第 一 章　概述

1. 什么是对象？什么是面向对象方法？这种方法有哪些特点？

答：①**对象**是构成世界的一个独立单位，它具有自己的静态特征和动态特征。面向对象方法中的对象，是系统中用来描述客观事物的一个实体，它是用来构成系统的一个基本单位，由一组属性和一组行为构成。

②**面向对象的方法**将数据及对数据的操作方法放在一起，作为一个相互依存、不可分离的整体--对象。对同类型对象抽象出其共性，形成类。类中的大多数数据，只能用本类的方法进行处理。类通过一个简单的外部接口，与外界发生关系，对象与对象之间通过消息进行通讯。

2.什么叫做封装？

答：封装是面向对象方法的一个重要原则，就是把对象的属性和服务结合成一个独立的系统单位，并尽可能隐蔽对象的内部细节。

第二章 C++简单程序设计

1. 使用关键字 const 而不是#define 语句的好处有哪些？

答：const 定义的常量是有类型的，所以在使用它们时编译器可以查错；而且，这些变量在调试时仍然是可见的。

2. 在下面的枚举类型中，**Blue** 的值是多少？

enum COLOR { WHITE， BLACK = 100， RED， BLUE， GREEN = 300 };

答：Blue = 102

3. 在一个 **for** 循环中，可以初始化多个变量吗？如何实现？

答：在 for 循环设置条件的第一个";"前，用，分隔不同的赋值表达式。

for (x = 0， y = 10; x < 100; x++， y++) //不能像 C# 第一个;前 int x=0

4. 执行完下列语句后，**n** 的值为多少？

int n;
for (n = 0; n < 100; n++)

答： n 的值为 100

5. 什么叫做作用域？什么叫做局部变量？什么叫做全局变量？

答：作用域是一个标识符在程序正文中有效的区域。局部变量，一般来讲就是具有块作用域的变量；全局变量，就是具有文件作用域的变量。

6. 打印 **ASCII** 码为 **32~127** 的字符。

答：

```
#include <iostream.h>
int main()
{
for (int i = 32; i<128; i++)
cout << (char) i; //将 int 转换成 char 型即可输出数字对应 ASCII 码
return 0;
}
```

程序运行输出：

!"#$%G'()*+，./0123456789:;<>?@ABCDEFGHIJKLMNOP_QRSTUVWXYZ[\]^'abcdefghijklmnopqrstuvwxyz<|>~s

7.什么叫常量？什么叫变量？

答：所谓常量是指在程序运行的整个过程中其值始终不可改变的量，除了用**文字表示常量**外，**也可以为常量命名，这就是符号常量**；在程序的执行过程中其值可以变化的量称为变量，变量是需要用名字来标识的。

8. 变量有哪几种存储类型？

答：变量有以下几种存储类型：

1.auto 存储类型：采用堆栈方式分配内存空间，属于一时性存储，其存储空间可以被若干变量多次覆盖使用；

2.register 存储类型：存放在通用寄存器中；

3.extern 存储类型：在所有函数和程序段中都可引用；

4.static 存储类型：在内存中是以固定地址存放的，在整个程序运行期间都有效。

9.位操作 若 a = 1，b = 2，c = 3，下列各式的结果是什么？

1. a | b - c

2. a ^ b & -c

3. a & b | c

4. ! a | a

5. a >> 2

答:

1． -1 ？取-是什么意思?

2． 1 。 ^异或，位不同才为 1。1^2 = 0000 0001 ^ 0000 0010 = 0000 0011(3)

3-3 = 0???

3． 3 。1&2 = 0000 0001 & 0000 0010 = 0000 0011 （3）

3|3 = 0000 0011 | 0000 0011 = 0000 0011 （3）

4.！？？

5.0。0000 0010>>2，右移 2 位补 0（补码**正**数）=0000 0000（0）

2-32 比较 Break 语句与 Continue 语句的不同用法。

解:

Break 使程序从循环体和 switch 语句内跳出，继续执行逻辑上的下一条语句，不能用在别处;

continue 语句结束本次循环，接着开始判断决定是否继续执行下一次循环;

2-33 定义一个表示时间的结构体，可以精确表示年、月、日、小时、分、秒；提示用户输入年、月、日、小时、分、秒的值，然后完整地显示出来。

解:

源程序见"实验指导"部分实验二

2-34 在程序中定义一个整型变量，赋以 1~100 的值，要求用户猜这个数，比较两个数的大小，把结果提示给用户，直到猜对为止。分别使用 while、do...while 语句实现循环。

解：

//使用 while 语句

```
#include <iostream.h>
void main() {
int n = 18;
int m = 0;
while(m != n)
{
cout << "请猜这个数的值为多少？(0~~100):";
cin >> m;
if (n > m)
cout << "你猜的值太小了！" << endl;
else if (n < m)
cout << "你猜的值太大了！" << endl;
else
cout << "你猜对了！" << endl;
}
}
```

//使用 do...while 语句

```
#include <iostream.h>
void main() {
int n = 18;
int m = 0;
do{
cout << "请猜这个数的值为多少？(0~~100):";
```

```
cin >> m;
if (n > m)
cout << "你猜的值太小了！ " << endl;
else if (n < m)
cout << "你猜的值太大了！ " << endl;
else
cout << "你猜对了！ " << endl;
}while(n != m);
}
```

程序运行输出：

```
请猜这个数的值为多少？(0~~100):50
你猜的值太大了!
请猜这个数的值为多少？(0~~100):25
你猜的值太大了!
请猜这个数的值为多少？(0~~100):10
你猜的值太小了!
请猜这个数的值为多少？(0~~100):15
你猜的值太小了!
请猜这个数的值为多少？(0~~100):18
你猜对了!
```

2-35 定义枚举类型 weekday，包括 Sunday 到 Saturday 七个元素在程序中定义 weekday

类型的变量，对其赋值，定义整型变量，看看能否对其赋 weekday 类型的值。

解：

```
#include <iostream.h>
enum weekday
{
Sunday,
Monday,
Tuesday,
Wednesday,
Thursday,
Friday,
Saturday
};
void main()
{
int i;
weekday d = Thursday;
cout << "d = " << d << endl;
i = d;
cout << "i = " << i << endl;
d = (weekday)6;
cout << "d = " << d << endl;
d = weekday( 4 );
cout << "d = " << d << endl;
}
```

程序运行输出：

```
d = 4
i = 4
d = 6
d = 4
```

第三章　　函数

3-1 C++中的函数是什么？什么叫主调函数，什么叫被调函数，二者之间有什么关系？如何调用一个函数？

解：

一个较为复杂的系统往往需要划分为若干子系统，高级语言中的子程序就是用来实现这种模块划分的。C 和 C++语言中的子程序就体现为函数。调用其它函数的函数被称为主调函数，被其它函数调用的函数称为被调函数。一个函数很可能既调用别的函数又被另外的函数调用，这样它可能在某一个调用与被调用关系中充当主调函数，而在另一个调用与被调用关系中充当被调函数。

调用函数之前先要声明函数原型。按如下形式声明：

类型标识符 被调函数名 (含类型说明的形参表);

声明了函数原型之后，便可以按如下形式调用子函数：

函数名（实参列表）

3-2 观察下面程序的运行输出，与你设想的有何不同？仔细体会引用的用法。

源程序：

```
#include <iostream.h>
int main()
{
int intOne;
int &rSomeRef = intOne;
intOne = 5;
cout << "intOne:\t\t" << intOne << endl;
cout << "rSomeRef:\t" << rSomeRef << endl;
int intTwo = 8;
rSomeRef = intTwo; // not what you think!
cout << "\nintOne:\t\t" << intOne << endl;
```

```
cout << "intTwo:\t\t" << intTwo << endl;
cout << "rSomeRef:\t" << rSomeRef << endl;
return 0;
}
```

程序运行输出：

```
intOne: 5
rSomeRef: 5
intOne: 8
intTwo: 8
rSomeRef: 8
```

3-3 比较值调用和引用调用的相同点与不同点。

解：

值调用是指当发生函数调用时，给形参分配内存空间，并用实参来初始化形参（直接将实参的值传递给形参)。这一过程是参数值的单向传递过程，一旦形参获得了值便与实参脱离关系，此后无论形参发生了怎样的改变，都不会影响到实参。

引用调用将引用作为形参，在执行主调函数中的调用语句时，系统自动用实参来初始化形参。这样形参就成为实参的一个别名，对形参的任何操作也就直接作用于实参。

3-4 什么叫内联函数?它有哪些特点?

解：

定义时使用关键字 inline 的函数叫做内联函数；

编译器在编译时在调用处用函数体进行替换,节省了参数传递、控制转移等开销；

内联函数体内不能有循环语句和 switch 语句；

内联函数的定义必须出现在内联函数第一次被调用之前；

对内联函数不能进行异常接口声明；

3-5 函数原型中的参数名与函数定义中的参数名以及函数调用中的参数名必须一致吗?

解:

不必一致，所有的参数是根据位置和类型而不是名字来区分的。

3-6 重载函数时通过什么来区分?

解:

重载的函数的函数名是相同的，但它们的参数的个数和数据类型不同，编译器根据实参和形参的类型及个数的最佳匹配，自动确定调用哪一个函数。

3-7 编写函数，参数为两个 unsigned short int 型数，返回值为第一个参数除以第二个参数的结果，数据类型为 short int；如果第二个参数为 0，则返回值为-1。在主程序中实现输入输出。

解:

源程序:

```
#include <iostream.h>
short int Divider(unsigned short int a, unsigned short int b)
{
if (b == 0)
return -1;
else
return a/b;
}
typedef unsigned short int USHORT;
typedef unsigned long int ULONG;
int main()
{
```

```
USHORT one, two;
short int answer;
cout << "Enter two numbers.\n Number one: ";
cin >> one;
cout << "Number two: ";
cin >> two;
answer = Divider(one, two);
if (answer > -1)
cout << "Answer: " << answer;
else
cout << "Error, can't divide by zero!";
return 0;
}
```

程序运行输出：

```
Enter two numbers.
Number one:8
Number two:2
Answer: 4
```

3-8 编写函数把华氏温度转换为摄氏温度，公式为：C = (F - 32) * 5/9；在主程序中提示用户输入一个华氏温度，转化后输出相应的摄氏温度。

解：

源程序见"实验指导"部分实验三

3-9 编写函数判断一个数是否是质数，在主程序中实现输入、输出。

解：

```
#include <iostream.h>
#include <math.h>
int prime(int i); //判一个数是否是质数的函数
void main()
```

```
{
int i;
cout << "请输入一个整数：";
cin >> i;
if (prime(i))
cout << i << "是质数." << endl;
else
cout << i << "不是质数." << endl;
}
int prime(int i)
{
int j,k,flag;
flag = 1;
k = sqrt(i);
for (j = 2; j <= k; j++)
{
if(i%j == 0)
{
flag = 0;
break;
}
}
if (flag)
return 1;
else
return 0;
}
```

程序运行输出：

请输入一个整数：1151

1151 是质数.

3-10 编写函数求两个整数的最大公约数和最小公倍数。

解：

源程序：

```
#include <iostream.h>
#include <math.h>
int fn1(int i,int j); //求最大公约数的函数
void main()
{
int i,j,x,y;
cout << "请输入一个正整数： ";
cin >> i ;
cout << "请输入另一个正整数： ";
cin >> j ;
x = fn1(i,j);
y = i * j / x;
cout << i << "和" << j << "的最大公约数是： " << x << endl;

cout << i << "和" << j << "的最小公倍数是： " << y << endl;
}
int fn1(int i, int j)
{
int temp;
if (i < j)
{
temp = i;
i = j;
j = i;
}
while(j != 0)
```

```
{
temp = i % j;
i = j;
j = temp;
}
return i;
}
```

程序运行输出：

请输入一个正整数：120

请输入另一个正整数：72

120 和 72 的最大公约数是：24

120 和 72 的最小公倍数是：360

3-11 什么叫作嵌套调用？什么叫作递归调用？

解：

函数允许嵌套调用，如果函数 1 调用了函数 2，函数 2 再调用函数 3，便形成了函数的嵌套调用。

函数可以直接或间接地调用自身，称为递归调用。

3-12 在主程序中提示输入整数 n，编写函数用递归的方法求 1 + 2 + ... + n 的值。

解：

```
#include <iostream.h>
#include <math.h>
int fn1(int i);
void main()
{
int i;
```

```
cout << "请输入一个正整数： ";
cin >> i ;
cout << "从 1 累加到" <<i << "的和为： " << fn1(i) << endl;
}
int fn1(int i)
{
if (i == 1)
return 1;
else
return i + fn1(i -1);
}
```

程序运行输出：

请输入一个正整数：100

从 1 累加到 100 的和为：5050

3-13 编写递归函数 GetPower(int x， int y)计算 x 的 y 次幂， 在主程序中实现输入输出。

解：

源程序：

```
#include <iostream.h>
long GetPower(int x， int y);
int main()
{
int number， power;
long answer;
cout << "Enter a number: ";
cin >> number;
cout << "To what power? ";
cin >> power;
```

```
answer = GetPower(number， power);

cout << number << " to the " << power << "th power is " <<answer << endl;
return 0;
}

long GetPower(int x， int y)

{
if(y == 1)
return x;
else

return (x * GetPower(x，y-1));

}
```

程序运行输出：

```
Enter a number: 3
To what power? 4
3 to the 4th power is 81
```

3-14 用递归的方法编写函数求 Fibonacci 级数，公式为 fib(n) = fib(n-1) + fib(n-2)，n>2;

fib(1) = fib(2) = 1;观察递归调用的过程。

解：

源程序见"实验指导"部分实验三

3-15 用递归的方法编写函数求 n 阶勒让德多项式的值，在主程序中实现输入、输出；

解：

```
#include <iostream.h>
float p(int n, int x);
void main()
{
int n,x;
```

```
cout << "请输入正整数 n： ";
cin >> n;
cout << "请输入正整数 x： ";
cin >> x;
cout << "n = " << n << endl;
cout << "x = " << x << endl;
cout << "P" << n << "(" << x << ") = " << p(n,x) << endl;
}
float p(int n, int x)
{
if (n == 0)
return 1;
else if (n == 1)
return x;
else
return ((2*n-1)*x*p(n-1,x) - (n-1)*p(n-2,x)) /n ;
}
```

程序运行输出：

```
请输入正整数 n： 1
请输入正整数 x： 2
n = 1
x = 2
P1(2) = 2
请输入正整数 n： 3
请输入正整数 x： 4
n = 3
x = 4
P3(4) = 154
```

3-16 使用模板函数实现 Swap(x， y)，函数功能为交换 x、y 的值。

解：

源程序：

```
#include <iostream.h>

template <typename T> void swap(T &x， T &y)

{

T z;

z = x;

x = y;

y = z;

}

void main()

{

int j = 1， k = 2;

double v = 3.0， w = 4.0;

cout << "j = " <<j << " k = " << k << endl;

cout << "v = " <<v << " w = " << w << endl;

swap(j， k); //int

swap(v， w); //double

cout << "After swap:" << endl;

cout << "j = " <<j << " k = " << k << endl;

cout << "v = " <<v << " w = " << w << endl;

}
```

程序运行输出：

```
j = 1 k = 2

v = 3.14 w = 4.35

After swap:
```

j = 2 k = 1

v = 4.35 w = 3.14

第 四 章　　类

4-1 解释 public 和 private 的作用，公有类型成员与私有类型成员有些什么区别?

解:

公有类型成员用 public 关键字声明，公有类型定义了类的外部接口；私有类型的成员用 private 关键字声明，只允许本类的函数成员来访问，而类外部的任何访问都是非法的，这样，私有的成员就整个隐蔽在类中，在类的外部根本就无法看到，实现了访问权限的有效控制。

4-2 protected 关键字有何作用?

解:

protected 用来声明保护类型的成员，保护类型的性质和私有类型的性质相似，其差别在于继承和派生时派生类的成员函数可以访问基类的保护成员。

4-3 构造函数和析构函数有什么作用?

解:

构造函数的作用就是在对象被创建时利用特定的值构造对象,将对象初始化为一个特定的状态，使此对象具有区别于彼对象的特征，完成的就是是一个从一般到具体的过程，构造函数在对象创建的时候由系统自动调用。

析构函数与构造函数的作用几乎正好相反，它是用来完成对象被删除前的一些清理工作，也就是专门作扫尾工作的。一般情况下，析构函数是在对象的生存期即将结束的时刻由系统自动调用的，它的调用完成之后，对象也就消失了，相应的内存空间也被释放。

4-4 数据成员可以为公有的吗？成员函数可以为私有的吗？

解：

可以，二者都是合法的。数据成员和成员函数都可以为公有或私有的。但数据成员最好定义为私有的。

4-5 已知 class A 中有数据成员 int a，如果定义了 A 的两个对象 A1、A2，它们各自的数据成员 a 的值可以不同吗？

解：

可以，类的每一个对象都有自己的数据成员。

4-6 什么叫做拷贝构造函数？拷贝构造函数何时被调用？

解：

拷贝构造函数是一种特殊的构造函数，具有一般构造函数的所有特性，其形参是本类的对象的引用，其作用是使用一个已经存在的对象，去初始化一个新的同类的对象。在以下三种情况下会被调用：在当用类的一个对象去初始化该类的另一个对象时；如果函数的形参是类对象，调用函数进行形参和实参结合时；如果函数的返回值是类对象，函数调用完成返回时；

4-7 拷贝构造函数与赋值运算符(=)有何不同？

解：

赋值运算符(=)作用于一个已存在的对象；而拷贝构造函数会创建一个新的对象。

4-8 定义一个 Dog 类，包含的 age、weight 等属性，以及对这些属性操作的方法。实现并测试这个类。

解:

源程序:

```
#include <iostream.h>
class Dog
{
public:
Dog (int initialAge = 0，  int initialWeight = 5);
~Dog();
int GetAge() { return itsAge;} // inline!
void SetAge (int age) { itsAge = age;} // inline!
int GetWeight() { return itsWeight;} // inline!
void SetWeight (int weight) { itsAge = weight;} // inline!
private:
int itsAge，  itsWeight;
};
Dog::Dog(int initialAge，  int initialWeight)
{
itsAge = initialAge;
itsWeight = initialWeight;
}
Dog::~Dog() //destructor，  takes no action
{
}
int main()
{
Dog Jack(2，10);
cout << "Jack is a Dog who is " ;
cout << Jack.GetAge() << " years old and";
cout << Jack.GetWeight() << " pounds weight.\n";
```

```
Jack.SetAge(7);
Jack.SetWeight(20);
cout << "Now Jack is " ;
cout << Jack.GetAge() << " years old and";
cout << Jack.GetWeight() << " pounds weight.";
return 0;
}
```

程序运行输出：

```
Jack is a Dog who is 2 years old and 10 pounds weight.
Now Jack is 7 years old 20 pounds weight.
```

4-9 设计并测试一个名为 Rectangle 的矩形类，其属性为矩形的左下角与右上角两个点的坐标，能计算矩形的面积。

解：

源程序：

```
#include <iostream.h>
class Rectangle
{
public:
Rectangle (int top, int left, int bottom, int right);
~Rectangle () {}
int GetTop() const { return itsTop; }
int GetLeft() const { return itsLeft; }
int GetBottom() const { return itsBottom; }
int GetRight() const { return itsRight; }
void SetTop(int top) { itsTop = top; }
void SetLeft (int left) { itsLeft = left; }
void SetBottom (int bottom) { itsBottom = bottom; }
void SetRight (int right) { itsRight = right; }
int GetArea() const;
private:
int itsTop;
```

```
int itsLeft;
int itsBottom;
int itsRight;
};
Rectangle::Rectangle(int top, int left, int bottom, int right)
{
itsTop = top;
itsLeft = left;
itsBottom = bottom;
itsRight = right;
}
int Rectangle::GetArea() const
{
int Width = itsRight-itsLeft;
int Height = itsTop - itsBottom;
return (Width * Height);
}
int main()
{
Rectangle MyRectangle (100, 20, 50, 80 );
int Area = MyRectangle.GetArea();
cout << "Area: " << Area << "\n";
return 0;
}
```

程序运行输出：

```
Area: 3000
Upper Left X Coordinate: 20
```

4-10 设计一个用于人事管理的 People（人员）类。考虑到通用性，这里只抽象出所有类型人员都具有的属性：number（编号）、sex（性别）、birthday（出生日期）、id（身份证号）等等。其中"出生日期"定义为一个"日期"类内嵌子对象。用成员函数实现对人员信息的录入和显示。要求包括：构造函数和析构函数、拷贝构造函数、内联成员函数、带缺省形参

值的成员函数、聚集。

解：

本题用作实验四的选做题，因此不给出答案。

4-11 定义一个矩形类，有长、宽两个属性，有成员函数计算矩形的面积

解：

```
#include <iostream.h>
class Rectangle
{
public:
Rectangle(float len, float width)
{
Length = len;
Width = width;
}
~Rectangle(){};
float GetArea() { return Length * Width; }
float GetLength() { return Length; }
float GetWidth() { return Width; }
private:
float Length;
float Width;
};
void main()
{
float length, width;
cout << "请输入矩形的长度：";
cin >> length;
cout << "请输入矩形的宽度：";
cin >> width;
```

```
Rectangle r(length, width);
cout << "长为" << length << "宽为" << width << "的矩形的面积为："
<< r.GetArea () << endl;
}
```

程序运行输出：

请输入矩形的长度：5

请输入矩形的宽度：4

长为 5 宽为 4 的矩形的面积为：20

4-12 定义一个"数据类型" datatype 类，能处理包含字符型、整型、浮点型三种类型的数据，给出其构造函数。

解：

```
#include <iostream.h>
class datatype{
enum{
character,
integer,
floating_point
} vartype;
union
{
char c;
int i;
float f;
};
public:
datatype(char ch) {
vartype = character;
c = ch;
```

```
}
datatype(int ii) {
vartype = integer;
i = ii;
}
datatype(float ff) {
vartype = floating_point;
f = ff;
}
void print();
};
void datatype::print() {
switch (vartype) {
case character:
cout << "字符型: " << c << endl;
break;
case integer:
cout << "整型: " << i << endl;
break;
case floating_point:
cout << "浮点型: " << f << endl;
break;
}
}
void main() {
datatype A('c'), B(12), C(1.44F);
A.print();
B.print();
C.print();
}
```

程序运行输出：

字符型: c

整型: 12

浮点型: 1.44

4-13 定义一个 Circle 类，有数据成员半径 Radius，成员函数 GetArea()，计算圆的面积，构造一个 Circle 的对象进行测试。

解：

```
#include <iostream.h>
class Circle
{
public:
Circle(float radius){ Radius = radius;}
~Circle(){}
float GetArea() { return 3.14 * Radius * Radius; }
private:
float Radius;
};
void main()
{
float radius;
cout << "请输入圆的半径： ";
cin >> radius;
Circle p(radius);
cout << "半径为" << radius << "的圆的面积为： " << p.GetArea ()
<< endl;
}
```

程序运行输出：

请输入圆的半径：5

半径为 5 的圆的面积为：78.5

4-14 定义一个 tree 类，有成员 ages，成员函数 grow(int years)对 ages 加上 years,age()显示 tree 对象的 ages 的值。

解：

```
#include <iostream.h>
class Tree {
int ages;
public:
Tree(int n=0);
~Tree();
void grow(int years);
void age();
};
Tree::Tree(int n) {
ages = n;
}
Tree::~Tree() {
age();
}
void Tree::grow(int years) {
ages += years;
}
void Tree::age() {
cout << "这棵树的年龄为" << ages << endl;
}
void main()
{
Tree t(12);
t.age();
t.grow(4);
}
```

程序运行输出：

这棵树的年龄为 12

这棵树的年龄为 16

第 五 章　　C++程序的基本结构

5-1 什么叫做作用域？有哪几种类型的作用域？

解：

作用域讨论的是标识符的有效范围，作用域是一个标识符在程序正文中有效的区域。C++的作用域分为函数原形作用域、块作用域(局部作用域)、类作用域和文件作用域.

