阵
6	48*48 点阵
7	56*56 点阵
8	64*64 点阵
9	72*72 点阵
10	80*80 点阵
0	用户定义的字符大小



程序代码：参见 “14-senior graphics\text.c”

实例 28 动画

④ 说明：本实例主要学习高级图形处理函数，并实现简单动画。

④ 要求：

(1) 掌握多边形、矩形、三角形等绘制函数（参考实验 13），在屏幕上绘制一辆简单的汽车模型，如图 14-1 所示（希望读者能够发挥自己的想像，能够绘制更逼真的模型），并要求通过 `floodfill` 函数（参考后文解析）对多边形进行颜色填充。

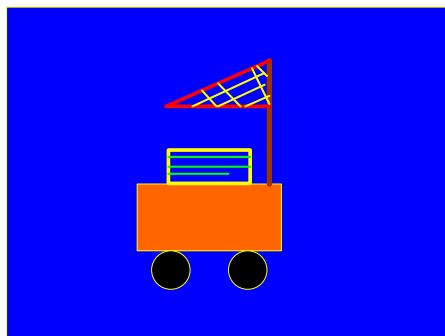


图 14-1 绘制小汽车

(1) 掌握 `getimage`, `setimage` 函数，实现一辆小汽车分别从屏幕左侧向右侧行驶，另一辆从右侧向左行驶。

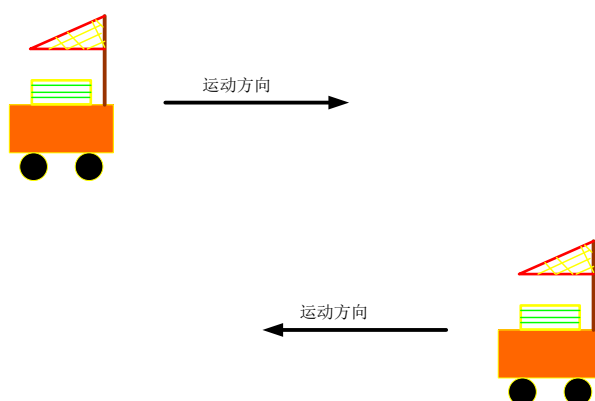


图 14-2 两辆小汽车相向运动

解析

1. 填充颜色

绘制圆及多边形等封闭区域后，可以通过下面的函数进行填充。

(1) `setfillstyle()` 设置填充图样和颜色函数

功能：setfillstyle()设置填充图样和颜色函数
功能： 函数 setfillstyle()为各种图形函数设置填充图样和颜色。
用法： 函数调用方式为 void setfillstyle(int pattern,int color);
说明： 参数 pattern 的值为填充图样，它们在头文件 graphics.h 中定义，详见表 14-5。
注： 参数 color 的值是填充色，它必须为当前显示模式所支持的有效值。填充图样与填充色是独立的，可以是不同的值。

表 14-5 填充图样

填充图样符号名	取值	说明
EMPTY_FILL	0	用背景色填充区域(空填)
SOLID_FILL	1	用实填充色填充(实填)
LINE_FILL	2	----填充
LTSLASH_FILL	3	///填充
SLASH_FILL	4	///用粗线填充
BKSLASH_FILL	5	\\用粗线填充
LTBKSLASH_FILL	6	\\填充
HATCH_FILL	7	网格线填充
xHATCH_FILL	8	斜网格线填充
INTEREAVE_FILL	9	间隔点填充
WIDE_DOT_FILL	10	大间隔点填充
CLOSE_DOT_FILL	11	小间隔点填充
USER_FILL	12	用户定义图样填充

注：除了 EMPTY_FILL，所有填充图样都使用当前填充色

(2) floodfill() 填充闭域函数

功能： 函数 floodfill()用当前填充图样和填充色填充一个由特定边界颜色（通常是当前绘图色）定义的有界封闭区域。
用法： 该函数调用方式为 void floodfill(int x,int y,int bordercolor);
说明： 这里参数(x,y)为指定填充区域中的某点，如果点(x,y)在该填充区域之外，那么外部区域将被填充，但受图形视口边界的限制。如果直线定义的区域出现间断，那么将导致泄漏，即使很小的间断，也将导致泄漏。也就是说，间断将引起区域外被填充。参数 bordercolor 为闭区域边界颜色。注：**由 border 指定的颜色值必须与图形轮廓的颜色值相同（否则出错！）**，但填充色可选任意颜色。

2. 实现动画绘制的相关函数

图像复制、擦除以及一般对屏幕图像的操作，这对应用程序是非常有用的，对动画制作是必不可少的。以下部分将介绍几个重要的图像操作函数：

(1) `imagesize()` 图像存储大小函数

功能：函数 `imagesize()` 返回存储一块屏幕图像所需的内存大小(即字节数)。

用法：此函数调用方式为 `unsigned imagesize(int left,int top,int right,int bottom);`

说明：参数 `(left,top)` 为所定义的一块图像屏幕左上角，`(right,bottom)` 为其右下角。

函数调用执行后，返回存储该块屏幕所需要的字节数。如果所需要字节数大于 64KB，那么将返回 -1。`imagesize()` 函数一般与下面 `getimage()` 函数联用。

这个函数对应的头文件为 `graphics.h`

返回值：返回一块图像屏幕存储所需的字节数，如果大于 64KB，则返回 -1。

例：确定左上角(10,10)与右下角(100,100)所定义的屏幕图像所需的字节数：

```
unsigned size;
size=imagesize(10,10,100,100);
```

(2) `getimage()` 保存图像函数

功能：函数 `getimage()` 保存左上角与右下角所定义的屏幕上像素图形到指定的内存区域。

用法：该函数调用方式为 `void getimage(int left,int top,int right,int bottom,void *buf);`

说明：函数中参数 `(left,top)` 为要保存的图像屏幕的左上角，`(right,bottom)` 为其右下角，`buf` 指向保存图像的内存地址。调用 `getimage()` 保存屏幕图像，可用 `imagesize()` 函数确定保存图像所需字节数，再用 `malloc()` 函数分配存储图像的内存（内存分配必须小于 64KB），还可以用下面函数 `putimage()` 输出 `getimage()` 保存的屏幕图像。

返回值：无

例：把带有两对角线的矩形拷贝到屏幕其它位置上

```
unsigned size;
void *buf;
sector(15);
rectangle(20,20,200,200);
setcolor(RED);
line(20,20,200,200);
setcolor(GREEN);
line(20,200,200,20);
getch();
size=imagesize(20,20,200,200);
if(size!=-1){
    buf=malloc(size);
    if(buf){
        getimage(20,20,200,200,buf);
        putimage(100,100,buf,COPY_PUT);
        putimage(300,50,buf,COPY_PUT);
    }
}
```

```
outtext("press a key");
```

(2) putimage() 输出图像函数

功能：函数 `putimage()` 将一个先前保存在内存中的图像输出到屏幕上。

用法：此函数调用方式为 `void putimage(int left,int top,void *buf,int ops);`

说明：参数 `(left,top)` 为输出屏幕图像的左上角，即输出图像的起始位置。`buf` 指向要输出的内存中图像。参数 `ops` 控制图像以何种方式输出到屏幕上。表 14-6 给出了图像输出方式。

