第 4 章　类与对象

类是实现 C++面向对象程序设计的基础，在 C++语言面向对象程序设计中占据着核心地位。利用它可以实现数据的封装、隐蔽，通过类的继承与派生，能够实现对问题的深入抽象描述。

本章介绍类的有关知识，包括类与对象的概念、构造函数与析构函数、类的组合、友元、类的静态成员、常对象与常成员函数，以及对象数组与对象指针等内容。

4.1　类与对象

4.1.1　类与对象的概念

从一般意义上讲，对象（object）是现实世界中的客观事物。类（class）是把具有相同属性的事物划分为一类，从而得出的抽象概念。在程序中，类是一组性质相同对象的程序描述，它由概括了一组对象共同性质的数据和函数组成。

面向对象程序设计中的对象，是系统中用来描述客观事物的一个实体，是构成系统的一个基本单位，对象由一组属性和一组行为构成。面向对象程序设计中的类，是具有相同属性和服务的一组对象的集合，它为属于该类的全部对象提供了抽象的描述。对象是类的实例，类是同种对象的抽象。

在面向过程的程序设计中，如 C 语言，函数是构成程序的基本单位。程序中的数据和处理数据的函数是相互分离的，当数据结构改变时，所有和该数据结构有关联的函数都要修改，程序的可维护性较差。

而在面向对象的程序设计中，如 C++语言，构成程序的基本单位是类。将描述一个对象的数据（在面向对象的术语中称为**属性**）和处理这些数据的函数（在面向对象的术语中称为**方法**）封装在一起就形成 C++的类。类中的大多数数据成员只能用本类中的成员函数进行处理，类通过简单的外部接口与外界联系，这样即使类中的数据结构发生改变，只要类的外部接口不变，使用该类的程序就不需要改变，使得软件开发和维护更加方便。

如具有确定大小和颜色的矩形都是一个个具体的对象，而将所有矩形的共同特点抽象出来，就是一个矩形类，这些共有的属性包括颜色（color），左上角坐标（left, top），长（length）和宽（width）等；对这些属性的处理包括改变矩形的颜色（SetColor）和大小（SetSize），移动矩形到新的位置（Move），绘出矩形（Draw）等。如果在 C 语言中就要声明一个结构，成员包括颜色，左上角坐标，长和宽，然后再定义独立的各种函数。而在 C++中就可以将矩形的这些属性和方法作为一个整体，封装在一起形成一个矩形类。

4.1.2　类的声明

类是一种用户自定义的数据类型，在 C++中用关键字 class 声明，它的一般定义格式如下：

```
class　类名
{
```

```
    private：
        私有数据成员和成员函数；
    protected：
        保护数据成员和成员函数；
    public：
        公有数据成员和成员函数；
};
```

其中 private、protected、public 表明跟随其后面成员的访问权限，分别为私有成员、保护成员和公有成员。其中的成员函数可以直接写出函数定义，也可以在类中只写函数原型，然后在类的外面写出函数定义。

例 4.1　定义一个矩形类 CRect，其数据成员包括颜色，左上角坐标，长和宽，其函数成员包括改变矩形的颜色（SetColor）和大小（SetSize），移动矩形到新的位置（Move），绘出矩形（Draw）。

```
class CRect
{
private:
    char color[10];
    int left;
    int top;
    int length;
    int width;
public:
    void SetColor(char *c);
    void SetSize(int l, int w);
    void Move(int t,int l);
    void Draw();
};
```

在 CRect 类中，定义了 5 个数据成员和 4 个成员函数，其中成员函数只给出函数的原型，即声明，还需要在类的外面给出函数的定义。

函数 SetColor()设置矩形的颜色，函数 SetSize()设置矩形的大小，即长和宽，函数 Move()将矩形的左上角移动到指定的点，函数 Draw()输出矩形的属性值，下面给出这 4 个成员函数的定义。

本书中类的命名规则是：第一个字母为大写字母 C，后面是代表具体类含义的单词，单词的第一个字母大写，其余字母小写，如 CRect。

成员函数的命名由表示具体含义的单词组成，可以是一个单词，也可以是多个单词的组合，每个单词的第一个字母大写，其余字母小写，如 SetSize()。

```
void CRect::SetColor(char *c)
{
    strcpy(color, c);
}
void CRect::SetSize(int l, int w)
{
    length=l;
    width = w;
```

```
}
void CRect::Move(int t,int l)
{
    top = t;
    left = l;
}
void CRect::Draw()
{
    cout << "矩形左上角坐标为(" << left << "," << top << ")" << endl;
    cout << "矩形长和宽分别为" << length << "," << width << endl;
    cout << "矩形的颜色是" << color << endl;
}
```

在类外进行函数定义时，为了指明函数是 CRect 类的成员函数，需要在函数名前用类名进行限制，其格式为：

类名::函数名（参数表）

如：CRect**::**SetColor(char *c)

我们将符号“**::**”称为域运算符。为了测试一下上面定义的矩形类，编写一个简单的主函数对其进行测试。

```
void main()
{
    CRect r;                          //定义 CRect 类的对象 r
    r.SetColor("Red");
    r.Move(10,20);
    r.SetSize(100,200);
    r.Draw();
    r.Move(50,50);
    r.SetColor("Blue");
    r.Draw();
}
```