5-2 什么叫做可见性？可见性的一般规则是什么？

解：

可见性是标识符是否可以引用的问题；

可见性的一般规则是：标识符要声明在前，引用在后，在同一作用域中，不能声明同名的标识符。对于在不同的作用域声明的标识符，遵循的原则是：若有两个或多个具有包含关系的作用域，外层声明的标识符如果在内层没有声明同名标识符时仍可见，如果内层声明了同名标识符则外层标识符不可见。

5-3 下面的程序的运行结果是什么，实际运行一下，看看与你的设想有何不同。

```
#include <iostream.h>
void myFunction();

int x = 5,    y = 7;

int main()
{
cout << "x from main: " << x << "\n";
```

```
cout << "y from main: " << y << "\n\n";
myFunction();
cout << "Back from myFunction!\n\n";
cout << "x from main: " << x << "\n";
cout << "y from main: " << y << "\n";
return 0;
}
void myFunction()
{
int y = 10;
cout << "x from myFunction: " << x << "\n";
cout << "y from myFunction: " << y << "\n\n";
}
```

解：

程序运行输出：

```
x from main: 5
y from main: 7
x from myFunction: 5
y from myFunction: 10
Back from myFunction!
x from main: 5
y from main: 7
```

5-4 假设有两个无关系的类 Engine 和 Fuel，使用时，怎样允许 Fuel 成员访问 Engine 中的私有和保护的成员?

解：

源程序：

```
class fuel;
class engine
{
```

```
friend class fuel;
private;
int powerlevel;
public;
engine(){ powerLevel = 0;}
void engine_fn(fuel &f);
};
class fuel
{
friend class engine;
private;
int fuelLevel;
public:
fuel(){ fuelLevel = 0;}
void fuel_fn( engine &e);
};
```

5-5 什么叫做静态数据成员？它有何特点？

解：

类的静态数据成员是类的数据成员的一种特例，采用 static 关键字来声明。对于类的普通数据成员，每一个类的对象都拥有一个拷贝，就是说每个对象的同名数据成员可以分别存储不同的数值，这也是保证对象拥有自身区别于其它对象的特征的需要，但是静态数据成员，每个类只要一个拷贝，由所有该类的对象共同维护和使用，这个共同维护、使用也就实现了同一类的不同对象之间的数据共享。

5-6 什么叫做静态函数成员？它有何特点？

解：

使用 static 关键字声明的函数成员是静态的，静态函数成员属于整个类，同一个类的所

有对象共同维护，为这些对象所共享。静态函数成员具有以下两个方面的好处，一是由于静态成员函数只能直接访问同一个类的静态数据成员，可以保证不会对该类的其余数据成员造成负面影响；二是同一个类只维护一个静态函数成员的拷贝，节约了系统的开销，提高程序的运行效率。

5-7 定义一个 Cat 类，拥有静态数据成员 HowManyCats，记录 Cat 的个体数目；静态成员函数 GetHowMany（），存取 HowManyCats。设计程序测试这个类，体会静态数据成员和静态成员函数的用法。

解：

源程序：

```
#include <iostream.h>
class Cat
{
public:
Cat(int age):itsAge(age){HowManyCats++; }
virtual ~Cat() { HowManyCats--; }
virtual int GetAge() { return itsAge; }
virtual void SetAge(int age) { itsAge = age; }
static int GetHowMany() { return HowManyCats; }
private:
int itsAge;
static int HowManyCats;
};
int Cat::HowManyCats = 0;
void TelepathicFunction();
int main()
{
const int MaxCats = 5;
Cat *CatHouse[MaxCats]; int i;
```

```
for (i = 0; i<MaxCats; i++)
{
CatHouse[i] = new Cat(i);
TelepathicFunction();
}
for ( i = 0; i<MaxCats; i++)
{
delete CatHouse[i];
TelepathicFunction();
}
return 0;
}
void TelepathicFunction()
{
cout << "There are " << Cat::GetHowMany() << " cats alive!\n";
}
```

程序运行输出：

```
There are 1 cats alive!
There are 2 cats alive!
There are 3 cats alive!
There are 4 cats alive!
There are 5 cats alive!
There are 4 cats alive!
There are 3 cats alive!
There are 2 cats alive!
There are 1 cats alive!
There are 0 cats alive!
```

5-8 什么叫做友元函数？什么叫做友元类？

解：

友元函数是使用 friend 关键字声明的函数，它可以访问相应类的保护成员和私有成员。友元类是使用 friend 关键字声明的类，它的所有成员函数都是相应类的友元函数。

5-9 如果类 A 是类 B 的友元，类 B 是类 C 的友元，类 D 是类 A 的派生类，那么类 B 是类 A 的友元吗？类 C 是类 A 的友元吗？类 D 是类 B 的友元吗？

解：

类 B 不是类 A 的友元，友元关系不具有交换性；

类 C 不是类 A 的友元，友元关系不具有传递性；

类 D 不是类 B 的友元，友元关系不能被继承。

5-10 静态成员变量可以为私有的吗？声明一个私有的静态整型成员变量。

解：

可以，例如：

```
private:
static int a;
```

5-11 在一个文件中定义一个全局变量 n，主函数 main()，在另一个文件中定义函数 fn1()，在 main()中对 n 赋值，再调用 fn1()，在 fn1()中也对 n 赋值，显示 n 最后的值。

解：

```
#include <iostream.h>
#include "fn1.h"
int n;
void main()
{
n = 20;
fn1();
cout << "n 的值为" <<n;
}
```

```
// fn1.h 文件
extern int n;
void fn1()
{
n=30;
}
```

程序运行输出：

n 的值为 30

5-12 在函数 fn1()中定义一个静态变量 n，fn1()中对 n 的值加 1，在主函数中，调用 fn1()十次，显示 n 的值。

解：

```
#include <iostream.h>
void fn1()
{
static int n = 0;
n++;
cout << "n 的值为" << n <<endl;
}
void main()
{
for(int i = 0; i < 10; i++)
fn1();
}
```

程序运行输出：

n 的值为 1

n 的值为 2

n 的值为 3

n 的值为 4

n 的值为 5

n 的值为 6

n 的值为 7

n 的值为 8

n 的值为 9

n 的值为 10

5-13 定义类 X、Y、Z，函数 h(X*)，满足：类 X 有私有成员 i，Y 的成员函数 g(X*)是 X 的友元函数，实现对 X 的成员 i 加 1，类 Z 是类 X 的友元类，其成员函数 f(X*)实现对 X 的成员 i 加 5，函数 h(X*)是 X 的友元函数，实现对 X 的成员 i 加 10。在一个文件中定义和实现类，在另一个文件中实现 main()函数。

解：

```
#include "my_x_y_z.h"
void main()
{
X x;
Z z;
z.f(&x);
}
```

// my_x_y_z.h 文件

```
#ifndef MY_X_Y_Z_H
class X;
class Y {
```

```
void g(X*);
};
class X
{
private:
int i;
public:
X(){i=0;}
friend void h(X*);
friend void Y::g(X*);
friend class Z;
};
void h(X* x) { x->i =+10; }
void Y::g(X* x) { x->i ++; }
class Z {
public:
void f(X* x) { x->i += 5; }
};
#endif // MY_X_Y_Z_H
```

程序运行输出：无

5-14 定义 Boat 与 Car 两个类，二者都有 weight 属性，定义二者的一个友元函数 totalWeight()，计算二者的重量和。

解：

源程序：

```
#include <iostream.h>
class Boat;
class Car
{
private:
int weight;
public:
```

```
Car(int j){weight = j;}
friend int totalWeight(Car &aCar,  Boat &aBoat);
};
class Boat
{
private:
int weight;
public:
Boat(int j){weight = j;}
friend int totalWeight(Car &aCar,  Boat &aBoat);
};
int totalWeight(Car &aCar,  Boat &aBoat)
{
return aCar.weight + aBoat.weight;
}
void main()
{
Car c1(4);
Boat b1(5);
cout << totalWeight(c1,  b1) << endl;
}
```

程序运行输出：

9

5-15 如果在类模板的定义中有一个静态数据成员，则在程序运行中会产生多少个相应的静态变量?

解：

这个类模板的每一个实例类都会产生一个相应的静态变量。

第 六 章　　数组、指针与字符串

6-1 数组 A[10][5][15]一共有多少个元素?

解:

10×5×15 = 750 个元素

1-2 在数组 A[20]中第一个元素和最后一个元素是哪一个?

解:

第一个元素是 A[0]，最后一个元素是 A[19]。

6-3 用一条语句定义一个有五个元素的整型数组，并依次赋予 1~5 的初值。

解:

源程序:

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

或：int IntegerArray[] = { 1， 2， 3， 4， 5 };

6-4 已知有一个数组名叫 oneArray，用一条语句求出其元素的个数。

解:

源程序:

```
nArrayLength = sizeof(oneArray) / sizeof(oneArray[0]);
```

6-5 用一条语句定义一个有 5×3 个元素的二维整型数组，并依次赋予 1~15 的初值。

解:

源程序:

```
int theArray[5][3] = { 1，2，3，4，5，6，7，8，9，10，11，12，13，14，15 };
```

或：int theArray[5][3] = { {1，2，3}， {4，5，6}， {7，8，9}， {10，11，12}，{13，14，15} };

6-6 运算符*和&的作用是什么？

解：

*称为指针运算符，是一个一元操作符，表示指针所指向的对象的值；&称为取地址运算符，也是一个一元操作符，是用来得到一个对象的地址。

6-7 什么叫做指针？指针中储存的地址和这个地址中的值有何区别？

解：

指针是一种数据类型，具有指针类型的变量称为指针变量。指针变量存放的是另外一个对象的地址，这个地址中的值就是另一个对象的内容。

6-8 定义一个整型指针，用 new 语句为其分配包含 10 个整型元素的地址空间。

解：

源程序：

```
int *pInteger = new int[10];
```

6-9 在字符串”Hello，world!”中结束符是什么？

解：

是 NULL 字符。

6-10 定义一个有五个元素的整型数组，在程序中提示用户输入元素值，最后再在屏幕上显示出来。

解：

源程序：

```
#include <iostream.h>
int main()
{
int myArray[5];
int i;
for ( i=0; i<5; i++)
{
cout << "Value for myArray[" << i << "]: ";
cin >> myArray[i];
}
for (i = 0; i<5; i++)
cout << i << ": " << myArray[i] << "\n";
return 0;
}
```

程序运行输出：

```
Value for myArray[0]: 2
Value for myArray[1]: 5
Value for myArray[2]: 7
Value for myArray[3]: 8
Value for myArray[4]: 3
0: 2
1: 5
2: 7
3: 8
4: 3
```

6-11 引用和指针有何区别？何时只能使用指针而不能使用引用？

解：

引用是一个别名，不能为 NULL 值，不能被重新分配；指针是一个存放地址的变量。当需要对变量重新赋以另外的地址或赋值为 NULL 时只能使用指针。

6-12 声明下列指针：float 类型变量的指针 pFloat，char 类型的指针 pString 和 struct

customer 型的指针 prec。

解：

```
float *pfloat;
char *pString;
struct customer *prec;
```

6-13 给定 float 类型的指针 fp，写出显示 fp 所指向的值的输出流语句。

解：

```
cout << "Value == " << *fp;
```

6-14 程序中定义一个 double 类型变量的指针。分别显示指针占了多少字节和指针所指的变量占了多少字节。

解：

```
double *counter;
cout << "\nSize of pointer == "sizeof(counter);
cout << '\nSize of addressed value == "<<sizeof(*counter);
```

6-15 const int * p1 和 int * const p2 的区别是什么?

解：

const int * p1 声明了一个指向整型常量的指针 p1，因此不能通过指针 p1 来改变它所指向的整型值；int * const p2 声明了一个指针型常量，用于存放整型变量的地址，这个指针一旦初始化后，就不能被重新赋值了。

6-16 定义一个整型变量 a，一个整型指针 p，一个引用 r，通过 p 把 a 的值改为 10，通过 r 把 a 的值改为 5

解：

```
void main()
{
```

```
int a;
int *p = &a;
int &r = a;
*p = 10;
r = 5;
}
```

6-17 下列程序有何问题，请仔细体会使用指针时应避免出现这个的问题。

```
#include <iostream.h>
int main()
{
int *p;
*pInt = 9;
cout << "The value at p: " << *p;
return 0;
}
```

解：

指针 p 没有初始化，也就是没有指向某个确定的内存单元，它指向内存中的一个随机地址，给这个随机地址赋值是非常危险的。

6-18 下列程序有何问题，请改正；仔细体会使用指针时应避免出现的这个问题。

```
#include <iostream.h>
int Fn1();
int main()
{
int a = Fn1();
cout << "the value of a is: " << a;
return 0;
}
int Fn1()
{
int * p = new int (5);
return *p;
```

```
}
```

解：

此程序中给*p 分配的内存没有被释放掉。

改正：

```
#include <iostream.h>
int* Fn1();
int main()
{
int *a = Fn1();
cout << "the value of a is: " << *a;
delete a;
return 0;
}
int* Fn1()
{
int * p = new int (5);
return p;
}
```

6-19 声明一个参数为整型，返回值为长整型的函数指针；声明类 A 的一个成员函数指针，其参数为整型，返回值长整型。

解：

```
long (* p_fn1)(int);
long ( A::*p_fn2)(int);
```

6-20 实现一个名为 SimpleCircle 的简单圆类，其数据成员 int *itsRadius 为一个指向其半径值的指针，设计对数据成员的各种操作，给出这个类的完整实现并测试这个类。

解：

源程序：

```
#include <iostream.h>
class SimpleCircle
{
public:
SimpleCircle();
SimpleCircle(int);
SimpleCircle(const SimpleCircle &);
~SimpleCircle() {}
void SetRadius(int);
int GetRadius()const;
private:
int *itsRadius;
};
SimpleCircle::SimpleCircle()
{
itsRadius = new int(5);
}
SimpleCircle::SimpleCircle(int radius)
{
itsRadius = new int(radius);
}
SimpleCircle::SimpleCircle(const SimpleCircle & rhs)
{
int val = rhs.GetRadius();
itsRadius = new int(val);
}
int SimpleCircle::GetRadius() const
{
return *itsRadius;
}
int main()
{
SimpleCircle CircleOne, CircleTwo(9);
cout << "CircleOne: " << CircleOne.GetRadius() << endl;
```

```
cout << "CircleTwo: " << CircleTwo.GetRadius() << endl;
return 0;
```

}程序运行输出：

```
CircleOne: 5
CircleTwo: 9
```

6-21 编写一个函数，统计一个英文句子中字母的个数，在主程序中实现输入、输出。

解：

源程序：

```
#include <iostream.h>
#include <stdio.h>
int count(char *str)
{
int i,num=0;
for (i=0; str[i]; i++)
{
if ( (str[i]>='a' && str[i]<='z') || (str[i]>='A' && str[i]<='Z') )
num++;
}
return num;
}
void main()
{
char text[100];
cout << "输入一个英语句子： " << endl;
gets(text);
cout << "这个句子里有" << count(text) << "个字母。" << endl;
}
```

程序运行输出：

输入一个英语句子：

It is very interesting!

这个句子里有 19 个字母。

6-22 编写函数 int index(char *s， char *t)，返回字符串 t 在字符串 s 中出现的最左边的位置，如果在 s 中没有与 t 匹配的子串，就返回-1。

解：

源程序：

```
#include <iostream.h>
int index( char *s, char *t)
{
int i,j,k;
for(i = 0; s[i] != '\0'; i++)
{
for(j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
;
if (t[k] =='\0')
return i;
}
return -1;
}
void main()
{
int n;
char str1[20],str2[20];
cout << "输入一个英语单词： ";
cin >> str1;
cout << "输入另一个英语单词： ";
cin >> str2;
```

```
n = index(str1,str2);
if (n > 0)
cout << str2 << "在" << str1 << "中左起第" << n+1
<< "个位置。"<<endl;
else
cout << str2 << "不在" << str1 << "中。" << endl;
}
```

程序运行输出：

输入一个英语单词：abcdefgh

输入另一个英语单词：de

de 在 abcdefghijk 中左起第 4 个位置。

6-23 编写函数 reverse(char *s)的倒序递归程序，使字符串 s 倒序。

解：

源程序：

```
#include <iostream.h>
#include <string.h>
void reverse(char *s, char *t)
{
char c;
if (s < t)
{
c = *s;
*s = *t;
*t = c;
reverse(++s, --t);
}
}
```

```
void reverse( char *s)
{
reverse(s, s + strlen(s) - 1);
}
void main()
{
char str1[20];
cout << "输入一个字符串：";
cin >> str1;
cout << "原字符串为：" << str1 << endl;
reverse(str1);
cout << "倒序反转后为：" << str1 << endl;
}
```

程序运行输出：

输入一个字符串：abcdefghijk

原字符串为：abcdefghijk

倒序反转后为：kjihgfedcba

6-24 设学生人数 N=8，提示用户输入 N 个人的考试成绩，然后计算出平均成绩，显示出来。

解：

源程序：

```
#include <iostream.h>
#include <string.h>
#define N 8
float grades[N]; //存放成绩的数组
```

```
void main()
{
int i;
float total,average;
//提示输入成绩
for(i = 0; i < N; i++ )
{
cout << "Enter grade #" <<(i +1) << ": ";
cin >> grades[i];
}
total = 0;
for (i = 0; i < N; i++)
total += grades[i];
average = total / N;
cout << "\nAverage grade: " << average << endl;
}
```

程序运行输出：

```
Enter grade #1: 86
Enter grade #2: 98
Enter grade #3: 67
Enter grade #4: 80
Enter grade #5: 78
Enter grade #6: 95
Enter grade #7: 78
Enter grade #8: 56
Average grade: 79.75
```

6-25 设计一个字符串类 MyString，具有构造函数、析构函数、拷贝构造函数，重载运算符+、=、+=、[]，尽可能地完善它，使之能满足各种需要。(运算符重载功能为选做，参见第 8 章)

解：

```
#include <iostream.h>
#include <string.h>
class MyString
{
public:
MyString();
MyString(const char *const);
MyString(const MyString &);
~MyString();
char & operator[](unsigned short offset);
char operator[](unsigned short offset) const;
MyString operator+(const MyString&);
void operator+=(const MyString&);
MyString & operator= (const MyString &);
unsigned short GetLen()const { return itsLen; }
const char * GetMyString() const { return itsMyString; }
private:
MyString (unsigned short); // private constructor
char * itsMyString;
unsigned short itsLen;
};
MyString::MyString()
{
itsMyString = new char[1];
itsMyString[0] = '\0';
itsLen=0;
}
MyString::MyString(unsigned short len)
{
itsMyString = new char[len+1];
for (unsigned short i = 0; i<=len; i++)
itsMyString[i] = '\0';
itsLen=len;
}
```

```
MyString::MyString(const char * const cMyString)
{
itsLen = strlen(cMyString);
itsMyString = new char[itsLen+1];
for (unsigned short i = 0; i<itsLen; i++)
itsMyString[i] = cMyString[i];
itsMyString[itsLen]='\0';
}
MyString::MyString (const MyString & rhs)
{
itsLen=rhs.GetLen();
itsMyString = new char[itsLen+1];
for (unsigned short i = 0; i<itsLen;i++)
itsMyString[i] = rhs[i];
itsMyString[itsLen] = '\0';
}
MyString::~MyString ()
{
delete [] itsMyString;
itsLen = 0;
}
MyString& MyString::operator=(const MyString & rhs)
{
if (this == &rhs)
return *this;
delete [] itsMyString;
itsLen=rhs.GetLen();
itsMyString = new char[itsLen+1];
for (unsigned short i = 0; i<itsLen;i++)
itsMyString[i] = rhs[i];
itsMyString[itsLen] = '\0';
return *this;
}
char & MyString::operator[](unsigned short offset)
```

```
{
if (offset > itsLen)
return itsMyString[itsLen-1];
else
return itsMyString[offset];
}
char MyString::operator[](unsigned short offset) const
{
if (offset > itsLen)
return itsMyString[itsLen-1];
else
return itsMyString[offset];
}
MyString MyString::operator+(const MyString& rhs)
{
unsigned short totalLen = itsLen + rhs.GetLen();
MyString temp(totalLen);
for (unsigned short i = 0; i<itsLen; i++)
temp[i] = itsMyString[i];
for (unsigned short j = 0; j<rhs.GetLen(); j++, i++)
temp[i] = rhs[j];
temp[totalLen]='\0';
return temp;
}
void MyString::operator+=(const MyString& rhs)
{
unsigned short rhsLen = rhs.GetLen();
unsigned short totalLen = itsLen + rhsLen;
MyString temp(totalLen);
for (unsigned short i = 0; i<itsLen; i++)
temp[i] = itsMyString[i];
for (unsigned short j = 0; j<rhs.GetLen(); j++, i++)
temp[i] = rhs[i-itsLen];
temp[totalLen]='\0';
```

```
*this = temp;
}
int main()
{
MyString s1("initial test");
cout << "S1:\t" << s1.GetMyString() << endl;
char * temp = "Hello World";
s1 = temp;
cout << "S1:\t" << s1.GetMyString() << endl;
char tempTwo[20];
strcpy(tempTwo,"; nice to be here!");
s1 += tempTwo;
cout << "tempTwo:\t" << tempTwo << endl;
cout << "S1:\t" << s1.GetMyString() << endl;
cout << "S1[4]:\t" << s1[4] << endl;
s1[4]='x';
cout << "S1:\t" << s1.GetMyString() << endl;
cout << "S1[999]:\t" << s1[999] << endl;
MyString s2(" Another myString");
MyString s3;
s3 = s1+s2;
cout << "S3:\t" << s3.GetMyString() << endl;
MyString s4;
s4 = "Why does this work?";
cout << "S4:\t" << s4.GetMyString() << endl;
return 0;
}
```

程序运行输出：

```
S1: initial test
S1: Hello World
tempTwo: ; nice to be here!
S1: Hello World; nice to be here!
S1[4]: o
```

S1: Hellx World; nice to be here!

S1[999]: !

S3: Hellx World; nice to be here! Another myString

S4: Why does this work?