表 14-6 图像输出方式

输出方式符号名	取值	含义
COPY_PUT	0	图像输出到屏幕上，取代原有图像
XOR_PUT	1	图像和原有像素作异或运算
OR_PUT	2	图像和原有像素作或运算
AND_PUT	3	图像和原有像素作与运算
NOT_PUT	4	把求反的位图像输出到屏幕上

1) COPY_PUT 输出方式

图像中每个像素都直接绘制到屏幕上，取代原有图像像素，包括空白的图像像素(背景)。完全空白的图像可以用来擦除其它图像或屏幕的一部分。但通常选择 `xOR_PUT` 输出方式擦除原有图像。

2) XOR_PUT 输出方式

原有屏幕每个像素与相应的图像字节作“异或”运算，其结果画到屏幕上。当某一图像和屏幕上原有图像作“异或”运算时，屏幕显示的是两个图像的合成。若相同的图像作异或运算，将有效地擦除该图像，留下原始屏幕。这种输出方式，对动画制作是非常有用的。

3) OR_PUT 输出方式

每个图像字节和相应的屏幕像素作“或”运算，再将结果画到屏幕上，这种输出方式也叫“两者取一”。记住，像素中的每位和图像中的每位作“或”运算，这样所得结果是背景和图像的彩色合成图像。

4) AND_PUT 输出方式

选择 `AND_PUT` 图像输出方式时，屏幕像素和图像字节中都显示的位，运算后仍显示，例如，星图像中的空白背景擦除了方块轮廓以及填充色，只有星图像复盖着的方块留下，即运算后，显示两者相同的图像。

5) NOT_PUT 输出方式

`NOT_PUT` 输出方式，除了把图像的每位求反---图像中所有黑的像素(0000)变成了白色(1111)，其它方面与 `COPY_PUT` 相同。背景图像被重画后将消失。

`putimage()` 函数对应的头文件为 `graphics.h`

返回值：无

例 下面代码模拟两个小球动态碰撞的动画过程：

```
#include<stdio.h>
#include<graphics.h>
int main()
{
    int i, gdriver, gmode, size;
    void *buf;
    gdriver=DETECT;
    initgraph(&gdriver, &gmode, "");
    setbkcolor(BLUE);
    cleardevice();
    setcolor(LIGHTRED);
    setlinestyle(0,0,1);
    setfillstyle(1, 10);
    circle(100, 200, 30);
    floodfill(100, 200, 12);
    size=imagesize(69, 169, 131, 231);
    buf=malloc(size);
    getimage(69, 169, 131, 231,buf);
    putimage(500, 269, buf, COPY_PUT);
    for(i=0; i<185; i++){
        putimage(70+i, 170, buf, COPY_PUT);
        putimage(500-i, 170, buf, COPY_PUT);
    }
    for(i=0; i<185; i++){
        putimage(255-i, 170, buf, COPY_PUT);
        putimage(315+i, 170, buf, COPY_PUT);
    }
    getch();
    closegraph();
}
```



程序代码：参见 “14-senior graphics\car.c”

综合实验 1



可视化窗口编辑器

④ 说明：参考 TC 编辑器，编程实现一个窗口程序，并实现文本文件（如*.txt, *.c, *.h 等文件）的编辑与保存。

④ 要求

(1) 主界面参考图 1，整个窗口就是应用程序的主窗口，包括如下元素：边框、菜单、装饰线及提示文字、快捷键提示信息等。

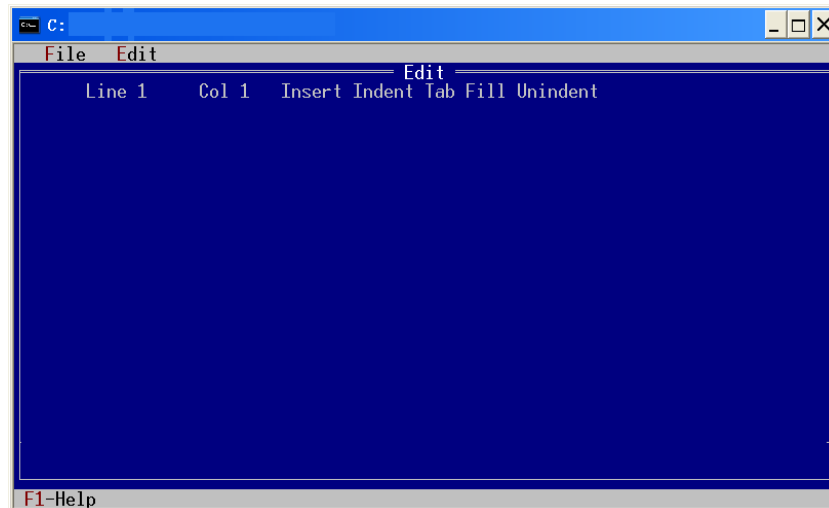


图 1 主窗口界面

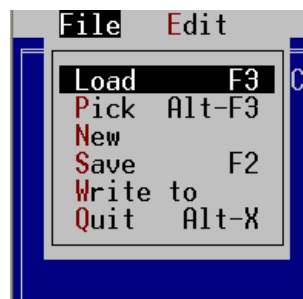


图 2 弹出式菜单

(2) 菜单包括主菜单项及弹出菜单。主菜单项如图 1 中的“File”、“Edit”所示。为每一个主菜单项设置一个快捷键，如“File”可以通过用户按‘Alt+F’选中。当用户选中主菜单项后，将生成对应的弹出式菜单，如图 2 所示。**要求：**主菜单项不少于 2 项，每个主菜单项对应的子菜单功能不少于 3 项。

(3) 通过处理键盘消息，为系统的某些功能设置对应的快捷键，如打开文件快捷键可以设为 F3，保存文件为 F4，打开软件说明为 F1，复制为 Ctrl+C，粘贴为 Ctrl+V 等。**要求：**快捷键不少于 6 个。

(4) 实现一个完整的编辑器功能：打开文件、新建文件、保存文件、编辑文件（添加文字、修改文字、删除文字等）。

(5) 为了设计用户界面良好的程序，要求能够响应相应的功能键，如退格键、PageUp、PageDown、上下左右方向键等。

综合实验 2



图书管理系统设计

④ 说明：参考学校图书馆管理系统，完成一套简单的图书及读者管理系统，实现图书信息管理、读者查询、借阅管理等。

④ 要求

(1) 主界面参考图 1。系统以图形化菜单方式工作，即用户通过选取相应的菜单项进入相关的功能模块。

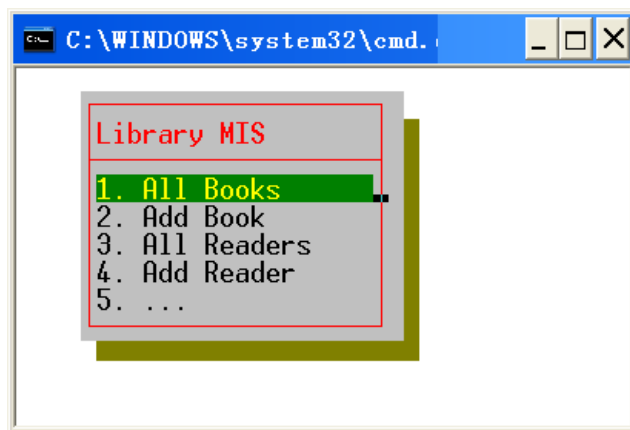


图 1 主窗口界面

(2) 图书管理信息包括：图书名称、图书编号、单价、作者、借阅状态（已借、未借出）、读者（即借书人）姓名、学号等。

(3) 功能包括：

- 新进图书基本信息的输入
- 图书基本信息的查询
- 对撤消图书信息的删除
- 为读者（即借书人）办理注册，即添加读者
- 查询：所有图书信息查询、所有读者信息查询
- 办理借书手续：即修改书籍的借阅状态，添加读者的信息等
- 办理还书手续：即修改书籍的借阅状态，删除读者的信息等