6-26 编写一个 3×3 矩阵转置的函数，在 main()函数中输入数据。

解：

```
#include <iostream.h>
void move (int matrix[3][3])
{
int i, j, k;
for(i=0; i<3; i++)
for (j=0; j<i; j++)
{
k = matrix[i][j];
matrix[i][j] = matrix[j][i];
matrix[j][i] = k;
}
}
void main()
{
int i, j;
int data[3][3];
cout << "输入矩阵的元素" << endl;
for(i=0; i<3; i++)
for (j=0; j<3; j++)
{
cout << "第" << i+1 << "行第" << j+1
<<"个元素为： ";
cin >> data[i][j];
```

```
}
cout << "输入的矩阵的为： " << endl;
for(i=0; i<3; i++)
{
for (j=0; j<3; j++)
cout << data[i][j] << " ";
cout << endl;
}
move(data);
cout << "转置后的矩阵的为： " << endl;
for(i=0; i<3; i++)
{
for (j=0; j<3; j++)
cout << data[i][j] << " ";
cout << endl;
}
}
```

程序运行输出：

输入矩阵的元素

第 1 行第 1 个元素为：1

第 1 行第 2 个元素为：2

第 1 行第 3 个元素为：3

第 2 行第 1 个元素为：4

第 2 行第 2 个元素为：5

第 2 行第 3 个元素为：6

第 3 行第 1 个元素为：7

第 3 行第 2 个元素为：8

第 3 行第 3 个元素为：9

输入的矩阵的为：

1 2 3

4 5 6

7 8 9

转置后的矩阵的为：

1 4 7

2 5 8

3 6 9

6-27 编写一个矩阵转置的函数，矩阵的维数在程序中由用户输入。

解：

```
#include <iostream.h>
void move (int *matrix ,int n)
{
int i, j, k;
for(i=0; i<n; i++)
for (j=0; j<i; j++)
{
k = *(matrix + i*n + j);
*(matrix + i*n + j) = *(matrix + j*n + i);
*(matrix + j*n + i) = k;
}
}
void main()
{
int n, i, j;
int *p;
cout << "请输入矩阵的维数： ";
```

```
cin >> n;
p = new int[n*n];
cout << "输入矩阵的元素" << endl;
for(i=0; i<n; i++)
for (j=0; j<n; j++)
{
cout << "第" << i+1 << "行第" << j+1
<<"个元素为： ";
cin >> p[i*n + j];
}
cout << "输入的矩阵的为： " << endl;
for(i=0; i<n; i++)
{
for (j=0; j<n; j++)
cout << p[i*n + j] << " ";
cout << endl;
}
move(p, n);
cout << "转置后的矩阵的为： " << endl;
for(i=0; i<n; i++)
{
for (j=0; j<n; j++)
cout << p[i*n + j] << " ";
cout << endl;
}
}
```

程序运行输出：

请输入矩阵的维数：3

输入矩阵的元素

第 1 行第 1 个元素为：1

第 1 行第 2 个元素为：2

第 1 行第 3 个元素为：3

第 2 行第 1 个元素为：4

第 2 行第 2 个元素为：5

第 2 行第 3 个元素为：6

第 3 行第 1 个元素为：7

第 3 行第 2 个元素为：8

第 3 行第 3 个元素为：9

输入的矩阵的为：

1 2 3

4 5 6

7 8 9

转置后的矩阵的为：

1 4 7

2 5 8

3 6 9

6-28 定义一个 Employee 类，其中包括表示姓名、街道地址、城市和邮编等属性，包括 chage_name()和 display()等函数；display()使用 cout 语句显示姓名、街道地址、城市和邮编等属性，函数 change_name()改变对象的姓名属性，实现并测试这个类。

解：

源程序：

```
#include <iostream.h>
#include <string.h>
class Employee
{
private:
char name[30];
char street[30];
char city[18];
char zip[6];
public:
Employee(char *n, char *str, char *ct, char *z);
void change_name(char *n);
void display();
};
Employee::Employee (char *n,char *str,char *ct, char *z)
{
strcpy(name, n);
strcpy(street, str);
strcpy(city, ct);
strcpy(zip, z);
}
void Employee::change_name (char *n)
{
strcpy(name, n);
}
void Employee::display ()
{
cout << name << " " << street << " ";
cout << city << " "<< zip;
}
void main(void)
{
```

```
Employee e1("张三","平安大街 3 号", "北京", "100000");
e1.display();
cout << endl;
e1.change_name("李四");
e1.display();
cout << endl;
}
```

程序运行输出：

```
张三 平安大街 3 号 北京 100000
李四 平安大街 3 号 北京 100000
```

第 七 章　　继承与派生

7-1 比较类的三种继承方式 public 公有继承、protected 保护继承、private 私有继承之间的差别。

解：

不同的继承方式，导致不同访问属性的基类成员在派生类中的访问属性也有所不同：

公有继承，使得基类 public(公有)和 protected(保护)成员的访问属性在派生类中不变，而基类 private(私有)成员不可访问。

私有继承，使得基类 public(公有)和 protected(保护)成员都以 private(私有)成员身份出现在派生类中，而基类 private(私有)成员不可访问。

保护继承中，基类 public(公有)和 protected(保护)成员都以 protected(保护)成员身份出现在派生类中，而基类 private(私有)成员不可访问。

7-2 派生类构造函数执行的次序是怎样的?

解:

派生类构造函数执行的一般次序为:调用基类构造函数;调用成员对象的构造函数;派生类的构造函数体中的内容。

7-3 如果在派生类 B 已经重载了基类 A 的一个成员函数 fn1(),没有重载成员函数 fn2(),如何调用基类的成员函数 fn1()、fn2()?

解:

调用方法为: A::fn1();

fn2();

7-4 什么叫做虚基类?有何作用?

解:

当某类的部分或全部直接基类是从另一个基类派生而来,这些直接基类中,从上一级基类继承来的成员就拥有相同的名称,派生类的对象的这些同名成员在内存中同时拥有多个拷贝,我们可以使用作用域分辨符来唯一标识并分别访问它们。我们也可以将直接基类的共同基类设置为虚基类,这时从不同的路径继承过来的该类成员在内存中只拥有一个拷贝,这样就解决了同名成员的唯一标识问题。

虚基类的声明是在派生类的定义过程,其语法格式为:

class 派生类名:virtual 继承方式 基类名

上述语句声明基类为派生类的虚基类,在多继承情况下,虚基类关键字的作用范围和继承方式关键字相同,只对紧跟其后的基类起作用。声明了虚基类之后,虚基类的成员在进一步派生过程中,和派生类一起维护一个内存数据拷贝。

7-5 定义一个 Shape 基类，在此基础上派生出 Rectangle 和 Circle，二者都有 GetArea() 函数计算对象的面积。使用 Rectangle 类创建一个派生类 Square。

解：

源程序：

```
#include <iostream.h>
class Shape
{
public:
Shape(){}
~Shape(){}
virtual float GetArea() { return -1; }
};
class Circle : public Shape
{
public:
Circle(float radius):itsRadius(radius){}
~Circle(){}
float GetArea() { return 3.14 * itsRadius * itsRadius; }
private:
float itsRadius;
};
class Rectangle : public Shape
{
public:
Rectangle(float len， float width): itsLength(len)， itsWidth(width){};
~Rectangle(){};
virtual float GetArea() { return itsLength * itsWidth; }
virtual float GetLength() { return itsLength; }
virtual float GetWidth() { return itsWidth; }
private:
float itsWidth;
```

```
float itsLength;
};
class Square : public Rectangle
{
public:
Square(float len);
~Square(){}
};
Square::Square(float len):
Rectangle(len，len)
{
}
void main()
{
Shape * sp;
sp = new Circle(5);
cout << "The area of the Circle is " << sp->GetArea () << endl;
delete sp;
sp = new Rectangle(4，6);
cout << "The area of the Rectangle is " << sp->GetArea() << endl;
delete sp;
sp = new Square(5);
cout << "The area of the Square is " << sp->GetArea() << endl;
delete sp;
}
```

程序运行输出：

```
The area of the Circle is 78.5
The area of the Rectangle is 24
The area of the Square is 25
```

7-6 定义一个哺乳动物 Mammal 类，再由此派生出狗 Dog 类，定义一个 Dog 类的对象，观察基类与派生类的构造函数与析构函数的调用顺序。

解：

源程序：

```
#include <iostream.h>

enum myColor{ BLACK,    WHITE };

class Mammal
{
public:
// constructors
Mammal();
~Mammal();
//accessors
int GetAge() const { return itsAge; }
void SetAge(int age) { itsAge = age; }
int GetWeight() const { return itsWeight; }
void SetWeight(int weight) { itsWeight = weight; }
//Other methods
void Speak() const { cout << "Mammal sound!\n"; }
protected:
int itsAge;
int itsWeight;
};
class Dog : public Mammal
{
public:
Dog();
~Dog();
myColor GetColor() const { return itsColor; }
void SetColor (myColor color) { itsColor = color; }
void WagTail() { cout << "Tail wagging...\n"; }
private:
myColor itsColor;
};
```

```
Mammal::Mammal():
itsAge(1),
itsWeight(5)
{
cout << "Mammal constructor...\n";
}
Mammal::~Mammal()
{
cout << "Mammal destructor...\n";
}
Dog::Dog(): itsColor (WHITE)
{
cout << "Dog constructor...\n";
}
Dog::~Dog()
{
cout << "Dog destructor...\n";
}
int main()
{
Dog Jack;
Jack.Speak();
Jack.WagTail();
cout << " Jack is " << Jack.GetAge() << " years old\n";
return 0;
}
```

程序运行输出：

```
Mammal constructor...
Dog constructor...
Mammal sound!
Tail wagging...
Fido is 1 years old
Dog destructor...
```

Mammal destructor...

7-7 定义一个基类，构造其派生类，在构造函数中输出提示信息，观察构造函数的执行情况。

解：

```
#include <iostream.h>
class BaseClass
{
public:
BaseClass();
};
BaseClass::BaseClass()
{
cout << "构造基类对象!" << endl;
}
class DerivedClass : public BaseClass
{
public:
DerivedClass();
};
DerivedClass::DerivedClass()
{
cout << "构造派生类对象!" << endl;
}
void main()
{
DerivedClass d;
}
```

程序运行输出：

构造基类对象!

构造派生类对象!

7-8 定义一个 Document 类，有 name 成员变量，从 Document 派生出 Book 类，增加 PageCount 变量。

解：

```
#include <iostream.h>
#include <string.h>
class Document
{
public:
Document(){};
Document( char *name );
char *Name; // Document name.
void PrintNameOf(); // Print name.
};
Document::Document( char *name )
{
Name = new char[ strlen( name ) + 1 ];
strcpy( Name, name );
};
void Document::PrintNameOf()
{
cout << Name << endl;
}
class Book : public Document
{
public:
Book( char *name, long pagecount );
void PrintNameOf();
private:
long PageCount;
```

```
};
Book::Book( char *name, long pagecount ):Document(name)
{
PageCount = pagecount;
}
void Book::PrintNameOf()
{
cout << "Name of book: ";
Document::PrintNameOf();
}
void main()
{
Document a("Document1");
Book b("Book1",100);
b.PrintNameOf();
}
```

程序运行输出：

```
Name of book: Book1
```

7-9 定义基类 Base，有两个共有成员函数 fn1()、fn2()，私有派生出 Derived 类，如果想在 Derived 类的对象中使用基类函数 fn1()，应怎么办?

解：

```
class Base
{
public:
int fn1() const { return 1; }
int fn2() const { return 2; }
};
class Derived : private Base
{
public:
int fn1() { return Base::fn1();};
```

```
int fn2() { return Base::fn2();};
};
void main()
{
Derived a;
a.fn1();
}
```

7-10 定义 object 类，有 weight 属性及相应的操作函数，由此派生出 box 类，增加 Height 和 width 属性及相应的操作函数，声明一个 box 对象，观察构造函数与析构函数的调用顺序。

解：

```
#include <iostream.h>
class object
{
private:
int Weight;
public:
object()
{
cout << "构造 object 对象" << endl;
Weight = 0;
}
int GetWeight(){ return Weight;}
void SetWeight(int n){ Weight = n;}
~object() { cout << "析构 object 对象" << endl;}
};
class box : public object
{
private:
int Height,Width;
public:
box()
```

```
{
cout << "构造 box 对象" << endl;
Height = Width = 0;
}
int GetHeight(){ return Height;}
void SetHeight(int n){ Height = n;}
int GetWidth(){ return Width;}
void SetWidth(int n){ Width = n;}
~box() { cout << "析构 box 对象" << endl;}
};
void main()
{
box a;
}
```

程序运行输出：

```
构造 object 对象
构造 box 对象
析构 box 对象
析构 object 对象
```

7-11 定义一个基类 BaseClass，从它派生出类 DerivedClass，BaseClass 有成员函数 fn1()、fn2()，DerivedClass 也有成员函数 fn1()、fn2()，在主程序中定义一个 DerivedClass 的对象，分别用 DerivedClass 的对象以及 BaseClass 和 DerivedClass 的指针来调用 fn1()、fn2()，观察运行结果。

解：

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

```
class BaseClass
{
public:
void fn1();
void fn2();
};
void BaseClass::fn1()
{
cout << "调用基类的函数 fn1()" << endl;
}
void BaseClass::fn2()
{
cout << "调用基类的函数 fn2()" << endl;
}
class DerivedClass : public BaseClass
{
public:
void fn1();
void fn2();
};
void DerivedClass::fn1()
{
cout << "调用派生类的函数 fn1()" << endl;
}
void DerivedClass::fn2()
{
cout << "调用派生类的函数 fn2()" << endl;
}
void main()
{
DerivedClass aDerivedClass;
```

```
DerivedClass *pDerivedClass = &aDerivedClass;
BaseClass *pBaseClass = &aDerivedClass;
aDerivedClass.fn1();
aDerivedClass.fn2();
pBaseClass->fn1();
pBaseClass->fn2();
pDerivedClass->fn1();
pDerivedClass->fn2();
}
```

程序运行输出：

调用派生类的函数 fn1()

调用派生类的函数 fn2()

调用基类的函数 fn1()

调用基类的函数 fn2()

调用派生类的函数 fn1()

调用派生类的函数 fn2()

7-12 为例 9-1 的吹泡泡程序加一版权（About）对话框。

然后修改例 9-1 的程序，加入以下内容：

程 序：

1．在程序首部加上文件包含命令

```
#include “resource.h”
```

2．在框架窗口类之前加入从 CDialog 类派生的对话框类：

```
// 对话框类
class CAboutDlg: public CDialog
```

```
{
public:
CAboutDlg();
enum {IDD = IDD_DIALOG1};
};
inline CAboutDlg::CAboutDlg():CDialog(CAboutDlg::IDD){}
```

3．在框架窗口类中添加响应鼠标右键消息的代码，包括消息响应函数说明、消息响应宏和消息响应函数定义。鼠标右键消息响应函数为：

```
void CMyWnd::OnRButtonDown(UINT nFlags, CPoint point)
{
CAboutDlg dlg;
dlg.DoModal();
}
```

7-13 签名留念簿程序。该程序模仿签名簿，用户使用鼠标左键点击窗口客户区后会弹出一个对话框，输入姓名后可在鼠标点击位置显示出该签名。签名的颜色、字体大小和方向随机确定。

说 明：项目建立及添加对话框模板资源的方法同例 14-1。修改对话框模板的 ID 为 IDD_NAMEDLG，Caption 为“签名对话框”，并添加一个静态文本控件（Caption 改为“签名”）和一个编辑控件（ID 改为 IDC_EDITNAME）。

程 序：

```
// Example 14-2：签名留念簿程序
#include <afxwin.h>
#include "resource.h"
// 对话框类
class CNameDlg: public CDialog
{
public:
```

```
CPoint m_pointTopLeft;
CString m_strNameEdit;
public:
CNameDlg();
enum {IDD = IDD_NAMEDLG};
protected:
virtual void DoDataExchange(CDataExchange* pDX);
virtual BOOL OnInitDialog();
};
```

// 对话框类的构造函数

```
CNameDlg::CNameDlg():CDialog(CNameDlg::IDD)
{
m_strNameEdit = _T("");
}
```

// 数据交换和数据检验

```
void CNameDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
DDX_Text(pDX, IDC_EDITNAME, m_strNameEdit);
DDV_MaxChars(pDX, m_strNameEdit, 20);
}
```

// 初始化对话框

```
BOOL CNameDlg::OnInitDialog()
{
CDialog::OnInitDialog();
CRect rect;
GetWindowRect(&rect);
rect = CRect(m_pointTopLeft, rect.Size());
MoveWindow(rect);
return TRUE;
}
```

```
// 签名类
class CSignal: public CObject
{
CString m_sName; // 姓名
CPoint m_pointSignal; // 签名位置
int m_nHeight; // 字体高
int m_nColor; // 签名颜色
int m_nEscapement; // 签名倾角
public:
CSignal(){}
void SetValue(CString name,CPoint point,int height,int color,
int escapement);
void ShowSignal(CDC *pDC);
};
// 签名类成员函数
void CSignal::SetValue(CString name,CPoint point,int height,int color,
int escapement)
{
m_sName = name;
m_pointSignal = point;
m_nHeight = height;
m_nColor = color;
m_nEscapement = escapement;
}
// 显示签名
void CSignal::ShowSignal(CDC *pDC)
{
CFont *pOldFont, font;
```

```
font.CreateFont(m_nHeight, 0, m_nEscapement,0, 400, FALSE,FALSE,
0, OEM_CHARSET, OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
DEFAULT_PITCH, "楷体");
pOldFont = pDC->SelectObject(&font);
switch(m_nColor)
{
case 0:
pDC->SetTextColor(RGB(0, 0, 0));
break;
case 1:
pDC->SetTextColor(RGB(255, 0, 0));
break;
case 2:
pDC->SetTextColor(RGB(0, 255, 0));
break;
case 3:
pDC->SetTextColor(RGB(0, 0, 255));
break;
}
pDC->TextOut(m_pointSignal.x, m_pointSignal.y, m_sName);
pDC->SelectObject(pOldFont);
}
// 框架窗口类
#define MAX_NAME 250
class CMyWnd: public CFrameWnd
{
CSignal m_signalList[MAX_NAME];
int m_nCount;
public:
CMyWnd(): m_nCount(0){}
protected:
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

```
afx_msg void OnPaint();
DECLARE_MESSAGE_MAP()
};
// 消息映射
BEGIN_MESSAGE_MAP(CMyWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_PAINT()
END_MESSAGE_MAP()
// 框架窗口类的成员函数
// 鼠标右键消息响应函数
void CMyWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
if(m_nCount < MAX_NAME)
{
CNameDlg dlg;
dlg.m_pointTopLeft = point;
if(dlg.DoModal() == IDOK)
{
int height = rand()%60+12;
int color = rand()%4;
int escapement = (rand()%1200)-600;
CString name = dlg.m_strNameEdit;
m_signalList[m_nCount].SetValue(name,point,height,
color,escapement);
m_nCount++;
Invalidate();
}
}
}
// 绘制框架窗口客户区函数
void CMyWnd::OnPaint()
```

```
{
CPaintDC dc(this);
for(int i=0; i<m_nCount; i++)
m_signalList[i].ShowSignal(&dc);
}
// 应用程序类
class CMyApp: public CWinApp
{
public:
BOOL InitInstance();
};
// 应用程序类的成员函数
BOOL CMyApp::InitInstance()
{
CMyWnd *pFrame = new CMyWnd;
pFrame->Create(0,_T("签字留念簿程序"));
pFrame->ShowWindow(m_nCmdShow);
this->m_pMainWnd = pFrame;
return TRUE;
}
// 全局应用程序对象
CMyApp ThisApp;
```

7-14 将例 14-2 的签名留念簿中的对话框改为无模式对话框。用户可用鼠标右键调出签名对话框，并在不退出该对话框的情况下用鼠标左键将输入的签名显示在窗口客户区。

说 明：在向项目中添加对话框模板资源时，要在其属性对话框的 More Styles 页中选择 Visible 项。其他同例 14-2。

程 序：

该程序中的签名类 CSignal 和应用程序类与上例相同，因此下面仅列出了框架窗口类和对话框类。由于这两个类的成员函数中存在相互引用的情况，所以我们将框架窗口类的声明放在前面，接下来是对话框类的定义，并在框架窗口类之前加入了一条对对话框类的声明。最后是这两个类的成员函数定义。

```
// 框架窗口类
#define MAX_NAME 250
class CNameDlg;
class CMyWnd: public CFrameWnd
{
CSignal m_signalList[MAX_NAME];
int m_nCount;
CNameDlg *m_pNameDlg;
public:
CMyWnd();
~CMyWnd();
protected:
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
afx_msg void OnPaint();
DECLARE_MESSAGE_MAP()
};
// 对话框类
class CNameDlg: public CDialog
{
public:
BOOL m_bActive;
CString m_strNameEdit;
enum {IDD = IDD_NAMEDLG};
CNameDlg();
BOOL Create();
```

```
protected:
virtual void DoDataExchange(CDataExchange* pDX);
virtual BOOL OnInitDialog();
virtual void OnOK();
virtual void OnCancel();
};
// 框架窗口类的消息映射
BEGIN_MESSAGE_MAP(CMyWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_RBUTTONDOWN()
ON_WM_PAINT()
END_MESSAGE_MAP()
// 框架窗口类的成员函数
// 框架窗口类的构造函数
CMyWnd::CMyWnd()
{
m_nCount = 0;
m_pNameDlg = new CNameDlg;
}
// 框架窗口类的析构函数
CMyWnd::~CMyWnd()
{
delete m_pNameDlg;
}
// 鼠标右键消息响应函数
void CMyWnd::OnRButtonDown(UINT nFlags, CPoint point)
{
if(m_pNameDlg->m_bActive)
m_pNameDlg->SetActiveWindow(); // 激活对话框
else
```

```
m_pNameDlg->Create(); // 显示对话框
}
// 鼠标左键消息响应函数
void CMyWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
if(m_nCount < MAX_NAME)
{
int height = rand()%60+12;
int color = rand()%4;
int escapement = (rand()%1200)-600;
CString name = m_pNameDlg->m_strNameEdit;
m_signalList[m_nCount].SetValue(name, point, height, color,
escapement);
m_nCount++;
Invalidate();
}
}
// 绘制框架窗口客户区函数
void CMyWnd::OnPaint()
{
CPaintDC dc(this);
for(int i=0; i<m_nCount; i++)
m_signalList[i].ShowSignal(&dc);
}
// 对话框类的成员函数
// 对话框类的构造函数
CNameDlg::CNameDlg():CDialog(CNameDlg::IDD)
{
m_bActive = FALSE;
m_strNameEdit = _T("");
```

```
}
// 数据交换
void CNameDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
DDX_Text(pDX, IDC_EDIT1, m_strNameEdit);
}
// 初始化对话框
BOOL CNameDlg::OnInitDialog()
{
CDialog::OnInitDialog();
CRect rect;
GetWindowRect(&rect);
MoveWindow(0, 0, rect.Width(), rect.Height());
return TRUE;
}
// 显示无模态对话框
BOOL CNameDlg::Create()
{
m_bActive = TRUE;
return CDialog::Create(CNameDlg::IDD);
}
// 退出对话框
void CNameDlg::OnCancel()
{
m_bActive = FALSE;
DestroyWindow();
}
// 更新数据
void CNameDlg::OnOK()
```

```
{
UpdateData(TRUE);
}
```

7-15 为例 14-2 的签名程序加上字体选择对话框。

说 明：本程序使用字体选择公用对话框（通过鼠标右键调出）选择签名的字体、字号和颜色等参数，在签名对话框中要输入姓名和签名与 X 轴的倾斜角。建立项目的方法与例 14-2 相似，只是要在签名对话框模板中再添加一个编辑控件用于输入签名的倾斜角，其标识符为 IDD_EDIT2。

程 序：

```
// Example 14-4：签名留念簿程序
#include <afxwin.h>
#include <afxdlgs.h>
#include <string.h>
#include "resource.h"
// 对话框类
class CNameDlg: public CDialog
{
public:
CPoint m_pointTopLeft; // 对话框位置
CString m_strNameEdit; // 签名
LONG m_lEscapement; // 签名倾角
public:
CNameDlg();
enum {IDD = IDD_NAMEDLG};
protected:
```

```
virtual void DoDataExchange(CDataExchange* pDX);
virtual BOOL OnInitDialog();
};
// 对话框类的构造函数
CNameDlg::CNameDlg():CDialog(CNameDlg::IDD), m_pointTopLeft(0, 0)
{
m_strNameEdit = _T("");
m_lEscapement = 0;
}
// 数据交换和数据检验
void CNameDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
DDX_Text(pDX, IDC_EDIT1, m_strNameEdit);
DDX_Text(pDX, IDC_EDIT2, m_lEscapement);
DDV_MaxChars(pDX, m_strNameEdit, 20);
DDV_MinMaxLong(pDX, m_lEscapement, -600, 600);
}
// 初始化对话框
BOOL CNameDlg::OnInitDialog()
{
CDialog::OnInitDialog();
CRect rect;
GetWindowRect(&rect);
rect = CRect(m_pointTopLeft, rect.Size());
MoveWindow(rect);
return TRUE;
}
// 签名类
class CSignal: public CObject
{
```

```
CString m_strSignal; // 姓名

COLORREF m_colorSignal; // 签名颜色

CPoint m_pointSignal; // 签名位置

LOGFONT m_fontSignal; // 签名字体

public:
CSignal(){}
void SetValue(CString signal, CPoint point, COLORREF color,
LONG escapement, LOGFONT *pfont);
void ShowSignal(CDC *pDC);
};

// 签名类成员函数

void CSignal::SetValue(CString signal, CPoint point, COLORREF color,
int escapement, LOGFONT *pfont)
{
m_strSignal = signal;
m_pointSignal = point;
m_colorSignal = color;
memcpy(&m_fontSignal, pfont, sizeof(LOGFONT));
m_fontSignal.lfEscapement = escapement;
}

// 显示签名

void CSignal::ShowSignal(CDC *pDC)
{
CFont font, *pOldFont;
font.CreateFontIndirect(&m_fontSignal);
pOldFont = pDC->SelectObject(&font);
pDC->SetTextColor(m_colorSignal);
pDC->TextOut(m_pointSignal.x, m_pointSignal.y, m_strSignal);
pDC->SelectObject(pOldFont);
}
```

```
// 框架窗口类

#define MAX_NAME 250
class CMyWnd: public CFrameWnd
{

CSignal m_signalList[MAX_NAME]; // 签名数组

int m_nCount; // 签名数量

LOGFONT m_fontSignal; // 签名字体

COLORREF m_colorSignal; // 签名颜色

public:
CMyWnd();
protected:
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
afx_msg void OnPaint();
DECLARE_MESSAGE_MAP()
};

// 消息映射

BEGIN_MESSAGE_MAP(CMyWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_RBUTTONDOWN()
ON_WM_PAINT()
END_MESSAGE_MAP()

// 框架窗口类的成员函数

CMyWnd::CMyWnd()
{
m_nCount = 0;
m_colorSignal = RGB(0, 0, 0);
m_fontSignal.lfHeight = 40;
m_fontSignal.lfWidth = 0;
```

```
m_fontSignal.lfEscapement = 0;
m_fontSignal.lfOrientation = 0;
m_fontSignal.lfWeight = 400;
m_fontSignal.lfItalic = FALSE;
m_fontSignal.lfUnderline = FALSE;
m_fontSignal.lfStrikeOut = 0;
m_fontSignal.lfCharSet = OEM_CHARSET;
m_fontSignal.lfOutPrecision = OUT_DEFAULT_PRECIS;
m_fontSignal.lfClipPrecision = CLIP_DEFAULT_PRECIS;
m_fontSignal.lfQuality = DEFAULT_QUALITY;
m_fontSignal.lfPitchAndFamily = DEFAULT_PITCH;
strcpy(m_fontSignal.lfFaceName, "Arial");
}
// 鼠标右键消息响应函数
void CMyWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
if(m_nCount < MAX_NAME)
{
CNameDlg dlg;
dlg.m_pointTopLeft = point;
if(dlg.DoModal() == IDOK)
{
LONG escapement = dlg.m_lEscapement;
CString name = dlg.m_strNameEdit;
m_signalList[m_nCount].SetValue(name, point, m_colorSignal,
escapement, &m_fontSignal);
m_nCount++;
Invalidate();
}
}
}
// 鼠标右键消息响应函数
void CMyWnd::OnRButtonDown(UINT nFlags, CPoint point)
```

```
{
CFontDialog dlg(&m_fontSignal);
if(dlg.DoModal() == IDOK)
{
dlg.GetCurrentFont(&m_fontSignal);
m_colorSignal = dlg.GetColor();
}
}
```

// 绘制框架窗口客户区函数

```
void CMyWnd::OnPaint()
{
CPaintDC dc(this);
for(int i=0; i<m_nCount; i++)
m_signalList[i].ShowSignal(&dc);
}
```

// 应用程序类

```
class CMyApp: public CWinApp
{
public:
BOOL InitInstance();
};
```

// 应用程序类的成员函数

```
BOOL CMyApp::InitInstance()
{
CMyWnd *pFrame = new CMyWnd;
pFrame->Create(0,_T("签字留念簿程序"));
pFrame->ShowWindow(SW_SHOWMAXIMIZED);
this->m_pMainWnd = pFrame;
return TRUE;
}
```

// 全局应用程序对象

CMyApp ThisApp;

7-16 为例 9-3 的吹泡泡程序添加颜色选择对话框，使其可以绘出五颜六色的泡泡。

程 序：在例 9-3 的程序基础上作如下修改：

1．在程序首部添加文件包含命令：

#include <afxdlgs.h>

2．在框架窗口类声明中添加一个 COLORREF 类型的数组，存放各泡泡的颜色：

COLORREF m_colorBubble [MAX_BUBBLE];

3．修改鼠标左键消息映射函数，添加使用颜色选择公用对话框的代码：

```
void CMyWnd::OnLButtonDown ( UINT nFlags, CPoint point )
{
if(m_nBubbleCount < MAX_BUBBLE)
{
m_colorBubble[m_nBubbleCount] = RGB(200, 200, 200);
CColorDialog dlg(m_colorBubble[m_nBubbleCount]);
if(dlg.DoModal() == IDOK)
m_colorBubble[m_nBubbleCount] = dlg.GetColor();
int r = rand()%50+10;
CRect rect(point.x-r, point.y-r, point.x+r, point.y+r);
m_rectBubble[m_nBubbleCount] = rect;
m_nBubbleCount++;
InvalidateRect(rect, FALSE);
}
}
```

4．修改 OnPaint（）成员函数，添加根据泡泡颜色使用画刷的代码：

```
void CMyWnd::OnPaint()
{
CPaintDC dc(this);
CBrush brushNew, *pbrushOld;
for(int i=0; i<m_nBubbleCount; i++)
```

```
{
brushNew.CreateSolidBrush(m_colorBubble[i]);
pbrushOld = dc.SelectObject(&brushNew);
dc.Ellipse(m_rectBubble[i]);
dc.SelectObject(pbrushOld);
brushNew.DeleteObject();
}
}
```

7-17 序列化。如果例 12-1 的吹泡泡程序使用一般的数组存放泡泡数据（参看例 9-1 的程序)：

```
CRect m_rectBubble[MAX_BUBBLE];
int m_nBubbleCount;
```

为其文档类重新设计 Serialize（）函数。

说 明：按例 12-1 的方法建立项目和输入源代码，但将文档类中的泡泡数据改为以上两行的形式。修改文档类的 Serialize（）函数，代码如下。

程 序：

```
// 序列化函数
void CMyDoc::Serialze(CArchive& ar)
{
if(ar.IsStoring())
{
ar << m_nBubbleCount;
for(int i=0; i<m_nBubbleCount; i++)
ar << m_rectBubble[i];
}
else
{
ar >> m_nBubbleCount;
for(int i=0; i<m_nBubbleCount; i++)
```

```
ar >> m_rectBubble[i];
}
}
```

7-18 修改例 12-1 的程序并观察其打印结果。

程 序:

在例 12-1 程序的视图类 CMyView 类的成员函数 OnDraw（）中，添加代码沿窗口客户区轮廓画一矩形:

```
void CMyView::OnDraw(CDC* pDC)
{
CRect rect;
GetClientRect(&rect);
pDC->Rectangle(rect);
CMyDoc* pDoc = GetDocument(); // 取文档指针
ASSERT_VALID(pDoc);
pDC->SelectStockObject(LTGRAY_BRUSH); // 在视图上显示文档数据
for(int i=0; i<pDoc->GetListSize(); i++)
pDC->Ellipse(pDoc->GetBubble(i));
}
```

7-19 改进吹泡泡程序，使之打印输出与屏幕显示的比例相近。

程 序:

在例 12-1 基础上修改。首先在 CMyView 类中重载虚函数 OnPrepareDC（)。在 CMyView 类的声明中增加一行:

```
virtual void OnPrepareDC(CDC *pDC, CPrintInfo *pInfo=NULL);
```

然后添加该函数的定义，设置映射模式为 MM_LOMETRIC:

```
// 设置映射模式
```

```
void CMyView::OnPrepareDC(CDC *pDC, CPrintInfo *pInfo)
{
pDC->SetMapMode(MM_LOMETRIC);
CView::OnPrepareDC(pDC, pInfo);
}
```

然后修改消息映射函数 OnLButtonDown（），将物理坐标转换为逻辑坐标：

```
// 响应点击鼠标左键消息
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
CMyDoc* pDoc = GetDocument(); // 取文档指针
ASSERT_VALID(pDoc);
CClientDC dc(this); // 设置设备环境
OnPrepareDC(&dc);
int r = rand()%50+5; // 生成泡泡
CRect rect(point.x-r, point.y-r, point.x+r, point.y+r);
InvalidateRect(rect, FALSE); // 更新视图
dc.DPtoLP(rect); // 转换物理坐标为逻辑坐标
pDoc->AddBubble(rect); // 修改文档数据
pDoc->SetModifiedFlag(); // 设置修改标志
}
```

7-20 声明一个 Person 类，并使之支持序列化。

程 序：

```
class CPerson: public CObject
{
DECLARE_SERIAL( CPerson)
```

```
LONG m_IDnumber; // 身份证号码

CString m_strName; // 姓名

CString m_strNation; // 民族

int m_nSex; // 性别

int m_nAge; // 年龄

BOOL m_bMarried; // 婚否

public:
CEmployee(){};
CPerson& operator = (CPerson& person);
void Serialize(CArchive& ar);
};
IMPLEMENT_SERIAL( CPerson, CObject, 1 )
CPerson& CPerson::operator = (CPerson& person)
{
m_IDnumber = person.m_IDnumber;
m_strName = person.m_strName;
m_strNation = person.m_strNation;
m_nSex = person.m_nSex;
m_nAge = person.m_nAge;
m_bMarried = person.m_bMarried;
return *this;
}
void CPerson::Serialize(CArchive& ar)
{

CObject::Serialize( ar); // 首先调用基类的 Serialize()方法

if(ar.IsStoring())
{
ar << m_IDnumber;
ar << m_strName;
```

```
ar << m_strNation;
ar << m_nSex;
ar << m_nAge;
ar << (int)m_bMarried;
}
else
{
ar >> m_IDnumber;
ar >> m_strName;
ar >> m_strNation;
ar >> m_nSex;
ar >> m_nAge;
ar >> (int)m_bMarried;
}
}
```

7-21 修改例 13-3 的吹泡泡程序，使其打印每个泡泡的数据值。打印格式为每页 40 行，页眉为文档名，页脚为页号。

说 明：首先为视图类添加一个数据成员 m_nLinePerPage，用来存放每页行数，并在视图类 CMyView 的构造函数中将 m_nLinePerPage 初始化为 40。

修改视图类成员函数 OnPrepareDC（），设置映射模式为 MM_TWIPS。该模式为每英寸 1440 点，很适合打印机输出。

程 序：

重载视图类的成员函数 OnPreparePrinting（），在其中添加计算打印页数的代码：

```
BOOL CMyView::OnPreparePrinting(CPrintInfo* pInfo)
{
CMyDoc *pDoc = GetDocument();
int nPageCount = pDoc->GetListSize()/m_nLinePerPage;
if(pDoc->GetListSize() % m_nLinePerPage)
nPageCount ++;
```

```
pInfo->SetMaxPage(nPageCount);
return DoPreparePrinting(pInfo);
}
```

最后重载视图类的 OnPrint（）函数并添加打印代码：

```
void CMyView::OnPrint( CDC* pDC, CPrintInfo* pInfo )
{
int nPage = pInfo->m_nCurPage; // 当前页号
int nStart = (nPage-1)*m_nLinePerPage; // 本页第一行
int nEnd = nStart+m_nLinePerPage; // 本页最后一行
CFont font; // 设置字体
font.CreateFont(-280, 0, 0, 0, 400, FALSE, FALSE,
0, ANSI_CHARSET, OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
DEFAULT_PITCH|FF_MODERN, "Courier New");
CFont *pOldFont = (CFont *)(pDC->SelectObject(&font));
CRect rectPaper = pInfo->m_rectDraw; // 取页面打印矩形
// 页眉: 页面顶端中央打印文档名称
CMyDoc *pDoc = GetDocument();
ASSERT_VALID(pDoc);
CString str;
str.Format("Bubble Report: %s", (LPCSTR)pDoc->GetTitle());
CSize sizeText = pDC->GetTextExtent(str);
CPoint point((rectPaper.Width()-sizeText.cx)/2, 0);
pDC->TextOut(point.x, point.y, str);
point.x = rectPaper.left; // 打印页眉下划线
point.y = rectPaper.top-sizeText.cy;
pDC->MoveTo(point);
point.x = rectPaper.right;
```

```
pDC->LineTo(point);

// 打印表头

str.Format("%6.6s %6.6s %6.6s %6.6s %6.6s",
"Index", "Left", "Top", "Right", "Bottom");
point.x = 720;
point.y -= 720;
pDC->TextOut(point.x, point.y, str);

TEXTMETRIC tm; // 取当前字体有关信息

pDC->GetTextMetrics(&tm);
int nHeight = tm.tmHeight+tm.tmExternalLeading;

point.y -= 360; // 下移 1/4 英寸

for(int i=nStart; i<nEnd; i++) // 打印表体

{
if(i >= pDoc->GetListSize())
break;
str.Format("%6d %6d %6d %6d %6d", i+1,
pDoc->GetBubble(i).left,
pDoc->GetBubble(i).top,
pDoc->GetBubble(i).right,
pDoc->GetBubble(i).bottom);
point.y -= nHeight;
pDC->TextOut(point.x, point.y, str);
}

// 在页面底部中央打印页号

str.Format("- %d -", nPage);
sizeText = pDC->GetTextExtent(str);
point.x = (rectPaper.Width()-sizeText.cx)/2;
point.y = rectPaper.Height()+sizeText.cy;
pDC->TextOut(point.x, point.y, str);

// 释放字体对象
```

```
pDC->SelectObject(pOldFont);
}
```

7-22 修改例 11-4 的拼图程序，使之在难度菜单的相应选项前打钩。

程 序:

首先在框架窗口类的消息响应函数声明处增加以下消息响应函数的声明:

```
afx_msg void CPuzzleWnd::OnUpdateGrad01(CCmdUI* pCmdUI);
afx_msg void CPuzzleWnd::OnUpdateGrad02(CCmdUI* pCmdUI);
afx_msg void CPuzzleWnd::OnUpdateGrad03(CCmdUI* pCmdUI);
```

然后在框架窗口类的消息映射宏中加入相应内容:

```
BEGIN_MESSAGE_MAP(CPuzzleWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_LBUTTONUP()
ON_WM_MOUSEMOVE()
ON_WM_PAINT()
ON_COMMAND(ID_SHOWFIG, OnShowFig)
ON_COMMAND(ID_GRAD01, OnGrad01)
ON_COMMAND(ID_GRAD02, OnGrad02)
ON_COMMAND(ID_GRAD03, OnGrad03)
ON_UPDATE_COMMAND_UI(ID_GRAD01, OnUpdateGrad01)
ON_UPDATE_COMMAND_UI(ID_GRAD02, OnUpdateGrad02)
ON_UPDATE_COMMAND_UI(ID_GRAD03, OnUpdateGrad03)
END_MESSAGE_MAP()
```

注意更新命令用户接口消息映射宏将菜单标识符与相应的消息映射函数联系在一起。最后编写相应的更新命令用户接口消息映射函数:

```
void CPuzzleWnd::OnUpdateGrad01(CCmdUI* pCmdUI)
{
pCmdUI->SetCheck(m_nColCount == 4);
}
void CPuzzleWnd::OnUpdateGrad02(CCmdUI* pCmdUI)
```

```
{
pCmdUI->SetCheck(m_nColCount == 8);
}
void CPuzzleWnd::OnUpdateGrad03(CCmdUI* pCmdUI)
{
pCmdUI->SetCheck(m_nColCount == 16);
}
```

输入输出：在选择拼图难度时，可在相应选项前打钩（图 13-4）。

```
// Example 13-7: 七巧板程序 ////////////////////////////////
#include <afxwin.h>
#include <afxext.h>
// 拼板类 ////////////////////////////////////////////////
#define MAX_POINTS 4
#define CHIP_WIDTH 240
#define DELTA 30
class CChip : public CObject
{
DECLARE_SERIAL(CChip)
int m_nType;
CPoint m_pointList[MAX_POINTS];
int m_nPointCount;
public:
CChip(){}
void SetChip(int type, POINT *ppointlist, int count);
void DrawChip(CDC *pDC);
BOOL PtInChip(POINT point);
LPCRECT GetRect();
void MoveTo(CSize offset);
void Rotation();
void Serialize(CArchive &ar);
};
IMPLEMENT_SERIAL(CChip, CObject, 1)
```

// 设置拼图块参数

```
void CChip::SetChip(int type, POINT *ppointlist, int count)
{
m_nType = type;
m_nPointCount = count;
for(int i=0; i<count; i++)
m_pointList[i] = ppointlist[i];
}
```

// 绘出拼图块

```
void CChip::DrawChip(CDC *pDC)
{
CPen penNew, *ppenOld;
CBrush brushNew, *pbrushOld;
switch(m_nType)
{
case 1:
brushNew.CreateSolidBrush(RGB(127, 127, 127));
break;
case 2:
brushNew.CreateSolidBrush(RGB(255, 0, 0));
break;
case 3:
brushNew.CreateSolidBrush(RGB(0, 255, 0));
break;
case 4:
brushNew.CreateSolidBrush(RGB(0, 0, 255));
break;
case 5:
brushNew.CreateSolidBrush(RGB(127, 127, 0));
break;
case 6:
brushNew.CreateSolidBrush(RGB(127, 0, 127));
break;
```

```
case 7:
brushNew.CreateSolidBrush(RGB(0, 127, 127));
break;
}
penNew.CreatePen(PS_SOLID, 1, RGB(0, 0, 0));
ppenOld = pDC->SelectObject(&penNew);
pbrushOld = pDC->SelectObject(&brushNew);
pDC->Polygon(m_pointList, m_nPointCount);
pDC->SelectObject(ppenOld);
pDC->SelectObject(pbrushOld);
}
// 检测一点是否在拼图块中
BOOL CChip::PtInChip(POINT point)
{
CRgn rgn;
rgn.CreatePolygonRgn(m_pointList, m_nPointCount, 0);
return rgn.PtInRegion(point);
}
// 取拼图块的包含矩形
LPCRECT CChip::GetRect()
{
static RECT rect;
CRgn rgn;
rgn.CreatePolygonRgn(m_pointList, m_nPointCount, 0);
rgn.GetRgnBox(&rect);
rect.right++;
rect.bottom++;
return &rect;
}
// 旋转拼图块
void CChip::Rotation()
{
```

```
CRect rect;
CRgn rgn;
rgn.CreatePolygonRgn(m_pointList, m_nPointCount, 0);
rgn.GetRgnBox(&rect);

double x = rect.left+rect.Width()/2; // 计算旋转中心

double y = rect.top+rect.Height()/2;
double dx, dy;

for(int i=0; i<m_nPointCount; i++) // 旋转各点

{
dx = m_pointList[i].x-x;
dy = m_pointList[i].y-y;
m_pointList[i].x = (int)(x+dx*0.7071-dy*0.7071);
m_pointList[i].y = (int)(y+dx*0.7071+dy*0.7071);
}
}
```

// 移动拼图块

```
void CChip::MoveTo(CSize offset)
{
for(int i=0; i<m_nPointCount; i++)
m_pointList[i] = m_pointList[i]+offset;
}
```

// 序列化

```
void CChip::Serialize(CArchive &ar)
{
if(ar.IsStoring())
{
ar << m_nType;
ar << m_nPointCount;
for(int i=0; i<m_nPointCount; i++)
ar << m_pointList[i];
}
```

```
else
{
ar >> m_nType;
ar >> m_nPointCount;
for(int i=0; i<m_nPointCount; i++)
ar >> m_pointList[i];
}
}
// 文档类 ////////////////////////////////////////////////
#define CHIP_COUNT 7
class CMyDoc : public CDocument
{
DECLARE_DYNCREATE(CMyDoc)
CChip m_chipList[CHIP_COUNT];
public:
void Reset();
virtual void DeleteContents();
virtual void Serialize(CArchive& ar);
};
IMPLEMENT_DYNCREATE(CMyDoc, CDocument)
// 初始化拼图块
void CMyDoc::Reset()
{
POINT pointList[MAX_POINTS];
pointList[0].x = DELTA;
pointList[0].y = DELTA;
pointList[1].x = DELTA+CHIP_WIDTH;
pointList[1].y = DELTA;
pointList[2].x = DELTA+CHIP_WIDTH/2;
pointList[2].y = DELTA+CHIP_WIDTH/2;
m_chipList[0].SetChip(1, pointList, 3);
pointList[0].x = DELTA;
pointList[0].y = DELTA;
```