(4) 要求使用二进制文件方式存储数据。

(5) 要求使用链表组织、管理图书信息。

(6) 对于管理信息系统 (MIS) 而言, 在编码前的系统设计往往非常重要, 甚至关乎软件是否能够顺利完成。要求在编写代码前进行充分设计: 进行系统结构设计、功能模块划分、模块内部流程设计、模块间调用关系等。在实验报告中要包括完整的**系统设计方案** (模块定义、工作流程等)、**系统结构图** (模块划分、模块间的调用关系等)、**用户使用手册**。

综合实验 3

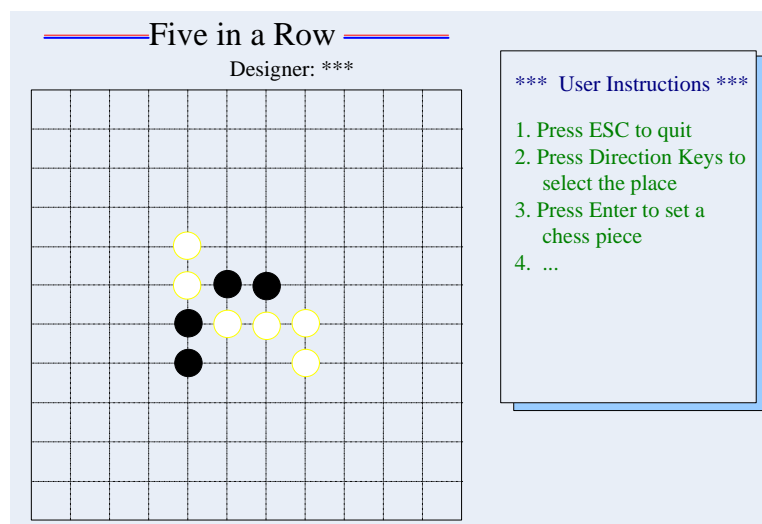


五子棋游戏软件开发

④ **说明：**设计一个五子棋游戏，具有良好的图形化界面，实现“人工智能”，即使计算机能够同人对战。

④ **要求**

(1) 主界面参考图 1。要求至少包括**标题信息**（见左上部）、**用户操作说明**（右侧方框中）、**棋盘**（左下部分）。图 1 仅作参考，其中文字颜色、棋盘颜色、边框样式及位置均可以根据需要调整，并且可以灵活添加所需要的额外信息。



(2) 在下棋过程中，用户随时可以按 **ESC** 键退出游戏。退出前需要弹出提示信息窗口，如“Are you sure want to quit? Y or N”，让用户确认是否真正退出，如果用户输入 **Y**，则退出程序，否则返回游戏继续执行。

(3) 掌握五子棋基本规则，通过自动计算最优棋盘格，让计算机实现自动下棋。

- 参考方法 1：根据规则，棋盘格局每次改变后，对每一个未填充的棋盘格进行计分，（例如，将能够连成 5 子成线的格设置为最高分）；然后选择最高分的棋盘格落子。
- 参考方法 2：贪心算法，请自行查找资料学习。

- 每次棋盘格局每次改变后，都需要判断最后下棋一方已经有 5 子连成线了，如果成立则进入胜负判断模块。

附录 A 本书使用到的库函数说明

1. **atoi** 功能：将字符串转换成整数
 - **原型**
`int atoi(const char *string);`
 - **参数**
string: 输入字符串
 - **返回值**
返回整数
 - **头文件**
`stdlib.h`
 - **示例**
见本书 P6。

2. **itoa** 功能：将整数转换成字符串
 - **原型**
`char *itoa(int value, char *string, int radix);`
 - **参数**
value: 输入数字
string: 转换后的字符串保存的位置
radix: *value* 数字的基，如 10 表示十进制
 - **返回值**
返回字符串首地址
 - **头文件**
`stdlib.h`
 - **示例**
见本书 P6。

3. **sprintf** 功能：将格式化的数据转换为字符串
 - **原型**
`int sprintf(char *buffer, const char *format [, argument] ...);`
 - **参数**
buffer: 转换的字符串首地址，用于保存结果
format: 控制字符串，类似于 `printf` 中的控制字符串
 - **返回值**
返回 *buffer* 中转换的字节数，不包括结尾的 `NULL`
 - **头文件**
`stdio.h`
 - **示例**
见本书 P7。
4. **system** 功能：根据用户输入执行系统命令

- **原型**
`int system(const char *command);`
 - **参数**
command : 要执行的用户命令
 - **返回值**
执行成功返回 0; 否则返回-1
 - **头文件**
`stdlib.h`
 - **示例**
见本书 P3。
5. **time** 功能: 返回从 1970 年 1 月 1 日 00:00:00 起距函数调用时相距的时间, 以秒为单位
- **原型**
`time_t time(time_t *timer);;`
 - **参数**
timer: 时间保存的位置, 如果为 NULL, 则不保存任何数据。
 - **返回值**
返回从 1970 年 1 月 1 日 00:00:00 起, 距函数调用时相距的时间, 以秒为单位。
 - **头文件**
`time.h`
 - **示例**
见本书 P13。

附录 B string 函数列表

引用头文件方法：

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

示例：

```
void main()
{
    string s="hello";
    printf("%s",s.data());
}
```

函数名	描述
begin	得到指向字符串开头的 Iterator
end	得到指向字符串结尾的 Iterator
rbegin	得到指向反向字符串开头的 Iterator
rend	得到指向反向字符串结尾的 Iterator
size	得到字符串的大小
length	和 size 函数功能相同
max_size	字符串可能的最大大小
capacity	在不重新分配内存的情况下，字符串可能的大小
empty	判断是否为空
operator[]	取第几个元素，相当于数组
c_str	取得 C 风格的 const char* 字符串
data	取得字符串内容地址
operator=	赋值操作符
reserve	预留空间
swap	交换函数
insert	插入字符
append	追加字符
push_back	追加字符
operator+=	+= 操作符
erase	删除字符串
clear	清空字符容器中所有内容
resize	重新分配空间
assign	和赋值操作符一样
replace	替代
copy	字符串到空间
find	查找
rfind	反向查找

find_first_of	查找包含子串中的任何字符，返回第一个位置
find_first_not_of	查找不包含子串中的任何字符，返回第一个位置
find_last_of	查找包含子串中的任何字符，返回最后一个位置
find_last_not_of	查找不包含子串中的任何字符，返回最后一个位置
substr	得到子串
compare	比较字符串
operator+	字符串链接
operator==	判断是否相等
operator!=	判断是否不等于
operator<	判断是否小于
operator>>	从输入流中读入字符串
operator<<	字符串写入输出流
getline	从输入流中读入一行