```
pointList[1].x = DELTA;
pointList[1].y = DELTA+CHIP_WIDTH;
pointList[2].x = DELTA+CHIP_WIDTH/2;
pointList[2].y = DELTA+CHIP_WIDTH/2;
m_chipList[1].SetChip(2, pointList, 3);
pointList[0].x = DELTA+CHIP_WIDTH;
pointList[0].y = DELTA;
pointList[1].x = DELTA+CHIP_WIDTH;
pointList[1].y = DELTA+CHIP_WIDTH/2;
pointList[2].x = DELTA+(CHIP_WIDTH*3)/4;
pointList[2].y = DELTA+CHIP_WIDTH/4;
m_chipList[2].SetChip(3, pointList, 3);
pointList[0].x = DELTA+CHIP_WIDTH/2;
pointList[0].y = DELTA+CHIP_WIDTH/2;
pointList[1].x = DELTA+CHIP_WIDTH/4;
pointList[1].y = DELTA+(CHIP_WIDTH*3)/4;
pointList[2].x = DELTA+(CHIP_WIDTH*3)/4;
pointList[2].y = DELTA+(CHIP_WIDTH*3)/4;
m_chipList[3].SetChip(4, pointList, 3);
pointList[0].x = DELTA+CHIP_WIDTH;
pointList[0].y = DELTA+CHIP_WIDTH/2;
pointList[1].x = DELTA+CHIP_WIDTH;
pointList[1].y = DELTA+CHIP_WIDTH;
pointList[2].x = DELTA+CHIP_WIDTH/2;
pointList[2].y = DELTA+CHIP_WIDTH;
m_chipList[4].SetChip(5, pointList, 3);
pointList[0].x = DELTA+(CHIP_WIDTH*3)/4;
pointList[0].y = DELTA+CHIP_WIDTH/4;
pointList[1].x = DELTA+CHIP_WIDTH/2;
pointList[1].y = DELTA+CHIP_WIDTH/2;
pointList[2].x = DELTA+(CHIP_WIDTH*3)/4;
pointList[2].y = DELTA+(CHIP_WIDTH*3)/4;
pointList[3].x = DELTA+CHIP_WIDTH;
pointList[3].y = DELTA+CHIP_WIDTH/2;
```

```
m_chipList[5].SetChip(6, pointList, 4);
pointList[0].x = DELTA;
pointList[0].y = DELTA+CHIP_WIDTH;
pointList[1].x = DELTA+CHIP_WIDTH/4;
pointList[1].y = DELTA+(CHIP_WIDTH*3)/4;
pointList[2].x = DELTA+(CHIP_WIDTH*3)/4;
pointList[2].y = DELTA+(CHIP_WIDTH*3)/4;
pointLis ¨t[3].x = DELTA+CHIP_WIDTH/2; pointList[3].y = DELTA+CHIP_WIDTH;
m_chipList[6].SetChip(7, pointList, 4); // 清理文档：关闭文档、建立新文档和打开文档前调用
void CMyDoc::DeleteContents()
{
Reset();
CDocument::DeleteContents();
}
// 系列化：读写文档时自动调用
void CMyDoc::Serialize(CArchive &ar)
{
for(int i=0; i<CHIP_COUNT; i++)
m_chipList[i].Serialize(ar);
}
// 视图类 ////////////////////////////////////////////////
class CMyView : public CView
{
DECLARE_DYNCREATE(CMyView)
BOOL m_bCaptured;
CPoint m_pointMouse;
int m_nCurrIndex;
public:
CMyView(){m_bCaptured = FALSE;}
CMyDoc* GetDocument(){return (CMyDoc*)m_pDocument;}
virtual void OnInitialUpdate();
```

```
virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
virtual void OnDraw(CDC* pDC);
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
DECLARE_MESSAGE_MAP()
};
IMPLEMENT_DYNCREATE(CMyView, CView)
BEGIN_MESSAGE_MAP(CMyView, CView)
ON_WM_LBUTTONDOWN()
ON_WM_LBUTTONUP()
ON_WM_MOUSEMOVE()
ON_WM_RBUTTONDOWN()
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
// 更新初始化：当建立新文档或打开文档时调用
void CMyView::OnInitialUpdate()
{
CView::OnInitialUpdate();
Invalidate();
}
// 绘制视图：程序开始运行或窗体发生变化时自动调用
void CMyView::OnDraw(CDC* pDC)
{
CMyDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
for(int i=0; i<CHIP_COUNT; i++)
pDoc->m_chipList[i].DrawChip(pDC);
}
```

```
// 消息响应：用户点击鼠标左键时调用
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
CMyDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
for(int i=CHIP_COUNT-1; i>=0; i--)
if(pDoc->m_chipList[i].PtInChip(point))
{
SetCapture();
m_bCaptured = TRUE;
m_pointMouse = point;
m_nCurrIndex = i;
break;
}
}
// 释放鼠标左键
void CMyView::OnLButtonUp(UINT nFlags, CPoint point)
{
if(m_bCaptured)
{
::ReleaseCapture();
m_bCaptured = FALSE;
}
}
// 移动鼠标左键
void CMyView::OnMouseMove(UINT nFlags, CPoint point)
{
if(m_bCaptured)
{
CMyDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
InvalidateRect(pDoc->m_chipList[m_nCurrIndex].GetRect());
```

```
CSize offset(point-m_pointMouse);
pDoc->m_chipList[m_nCurrIndex].MoveTo(offset);
InvalidateRect(pDoc->m_chipList[m_nCurrIndex].GetRect());
m_pointMouse = point;
pDoc->SetModifiedFlag();
}
}
// 按下鼠标右键: 旋转拼图块
void CMyView::OnRButtonDown(UINT nFlags, CPoint point)
{
CMyDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
for(int i=CHIP_COUNT-1; i>=0; i--)
if(pDoc->m_chipList[i].PtInChip(point))
{
InvalidateRect(pDoc->m_chipList[i].GetRect());
pDoc->m_chipList[i].Rotation();
InvalidateRect(pDoc->m_chipList[i].GetRect(), FALSE);
pDoc->SetModifiedFlag();
break;
}
}
// 准备打印：设置打印参数
BOOL CMyView::OnPreparePrinting(CPrintInfo* pInfo)
{
pInfo->SetMaxPage(1);
return DoPreparePrinting(pInfo);
}
// 主框架类 ////////////////////////////////////////////////
class CMainFrame : public CFrameWnd
{
DECLARE_DYNCREATE(CMainFrame)
```

```
};
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

// 应用程序类 ////////////////////////////////////////////

#define IDR_MAINFRAME 128 // 主框架的资源代号

class CMyApp : public CWinApp
{
public:
virtual BOOL InitInstance();
DECLARE_MESSAGE_MAP()
};
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

// 初始化程序实例：建立并登记文档模板

BOOL CMyApp::InitInstance()
{
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
IDR_MAINFRAME,
RUNTIME_CLASS(CMyDoc),
RUNTIME_CLASS(CMainFrame),
RUNTIME_CLASS(CMyView));
AddDocTemplate(pDocTemplate);
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
if (!ProcessShellCommand(cmdInfo))
return FALSE;
m_pMainWnd->ShowWindow(SW_SHOWMAXIMIZED);
return TRUE;
}
```

```
// 全局应用程序对象
CMyApp theApp;
```

7-23 为例 9-3 的吹泡泡程序添加一标识符为 IDI_MAINICON 的图标(该图标应已按 11.8：“向项目中添加资源”中的方法建立并加入项目)。

说 明：建立项目的方法见 9.8：“用 Visual C++集成开发环境开发 Win32 应用程序”。

程 序：

在例 9-3 程序前面添加一文件包含命令：

```
#include ”resource.h”
```

并将 CMyApp::InitInstance（）函数修改为：

```
BOOL CMyApp::InitInstance()
{
HICON hIcon;
hIcon = LoadIcon(IDI_MAINICON); // 载入图标
CMyWnd *pFrame = new CMyWnd;
pFrame->Create(0,_T("吹泡泡程序"));
pFrame->SetIcon(hIcon, TRUE); // 设置大图标
pFrame->SetIcon(hIcon, FALSE); // 设置小图标
pFrame->ShowWindow(m_nCmdShow);
this->m_pMainWnd = pFrame;
return TRUE;
}
```

7-24 显示一张位图文件（.BMP)。

说 明：首先建立 Win32 Application 空白项目和源代码文件（不要忘记设置项目使之可以使用 MFC 类库)，然后按 11.8：“向项目中添加资源”的方法为项目建立资源文件，并将

待显示的位图文件作为资源装入项目。

程 序:

```
// Example 11-2：显示 BMP 图片
#include <afxwin.h>
#include "resource.h"
// 框架窗口类
class CMyWnd: public CFrameWnd
{
CBitmap m_Bitmap;
int m_nHeight;
int m_nWidth;
public:
CMyWnd();
protected:
afx_msg void OnPaint();
DECLARE_MESSAGE_MAP()
};
// 消息映射
BEGIN_MESSAGE_MAP(CMyWnd, CFrameWnd)
ON_WM_PAINT()
END_MESSAGE_MAP()
// 框架窗口类的成员函数
CMyWnd::CMyWnd()
{
m_Bitmap.LoadBitmap(IDB_BITMAP1);
BITMAP BM;
m_Bitmap.GetBitmap(&BM);
m_nWidth = BM.bmWidth;
m_nHeight = BM.bmHeight;
}
```

```
// 响应绘制窗口客户区消息
void CMyWnd::OnPaint()
{
CPaintDC dc(this);
CDC MemDC;
MemDC.CreateCompatibleDC(NULL);
MemDC.SelectObject(&m_Bitmap);
dc.BitBlt(0,0,m_nWidth,m_nHeight,&MemDC,0,0,SRCCOPY);
}
// 应用程序类
class CMinMFCApp: public CWinApp
{
public:
BOOL InitInstance();
};
// 应用程序类的成员函数
// 初始化应用程序实例
BOOL CMinMFCApp::InitInstance()
{
CMyWnd *pFrame = new CMyWnd;
pFrame->Create(0,_T("Beautiful Cats”));
pFrame->ShowWindow(m_nCmdShow);
this->m_pMainWnd = pFrame;
return TRUE;
}
// 全局应用程序对象
CMinMFCApp ThisApp;
```

7-25 修改 11-2，使之可以不同比例放大或缩小图象。

说 明：使用 StretchBlt（）函数代替 BitBlt（）函数就可实现图象的缩放显示。在项目中加入一个弹出式菜单（将标识符改为 IDR_MAINMENU），内含 3 个菜单选项：缩小 1 倍显示，按原尺寸显示和放大 1 倍显示，其标识符分别改为 ID_SHRINK，ID_BESTFIT 和 ID_ZOOMOUT。

程 序：

```
// Example 11-3：以不同尺寸显示 BMP 图片
#include <afxwin.h>
#include "resource.h"
// 框架窗口类
class CMyWnd: public CFrameWnd
{
CBitmap m_Bitmap;
float m_fTimes;
int m_nHeight;
int m_nWidth;
public:
CMyWnd();
BOOL PreCreateWindow(CREATESTRUCT &cs);
protected:
afx_msg void OnPaint();
afx_msg void OnShrink();
afx_msg void OnBestFit();
afx_msg void OnZoomOut();
DECLARE_MESSAGE_MAP()
};
// 消息映射
BEGIN_MESSAGE_MAP(CMyWnd, CFrameWnd)
ON_WM_PAINT()
ON_COMMAND(ID_SHRINK, OnShrink)
```

```
ON_COMMAND(ID_BESTFIT, OnBestFit)
ON_COMMAND(ID_ZOOMOUT, OnZoomOut)
END_MESSAGE_MAP()
// 主窗口类的成员函数
CMyWnd::CMyWnd()
{
BITMAP BM;
m_Bitmap.LoadBitmap(IDB_BITMAP1);
m_Bitmap.GetBitmap(&BM);
m_nWidth = BM.bmWidth;
m_nHeight = BM.bmHeight;
m_fTimes = 1.0;
}
// 装入菜单
BOOL CMyWnd::PreCreateWindow(CREATESTRUCT &cs)
{
cs.hMenu = LoadMenu(NULL, MAKEINTRESOURCE(IDR_MAINMENU));
return CFrameWnd::PreCreateWindow(cs);
}
// 缩小图象
void CMyWnd::OnShrink()
{
m_fTimes = 0.5;
Invalidate();
}
// 放大图象
void CMyWnd::OnZoomOut()
{
m_fTimes = 2.0;
Invalidate();
}
```

```
// 原样显示
void CMyWnd::OnBestFit()
{
m_fTimes = 1.0;
Invalidate();
}
// 响应绘制窗口客户区消息
void CMyWnd::OnPaint()
{
CPaintDC dc(this);
CDC MemDC;
MemDC.CreateCompatibleDC(NULL);
MemDC.SelectObject(&m_Bitmap);
dc.StretchBlt(0, 0, (int)(m_nWidth*m_fTimes),
(int)(m_nHeight*m_fTimes),
&MemDC, 0, 0, m_nWidth, m_nHeight, SRCCOPY);
}
// 应用程序类
class CMyApp: public CWinApp
{
public:
BOOL InitInstance();
};
// 应用窗口类的成员函数
// 初始化应用程序实例
BOOL CMyApp::InitInstance()
{
CMyWnd *pFrame = new CMyWnd;
pFrame->Create(0,_T("Show Bitmap 1.0"));
pFrame->ShowWindow(SW_SHOWMAXIMIZED);
```

```
pFrame->UpdateWindow();
this->m_pMainWnd = pFrame;
return TRUE;
}
// 全局应用程序对象
CMyApp ThisApp;
```

7-26 拼图程序。

设计思想：将一张图片切分成若干小片，打乱顺序任意显示。用户可用鼠标拖动各小片到正确位置以恢复用来的图象。

说 明：首先创建一 Win32 Application 空项目，然后向项目中添加 C++ Source File（源程序）和 Resource Script（资源描述）文件。在添加资源描述文件后应将其关闭。

按本单元介绍的有关内容为本程序添加图标、字符串（窗口标题）和下拉菜单（标题为“游戏”）等资源，其标识符均取 IDR_MAINFRAME。

为“游戏”下拉菜单添 4 个菜单选项，一个是“自动拼图”， 和一个“结束”，标识符分别为 ID_BEGIN 和 ID_END。剩下 3 个用来控制拼图的难度，标识符分别为 ID_GRAD01，ID_GRAD02 和 ID_GRAD03。

另选一幅漂亮的图片，作为资源装入项目。该图片即拼图的底图。然后输入以下程序，注意在编译前选择使用 MFC。

程 序：

```
// Example 11-4: 拼图程序
#include <afxwin.h>
#include <afxext.h>
#include <stdlib.h>
```

```
#include <time.h>
#include "resource.h"
#define MAX_CHIPS 500
// 派生一个框架窗口类
class CPuzzleWnd: public CFrameWnd
{
CBitmap m_bmpPuzzle; // 位图
int m_nPuzzleWidth; // 位图宽
int m_nPuzzleHeight; // 位图高
int m_nColCount; // 每行拼图块数
int m_nRowCount; // 每列拼图块数
CRect m_rectChips[MAX_CHIPS]; // 每个拼图块的位置
int m_nChipWidth; // 拼图块宽
int m_nChipHeight; // 拼图块高
BOOL m_bCaptured;
CPoint m_pointMouse;
int m_nCurrIndex;
public:
CPuzzleWnd();
void InitPuzzle(int colcount, int rowcount);
protected:
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
afx_msg void OnPaint();
afx_msg void OnShowFig();
afx_msg void OnGrad01();
```

```
afx_msg void OnGrad02();
afx_msg void OnGrad03();
DECLARE_MESSAGE_MAP()
};
// 消息映射
BEGIN_MESSAGE_MAP(CPuzzleWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_LBUTTONUP()
ON_WM_MOUSEMOVE()
ON_WM_PAINT()
ON_COMMAND(ID_SHOWFIG, OnShowFig)
ON_COMMAND(ID_GRAD01, OnGrad01)
ON_COMMAND(ID_GRAD02, OnGrad02)
ON_COMMAND(ID_GRAD03, OnGrad03)
END_MESSAGE_MAP()
// 主窗口类的成员函数
// 构造函数: 装入位图资源
CPuzzleWnd::CPuzzleWnd()
{
BITMAP BM;
m_bmpPuzzle.LoadBitmap(IDB_BITMAP1);
m_bmpPuzzle.GetObject(sizeof(BM), &BM);
m_nPuzzleWidth = BM.bmWidth;
m_nPuzzleHeight = BM.bmHeight;
InitPuzzle(4, 3);
m_bCaptured = FALSE;
}
// 初始化拼图数组
void CPuzzleWnd::InitPuzzle(int colcount, int rowcount)
{
m_nColCount = colcount;
```

```
m_nRowCount = rowcount;
m_nChipWidth = m_nPuzzleWidth/m_nColCount;
m_nChipHeight = m_nPuzzleHeight/m_nRowCount;
srand((unsigned)time( NULL));
for(int row=0; row<m_nRowCount; row++)
for(int col=0; col<m_nColCount; col++)
{
int index = row*m_nColCount+col;
m_rectChips[index].left = rand()%m_nPuzzleWidth+20;
m_rectChips[index].top = rand()%m_nPuzzleHeight+20;
m_rectChips[index].right = m_rectChips[index].left
+m_nChipWidth;
m_rectChips[index].bottom = m_rectChips[index].top
+m_nChipHeight;
}
}
```

// 自动完成拼图

```
void CPuzzleWnd::OnShowFig()
{
for(int row=0; row<m_nRowCount; row++)
for(int col=0; col<m_nColCount; col++)
{
int index = row*m_nColCount+col;
m_rectChips[index].left = col*m_nChipWidth;
m_rectChips[index].top = row*m_nChipHeight;
m_rectChips[index].right = m_rectChips[index].left
+m_nChipWidth;
m_rectChips[index].bottom = m_rectChips[index].top
+m_nChipHeight;
}
Invalidate();
}
```

// 设置一级难度

```
void CPuzzleWnd::OnGrad01()
{
InitPuzzle(4, 3);
Invalidate();
}
// 设置二级难度
void CPuzzleWnd::OnGrad02()
{
InitPuzzle(8, 6);
Invalidate();
}
// 设置三级难度
void CPuzzleWnd::OnGrad03()
{
InitPuzzle(16, 12);
Invalidate();
}
// 按下鼠标左键
void CPuzzleWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
for(int row=m_nRowCount-1; row>=0; row--)
for(int col=m_nColCount-1; col>=0; col--)
{
int index = row*m_nColCount+col;
if(m_rectChips[index].PtInRect(point))
{
SetCapture();
m_bCaptured = TRUE;
m_pointMouse = point;
m_nCurrIndex = index;
return;
}
```

```
}
}
// 释放鼠标左键
void CPuzzleWnd::OnLButtonUp(UINT nFlags, CPoint point)
{
if(m_bCaptured)
{
::ReleaseCapture();
m_bCaptured = FALSE;
}
}
// 移动鼠标左键
void CPuzzleWnd::OnMouseMove(UINT nFlags, CPoint point)
{
if(m_bCaptured)
{
InvalidateRect(m_rectChips[m_nCurrIndex]);
CSize offset(point-m_pointMouse);
m_rectChips[m_nCurrIndex] += offset;
InvalidateRect(m_rectChips[m_nCurrIndex], FALSE);
m_pointMouse = point;
}
}
// 显示当前拼图
void CPuzzleWnd::OnPaint()
{
CPaintDC dc(this);
CDC MemDC;
MemDC.CreateCompatibleDC(NULL);
MemDC.SelectObject(&m_bmpPuzzle);
for(int row=0; row<m_nRowCount; row++)
for(int col=0; col<m_nColCount; col++)
```

```
{
int index = row*m_nColCount+col;
dc.BitBlt(m_rectChips[index].left,
m_rectChips[index].top,
m_nChipWidth,
m_nChipHeight,
&MemDC,
col*m_nChipWidth,
row*m_nChipHeight,
SRCCOPY);
}
}
// 应用程序类
class CPuzzleApp: public CWinApp
{
public:
BOOL InitInstance();
};
// 初始化应用程序实例
BOOL CPuzzleApp::InitInstance()
{
CPuzzleWnd *pFrame = new CPuzzleWnd;
pFrame->LoadFrame(IDR_MAINMENU);
pFrame->ShowWindow(SW_SHOWMAXIMIZED);
this->m_pMainWnd = pFrame;
return TRUE;
}
// 全局应用程序对象
CPuzzleApp TheApp;
// Example 11-5: 地空战游戏程序
#include <afxwin.h>
```

```
#include "resource.h"
// 定义飞机类
class CPlane: public CObject
{
CPoint m_pointPlane; // 飞机位置
CBitmap m_bmpPlane; // 飞机图象
int m_nWidth; // 飞机图象宽
int m_nHeight; // 飞机图象高
public:
CPlane();
void ShowPlane(CDC *pDC, CDC *pMemDC, CRect Client);
CRect GetPlane(){return CRect(m_pointPlane.x, m_pointPlane.y,
m_pointPlane.x+m_nWidth, m_pointPlane.y+m_nHeight);}
void ChangePos();
void ResetPos(){m_pointPlane.x = 0;}
};
// 飞机类的成员函数
// 构造函数
CPlane::CPlane()
{
m_pointPlane = CPoint(0, 50);
m_bmpPlane.LoadBitmap(IDB_PLANE);
BITMAP BM;
m_bmpPlane.GetBitmap(&BM);
m_nWidth = BM.bmWidth;
m_nHeight = BM.bmHeight;
}
// 显示飞机
```

```
void CPlane::ShowPlane(CDC *pDC, CDC *pMemDC, CRect Client)
{
pMemDC->SelectObject(&m_bmpPlane);
pDC->BitBlt(m_pointPlane.x, m_pointPlane.y,
m_nWidth, m_nHeight, pMemDC,0,0,SRCAND);
}
// 改变飞机位置
void CPlane::ChangePos()
{
if(m_pointPlane.x>788)
m_pointPlane.x = 0;
else
m_pointPlane.x += 10;
}
// 定义炸弹类
class CBomb: public CObject
{
CPoint m_pointBomb; // 炸弹位置

CBitmap m_bmpBomb; // 炸弹图象

int m_nWidth; // 炸弹图象高

int m_nHeight; // 炸弹图象宽

public:
CBomb();
void ShowBomb(CDC *pDC, CDC *pMemDC, CRect Client);
CRect GetBomb(){return CRect(m_pointBomb.x, m_pointBomb.y,
m_pointBomb.x+m_nWidth, m_pointBomb.y+m_nHeight);}
void ChangePos(int x);
void ResetPos(){m_pointBomb.x=0, m_pointBomb.y=80;}
};
```

```
// 炸弹类成员函数

// 炸弹类构造函数

CBomb::CBomb()
{
m_pointBomb.x = 0;
m_pointBomb.y = 80;
m_bmpBomb.LoadBitmap(IDB_BOMB);
BITMAP BM;
m_bmpBomb.GetBitmap(&BM);
m_nWidth = BM.bmWidth;
m_nHeight = BM.bmHeight;
}

// 显示炸弹

void CBomb::ShowBomb(CDC *pDC, CDC *pMemDC, CRect Client)
{
pMemDC->SelectObject(&m_bmpBomb);
pDC->BitBlt(m_pointBomb.x, m_pointBomb.y,
m_nWidth, m_nHeight, pMemDC, 0, 0, SRCAND);
}

// 改变位置

void CBomb::ChangePos(int x)
{
m_pointBomb.y += 20;
if(m_pointBomb.y>480)
m_pointBomb.y = 80;
m_pointBomb.x = x;
}

// 定义高炮类

class CTank: public CObject
{
```

```
CPoint m_pointTank; // 高炮位置

CBitmap m_bmpTank; // 高炮图象

int m_nWidth; // 高炮图象宽

int m_nHeight; // 高炮图象高

public:
CTank();
void ShowTank(CDC *pDC, CDC *pMemDC, CRect Client);
CRect GetTank(){return CRect(m_pointTank.x, m_pointTank.y,
m_pointTank.x+m_nWidth, m_pointTank.y+m_nHeight);}
void ChangePos(int tag);
void ResetPos(){m_pointTank.x = 350;}
};

// 高炮类成员函数

// 高炮类构造函数

CTank::CTank()
{
m_pointTank.x = 350;
m_pointTank.y = 450;
m_bmpTank.LoadBitmap(IDB_TANK);
BITMAP BM;
m_bmpTank.GetBitmap(&BM);
m_nWidth = BM.bmWidth;
m_nHeight = BM.bmHeight;
}

// 显示高炮

void CTank::ShowTank(CDC *pDC, CDC *pMemDC, CRect Client)
{
pMemDC->SelectObject(&m_bmpTank);
pDC->BitBlt(m_pointTank.x, m_pointTank.y,
```

```
m_nWidth, m_nHeight, pMemDC,0,0,SRCAND);
}
// 改变位置
void CTank::ChangePos(int tag)
{
if(tag == 0 && m_pointTank.x > 0)
m_pointTank.x -= 20;
else if(tag == 1 && m_pointTank.x+m_nWidth < 798)
m_pointTank.x += 20;
}
// 定义炮弹类
class CStone: public CObject
{
CPoint m_pointStone; // 炮弹位置

CBitmap m_bmpStone; // 炮弹图象

int m_nWidth; // 炮弹图象宽

int m_nHeight; // 炮弹图象高

BOOL m_bShot; // 是否已发射

public:
CStone();
BOOL HaveStone(){return !m_bShot;}
void Shot(int x);
void ShowStone(CDC *pDC, CDC *pMemDC, CRect Client);
CRect GetStone(){return CRect(m_pointStone.x, m_pointStone.y,
m_pointStone.x+m_nWidth, m_pointStone.y+m_nHeight);}
void ChangePos();
void ResetPos(){m_bShot = FALSE;}
void Refill(){m_bShot = FALSE;}
};
```

```
// 炮弹类成员函数

// 炮弹类构造函数

CStone::CStone()
{
m_bShot = FALSE;
m_bmpStone.LoadBitmap(IDB_STONE);
BITMAP BM;
m_bmpStone.GetBitmap(&BM);
m_nWidth = BM.bmWidth;
m_nHeight = BM.bmHeight;
}

// 发射

void CStone::Shot(int x)
{
m_bShot = TRUE;
m_pointStone.x = x+48;
m_pointStone.y = 440;
}

// 显示炮弹

void CStone::ShowStone(CDC *pDC, CDC *pMemDC, CRect Client)
{
if(m_bShot)
{
pMemDC->SelectObject(&m_bmpStone);
pDC->BitBlt(m_pointStone.x, m_pointStone.y,
m_nWidth, m_nHeight, pMemDC,0,0,SRCAND);
}
}

// 改变位置

void CStone::ChangePos()
```

```
{
if(m_bShot)
m_pointStone.y -= 20;
}
```

// 自定义消息

```
#define ON_WM_GAMEOVER 0x0402
#define I_SHOT_YOU 0
#define YOU_SHOT_ME 1
```

// 定义框架窗口类

```
class CMyWnd: public CFrameWnd
{
CPlane m_Plane;
CBomb m_Bomb;
CTank m_Tank;
CStone m_Stone;
CPoint m_pointMoutain[5];
public:
CMyWnd();
void DrawFields(CDC *pDC, CRect Client);
BOOL ShotOn(CRect &body1, CRect &body2);
protected:
afx_msg void OnPaint();
afx_msg void OnBegin();
afx_msg void OnEnd();
afx_msg void OnTimer(UINT nIDEvent);
afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
afx_msg void OnGameOver(UINT tag);
DECLARE_MESSAGE_MAP()
};
```

// 消息映射

```
BEGIN_MESSAGE_MAP(CMyWnd,CFrameWnd)
ON_WM_PAINT()
```

```
ON_COMMAND(ID_BEGIN, OnBegin)
ON_COMMAND(ID_END, OnEnd)
ON_WM_TIMER()
ON_WM_KEYDOWN()
ON_MESSAGE(ON_WM_GAMEOVER, OnGameOver)
END_MESSAGE_MAP()

// 框架窗口类的成员函数

// 构造函数

CMyWnd::CMyWnd()
{
m_pointMoutain[0] = CPoint(300, 400);
m_pointMoutain[1] = CPoint(400, 300);
m_pointMoutain[2] = CPoint(500, 350);
m_pointMoutain[3] = CPoint(600, 250);
m_pointMoutain[4] = CPoint(800, 400);
}

// 更新窗口客户区

void CMyWnd::OnPaint()
{
CPaintDC dc(this);
CDC MemDC;
MemDC.CreateCompatibleDC(NULL);
CRect rect;
GetClientRect(&rect);

DrawFields(&dc, rect); // 画战场

m_Plane.ShowPlane(&dc, &MemDC, rect); // 画对方飞机

m_Bomb.ShowBomb(&dc, &MemDC, rect); // 画炸弹

m_Tank.ShowTank(&dc, &MemDC, rect); // 画己方防空导弹车
```

```
m_Stone.ShowStone(&dc, &MemDC, rect); // 画导弹
}
// 响应菜单消息: 开始游戏
void CMyWnd::OnBegin()
{
m_Bomb.ResetPos();
m_Plane.ResetPos();
m_Tank.ResetPos();
m_Stone.ResetPos();
Invalidate();
SetTimer(1, 100, NULL);
}
// 响应菜单消息: 游戏结束
void CMyWnd::OnEnd()
{
KillTimer(1);
PostMessage(WM_QUIT);
}
// 定时器消息响应函数
void CMyWnd::OnTimer(UINT nIDEvent)
{
InvalidateRect(m_Plane.GetPlane()); // 修改飞机位置
m_Plane.ChangePos();
InvalidateRect(m_Plane.GetPlane(), FALSE);
InvalidateRect(m_Bomb.GetBomb()); // 修改炸弹位置
m_Bomb.ChangePos(m_Plane.GetPlane().left);
InvalidateRect(m_Bomb.GetBomb(), FALSE);
if(!m_Stone.HaveStone()) // 修改炮弹位置
{
```

```
CRect rect = m_Stone.GetStone();
if(rect.top < 3)
m_Stone.Refill();
else
{
InvalidateRect(rect);
m_Stone.ChangePos();
}
InvalidateRect(m_Stone.GetStone(), FALSE);
}
// 判断射击效果
if(!m_Stone.HaveStone() && ShotOn(m_Plane.GetPlane(), m_Stone.GetStone()))
SendMessage(ON_WM_GAMEOVER, YOU_SHOT_ME);
else if(ShotOn(m_Tank.GetTank(), m_Bomb.GetBomb()))
SendMessage(ON_WM_GAMEOVER, I_SHOT_YOU);
}
// 自定义的游戏结束消息响应函数
void CMyWnd::OnGameOver(UINT tag)
{
KillTimer(1);
if(tag == I_SHOT_YOU)
MessageBox("我砸到你了!!!");
else
MessageBox("你击中我了!!!");
}
// 按键消息响应函数
void CMyWnd::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
if(nChar == VK_LEFT) // 使用 < 键左移高炮
{
```

```
InvalidateRect(m_Tank.GetTank());
m_Tank.ChangePos(0);
InvalidateRect(m_Tank.GetTank(), FALSE);
}
else if(nChar == VK_RIGHT) // 使用 > 键右移高炮
{
InvalidateRect(m_Tank.GetTank());
m_Tank.ChangePos(1);
InvalidateRect(m_Tank.GetTank());
}
else if(nChar == 32 && m_Stone.HaveStone()) // 使用空格键发炮
m_Stone.Shot(m_Tank.GetTank().left);
}
// 画战场
void CMyWnd::DrawFields(CDC *pDC, CRect Client)
{
CBrush *pOldBrush, brushSky, brushGrass, brushMoutain;
CRect rect(Client);
brushSky.CreateSolidBrush(RGB(127, 200, 255)); // 画天空
pOldBrush = pDC->SelectObject(&brushSky);
pDC->Rectangle(rect);
pDC->SelectObject(pOldBrush);
brushGrass.CreateSolidBrush(RGB(0, 255, 0)); // 画草地
pOldBrush = pDC->SelectObject(&brushGrass);
rect.top = 400;
pDC->Rectangle(rect);
pDC->SelectObject(pOldBrush);
brushMoutain.CreateSolidBrush(RGB(125, 50, 0)); // 画大山
pOldBrush = pDC->SelectObject(&brushMoutain);
pDC->Polygon(m_pointMoutain, 5);
```

```
pDC->SelectObject(pOldBrush);
}
// 判断射击效果
BOOL CMyWnd::ShotOn(CRect &body1, CRect &body2)
{
return body1.PtInRect(body2.TopLeft());
}
// 定义应用程序类
class CMyApp: public CWinApp
{
public:
BOOL InitInstance();
};
// 初始化实例函数
BOOL CMyApp::InitInstance()
{
CMyWnd *pFrame = new CMyWnd;
pFrame->LoadFrame(IDR_MAINFRAME);
pFrame->ShowWindow(SW_SHOWMAXIMIZED);
this->m_pMainWnd = pFrame;
return TRUE;
}
// 说明应用程序类的全局对象
CMyApp ThisApp;
```

第 八 章　　多态性

8-1 什么叫做多态性 ?在 C++中是如何实现多态的?

解:

多态是指同样的消息被不同类型的对象接收时导致完全不同的行为,是对类的特定成员

函数的再抽象。C++支持的多态有多种类型，重载(包括函数重载和运算符重载)和虚函数是其中主要的方式。

8-2 什么叫做抽象类？抽象类有何作用？抽象类的派生类是否一定要给出纯虚函数的实现?

解:

带有纯虚函数的类是抽象类。抽象类的主要作用是通过它为一个类族建立一个公共的接口,使它们能够更有效地发挥多态特性。抽象类声明了一组派生类共同操作接口的通用语义，而接口的完整实现，即纯虚函数的函数体，要由派生类自己给出。但抽象类的派生类并非一定要给出纯虚函数的实现，如果派生类没有给出纯虚函数的实现，这个派生类仍然是一个抽象类。

8-3 声明一个参数为整型，无返回值，名为 fn1 的虚函数。

解:

```
virtual void fn1( int );
```

8-4 在 C++中，能否声明虚构造函数？为什么？能否声明虚析构函数？有何用途?

解:

在 C++中，不能声明虚构造函数，多态是不同的对象对同一消息有不同的行为特性，虚函数作为运行过程中多态的基础,主要是针对对象的,而构造函数是在对象产生之前运行的,因此虚构造函数是没有意义的；可以声明虚析构函数，析构函数的功能是在该类对象消亡之前进行一些必要的清理工作，如果一个类的析构函数是虚函数，那么，由它派生而来的所有子类的析构函数也是虚函数。析构函数设置为虚函数之后,在使用指针引用时可以动态联编,实现运行时的多态,保证使用基类的指针就能够调用适当的析构函数针对不同的对象进行清

理工作。

8-5 实现重载函数 Double(x)，返回值为输入参数的两倍；参数分别为整型、长整型、浮点型、双精度型，返回值类型与参数一样。

解：

源程序：

```
#include <iostream.h>
int Double(int);
long Double(long);
float Double(float);
double Double(double);
int main()
{
int myInt = 6500;
long myLong = 65000;
float myFloat = 6.5F;
double myDouble = 6.5e20;
int doubledInt;
long doubledLong;
float doubledFloat;
double doubledDouble;
cout << "myInt: " << myInt << "\n";
cout << "myLong: " << myLong << "\n";
cout << "myFloat: " << myFloat << "\n";
cout << "myDouble: " << myDouble << "\n";
doubledInt = Double(myInt);
doubledLong = Double(myLong);
doubledFloat = Double(myFloat);
doubledDouble = Double(myDouble);
cout << "doubledInt: " << doubledInt << "\n";
cout << "doubledLong: " << doubledLong << "\n";
cout << "doubledFloat: " << doubledFloat << "\n";
```

```
cout << "doubledDouble: " << doubledDouble << "\n";
return 0;
}
int Double(int original)
{
cout << "In Double(int)\n";
return 2 * original;
}
long Double(long original)
{
cout << "In Double(long)\n";
return 2 * original;
}
float Double(float original)
{
cout << "In Double(float)\n";
return 2 * original;
}
double Double(double original)
{
cout << "In Double(double)\n";
return 2 * original;
}
```

程序运行输出：

```
myInt: 6500
myLong: 65000
myFloat: 6.5
myDouble: 6.5e+20
In Double(int)
In Double(long)
In Double(float)
In Double(double)
DoubledInt: 13000
```

DoubledLong: 130000

DoubledFloat: 13

DoubledDouble: 1.3e+21

8-6 定义一个Rectangle类,有长itsWidth、宽itsLength等属性,重载其构造函数Rectangle()和Rectangle(int width, int length)。

解:

源程序:

```
#include <iostream.h>
class Rectangle
{
public:
Rectangle();
Rectangle(int width, int length);
~Rectangle() {}
int GetWidth() const { return itsWidth; }
int GetLength() const { return itsLength; }
private:
int itsWidth;
int itsLength;
};
Rectangle::Rectangle()
{
itsWidth = 5;
itsLength = 10;
}
Rectangle::Rectangle (int width, int length)
{
itsWidth = width;
itsLength = length;
}
int main()
```

```
{
Rectangle Rect1;
cout << "Rect1 width: " << Rect1.GetWidth() << endl;
cout << "Rect1 length: " << Rect1.GetLength() << endl;
int aWidth, aLength;
cout << "Enter a width: ";
cin >> aWidth;
cout << "\nEnter a length: ";
cin >> aLength;
Rectangle Rect2(aWidth, aLength);
cout << "\nRect2 width: " << Rect2.GetWidth() << endl;
cout << "Rect2 length: " << Rect2.GetLength() << endl;
return 0;
}
```

程序运行输出：

```
Rect1 width: 5
Rect1 length: 10
Enter a width: 20
Enter a length: 50
Rect2 width: 20
Rect2 length: 50
```

8-7 定义计数器 Counter 类，对其重载运算符 + 。

解：

源程序：

```
typedef unsigned short USHORT;
#include <iostream.h>
class Counter
{
public:
Counter();
Counter(USHORT initialValue);
```

```
~Counter(){}
USHORT GetItsVal()const { return itsVal; }
void SetItsVal(USHORT x) {itsVal = x; }
Counter operator+ (const Counter &);
private:
USHORT itsVal;
};
Counter::Counter(USHORT initialValue):
itsVal(initialValue)
{
}
Counter::Counter():
itsVal(0)
{
}
Counter Counter::operator+ (const Counter & rhs)
{
return Counter(itsVal + rhs.GetItsVal());
}
int main()
{
Counter varOne(2), varTwo(4), varThree;
varThree = varOne + varTwo;
cout << "varOne: " << varOne.GetItsVal()<< endl;
cout << "varTwo: " << varTwo.GetItsVal() << endl;
cout << "varThree: " << varThree.GetItsVal() << endl;
return 0;
}
```

程序运行输出：

```
varOne: 2
varTwo: 4
varThree: 6
```

8-8 定义一个哺乳动物 Mammal 类，再由此派生出狗 Dog 类，二者都定义 Speak()成员函数，基类中定义为虚函数，定义一个 Dog 类的对象，调用 Speak 函数，观察运行结果。

解：

源程序：

```
#include <iostream.h>
class Mammal
{
public:
Mammal():itsAge(1) { cout << "Mammal constructor...\n"; }
~Mammal() { cout << "Mammal destructor...\n"; }
virtual void Speak() const { cout << "Mammal speak!\n"; }
};
class Dog : public Mammal
{
public:
Dog() { cout << "Dog Constructor...\n"; }
~Dog() { cout << "Dog destructor...\n"; }
void Speak()const { cout << "Woof!\n"; }
};
int main()
{
Mammal *pDog = new Dog;
pDog->Speak();
return 0;
}
```

程序运行输出：

```
Mammal constructor...
Dog constructor...
Woof!
Dog destructor...
Mammal destructor...
```

8-9 定义一个 Shape 抽象类，在此基础上派生出 Rectangle 和 Circle，二者都有 GetArea() 函数计算对象的面积，GetPerim()函数计算对象的周长。

解：

源程序：

```
#include <iostream.h>
class Shape
{
public:
Shape(){}
~Shape(){}
virtual float GetArea() =0 ;
virtual float GetPerim () =0 ;
};
class Circle : public Shape
{
public:
Circle(float radius):itsRadius(radius){}
~Circle(){}
float GetArea() { return 3.14 * itsRadius * itsRadius; }
float GetPerim () { return 6.28 * itsRadius; }
private:
float itsRadius;
};
class Rectangle : public Shape
{
public:
Rectangle(float len， float width): itsLength(len)， itsWidth(width){};
~Rectangle(){};
virtual float GetArea() { return itsLength * itsWidth; }
float GetPerim () { return 2 * itsLength + 2 * itsWidth; }
virtual float GetLength() { return itsLength; }
```

```
virtual float GetWidth() { return itsWidth; }
private:
float itsWidth;
float itsLength;
};
void main()
{
Shape * sp;
sp = new Circle(5);
cout << "The area of the Circle is " << sp->GetArea () << endl;
cout << "The perimeter of the Circle is " << sp->GetPerim () << endl;
delete sp;

sp = new Rectangle(4，6);

cout << "The area of the Rectangle is " << sp->GetArea() << endl;
cout << "The perimeter of the Rectangle is " << sp->GetPerim () << endl;
delete sp;
}
```

程序运行输出：

```
The area of the Circle is 78.5
The perimeter of the Circle is 31.4
The area of the Rectangle is 24
The perimeter of the Rectangle is 20
```

8-10 对 Point 类重载++（自增）、--（自减）运算符

解：

```
#include <iostream.h>
class Point
{
public:
Point& operator++();
Point operator++(int);
Point& operator--();
```

```
Point operator--(int);
Point() { _x = _y = 0; }
int x() { return _x; }
int y() { return _y; }
private:
int _x, _y;
};
Point& Point::operator++()
{
_x++;
_y++;
return *this;
}
Point Point::operator++(int)
{
Point temp = *this;
++*this;
return temp;
}
Point& Point::operator--()
{
_x--;
_y--;
return *this;
}
Point Point::operator--(int)
{
Point temp = *this;
--*this;
return temp;
}
void main()
{
Point A;
```

```
cout << "A 的值为： " << A.x() << " , " << A.y() << endl;
A++;
cout << "A 的值为： " << A.x() << " , " << A.y() << endl;
++A;
cout << "A 的值为： " << A.x() << " , " << A.y() << endl;
A--;
cout << "A 的值为： " << A.x() << " , " << A.y() << endl;
--A;
cout << "A 的值为： " << A.x() << " , " << A.y() << endl;
}
```

程序运行输出：

```
A 的值为： 0 , 0
A 的值为： 1 , 1
A 的值为： 2 , 2
A 的值为： 1 , 1
A 的值为： 0 , 0
```

8-11 定义一个基类 BaseClass，从它派生出类 DerivedClass，BaseClass 有成员函数 fn1()、fn2()，fn1()是虚函数，DerivedClass 也有成员函数 fn1()、fn2()，在主程序中定义一个 DerivedClass 的对象，分别用 BaseClass 和 DerivedClass 的指针来调用 fn1()、fn2()，观察运行结果。

解：

```
#include <iostream.h>
class BaseClass
{
```

```
public:
virtual void fn1();
void fn2();
};
void BaseClass::fn1()
{
cout << "调用基类的虚函数 fn1()" << endl;
}
void BaseClass::fn2()
{
cout << "调用基类的非虚函数 fn2()" << endl;
}
class DerivedClass : public BaseClass
{
public:
void fn1();
void fn2();
};
void DerivedClass::fn1()
{
cout << "调用派生类的函数 fn1()" << endl;
}
void DerivedClass::fn2()
{
cout << "调用派生类的函数 fn2()" << endl;
}
void main()
{
DerivedClass aDerivedClass;
DerivedClass *pDerivedClass = &aDerivedClass;
BaseClass *pBaseClass = &aDerivedClass;
```

```
pBaseClass->fn1();
pBaseClass->fn2();
pDerivedClass->fn1();
pDerivedClass->fn2();
}
```

程序运行输出：

调用派生类的函数 fn1()

调用基类的非虚函数 fn2()

调用派生类的函数 fn1()

调用派生类的函数 fn2()

8-12 定义一个基类 BaseClass，从它派生出类 DerivedClass，BaseClass 中定义虚析构函数，在主程序中将一个 DerivedClass 的对象地址赋给一个 BaseClass 的指针，观察运行过程。

解：

```
#include <iostream.h>
class BaseClass {
public:
virtual ~BaseClass() {
cout << "~BaseClass()" << endl;
}
};
class DerivedClass : public BaseClass {
public:
~DerivedClass() {
cout << "~DerivedClass()" << endl;
}
};
void main()
```

```
{
BaseClass* bp = new DerivedClass;
delete bp;
}
```

程序运行输出：

```
~DerivedClass()
~BaseClass()
```

8-13 定义 Point 类，有成员变量 X、Y，为其定义友元函数实现重载+。

解：

```
#include <iostream.h>
class Point
{
public:
Point() { X = Y = 0; }
Point( unsigned x, unsigned y ) { X = x; Y = y; }
unsigned x() { return X; }
unsigned y() { return Y; }
void Print() { cout << "Point(" << X << ", " << Y << ")" << endl; }
friend Point operator+( Point& pt, int nOffset );
friend Point operator+( int nOffset, Point& pt );
private:
unsigned X;
unsigned Y;
};
Point operator+( Point& pt, int nOffset )
{
Point ptTemp = pt;
ptTemp.X += nOffset;
ptTemp.Y += nOffset;
return ptTemp;
}
Point operator+( int nOffset, Point& pt )
```

```
{
Point ptTemp = pt;
ptTemp.X += nOffset;
ptTemp.Y += nOffset;
return ptTemp;
}
void main()
{
Point pt( 10, 10 );
pt.Print();
pt = pt + 5; // Point + int
pt.Print();
pt = 10 + pt; // int + Point
pt.Print();
}
```

程序运行输出：