string 类常用函数用法

string 类的构造函数：

string(const char *s); //用 c 字符串 s 初始化

string(int n,char c); //用 n 个字符 c 初始化

此外，string 类还支持默认构造函数和复制构造函数，如 string s1; string s2="hello"; 都是正确的写法。当构造的 string 太长而无法表达时会抛出 length_error 异常

string 类的字符操作：

const char &operator[](int n)const;

const char &at(int n)const;

char &operator[](int n);

char &at(int n);

operator[]和 at()均返回当前字符串中第 n 个字符的位置，但 at 函数提供范围检查，当越界时会抛出 out_of_range 异常，下标运算符[]不提供检查访问。

const char *data()const;//返回一个非 null 终止的 c 字符数组

const char *c_str()const;//返回一个以 null 终止的 c 字符串

int copy(char *s, int n, int pos = 0) const;//把当前串中以 pos 开始的 n 个字符拷贝到以 s 为起始位置的字符数组中，返回实际拷贝的数目

string 的特性描述：

int capacity()const; //返回当前容量（即 string 中不必增加内存即可存放的元素个数）

int max_size()const; //返回 string 对象中可存放的最大字符串的长度

int size()const; //返回当前字符串的大小

int length()const; //返回当前字符串的长度

bool empty()const; //当前字符串是否为空

void resize(int len,char c);//把字符串当前大小置为 len，并用字符 c 填充不足的部分

string 类的输入输出操作:

string 类重载运算符 `operator<<` 用于输入, 同样重载运算符 `operator>>` 用于输出操作。

函数 `getline(istream &in, string &s);` 用于从输入流 `in` 中读取字符串到 `s` 中, 以换行符 `'\n'` 分开。

string 的赋值:

`string &operator=(const string &s);` // 把字符串 `s` 赋给当前字符串

`string &assign(const char *s);` // 用 `c` 类型字符串 `s` 赋值

`string &assign(const char *s, int n);` // 用 `c` 字符串 `s` 开始的 `n` 个字符赋值

`string &assign(const string &s);` // 把字符串 `s` 赋给当前字符串

`string &assign(int n, char c);` // 用 `n` 个字符 `c` 赋值给当前字符串

`string &assign(const string &s, int start, int n);` // 把字符串 `s` 中从 `start` 开始的 `n` 个字符赋给当前字符串

`string &assign(const_iterator first, const_iterator last);` // 把 `first` 和 `last` 迭代器之间的部分赋给字符串

string 的连接:

`string &operator+=(const string &s);` // 把字符串 `s` 连接到当前字符串的结尾

`string &append(const char *s);` // 把 `c` 类型字符串 `s` 连接到当前字符串结尾

`string &append(const char *s, int n);` // 把 `c` 类型字符串 `s` 的前 `n` 个字符连接到当前字符串结尾

`string &append(const string &s);` // 同 `operator+=()`

`string &append(const string &s, int pos, int n);` // 把字符串 `s` 中从 `pos` 开始的 `n` 个字符连接到当前字符串的结尾

`string &append(int n, char c);` // 在当前字符串结尾添加 `n` 个字符 `c`

`string &append(const_iterator first, const_iterator last);` // 把迭代器 `first` 和 `last` 之间的部分连接到当前字符串的结尾

string 的比较:

`bool operator==(const string &s1, const string &s2) const;` // 比较两个字符串是否相等

运算符 `>`, `<`, `>=`, `<=`, `!=` 均被重载用于字符串的比较;

`int compare(const string &s) const;` // 比较当前字符串和 `s` 的大小

`int compare(int pos, int n, const string &s) const;` // 比较当前字符串从 `pos` 开始的 `n` 个字符组成的字符串与 `s` 的大小

`int compare(int pos, int n, const string &s, int pos2, int n2) const;` // 比较当前字符串从 `pos` 开始的 `n` 个字符组成的字符串与 `s` 中 `pos2` 开始的 `n2` 个字符组成的字符串的大小

`int compare(const char *s) const;`

`int compare(int pos, int n, const char *s) const;`

```
int compare(int pos, int n,const char *s, int pos2) const;
```

compare 函数在>时返回 1, <时返回-1, ==时返回 0

string 的子串:

```
string substr(int pos = 0,int n = npos) const;//返回 pos 开始的 n 个字符组成的字符串
```

string 的交换:

```
void swap(string &s2); //交换当前字符串与 s2 的值
```

string 类的查找函数:

```
int find(char c, int pos = 0) const;//从 pos 开始查找字符 c 在当前字符串的位置
```

```
int find(const char *s, int pos = 0) const;//从 pos 开始查找字符串 s 在当前串中的位置
```

```
int find(const char *s, int pos, int n) const;//从 pos 开始查找字符串 s 中前 n 个字符在当前串中的位置
```

```
int find(const string &s, int pos = 0) const;//从 pos 开始查找字符串 s 在当前串中的位置
```

//查找成功时返回所在位置, 失败返回 string::npos 的值

```
int rfind(char c, int pos = npos) const;//从 pos 开始从后向前查找字符 c 在当前串中的位置
```

```
int rfind(const char *s, int pos = npos) const;
```

```
int rfind(const char *s, int pos, int n = npos) const;
```

```
int rfind(const string &s,int pos = npos) const;
```

//从 pos 开始从后向前查找字符串 s 中前 n 个字符组成的字符串在当前串中的位置, 成功返回所在位置, 失败时返回 string::npos 的值

```
int find_first_of(char c, int pos = 0) const;//从 pos 开始查找字符 c 第一次出现的位置
```

```
int find_first_of(const char *s, int pos = 0) const;
```

```
int find_first_of(const char *s, int pos, int n) const;
```

```
int find_first_of(const string &s,int pos = 0) const;
```

//从 pos 开始查找当前串中第一个在 s 的前 n 个字符组成的数组里的字符的位置。查找失败返回 string::npos

```
int find_first_not_of(char c, int pos = 0) const;
```

```
int find_first_not_of(const char *s, int pos = 0) const;
```

```
int find_first_not_of(const char *s, int pos,int n) const;
```

```
int find_first_not_of(const string &s,int pos = 0) const;
```

//从当前串中查找第一个不在串 s 中的字符出现的位置, 失败返回 string::npos

```
int find_last_of(char c, int pos = npos) const;
```

```
int find_last_of(const char *s, int pos = npos) const;
```

```
int find_last_of(const char *s, int pos, int n = npos) const;
```

```
int find_last_of(const string &s,int pos = npos) const;
```

```
int find_last_not_of(char c, int pos = npos) const;
```

```
int find_last_not_of(const char *s, int pos = npos) const;
```

```
int find_last_not_of(const char *s, int pos, int n) const;
```

```
int find_last_not_of(const string &s,int pos = npos) const;
```