```
Point(10, 10)
Point(15, 15)
Point(25, 25)
```

8-14 为某公司设计一个人事管理系统，其基本功能为输入、编辑、查看和保存公司的人事档案。职工人事档案包括姓名、性别、出生日期、婚姻状况、所在部门、职务和工资。

程 序：

由于列表框尚未初始化，所以为 CEmpDlg 类重载 OnInitDialog（）成员函数（可使用 ClassWizard 完成），并添加相应代码：

```
BOOL CEmpDlg::OnInitDialog()
{
CListBox *pLB = (CListBox *)GetDlgItem(IDC_DEPT);
pLB->InsertString(-1, "办公室");
pLB->InsertString(-1, "开发部");
```

```
pLB->InsertString(-1, "生产部");

pLB->InsertString(-1, "销售部");

pLB->InsertString(-1, "人事部");

return CDialog::OnInitDialog();
}
```

其中 GetDlgItem（）为对话框类的成员函数，用于取对话框控件的指针。

为项目添加有关自定义的职工类 CEmployee。选择 Developer Studio 菜单的 Insert/New Class...选项，调出 New Class 对话框。在 Class Type 组合框中选择 Generic（普通类），填写类名 CEmployee，在对话框下方的 Base class (es)框中输入基类 CObject。

在 Workspace 窗口的 Class View 中选择生成的 CEmployee 类的定义，添加代码：

```
class CEmployee : public CObject
{
DECLARE_SERIAL(CEmployee)
public:

CString m_strName; //  姓名

int m_nSex; //  性别

COleDateTime m_tBirthdate; //  出生日期

BOOL m_bMarried; //  婚否

CString m_strDept; //  工作部门

CString m_strPosition; //  职务

float m_fSalary; //  工资

CEmployee(){}
CEmployee& operator = (CEmployee& e);
```

```
virtual ~CEmployee();
virtual void Serialize(CArchive &ar);
};
```

CEmployee 类的对象即为一个职工的档案，我们用序列化实现文档的存取，所以要为 CEmployee 类编写序列化代码。这包括 DECLARE_SERIAL（）宏和 IMPLEMENT_SERIAL（）宏（在 CEmployee 类的源代码文件中），一个没有参数的构造函数，重载的赋值运算符和 Serialize（）成员函数。在 CEmployee 类的源代码文件中添加以下代码：

```
IMPLEMENT_SERIAL(CEmployee, CObject, 1)
// 重载的赋值运算符
CEmployee& CEmployee::operator = (CEmployee& e)
{
m_strName = e.m_strName;
m_nSex = e.m_nSex;
m_tBirthdate = e.m_tBirthdate;
m_bMarried = e.m_bMarried;
m_strDept = e.m_strDept;
m_strPosition = e.m_strPosition;
m_fSalary = e.m_fSalary;
return *this;
}
// 序列化函数
void CEmployee::Serialize(CArchive& ar)
{
CObject::Serialize(ar);
if(ar.IsStoring())
{
ar << m_strName;
ar << m_nSex;
ar << m_tBirthdate;
ar << m_bMarried;
```

```
ar << m_strDept;
ar << m_strPosition;
ar << m_fSalary;
}
else
{
ar >> m_strName;
ar >> m_nSex;
ar >> m_tBirthdate;
ar >> m_bMarried;
ar >> m_strDept;
ar >> m_strPosition;
ar >> m_fSalary;
}
}
```

然后修改文档类 CMyDocument 类定义，添加一个 CEmployee 类的数组：

```
#include "employee.h"
#define MAX_EMPLOYEE 1000
class CMy1501Doc : public CDocument
{
DECLARE_DYNCREATE(CMy1501Doc)
public:
CEmployee m_empList[MAX_EMPLOYEE];
int m_nCount;
public:
virtual BOOL OnNewDocument();
virtual void Serialize(CArchive& ar);
virtual void DeleteContents();
DECLARE_MESSAGE_MAP()
};
```

为了节省篇幅，这段程序经过删节，与原来由 AppWizard 生成的程序有所不同。其中黑体部分为要添加的代码。注意重载成员函数 DeleteContents（）可以手工进行，也可以通过

ClassWizard 进行。Serialize（）和 DeleteContents（）两个成员函数的代码如下：

```
void CMy1501Doc::Serialize(CArchive& ar)
{
if(ar.IsStoring())
ar << m_nCount;
else
ar >> m_nCount;
for(int i=0; i<m_nCount; i++)
m_empList[i].Serialize(ar);
}
void CMy1501Doc::DeleteContents()
{
m_nCount = 0; // 在打开文件和建立新文件时将数组大小置 0
CDocument::DeleteContents();
}
```

即在文档类的 Serialize()函数中，数据的序列化工作是通过调用 Cemployee 类的 Serialize（）函数实现的。

实际上，要为本程序添加的大部分代码均在视图类中。首先在视图类 CmyView 类的定义中添加一个用于记录当前操作的是哪个记录的数据成员：

```
int m_nCurrEmp;
```

并为视图类重载 OnInitialUpdate（）成员函数，在其中初始化该变量：

```
void CMy1501View::OnInitialUpdate()
{
CView::OnInitialUpdate();
m_nCurrEmp = 0;
Invalidate();
}
```

视图类的 OnDraw（）成员函数用于显示正在操作的职工档案：

```
void CMy1501View::OnDraw(CDC* pDC)
```

```
{
CMy1501Doc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// 显示职工人数和当前职工编号
CString s;
s.Format("职工人数: %d", pDoc->m_nCount);
pDC->SetTextColor(RGB(255, 0, 0));
pDC->TextOut( 40, 40, s);
s.Format("职工编号: %d", m_nCurrEmp+1);
pDC->TextOut(340, 40, s);
pDC->MoveTo( 40, 70);
pDC->LineTo(600, 70);
// 如果档案非空，显示当前记录
if(pDoc->m_nCount > 0)
{
// 显示栏目名称
pDC->SetTextColor(RGB(0, 0, 0));
pDC->TextOut(140, 90, "姓  名:");
pDC->TextOut(140, 130, "性  别:");
pDC->TextOut(140, 170, "出生日期: ");
pDC->TextOut(140, 210, "婚姻状态:");
pDC->TextOut(140, 250, "部  门:");
pDC->TextOut(140, 290, "职  务:");
pDC->TextOut(140, 330, "工  资:");
// 显示栏目内容
```

```
pDC->SetTextColor(RGB(0, 0, 255));
pDC->TextOut(300, 90, pDoc->m_empList[m_nCurrEmp].m_strName);
if(pDoc->m_empList[m_nCurrEmp].m_nSex==0)
pDC->TextOut(300, 130, "男");
else
pDC->TextOut(300, 130, "女");
s = pDoc->m_empList[m_nCurrEmp].m_tBirthdate.Format("%Y.%m.%d");
pDC->TextOut(300, 170, s);
if(pDoc->m_empList[m_nCurrEmp].m_bMarried)
pDC->TextOut(300, 210, "已婚");
else
pDC->TextOut(300, 210, "未婚");
pDC->TextOut(300, 250, pDoc->m_empList[m_nCurrEmp].m_strDept);
pDC->TextOut(300, 290, pDoc->m_empList[m_nCurrEmp].m_strPosition);
s.Format("%8.2f", pDoc->m_empList[m_nCurrEmp].m_fSalary);
pDC->TextOut(300, 330, s);
}
}
```

在编辑资源时，我们框架窗口添加了 5 个菜单选项，并将对应的消息响应函数映射到了视图类中。这些消息响应函数的代码如下：

```
void CMy1501View::OnAppend()
{
CMy1501Doc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
CEmpDlg dlg;
if(dlg.DoModal() == IDOK)
{
pDoc->m_nCount++;
m_nCurrEmp = pDoc->m_nCount-1;
pDoc->m_empList[m_nCurrEmp].m_strName = dlg.m_strName;
```

```
pDoc->m_empList[m_nCurrEmp].m_nSex = dlg.m_nSex;
pDoc->m_empList[m_nCurrEmp].m_tBirthdate = dlg.m_tBirthdate;
pDoc->m_empList[m_nCurrEmp].m_bMarried = dlg.m_bMarried;
pDoc->m_empList[m_nCurrEmp].m_strDept = dlg.m_strDept;
pDoc->m_empList[m_nCurrEmp].m_strPosition = dlg.m_strPosition;
pDoc->m_empList[m_nCurrEmp].m_fSalary = dlg.m_fSalary;
pDoc->SetModifiedFlag();
Invalidate();
}
}
```

// 删除当前记录

```
void CMy1501View::OnDelete()
{
CMy1501Doc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
if(pDoc->m_nCount)
{
for(int i=m_nCurrEmp; i<pDoc->m_nCount-1; i++)
pDoc->m_empList[i] = pDoc->m_empList[i+1];
pDoc->m_nCount--;
if(m_nCurrEmp > pDoc->m_nCount-1)
m_nCurrEmp = pDoc->m_nCount-1;
pDoc->SetModifiedFlag();
Invalidate();
}
}
```

// 编辑当前记录

```
void CMy1501View::OnEdit()
{
CMy1501Doc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
if(pDoc->m_nCount)
{
```

```
CEmpDlg dlg;
dlg.m_strName = pDoc->m_empList[m_nCurrEmp].m_strName;
dlg.m_nSex = pDoc->m_empList[m_nCurrEmp].m_nSex;
dlg.m_tBirthdate = pDoc->m_empList[m_nCurrEmp].m_tBirthdate;
dlg.m_bMarried = pDoc->m_empList[m_nCurrEmp].m_bMarried;
dlg.m_strDept = pDoc->m_empList[m_nCurrEmp].m_strDept;
dlg.m_strPosition = pDoc->m_empList[m_nCurrEmp].m_strPosition;
dlg.m_fSalary = pDoc->m_empList[m_nCurrEmp].m_fSalary;
if(dlg.DoModal() == IDOK)
{
pDoc->m_empList[m_nCurrEmp].m_strName = dlg.m_strName;
pDoc->m_empList[m_nCurrEmp].m_nSex = dlg.m_nSex;
pDoc->m_empList[m_nCurrEmp].m_tBirthdate = dlg.m_tBirthdate;
pDoc->m_empList[m_nCurrEmp].m_bMarried = dlg.m_bMarried;
pDoc->m_empList[m_nCurrEmp].m_strDept = dlg.m_strDept;
pDoc->m_empList[m_nCurrEmp].m_strPosition = dlg.m_strPosition;
pDoc->m_empList[m_nCurrEmp].m_fSalary = dlg.m_fSalary;
pDoc->SetModifiedFlag();
Invalidate();
}
}
}
```

// 查看后一记录

```
void CMy1501View::OnNext()
{
CMy1501Doc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
if(pDoc->m_nCount > 1)
{
if(m_nCurrEmp == pDoc->m_nCount-1)
m_nCurrEmp = 0;
else
m_nCurrEmp++;
```

```
}
Invalidate();
}
// 查看前一记录
void CMy1501View::OnPrev()
{
CMy1501Doc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
if(pDoc->m_nCount > 1)
{
if(m_nCurrEmp == 0)
m_nCurrEmp = pDoc->m_nCount-1;
else
m_nCurrEmp--;
}
Invalidate();
}
```

8-15 用基于对话框的应用程序结构实现例 14-5 的彩色吹泡泡程序。由于对话框本身结构简单，没有明显的客户区，颜色也不醒目，所以我们在对话框上自行建立一个矩形区域作为吹泡泡的客户区，并通过一个“颜色设置”按钮来设置泡泡的颜色。

说 明：用 AppWizard 建立一个基于对话框的应用程序框架（参看 15.4：“用 AppWizard 生成基于对话框的应用程序”），所有设置均使用缺省值。

使用对话框模板编辑器编辑作为主界面窗口的对话框模板，将其上的静态文本控件和“Cancel”按钮删除，将“OK”按钮的 Caption 设置为“完成”，并将对话框大小调整为 400×300 左右。

为对话框模板添加一个 Picture 控件，将其 Type 设置为 Frame，Color 设置为 Black，并

设置 Sunken 属性（在 Styles 选项卡中）。调整其位置为（7，7），大小为 287×287。这个框中即为自定义的吹泡泡客户区，所有的吹泡泡活动均在该区域中进行。

为对话框模板添加一个按钮，将其 ID 改为 IDC_COLOR，Caption 改为“颜色设置”。

使用 ClassWizard 为对话框类添加一个鼠标左键消息响应函数 OnLButtonDown（）和一个按钮命令消息响应函数 OnColor（）。

程 序：

在对话框类的头文件前面添加一行：

```
#define MAX_BUBBLE 250
```

并在对话框类定义中添加存放泡泡的几何参数和颜色的数组数据成员：

```
CRect m_rectBubble[MAX_BUBBLE];
COLORREF m_colorBubble[MAX_BUBBLE];
int m_nBubbleCount;
```

以及一个存放自定义客户区矩形的数据成员和一个存放当前泡泡颜色设置的数据成员：

```
CRect m_rectClient;
COLORREF m_colorCurrent;
```

修改对话框类的 OnInitDialog（）成员函数，添加计算自定义客户区位置和大小的代码，并将泡泡的数目初始化为 0：

```
BOOL CMyDlg::OnInitDialog()
{
CDialog::OnInitDialog();
CStatic *pST = (CStatic *)GetDlgItem(IDC_CLIENT);
pST->GetWindowRect(&m_rectClient);
ScreenToClient(&m_rectClient);
m_nBubbleCount = 0;
return TRUE;
}
```

修改 OnPaint（）成员函数，添加画出泡泡的有关代码：

```
void CMyDlg::OnPaint()
{
CPaintDC dc(this);
CRgn rgn;

rgn.CreateRectRgnIndirect(&m_rectClient); // 生成一个区域对象

dc.SelectClipRgn(&rgn); // 选择区域

dc.Rectangle(m_rectClient); // 将客户区背景设置

CBrush brushNew, *pbrushOld; // 白色

for(int i=0; i<m_nBubbleCount; i++)
{
brushNew.CreateSolidBrush(m_colorBubble[i]);
pbrushOld = dc.SelectObject(&brushNew);
dc.Ellipse(m_rectBubble[i]);
dc.SelectObject(pbrushOld);
brushNew.DeleteObject();
}
}
```

修改由 ClassWizard 生成的鼠标左键消息响应函数 OnLButtonDown（），添加吹泡泡的有关代码：

```
void CMyDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
if(m_nBubbleCount < MAX_BUBBLE)
{
int r = rand()%50+10;
CRect rect(point.x-r, point.y-r, point.x+r, point.y+r);
m_rectBubble[m_nBubbleCount] = rect;
m_colorBubble[m_nBubbleCount] = m_colorCurrent;
m_nBubbleCount++;
```

```
InvalidateRect(rect, FALSE);
}
}
```

最后修改由 ClassWizard 生成的按钮消息响应函数 OnColor（），添加调用颜色设置公用对话框的代码：

```
void CMyDlg::OnColor()
{
m_colorCurrent = RGB(200, 200, 200);
CColorDialog dlg(m_colorCurrent);
if(dlg.DoModal() == IDOK)
m_colorCurrent = dlg.GetColor();
}
```

[例 15-3] 动画播放器程序，可用文件查找公用对话框打开 AVI 文件并播放。

程 序：首先在对话框类定义中添加一个存放 AVI 文件名的变量：

```
CString m_strAviname;
```

并在 OnInitDialog（）函数中添加初始化代码：

```
m_strAviname = _T("");
```

修改对应于 3 个按钮的消息响应函数：

```
void CMy1503Dlg::OnOpen()
{
CFileDialog dlg(TRUE, ".AVI", "*.AVI",
OFN_FILEMUSTEXIST|OFN_LONGNAMES|OFN_PATHMUSTEXIST,
"*.AVI", this);
if(dlg.DoModal() == IDOK)
{
m_ctrAvi.Close();
m_strAviname = dlg.GetPathName();
m_ctrAvi.Open(LPCTSTR(m_strAviname));
}
}
```

```
void CMy1503Dlg::OnPlay()
{
m_ctrAvi.Play(0, 0xffff, 0xffff);
}
void CMy1503Dlg::OnStop()
{
m_ctrAvi.Stop();
}
```

8-16 编写一个计算器程序。该计算器使用编辑控件直接输入数据，并有“加”、“减”、“乘”、“除”、“平方根”和“倒数”计算功能，如图 15-4。

程 序：

在对话框类中添加以下数据成员：

```
int m_nOP; // 运算符

double m_fResult; // 运算中间结果
```

并在 OnInitDialog（）函数中添加相应的初始化代码：

```
m_fResult = 0.0;
m_nOP = 0;
```

为对话框类添加一个 Calc（）成员函数：

```
void CMy1504Dlg::Calc() // 计算
{
UpdateData(TRUE);
switch(m_nOP)
{
case 0: // 第 1 运算对象
m_fResult = m_fInput;
break;
case 1: // +
```

```
m_fResult += m_fInput;
break;
case 2: // -
m_fResult -= m_fInput;
break;
case 3: // *
m_fResult *= m_fInput;
break;
case 4: // /
m_fResult -= m_fInput;
break;
case 5: // 1/X
m_fResult = 1/m_fInput;
break;
case 6: // sqrt(X)
m_fResult = sqrt(m_fInput);
break;
}
m_fInput = m_fResult;
UpdateData(FALSE);
}
```

最后为所有按钮的消息响应函数添加代码：

```
void CMy1504Dlg::OnAdd() //  加法
{
Calc();
m_nOP = 1;
}
void CMy1504Dlg::OnSub() //  减法
{
Calc();
m_nOP = 2;
}
```

```
void CMy1504Dlg::OnMul() // 乘法
{
Calc();
m_nOP = 3;
}
void CMy1504Dlg::OnDiv() // 除法
{
Calc();
m_nOP = 4;
}
void CMy1504Dlg::OnReciprocal() // 倒数
{
m_nOP = 5;
Calc();
m_nOP = 0;
}
void CMy1504Dlg::OnSqrt() // 平方根
{
m_nOP = 6;
Calc();
m_nOP = 0;
}
void CMy1504Dlg::OnSetfocusInput() // 为输入数据做准备
{
m_fInput = 0.0;
UpdateData(FALSE);
}
void CMy1504Dlg::OnClear() // 清除
{
m_fResult = 0.0;
```

```
m_fInput = 0.0;
m_nOP = 0;
UpdateData(FALSE);
}

void CMy1504Dlg::OnCalc() // 显示计算结果
{
Calc();
m_nOP = 0;
}
```

第 十 章　　群体数据的组织

10-1 简单说明插入排序的算法思想。

解：

插入排序的基本思想是：每一步将一个待排序元素按其关键字值的大小插入到已排序序列的适当位置上，直到待排序元素插入完为止。

10-2 初始化整型数组 data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20}，调用教材中的直接插入排序函数模板进行排序，对此函数模板稍做修改，加入输出语句，在每插入一个待排序元素后显示整个数组，观察排序过程中数据的变化，加深对插入排序算法的理解。

解：

```
#include <iostream.h>
template <class T>
void InsertionSort(T A[], int n)
{
int i, j;
T temp;
// 将下标为 1 ~ n-1 的元素逐个插入到已排序序列中适当的位置
```

```
for (i = 1; i < n; i++)
{
//从 A[i-1]开始向 A[0]方向扫描各元素,寻找适当位置插入 A[i]
j = i;
temp = A[i];
while (j > 0 && temp < A[j-1])
{ //逐个比较，直到 temp>=A[j-1]时，j 便是应插入的位置。
//若达到 j==0，则 0 是应插入的位置。
A[j] = A[j-1];
//将元素逐个后移，以便找到插入位置时可立即插入。
j--;
}
// 插入位置已找到，立即插入。
A[j] = temp;
//输出数据
for(int k=0;k<n;k++)
cout << A[k] << " ";
cout << endl;
//结束输出
}
}
void main()
{
int i;
int data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20};
cout << "排序前的数据： " << endl;
for(i=0;i<20;i++)
cout << data1[i] << " ";
```

```
cout << endl;

cout << "开始排序..." << endl;

InsertionSort(data1, 20);

cout << "排序后的数据：" << endl;

for(i=0;i<20;i++)
cout << data1[i] << " ";
cout << endl;
}
```

程序运行输出：

排序前的数据：

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

开始排序...

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 2 3 5 7 9 11 13 15 17 19 4 6 8 10 12 14 16 18 20

1 2 3 4 5 7 9 11 13 15 17 19 6 8 10 12 14 16 18 20

1 2 3 4 5 6 7 9 11 13 15 17 19 8 10 12 14 16 18 20

1 2 3 4 5 6 7 8 9 11 13 15 17 19 10 12 14 16 18 20

1 2 3 4 5 6 7 8 9 10 11 13 15 17 19 12 14 16 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 15 17 19 14 16 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 19 16 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

排序后的数据：

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

10-3 简单说明选择排序的算法思想。

解：

选择类排序的基本思想是：每次从待排序序列中选择一个关键字最小的元素，（当需要按关键字升序排列时），顺序排在已排序序列的最后，直至全部排完。

10-4 初始化整型数组 data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20}，调用教材中的直接选择排序函数模板进行排序，对此函数模板稍做修改，加入输出语句，每对一个待排序元素排序后显示整个数组，观察排序过程中数据的变化，加深对直接选择排序算法的理解。

解：

```
#include <iostream.h>
// 辅助函数：交换 x 和 y 的值
template <class T>
void Swap (T &x, T &y)
{
T temp;
temp = x;
x = y;
y = temp;
}
// 用选择法对数组 A 的 n 个元素进行排序
template <class T>
void SelectionSort(T A[], int n)
```

```
{
int smallIndex; //每以趟中选出的最小元素之下标
int i, j;
// sort A[0]..A[n-2], and A[n-1] is in place
for (i = 0; i < n-1; i++)
{
smallIndex = i; //最小元素之下标初值设为 i
//  在元素 A[i+1]..A[n-1]中逐个比较显出最小值
for (j = i+1; j < n; j++)
// smallIndex 始终记录当前找到的最小值的下标
if (A[j] < A[smallIndex])
smallIndex = j;
//  将这一趟找到的最小元素与 A[i]交换
Swap(A[i], A[smallIndex]);
//输出数据
for(int k=0;k<n;k++)
cout << A[k] << " ";
cout << endl;
//结束输出
}
}
void main()
{
int i;
int data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20};
cout << "排序前的数据：" << endl;
for(i=0;i<20;i++)
cout << data1[i] << " ";
```

```
cout << endl;

cout << "开始排序..." << endl;