//find_last_of 和 find_last_not_of 与 find_first_of 和 find_first_not_of 相似, 只不过是从后向前查找

string 类的替换函数:

string &replace(int p0, int n0,const char *s);//删除从 p0 开始的 n0 个字符, 然后在 p0 处插入串 s

string &replace(int p0, int n0,const char *s, int n);//删除 p0 开始的 n0 个字符, 然后在 p0 处插入字符串 s 的前 n 个字符

string &replace(int p0, int n0,const string &s);//删除从 p0 开始的 n0 个字符, 然后在 p0 处插入串 s

string &replace(int p0, int n0,const string &s, int pos, int n);//删除 p0 开始的 n0 个字符, 然后在 p0 处插入串 s 中从 pos 开始的 n 个字符

string &replace(int p0, int n0,int n, char c);//删除 p0 开始的 n0 个字符, 然后在 p0 处插入 n 个字符 c

string &replace(iterator first0, iterator last0,const char *s);//把[first0, last0) 之间的部分替换为字符串 s

string &replace(iterator first0, iterator last0,const char *s, int n);//把[first0, last0) 之间的部分替换为 s 的前 n 个字符

string &replace(iterator first0, iterator last0,const string &s);//把[first0, last0) 之间的部分替换为串 s

string &replace(iterator first0, iterator last0,int n, char c);//把[first0, last0) 之间的部分替换为 n 个字符 c

string &replace(iterator first0, iterator last0,const_iterator first, const_iterator last);//把 [first0, last0) 之间的部分替换成[first, last) 之间的字符串

string 类的插入函数:

string &insert(int p0, const char *s);

string &insert(int p0, const char *s, int n);

string &insert(int p0,const string &s);

string &insert(int p0,const string &s, int pos, int n);

//前 4 个函数在 p0 位置插入字符串 s 中 pos 开始的前 n 个字符

string &insert(int p0, int n, char c);//此函数在 p0 处插入 n 个字符 c

iterator insert(iterator it, char c);//在 it 处插入字符 c, 返回插入后迭代器的位置

void insert(iterator it, const_iterator first, const_iterator last);//在 it 处插入[first, last) 之间的字符

void insert(iterator it, int n, char c);//在 it 处插入 n 个字符 c

string 类的删除函数

iterator erase(iterator first, iterator last);//删除[first, last) 之间的所有字符, 返回删除后迭代器的位置

iterator erase(iterator it);//删除 it 指向的字符, 返回删除后迭代器的位置

string &erase(int pos = 0, int n = npos);//删除 pos 开始的 n 个字符, 返回修改后的字符串

string 类的迭代器处理：

string 类提供了向前和向后遍历的迭代器 iterator，迭代器提供了访问各个字符的语法，类似于指针操作，迭代器不检查范围。

用 string::iterator 或 string::const_iterator 声明迭代器变量，const_iterator 不允许改变迭代的内容。常用迭代器函数有：

```
const_iterator begin()const;
```

```
iterator begin();           //返回 string 的起始位置
```

```
const_iterator end()const;
```

```
iterator end();           //返回 string 的最后一个字符后面的位置
```

```
const_iterator rbegin()const;
```

```
iterator rbegin();        //返回 string 的最后一个字符的位置
```

```
const_iterator rend()const;
```

```
iterator rend();          //返回 string 第一个字符位置的前面
```

rbegin 和 rend 用于从后向前的迭代访问，通过设置迭代器

string::reverse_iterator, string::const_reverse_iterator 实现

附录 C C 语言文件操作函数大全

clearerr (清除文件流的错误旗标)

相关函数 feof

表头文件 #include<stdio.h>

定义函数 void clearerr(FILE * stream);

函数说明 clearerr () 清除参数 stream 指定的文件流所使用的错误旗标。

返回值

fclose (关闭文件)

相关函数 close, fflush, fopen, setbuf

表头文件 #include<stdio.h>

定义函数 int fclose(FILE * stream);

函数说明 fclose()用来关闭先前 fopen()打开的文件。此动作会让缓冲区内的数据写入文件中,并释放系统所提供的文件资源。

返回值 若关文件动作成功则返回 0,有错误发生时则返回 EOF 并把错误代码存到 errno。

错误代码 EBADF 表示参数 stream 非已打开的文件。

范例 请参考 fopen ()。

fdopen (将文件描述词转为文件指针)

相关函数 fopen, open, fclose

表头文件 #include<stdio.h>

定义函数 FILE * fdopen(int fildes,const char * mode);

函数说明 fdopen()会将参数 fildes 的文件描述词,转换为对应的文件指针后返回。参数 mode 字符串则代表着文件指针的流形态,此形态必须和原先文件描述词读写模式相同。

关于 mode 字符串格式请参考 fopen()。

返回值 转换成功时返回指向该流的文件指针。失败则返回 NULL,并把错误代码存在 errno 中。

范例

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

```
main()
```

```
{
```

```
FILE * fp =fdopen(0," w+" );
```

```
fprintf(fp," %s\n" ," hello!" );
```

```
fclose(fp);  
}
```

执行 hello!

feof (检查文件流是否读到了文件尾)

相关函数 fopen, fgetc, fgets, fread

表头文件 #include<stdio.h>

定义函数 int feof(FILE * stream);

函数说明 feof()用来侦测是否读取到了文件尾,尾数 stream 为 fopen () 所返回之文件指针。如果已到文件尾则返回非零值,其他情况返回 0。

返回值 返回非零值代表已到达文件尾。

fflush (更新缓冲区)

相关函数 write, fopen, fclose, setbuf

表头文件 #include<stdio.h>

定义函数 int fflush(FILE* stream);

函数说明 fflush()会强迫将缓冲区内的数据写回参数 stream 指定的文件中。如果参数 stream 为 NULL, fflush()会将所有打开的文件数据更新。

返回值 成功返回 0, 失败返回 EOF, 错误代码存于 errno 中。

错误代码 EBADF 参数 stream 指定的文件未被打开, 或打开状态为只读。其它错误代码参考 write ()。

fgetc (由文件中读取一个字符)

相关函数 open, fread, fscanf, getc

表头文件 include<stdio.h>

定义函数 nt fgetc(FILE * stream);

函数说明 fgetc()从参数 stream 所指的文件中读取一个字符。若读到文件尾而无数据时便返回 EOF。

返回值 getc()会返回读取到的字符, 若返回 EOF 则表示到了文件尾。

范例

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

```
main()
```

```
{
```

```
FILE *fp;
```

```
int c;
```

```
fp=fopen( "exist" , "r" );
```

```
while((c=fgetc(fp))!=EOF)
printf( "%c" ,c);
fclose(fp);
}
```

fgets（由文件中读取一字符串）

相关函数 open, fread, fscanf, getc

表头文件 include<stdio.h>

定义函数 `char * fgets(char * s,int size,FILE * stream);`

函数说明 fgets()用来从参数 stream 所指的文件内读入字符并存到参数 s 所指的内存空间，直到出现换行字符、读到文件尾或是已读了 size-1 个字符为止，最后会加上 NULL 作为字符串结束。

返回值 gets()若成功则返回 s 指针，返回 NULL 则表示有错误发生。

范例

```
#include<stdio.h>
main()
{
char s[80];
fputs(fgets(s,80,stdin),stdout);
}
```

执行 this is a test /*输入*/

this is a test /*输出*/

fileno（返回文件流所使用的文件描述词）

相关函数 open, fopen

表头文件 #include<stdio.h>

定义函数 `int fileno(FILE * stream);`

函数说明 fileno()用来取得参数 stream 指定的文件流所使用的文件描述词。

返回值 返回文件描述词。

范例

```
#include<stdio.h>
main()
{
FILE * fp;
int fd;
```

```

fp=fopen( "/etc/passwd" ," r" );
fd=fileno(fp);
printf( "fd=%d\n" ,fd);
fclose(fp);
}

```

执行 fd=3

fopen（打开文件）

相关函数 open，fclose

表头文件 #include<stdio.h>

定义函数 FILE * fopen(const char * path,const char * mode);

函数说明 参数 path 字符串包含欲打开的文件路径及文件名，参数 mode 字符串则代表着流形态。

mode 有下列几种形态字符串：

r 打开只读文件，该文件必须存在。

r+ 打开可读写的文件，该文件必须存在。

w 打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在则建立该文件。

w+ 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。

a 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。

a+ 以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。

上述的形态字符串都可以再加一个 b 字符，如 rb、w+b 或 ab+ 等组合，加入 b 字符用来告诉函数库打开的文件为二进制文件，而非纯文字文件。不过在 POSIX 系统，包含 Linux 都会忽略该字符。由 fopen() 所建立的新文件会具有 S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH(0666) 权限，此文件权限也会参考 umask 值。

返回值 文件顺利打开后，指向该流的文件指针就会被返回。若果文件打开失败则返回 NULL，并把错误代码存在 errno 中。

附加说明 一般而言，开文件后会作一些文件读取或写入的动作，若开文件失败，接下来的读写动作也无法顺利进行，所以在 fopen() 后请作错误判断及处理。

范例

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



```

main()
{
FILE * fp;
fp=fopen( "noexist" , " a+" );
if(fp==NULL) return;
fclose(fp);
}

```

fputc (将一指定字符写入文件流中)

相关函数 fopen, fwrite, fscanf, putc

表头文件 #include<stdio.h>

定义函数 int fputc(int c,FILE * stream);

函数说明 fputc 会将参数 c 转为 unsigned char 后写入参数 stream 指定的文件中。

返回值 fputc()会返回写入成功的字符，即参数 c。若返回 EOF 则代表写入失败。

范例

```

#include<stdio.h>
main()
{
FILE * fp;
char a[26]=" abcdefghijklmnopqrstuvwxyz" ;
int i;
fp= fopen( "noexist" , " w" );
for(i=0;i<26;i++)
fputc(a,fp);
fclose(fp);
}

```

fputs (将一指定的字符串写入文件内)

相关函数 fopen, fwrite, fscanf, fputc, putc

表头文件 #include<stdio.h>

定义函数 int fputs(const char * s,FILE * stream);

函数说明 fputs()用来将参数 s 所指的字符串写入到参数 stream 所指的文件内。

返回值 若成功则返回写出的字符个数，返回 EOF 则表示有错误发生。

范例 请参考 fgets () 。

fread (从文件流读取数据)

相关函数 fopen, fwrite, fseek, fscanf

表头文件 #include<stdio.h>

定义函数 `size_t fread(void * ptr,size_t size,size_t nmemb,FILE * stream);`

函数说明 `fread()`用来从文件流中读取数据。参数 `stream` 为已打开的文件指针，参数 `ptr` 指向欲存放读取进来的数据空间，读取的字符数以参数 `size*nmemb` 来决定。`Fread()`会返回实际读取到的 `nmemb` 数目，如果此值比参数 `nmemb` 来得小，则代表可能读到了文件尾或有错误发生，这时必须用 `feof()`或 `ferror()`来决定发生什么情况。

返回值 返回实际读取到的 `nmemb` 数目。

附加说明

范例

```
#include<stdio.h>
#define nmemb 3
struct test
{
char name[20];
int size;
}s[nmemb];
int main(){
FILE * stream;
int i;
stream = fopen( “/tmp/fwrite” ,” r” );
fread(s,sizeof(struct test),nmemb,stream);
fclose(stream);
for(i=0;i<nmemb;i++)
printf( “name[%d]=%-20s:size[%d]=%d\n” ,i,s.name,i,s.size);
}
```

执行

```
name[0]=Linux! size[0]=6
name[1]=FreeBSD! size[1]=8
name[2]=Windows2000 size[2]=11
```

`freopen`（打开文件）

相关函数 `fopen`，`fclose`

表头文件 `#include<stdio.h>`

定义函数 `FILE * freopen(const char * path,const char * mode,FILE * stream);`

函数说明 参数 path 字符串包含欲打开的文件路径及文件名，参数 mode 请参考 fopen() 说明。参数 stream 为已打开的文件指针。Freopen() 会将原 stream 所打开的文件流关闭，然后打开参数 path 的文件。

返回值 文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回 NULL，并把错误代码存在 errno 中。

范例

```
#include<stdio.h>

main()
{
FILE * fp;
fp=fopen( "/etc/passwd" ," r" );
fp=freopen( "/etc/group" ," r" ,fp);
fclose(fp);
}
```

fseek (移动文件流的读写位置)

相关函数 rewind, ftell, fgetpos, fsetpos, lseek

表头文件 #include<stdio.h>

定义函数 int fseek(FILE * stream,long offset,int whence);

函数说明 fseek() 用来移动文件流的读写位置。参数 stream 为已打开的文件指针，参数 offset 为根据参数 whence 来移动读写位置的位移数。

参数 whence 为下列其中一种：

SEEK_SET 从距文件开头 offset 位移量为新的读写位置。SEEK_CUR 以目前的读写位置往后增加 offset 个位移量。

SEEK_END 将读写位置指向文件尾后再增加 offset 个位移量。

当 whence 值为 SEEK_CUR 或 SEEK_END 时，参数 offset 允许负值的出现。

下列是较特别的使用方式：

1) 欲将读写位置移动到文件开头时:fseek(FILE *stream,0,SEEK_SET);

2) 欲将读写位置移动到文件尾时:fseek(FILE *stream,0,SEEK_END);

返回值 当调用成功时则返回 0，若有错误则返回-1，errno 会存放错误代码。

附加说明 fseek() 不像 lseek() 会返回读写位置，因此必须使用 ftell() 来取得目前读写的位置。

范例

```
#include<stdio.h>

main()
{
FILE * stream;
```

```

long offset;
fpos_t pos;
stream=fopen( "/etc/passwd" ," r" );
fseek(stream,5,SEEK_SET);
printf( "offset=%d\n" ,ftell(stream));
rewind(stream);
fgetpos(stream,&pos);
printf( "offset=%d\n" ,pos);
pos=10;
fsetpos(stream,&pos);
printf( "offset = %d\n" ,ftell(stream));
fclose(stream);
}

```

执行 offset = 5

offset =0

offset=10

ftell（取得文件流的读取位置）

相关函数 `fseek`, `rewind`, `fgetpos`, `fsetpos`

表头文件 `#include<stdio.h>`

定义函数 `long ftell(FILE * stream);`

函数说明 `ftell()`用来取得文件流目前的读写位置。参数 `stream` 为已打开的文件指针。

返回值 当调用成功时则返回目前的读写位置，若有错误则返回-1，`errno` 会存放错误代码。

错误代码 `EBADF` 参数 `stream` 无效或可移动读写位置的文件流。

范例 参考 `fseek()`。

fwrite（将数据写至文件流）

相关函数 `fopen`, `fread`, `fseek`, `fscanf`

表头文件 `#include<stdio.h>`

定义函数 `size_t fwrite(const void * ptr,size_t size,size_t nmemb,FILE * stream);`

函数说明 `fwrite()`用来将数据写入文件流中。参数 `stream` 为已打开的文件指针，参数 `ptr` 指向欲写入的数据地址，总共写入的字符数以参数 `size*nmemb` 来决定。`Fwrite()`会返回实际写入的 `nmemb` 数目。