SelectionSort(data1, 20);

cout << "排序后的数据：  " << endl;

for(i=0;i<20;i++)
cout << data1[i] << " ";
cout << endl;
}
```

程序运行输出：

排序前的数据：

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

1 2 5 7 9 11 13 15 17 19 3 4 6 8 10 12 14 16 18 20

1 2 3 7 9 11 13 15 17 19 5 4 6 8 10 12 14 16 18 20

1 2 3 4 9 11 13 15 17 19 5 7 6 8 10 12 14 16 18 20

1 2 3 4 5 11 13 15 17 19 9 7 6 8 10 12 14 16 18 20

1 2 3 4 5 6 13 15 17 19 9 7 11 8 10 12 14 16 18 20

1 2 3 4 5 6 7 15 17 19 9 13 11 8 10 12 14 16 18 20

1 2 3 4 5 6 7 8 17 19 9 13 11 15 10 12 14 16 18 20

1 2 3 4 5 6 7 8 9 19 17 13 11 15 10 12 14 16 18 20

1 2 3 4 5 6 7 8 9 10 17 13 11 15 19 12 14 16 18 20

1 2 3 4 5 6 7 8 9 10 11 13 17 15 19 12 14 16 18 20

1 2 3 4 5 6 7 8 9 10 11 12 17 15 19 13 14 16 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 15 19 17 14 16 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 19 17 15 16 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 19 16 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 17 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 18 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

排序后的数据：

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

10-5 简单说明交换排序的算法思想。

解：

交换排序的基本思想是：两两比较待排序序列中的元素，并交换不满足顺序要求的各对元素，直到全部满足顺序要求为止。

10-6 初始化整型数组 data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20}，调用教材中的起泡排序函数模板进行排序，对此函数模板稍做修改，加入输出语句，每完成一趟起泡排序后显示整个数组，观察排序过程中数据的变化，加深对起泡排序算法的理解。

解：

```
#include <iostream.h>
// 辅助函数：交换 x 和 y 的值
template <class T>
void Swap (T &x, T &y)
{
T temp;
temp = x;
x = y;
y = temp;
}
// 用起泡法对数组 A 的 n 个元素进行排序
template <class T>
void BubbleSort(T A[], int n)
{
int i,j;
int lastExchangeIndex;
//用于记录每趟被交换的最后一对元素中较小的下标
```

```
i = n-1; // i 是下一趟需参与排序交换的元素之最大下标
while (i > 0)
//持续排序过程，直到最后一趟排序没有交换发生，或已达 n-1 趟
{
lastExchangeIndex = 0;
//每一趟开始时，设置交换标志为 0（未交换）
for (j = 0; j < i; j++) //每一趟对元素 A[0]..A[i]进行比较和交换
if (A[j+1] < A[j]) //如果元素 A[j+1] < A[j]，交换之
{
Swap(A[j],A[j+1]);
lastExchangeIndex = j;
//记录被交换的一对元素中较小的下标
}
// 将 i 设置为本趟被交换的最后一对元素中较小的下标
i = lastExchangeIndex;
//输出数据
for(int k=0;k<n;k++)
cout << A[k] << " ";
cout << endl;
//结束输出
}
}
void main()
{
int i;
int data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20};
cout << "排序前的数据： " << endl;
```

```
for(i=0;i<20;i++)
cout << data1[i] << " ";
cout << endl;
cout << "开始排序..." << endl;
BubbleSort(data1, 20);
cout << "排序后的数据：  " << endl;
for(i=0;i<20;i++)
cout << data1[i] << " ";
cout << endl;
}
```

程序运行输出：

排序前的数据：

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

开始排序...

1 3 5 7 9 11 13 15 17 2 4 6 8 10 12 14 16 18 19 20

1 3 5 7 9 11 13 15 2 4 6 8 10 12 14 16 17 18 19 20

1 3 5 7 9 11 13 2 4 6 8 10 12 14 15 16 17 18 19 20

1 3 5 7 9 11 2 4 6 8 10 12 13 14 15 16 17 18 19 20

1 3 5 7 9 2 4 6 8 10 11 12 13 14 15 16 17 18 19 20

1 3 5 7 2 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20

1 3 5 2 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

1 3 2 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

排序后的数据：

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

10-7 教材中的排序算法都是升序排序，稍做修改后即可完成降序排序，以起泡排序法为例，完成降序的起泡排序函数模板，初始化整型数组

data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20}，调用降序起泡排序函数模板进行排序，加入输出语句，每完成一趟起泡排序后显示整个数组，观察排序过程中数据的变化。

解：

```
#include <iostream.h>
// 辅助函数：交换 x 和 y 的值
template <class T>
void Swap (T &x, T &y)
{
T temp;
temp = x;
x = y;
y = temp;
}
// 用起泡法对数组 A 的 n 个元素进行排序
template <class T>
void BubbleSort(T A[], int n)
{
int i,j;
int lastExchangeIndex;
//用于记录每趟被交换的最后一对元素中较小的下标

i = n-1; // i 是下一趟需参与排序交换的元素之最大下标
while (i > 0)
//持续排序过程，直到最后一趟排序没有交换发生，或已达 n-1 趟
{
lastExchangeIndex = 0;
//每一趟开始时，设置交换标志为 0（未交换）

for (j = 0; j < i; j++) //每一趟对元素 A[0]..A[i]进行比较和交换
```

```
if (A[j+1] > A[j]) //如果元素 A[j+1] < A[j]，交换之
{
Swap(A[j],A[j+1]);
lastExchangeIndex = j;
//记录被交换的一对元素中较小的下标
}
//  将 i 设置为本趟被交换的最后一对元素中较小的下标
i = lastExchangeIndex;
//输出数据
for(int k=0;k<n;k++)
cout << A[k] << " ";
cout << endl;
//结束输出
}
}
void main()
{
int i;
int data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20};
cout << "排序前的数据：" << endl;
for(i=0;i<20;i++)
cout << data1[i] << " ";
cout << endl;
cout << "开始排序..." << endl;
BubbleSort(data1, 20);
cout << "排序后的数据：" << endl;
for(i=0;i<20;i++)
cout << data1[i] << " ";
```

```
cout << endl;
}
```

程序运行输出：

排序前的数据：

1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20

开始排序…

3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20 1

5 7 9 11 13 15 17 19 3 4 6 8 10 12 14 16 18 20 2 1

7 9 11 13 15 17 19 5 4 6 8 10 12 14 16 18 20 3 2 1

9 11 13 15 17 19 7 5 6 8 10 12 14 16 18 20 4 3 2 1

11 13 15 17 19 9 7 6 8 10 12 14 16 18 20 5 4 3 2 1

13 15 17 19 11 9 7 8 10 12 14 16 18 20 6 5 4 3 2 1

15 17 19 13 11 9 8 10 12 14 16 18 20 7 6 5 4 3 2 1

17 19 15 13 11 9 10 12 14 16 18 20 8 7 6 5 4 3 2 1

19 17 15 13 11 10 12 14 16 18 20 9 8 7 6 5 4 3 2 1

19 17 15 13 11 12 14 16 18 20 10 9 8 7 6 5 4 3 2 1

19 17 15 13 12 14 16 18 20 11 10 9 8 7 6 5 4 3 2 1

19 17 15 13 14 16 18 20 12 11 10 9 8 7 6 5 4 3 2 1

19 17 15 14 16 18 20 13 12 11 10 9 8 7 6 5 4 3 2 1

19 17 15 16 18 20 14 13 12 11 10 9 8 7 6 5 4 3 2 1

19 17 16 18 20 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

19 17 18 20 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

19 18 20 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

19 20 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

排序后的数据：

20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

10-8 简单说明顺序查找的算法思想。

解：

顺序查找是一种最简单、最基本的查找方法。

其基本思想是：从数组的首元素开始，逐个元素与待查找的关键字进行比较，直到找到相等的。若整个数组中没有与待查找关键字相等的元素，就是查找不成功。

10-9 初始化整型数组 data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20}，提示用户输入一个数字，调用教材中的顺序查找函数模板找出它的位置。

解：

```
#include <iostream.h>
template <class T>
int SeqSearch(T list[], int n, T key)
{
for(int i=0;i < n;i++)
if (list[i] == key)
return i;
return -1;
}
void main()
{
int i, n;
int data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20};
cout << "输入想查找的数字(1~20)： ";
cin >> n;
cout << "数据为： " << endl;
for(i=0;i<20;i++)
cout << data1[i] << " ";
cout << endl;
i = SeqSearch ( data1 , 20 , n );
if (i<0)
cout << "没有找到数字" << n << endl;
```

```
else
cout << n << "是第" << i+1 << "个数字" << endl;
}
```

程序运行输出：

```
输入想查找的数字(1~20)：6
数据为：
1 3 5 7 9 11 13 15 17 19 2 4 6 8 10 12 14 16 18 20
6 是第 13 个数字
```

10-10 简单说明折半查找的算法思想。

解：

如果是在一个元素排列有序的数组中进行查找，可以采用折半查找方法。

折半查找方法的基本思想是：对于已按关键字排序的序列，经过一次比较，可将序列分割成两部分，然后只在有可能包含待查元素的一部分中继续查找，并根据试探结果继续分割，逐步缩小查找范围，直至找到或找不到为止。

10-11 初始化整型数组 data1[]={1,3,5,7,9,11,13,15,17,19,2,4,6,8,10,12,14,16,18,20}，提示用户输入一个数字，调用教材中的折半查找函数模板找出它的位置。

解：

```
#include <iostream.h>
// 用折半查找方法，在元素呈升序排列的数组 list 中查找值为 key 的元素
template <class T>
int BinSearch(T list[], int n, T key)
{
```

```
int mid, low, high;
T midvalue;
low=0;
high=n-1;
while (low <= high) // low <= high 表示整个数组尚未查找完
{
mid = (low+high)/2; // 求中间元素的下标
midvalue = list[mid]; // 取出中间元素的值
if (key == midvalue)
return mid; // 若找到,返回下标
else if (key < midvalue)
high = mid-1;
// 若 key < midvalue 将查找范围缩小到数组的前一半
else
low = mid+1; // 否则将查找范围缩小到数组的后一半
}
return -1; // 没有找到返回-1
}
void main()
{
int i, n;
int data1[]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
cout << "输入想查找的数字(1~20)： ";
cin >> n;
cout << "数据为： " << endl;
for(i=0;i<20;i++)
cout << data1[i] << " ";
```

```
cout << endl;
i = BinSearch ( data1 , 20 , n );
if (i<0)
cout << "没有找到数字" << n << endl;
else
cout << n << "是第" << i+1 << "个数字" << endl;
}
```

程序运行输出：

```
输入想查找的数字(1~20)：9
数据为：
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
9 是第 9 个数字
```

第 十一 章　　流类库与输入/输出

11-1 什么叫做流？流的提取和插入是指什么？I/O 流在 C++中起着怎样的作用？

解：

流是一种抽象，它负责在数据的生产者和数据的消费者之间建立联系，并管理数据的流动，一般意义下的读操作在流数据抽象中被称为（从流中）提取，写操作被称为（向流中）插入。操作系统是将键盘、屏幕、打印机和通信端口作为扩充文件来处理的，I/O 流类就是用来与这些扩充文件进行交互，实现数据的输入与输出。

11-2 cerr 和 clog 有何区别？

解：

cerr 标准错误输出，没有缓冲，发送给它的内容立即被输出，适用于立即向屏幕输出

的错误信息；clog 类似于 cerr，但是有缓冲，缓冲区满时被输出，在向磁盘输出时效率更高。

11-3 使用 I/O 流以文本方式建立一个文件 test1.txt，写入字符“已成功写入文件!”，用其它字处理程序（例如 windows 的记事本程序 Notepad）打开，看看是否正确写入。

解：

```
#include <fstream.h>
void main()
{
ofstream file1("test.txt");
file1 << "已成功写入文件！";
file1.close();
}
```

程序运行后 test1.txt 的内容为：已成功写入文件!

11-4 使用 I/O 流以文本方式打开上一题建立的文件 test1.txt，读出其内容显示出来，看看是否正确。

解：

```
#include <fstream.h>
void main()
{
char ch;
ifstream file2("test.txt");
while (file2.get(ch))
cout << ch;
file2.close();
}
```

程序运行输出：

已成功写入文件!

11-5 使用 I/O 流以文本方式打开上题建立的文件 test1.txt，在次此文件后面添加字符“已成功添加字符！”，然后读出整个文件的内容显示出来，看看是否正确。

解：

```
#include <fstream.h>
void main()
{
ofstream file1("test.txt",ios::app);
file1 << "已成功添加字符！";
file1.close ();
char ch;
ifstream file2("test.txt");
while (file2.get(ch))
cout << ch;
file2.close();
}
```

程序运行输出：

已成功写入文件！已成功添加字符！

11-6 定义一个 dog 类，包含体重和年龄两个成员变量及相应的成员函数，声明一个实例 dog1，体重为 5，年龄为 10，使用 I/O 流把 dog1 的状态写入磁盘文件，再声明另一个实例 dog2，通过读文件把 dog1 的状态赋给 dog2。分别使用文本方式和二进制方式操作文件，看看结果有何不同；再看看磁盘文件的 ASCII 码有何不同。

解：

以两种方式操作，程序运行结果一样，但磁盘文件的 ASCII 码不同，使用二进制方式时，磁盘文件的 ASCII 码为 05 00 00 00 0A 00 00 00，使用文本方式时，磁盘文件的 ASCII 码为 05 00 00 00 0D 0A 00 00 00，这是因为此时系统自动把 0A 转换为了 0D 0A。

```
#include <fstream.h>
class dog
{
public:
dog(int weight, long days):itsWeight(weight),
itsNumberDaysAlive(days){}
~dog(){}
int GetWeight()const { return itsWeight; }
void SetWeight(int weight) { itsWeight = weight; }
long GetDaysAlive()const { return itsNumberDaysAlive; }
void SetDaysAlive(long days) { itsNumberDaysAlive = days; }
private:
int itsWeight;
long itsNumberDaysAlive;
};
int main() // returns 1 on error
{
char fileName[80];
cout << "Please enter the file name: ";
cin >> fileName;
ofstream fout(fileName);
// ofstream fout(fileName,ios::binary);
if (!fout)
{
cout << "Unable to open " << fileName << " for writing.\n";
return(1);
}
dog Dog1(5,10);
fout.write((char*) &Dog1,sizeof Dog1);
fout.close();
ifstream fin(fileName);
// ifstream fin(fileName,ios::binary);
if (!fin)
{
```

```
cout << "Unable to open " << fileName << " for reading.\n";
return(1);
}
dog Dog2(2,2);
cout << "Dog2 weight: " << Dog2.GetWeight() << endl;
cout << "Dog2 days: " << Dog2.GetDaysAlive() << endl;
fin.read((char*) &Dog2, sizeof Dog2);
cout << "Dog2 weight: " << Dog2.GetWeight() << endl;
cout << "Dog2 days: " << Dog2.GetDaysAlive() << endl;
fin.close();
return 0;
}
```

程序运行输出：

```
Please enter the file name: a
Dog2 weight: 2
Dog2 days: 2
Dog2 weight: 5
Dog2 days: 10
```

11-7 观察下面的程序，说明每条语句的作用，看看程序执行的结果。

```
#include <iostream>
using namespace ::std;
void main()
{
ios_base::fmtflags original_flags = cout.flags(); //1
cout<< 812<<'|';
cout.setf(ios_base::left,ios_base::adjustfield); //2
cout.width(10); //3
cout<< 813 << 815 << '\n';
cout.unsetf(ios_base::adjustfield); //4
cout.precision(2);
cout.setf(ios_base::uppercase|ios_base::scientific); //5
cout << 831.0 ;
cout.flags(original_flags); //6
```

}

解：

//1 保存现在的格式化参数设置，以便将来恢复这些设置；

//2 把对齐方式由缺省的右对齐改为左对齐；

//3 把输出域的宽度由缺省值 0 改为 10；

//4 清除对齐方式的设置；

//5 更改浮点数的显示设置；

//6 恢复原来的格式化参数设置。

程序运行输出：

812|813 815

8.31E+02

11-8 提示用户输入一个十进制整数，分别用十进制、八进制和十六进制形式输出。

解：

```
#include <iostream.h>
void main() {
int n;
cout << "请输入一个十进制整数： ";
cin >> n;
cout << "这个数的十进制形式为： " << dec << n << endl;
cout << "这个数的八进制形式为： " << oct << n << endl;
cout << "这个数的十六进制形式为： " << hex << n << endl;
}
```

程序运行输出：

请输入一个十进制整数：15

这个数的十进制形式为：15

这个数的八进制形式为：17

这个数的十六进制形式为：f

11-9 编写程序实现如下功能：打开指定的一个文本文件，在每一行前加行号。

解：

```
//b.cpp
#include <fstream.h>
#include <strstrea.h>
#include <stdlib.h>
void main(int argc, char* argv[])
{
strstream textfile;
{
ifstream in(argv[1]);
textfile << in.rdbuf();
}
ofstream out(argv[1]);
const int bsz = 100;
char buf[bsz];
int line = 0;
while(textfile.getline(buf, bsz)) {
out.setf(ios::right, ios::adjustfield);
out.width(1);
out << ++line << ". " << buf << endl;
}
}
```

编译后运行程序 b text1.txt

运行前 text1.txt 的内容为：

aaaaaaaaaaaa

bbbbbbbbbbbb

cccccccccccc

dddddddddddd

eeeeeeeeeeee

ffffffffffff

gggggggggggg

hhhhhhhhhhhh

运行后 text1.txt 的内容为：

1. aaaaaaaaaaaa

2. bbbbbbbbbbbb

3. cccccccccccc

4. dddddddddddd

5. eeeeeeeeeeee

6. ffffffffffff

7. gggggggggggg

8. hhhhhhhhhhhh

第 十二 章　　异常处理

12-1 什么叫做异常？什么叫做异常处理？

解：

当一个函数在执行的过程中出现了一些不平常的情况，或运行结果无法定义的情况，使得操作不得不被中断时，我们说出现了异常。异常通常是用 throw 关键字产生的一个对象，用来表明出现了一些意外的情况。我们在设计程序时，就要充分考虑到各种意外情况，并给与恰当的处理。这就是我们所说的异常处理。

12-2 C++的异常处理机制有何优点？

解：

C++的异常处理机制使得异常的引发和处理不必在同一函数中，这样底层的函数可以着重解决具体问题，而不必过多地考虑对异常的处理。上层调用者可以在适当的位置设计对不同类型异常的处理。

12-3 举例 throw 、try、catch 语句的用法?

解:

throw 语句用来引发异常，用法为:

throw 表达式;

例如: throw 1.0E-10;

catch 语句用来处理某中类型的异常，它跟在一个 try 程序块后面处理这个 try 程序块产生的异常，如果一个函数要调用一个可能会引发异常的函数， 并且想在异常真的出现后处理异常，就必须使用 try 语句来捕获异常。

例如:

try{

语句 //可能会引发多种异常

}

catch（参数声明 1）

{

语句 //异常处理程序

}

12-4 设计一个异常 Exception 抽象类，在此基础上派生一个 OutOfMemory 类响应内存不足，一个 RangeError 类响应输入的数不在指定范围内，实现并测试这几个类。

解:

源程序:

```
#include <iostream.h>
class Exception
{
public:
Exception(){}
virtual ~Exception(){}
virtual void PrintError() = 0;
};
class OutOfMemory : public Exception
{
public:
OutOfMemory(){}
~OutOfMemory(){}
virtual void PrintError();
};
void OutOfMemory::PrintError()
{
cout << "Out of Memory!!\n";
}
class RangeError : public Exception
{
public:
RangeError(unsigned long number){BadNum = number;}
~RangeError(){}
virtual void PrintError();
virtual unsigned long GetNumber() { return BadNum; }
virtual void SetNumber(unsigned long number) {BadNum = number;}
private:
unsigned long BadNum;
};
void RangeError::PrintError()
{
```

```
cout << "Number out of range. You used " << GetNumber() << " !\n";
}
void fn1();
unsigned int * fn2();
void fn3(unsigned int *);
int main()
{
try
{
fn1();
}
catch (Exception& theException)
{
theException.PrintError();
}
return 0;
}
unsigned int * fn2()
{
unsigned int *n = new unsigned int;
if (n == 0)
throw OutOfMemory();
return n;
}
void fn1()
{
unsigned int *p = fn2();
fn3(p);
cout << "The number is : " << *p << endl;
delete p;
}
void fn3(unsigned int *p)
{
long Number;
```

```
cout << "Enter an integer(0~~1000): ";
cin >> Number;
if (Number > 1000 || Number < 0)
throw RangeError(Number);
*p = Number;
}
```

程序运行输出：

```
Enter an integer(0~~1000): 56
The number is : 56
Enter an integer(0~~1000): 2000
Number out of range. You used 2000 !
```

12-5 练习使用 try、catch 语句，在程序中用 new 分配内存时，如果操作未成功，则用 try 语句触发一个字符型异常，用 catch 语句捕获此异常。

解：

```
#include <iostream.h>
void main()
{
char *buf;
try
{
buf = new char[512];
if( buf == 0 )
throw "内存分配失败!";
}
catch( char * str )
{
cout << "有异常产生： " << str << endl;
}
}
```

12-6 定义一个异常类 CException，有成员函数 Reason()，用来显示异常的类型，定义函数 fn1()触发异常，在主函数的 try 模块中调用 fn1(),在 catch 模块中捕获异常，观察程序的执行流程。

解：

```
#include <iostream.h>
class CException
{
public:
CException(){};
~CException(){};
const char *Reason() const { return "CException 类中的异常。"; }
};
void fn1()
{
cout<< "在子函数中触发 CException 类异常" << endl;
throw CException();
}
void main()
{
cout << "进入主函数" << endl;
try
{
cout << "在 try 模块中，调用子函数" << endl;
fn1();
}
catch( CException E )
{
cout << "在 catch 模块中，捕获到 CException 类型异常： ";
```

```
cout << E.Reason() << endl;
}
catch( char *str )
{
cout << "捕获到其它类型异常： " << str << endl;
}
cout << "回到主函数，异常已被处理" << endl;
}
```

程序运行输出：

进入主函数

在 try 模块中，调用子函数

在子函数中触发 CException 类异常

在 catch 模块中，捕获到 CException 类型异常：CException 类中的异常。

回到主函数，异常已被处理

第 十三 章　　MFC 类库与 Windows 程序开发简介

13-1 在 MS-DOS 环境下的 C++程序中，main()函数必不可少，在 Windows 程序中，什么函数代替了 main()函数，它有何特点?

解：

Windows 程序中替代 main()函数是 WinMain()函数，每一个 Windows 程序都需要有一个 WinMain()函数，该函数主要是建立应用程序的主窗口。与 MS-DOS 程序的根本差别在于：MS-DOS 程序是通过调用操作系统的功能来获得用户输入的，而 Windows 程序则是通过操作系统发送的消息来处理用户输入的，程序的主窗口中需要包含处理 Windows 所发送消息的代码。

13-2 什么叫做类库?

解:

类库是一个可以在应用程序中使用的相互关联的 C++类的集合。

13-3 当我们用应用程序向导生成 MFC 应用程序时，在源代码中找不到 WinMain()函数，这是为什么?

解:

当使用应用程序向导生成 MFC 应用程序时，WinMain()函数已被封装在 MFC 类库中了。对于大多数 Windows 程序，都必须从 CwinApp 类派生出自己的应用程序类，WinMain()就封装在 CwinApp 类里。