返回值 返回实际写入的 `nmemb` 数目。

范例

```
#include<stdio.h>
#define set_s (x,y) {strcpy(s[x].name,y);s[x].size=strlen(y);}
#define nmemb 3
struct test
{
char name[20];
int size;
}s[nmemb];
main()
{
FILE * stream;
set_s(0," Linux!" );
set_s(1," FreeBSD!" );
set_s(2," Windows2000." );
stream=fopen( "/tmp/fwrite" ," w" );
fwrite(s,sizeof(struct test),nmemb,stream);
fclose(stream);
}
```

执行 参考 fread () 。

getc (由文件中读取一个字符)

相关函数 read, fopen, fread, fgetc

表头文件 #include<stdio.h>

定义函数 int getc(FILE * stream);

函数说明 getc()用来从参数 stream 所指的文件中读取一个字符。若读到文件尾而无数据时便返回 EOF。虽然 getc()与 fgetc()作用相同，但 getc()为宏定义，非真正的函数调用。

返回值 getc()会返回读取到的字符，若返回 EOF 则表示到了文件尾。

范例 参考 fgetc()。

getchar (由标准输入设备内读进一字符)

相关函数 fopen, fread, fscanf, getc

表头文件 #include<stdio.h>

定义函数 int getchar(void);

函数说明 getchar()用来从标准输入设备中读取一个字符。然后将该字符从 unsigned char 转换成 int 后返回。

返回值 `getchar()` 会返回读取到的字符，若返回 EOF 则表示有错误发生。

附加说明 `getchar()` 非真正函数，而是 `getc(stdin)` 宏定义。

范例

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

```
main()
```

```
{
```

```
FILE * fp;
```

```
int c,i;
```

```
for(i=0;i<5;i++)
```

```
{
```

```
c=getchar();
```

```
putchar(c);
```

```
}
```

```
}
```

执行 1234 /*输入*/

1234 /*输出*/

`gets`（由标准输入设备内读进一字符串）

相关函数 `fopen`, `fread`, `fscanf`, `fgets`

表头文件 `#include<stdio.h>`

定义函数 `char * gets(char *s);`

函数说明 `gets()` 用来从标准设备读入字符并存到参数 `s` 所指的内存空间，直到出现换行字符或读到文件尾为止，最后加上 `NULL` 作为字符串结束。

返回值 `gets()` 若成功则返回 `s` 指针，返回 `NULL` 则表示有错误发生。

附加说明 由于 `gets()` 无法知道字符串 `s` 的大小，必须遇到换行字符或文件尾才会结束输入，因此容易造成缓冲溢出的安全性问题。建议使用 `fgets()` 取代。

范例 参考 `fgets()`

`mktemp`（产生唯一的临时文件名）

相关函数 `tmpfile`

表头文件 `#include<stdlib.h>`

定义函数 `char * mktemp(char * template);`

函数说明 `mktemp()` 用来产生唯一的临时文件名。参数 `template` 所指的文件名称字符串中最后六个字符必须是 `XXXXXX`。产生后的文件名会借字符串指针返回。

返回值 文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回 `NULL`，并把错误代码存在 `errno` 中。

附加说明 参数 `template` 所指的文件名称字符串必须声明为数组，如：

```
char template[]=" template-XXXXXX" ;
```

不可用 `char * template=" template-XXXXXX" ;`

范例

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

```
main()
```

```
{
```

```
char template[]=" template-XXXXXX" ;
```

```
mktemp(template);
```

```
printf( "template=%s\n",template);
```

```
}
```

`putc`（将一指定字符写入文件中）

相关函数 `fopen`, `fwrite`, `fscanf`, `fputc`

表头文件 `#include<stdio.h>`

定义函数 `int putc(int c,FILE * stream);`

函数说明 `putc()`会将参数 `c` 转为 `unsigned char` 后写入参数 `stream` 指定的文件中。虽然 `putc()` 与 `fputc()`作用相同，但 `putc()`为宏定义，非真正的函数调用。

返回值 `putc()`会返回写入成功的字符，即参数 `c`。若返回 `EOF` 则代表写入失败。

范例 参考 `fputc()`。

`putchar`（将指定的字符写到标准输出设备）

相关函数 `fopen`, `fwrite`, `fscanf`, `fputc`

表头文件 `#include<stdio.h>`

定义函数 `int putchar (int c);`

函数说明 `putchar()`用来将参数 `c` 字符写到标准输出设备。

返回值 `putchar()`会返回输出成功的字符，即参数 `c`。若返回 `EOF` 则代表输出失败。

附加说明 `putchar()`非真正函数，而是 `putc(c, stdout)`宏定义。

范例 参考 `getchar()`。

`rewind`（重设文件流的读写位置为文件开头）

相关函数 `fseek`, `ftell`, `fgetpos`, `fsetpos`

表头文件 `#include<stdio.h>`

定义函数 `void rewind(FILE * stream);`

函数说明 `rewind()`用来把文件流的读写位置移至文件开头。参数 `stream` 为已打开的文件指针。此函数相当于调用 `fseek(stream,0,SEEK_SET)`。

返回值

范例 参考 fseek()

setbuf (设置文件流的缓冲区)

相关函数 setbuffer, setlinebuf, setvbuf

表头文件 #include<stdio.h>

定义函数 void setbuf(FILE * stream, char * buf);

函数说明 在打开文件流后, 读取内容之前, 调用 setbuf() 可以用来设置文件流的缓冲区。

参数 stream 为指定的文件流, 参数 buf 指向自定的缓冲区起始地址。如果参数 buf 为 NULL 指针, 则为无缓冲 IO。Setbuf() 相当于调用: setvbuf(stream, buf, buf? _IOFBF: _IONBF, BUFSIZ)

返回值

setbuffer (设置文件流的缓冲区)

相关函数 setlinebuf, setbuf, setvbuf

表头文件 #include<stdio.h>

定义函数 void setbuffer(FILE * stream, char * buf, size_t size);

函数说明 在打开文件流后, 读取内容之前, 调用 setbuffer() 可用来设置文件流的缓冲区。

参数 stream 为指定的文件流, 参数 buf 指向自定的缓冲区起始地址, 参数 size 为缓冲区大小。

返回值

setlinebuf (设置文件流为线性缓冲区)

相关函数 setbuffer, setbuf, setvbuf

表头文件 #include<stdio.h>

定义函数 void setlinebuf(FILE * stream);

函数说明 setlinebuf() 用来设置文件流以换行为依据的无缓冲 IO。相当于调用: setvbuf(stream, (char *) NULL, _IOLBF, 0); 请参考 setvbuf()。

返回值

setvbuf (设置文件流的缓冲区)

相关函数 setbuffer, setlinebuf, setbuf

表头文件 #include<stdio.h>

定义函数 int setvbuf(FILE * stream, char * buf, int mode, size_t size);

函数说明 在打开文件流后, 读取内容之前, 调用 setvbuf() 可以用来设置文件流的缓冲区。参数 stream 为指定的文件流, 参数 buf 指向自定的缓冲区起始地址, 参数 size 为缓冲区大小, 参数 mode 有下列几种

`_IONBF` 无缓冲 IO

`_IOLBF` 以换行为依据的无缓冲 IO

`_IOFBF` 完全无缓冲 IO。如果参数 `buf` 为 `NULL` 指针，则为无缓冲 IO。

返回值

`ungetc` (将指定字符写回文件流中)

相关函数 `fputc`, `getchar`, `getc`

表头文件 `#include <stdio.h>`

定义函数 `int ungetc(int c, FILE * stream);`

函数说明 `ungetc()` 将参数 `c` 字符写回参数 `stream` 所指定的文件流。这个写回的字符会由下一个读取文件流的函数取得。

返回值 成功则返回 `c` 字符，若有错误则返回 `EOF`。

附录 D 标准 ASCII 码表及扩展字符集

1. 标准 ASCII 码表

Ctrl	十进制	十六进制	字符	代码	十进制	十六进制	字符	十进制	十六进制	字符	十进制	十六进制	字符
^@	0	00		NUL	32	20	!	64	40	@	96	60	'
^A	1	01		SOH	33	21	!	65	41	A	97	61	a
^B	2	02		STX	34	22	"	66	42	B	98	62	b
^C	3	03		ETX	35	23	#	67	43	C	99	63	c
^D	4	04		EOT	36	24	\$	68	44	D	100	64	d
^E	5	05		ENQ	37	25	%	69	45	E	101	65	e
^F	6	06		ACK	38	26	&	70	46	F	102	66	f
^G	7	07		BEL	39	27	'	71	47	G	103	67	g
^H	8	08		BS	40	28	(72	48	H	104	68	h
^I	9	09		HT	41	29)	73	49	I	105	69	i
^J	10	0A		LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B		VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C		FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D		CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E		SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F		SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10		DLE	48	30	0	80	50	P	112	70	p
^Q	17	11		DC1	49	31	1	81	51	Q	113	71	q
^R	18	12		DC2	50	32	2	82	52	R	114	72	r
^S	19	13		DC3	51	33	3	83	53	S	115	73	s
^T	20	14		DC4	52	34	4	84	54	T	116	74	t
^U	21	15		NAK	53	35	5	85	55	U	117	75	u
^V	22	16		SYN	54	36	6	86	56	V	118	76	v
^W	23	17		ETB	55	37	7	87	57	W	119	77	w
^X	24	18		CAN	56	38	8	88	58	X	120	78	x
^Y	25	19		EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A		SUB	58	3A	:	90	5A	Z	122	7A	z
^[27	1B		ESC	59	3B	;	91	5B	[123	7B	{
^\	28	1C		FS	60	3C	<	92	5C	\	124	7C	
^]	29	1D		GS	61	3D	=	93	5D]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^-	31	1F	▼	US	63	3F	?	95	5F	-	127	7F	ÿ

* ASCII 代码 127 拥有代码 DEL。在 MS-DOS 下，此代码具有与 ASCII 8 (BS) 相同的效果。
DEL 代码可由 CTRL + BKSP 键生成。

2. 扩展字符集

十进制	十六进制	字符	十进制	十六进制	字符	十进制	十六进制	字符	十进制	十六进制	字符
128	80	Ç	160	A0	á	192	C0	Ł	224	E0	α
129	81	ü	161	A1	î	193	C1	⊥	225	E1	Β
130	82	ë	162	A2	ó	194	C2	⊤	226	E2	Γ
131	83	â	163	A3	ô	195	C3	⊢	227	E3	Π
132	84	ä	164	A4	û	196	C4	—	228	E4	Σ
133	85	å	165	A5	ñ	197	C5	⊥	229	E5	σ
134	86	ä	166	A6	à	198	C6	⊥	230	E6	μ
135	87	ç	167	A7	ø	199	C7	⊥	231	E7	Υ
136	88	ê	168	A8	ç	200	C8	⊥	232	E8	ϕ
137	89	ë	169	A9	¸	201	C9	⊥	233	E9	θ
138	8A	è	170	AA	½	202	CA	⊥	234	EA	Ω
139	8B	ï	171	AB	¼	203	CB	⊥	235	EB	δ
140	8C	î	172	AC	¼	204	CC	⊥	236	EC	∞
141	8D	ï	173	AD	¡	205	CD	=	237	ED	Φ
142	8E	Ä	174	AE	«	206	CE	⊥	238	EE	ε
143	8F	Å	175	AF	»	207	CF	⊥	239	EF	Π
144	90	É	176	B0	░	208	D0	⊥	240	F0	≡
145	91	æ	177	B1	▒	209	D1	⊥	241	F1	±
146	92	ē	178	B2	▓	210	D2	⊥	242	F2	≥
147	93	ô	179	B3	⏏	211	D3	⊥	243	F3	≤
148	94	ö	180	B4	⏏	212	D4	⊥	244	F4	∫
149	95	ò	181	B5	⏏	213	D5	⊥	245	F5	∫
150	96	û	182	B6	⏏	214	D6	⊥	246	F6	+
151	97	ü	183	B7	⏏	215	D7	⊥	247	F7	≈
152	98	ÿ	184	B8	⏏	216	D8	⊥	248	F8	°
153	99	ö	185	B9	⏏	217	D9	⏏	249	F9	•
154	9A	Ü	186	BA	⏏	218	DA	⏏	250	FA	·
155	9B	¢	187	BB	⏏	219	DB	■	251	FB	√
156	9C	£	188	BC	⏏	220	DC	■	252	FC	n
157	9D	¥	189	BD	⏏	221	DD	■	253	FD	2
158	9E	℞	190	BE	⏏	222	DE	■	254	FE	■
159	9F	f	191	BF	⏏	223	DF	■	255	FF	

附录 E 键盘扫描码

1. 特殊键

按键	扫描码
Backspace	0E
Caps Lock	3A
Delete	53
End	4F
Enter	1C
Escape	01
HOME	47
Insert	52
Left Alt	38
Left Ctrl	1D
Left Shift	2A
Left Windows	5B
Num Lock	45
Page Down	51
Page Up	49
Power	5E
Prt Screen	37
Right Alt	38
Right Ctrl	1D
Right Shift	36
Right Windows	5C
Scroll Lock	46
Sleep	5F
Space	39
Tab	0F
Wake	63

2. 数字小键盘

按键	扫描码
0	52
1	4F
2	50
3	51
4	4B
5	4C
6	4D
7	47
8	48
9	49
-	4A
*	37
.	53
/	35
+	4E
Enter	1C

3. 功能键

按键	扫描码
F1	3B
F2	3C
F3	3D
F4	3E
F5	3F
F6	40
F7	41
F8	42
F9	43
F10	44
F11	57
F12	58
F13	64
F14	65
F15	66

4. 箭头键

按键	扫描码
Down	50
Left	4B
Right	4D
Up	48

5. 字符键

按键	扫描码
' "	28
- _	0C
, <	33
. >	34
/ ?	35
; :	27
[{	1A
\	2B
] }	1B
` ~	29
= +	0D
0)	0B
1 !	02
2 @	03
3 #	04
4 \$	05
5 %	06
6 ^	07

7 &	08
8 *	09
9 (0A
A	1E
B	30
C	2E
D	20
E	12
F	21
G	22
H	23
I	17
J	24
K	25
L	26
M	32
N	31
O	18
P	19
Q	10
R	13
S	1F
T	14
U	16
V	2F
W	11
X	2D
Y	15
Z	2C