函数的第一行定义了一个 CRect 类的对象 r，与定义普通变量类似，其一般格式为：

类名　对象名 1，对象名 2，……；

定义了对象之后，就可以访问对象的公有成员，如 r.SetColor("Red")，访问对象公有成员的一般格式为：

对象名.公有成员函数名（参数表）

对象名.公有数据成员名

为了使用 cout 和函数 strcpy()，需要加入以下两行文件包含语句：

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

将上面定义的类和主函数放在一个 C++源程序文件中，就可以编译，运行，查看运行结果，分析输出的内容。

程序运行结果为：

```
矩形左上角坐标为(20,10)
矩形长和宽分别为 100,200
矩形的颜色是 Red
```

```
矩形左上角坐标为(50,50)
矩形长和宽分别为 100,200
矩形的颜色是 Blue
```

4.1.3　成员的访问控制

在类的结构中，有数据成员和成员函数。在程序中，对这些成员的使用受成员访问权限的限制。类成员的访问权限分为三个等级，分别是私有访问权限（private），保护访问权限（protected）和公有访问权限（public）。

在关键字 private 后面声明的数据成员或成员函数具有私有访问权限，只允许类中的成员函数访问，其他函数不能访问。

在关键字 protected 后面声明的数据成员或成员函数具有保护访问权限，将在第 5 章“继承与派生”中详细介绍。

在关键字 public 后面声明的数据成员或成员函数具有公有访问权限，在任何函数中都可以访问。

关键字 private、protected 和 public 的作用范围是从该关键字开始，直到遇到下一个关键字之一结束。例如在上面的矩形类中，private 的作用范围是从 char color[10]开始一直到 int width，之后遇到 public，private 的作用结束，进入 public 控制的区域。当然如果在之后还需要声明私有成员，可以再使用 private 关键字。

为了测试一下私有成员的访问特点，将前面的主函数修改如下：

```
void main()
{
    CRect r;
    strcpy(r.color, "Red");          //错误，在 main()函数中不能访问私有成员
    r.top = 10;                      //错误，在 main()函数中不能访问私有成员
    r.left = 20;                     //错误，在 main()函数中不能访问私有成员
    r.SetSize(100,200);
    r.Draw();
    r.Move(50,50);
    r.SetColor("Blue");
    r.Draw();
}
```

重新编译，上面的三行代码会发生编译错误，不能访问类中的私有成员。因为 color、top、left 都是矩形类的私有成员，在这里不能访问。

如果不指定访问权限，在 class 中声明的成员默认为私有的。如上面的矩形类可以声明如下：

```
class CRect
{
    char color[10];
    int left;
    int top;
    int length;
    int width;
public:
    void SetColor(char *c);
```

```
    void SetSize(int l, int w);
    void Move(int t,int l);
    void Draw();
};
```

前 5 个数据成员没有指定访问权限，被认为是具有私有访问权限，和 4.1.2 节中的声明是完全一样的。在程序设计中，为了提高程序的清晰性，我们提倡使用关键字明确指明成员的访问权限。

事实上，类也可以使用关键字 struct 声明，strust 与 class 的区别是：如果不指定访问权限，前者默认的访问权限是公有的，而后者是私有的。因此也可以用 struct 声明前面的矩形类。

```
struct CRect
{
    void SetColor(char *c);
    void SetSize(int l, int w);
    void Move(int t,int l);
    void Draw();
private:
    char color[10];
    int left;
    int top;
    int length;
    int width;
};
```

这种声明与前面用 class 声明类的结果完全相同。

4.1.4 类的成员函数

1. 类成员函数的定义方式

类的成员函数可以在类中直接定义，也可以在类外部定义，在上面的例子中就是在类外定义的成员函数，其一般格式为：

```
函数类型  类名::成员函数名（参数说明）
{
   函数体
}
```

成员函数也可以直接在类中定义，例如可以将矩形类的成员函数 SetColor()直接定义在类中：

```
class CRect
{
private:
    char color[10];
    int left;
    int top;
    int length;
    int width;
public:
    void SetColor(char *c)
```

```
    {
        strcpy(color, c);
    }
    void SetSize(int l, int w);
    void Move(int t,int l);
    void Draw();
};
```

2. 内联成员函数

与一般函数一样，类的成员函数也可以是内联函数，欲使一个成员函数成为内联函数，有两种方法，一种方法就是将成员函数的定义直接写在类中，如上面的函数 SetColor()，如果函数足够简单，C++编译系统会将类中定义的成员函数当作内联函数处理。另一种方法就是在定义函数时用关键字 inline 指定为内联函数，如下例：

```
inline void CRect::SetColor(char *c)
{
    strcpy(color, c);
}
```

3. 带默认参数值的成员函数

与一般函数一样，类的成员函数也可以带有默认的参数值，需要指出的是，默认参数值只能在声明或定义中的一处给出，而不能在两处都给出默认值，如在类中的函数声明已经给出默认参数值：

```
void SetSize(int l=100, int w=100);
```

则在函数定义时就不能再给出默认值。同样如果在定义时给出了默认值：

```
void CRect::SetSize(int l=100, int w=100)
{
    length=l;
    width = w;
}
```

在声明处就不能再给出默认值了。

4.2　构造函数与析构函数

类和对象的关系相当于简单数据类型与它的变量的关系，在定义变量时可以为变量提供初始值，即初始化，同样在定义对象时也可以为对象的属性提供初始值。在定义对象时为数据成员设置初始值，称为对象初始化。C++提供了构造函数，可以对对象进行初始化。

与变量一样，在函数内部定义的对象为局部对象，当程序运行离开对象所在的函数后，该对象就消失，由于类中的数据成员可以有指针，对象中可能会有动态分配的内存，就需要在对象消失前将内存释放。C++提供了析构函数，可以进行内存释放等清理工作。

4.2.1　构造函数

1. 构造函数的特点

构造函数也是类的一个成员函数，除具有一般成员函数的特征之外，还有以下一些特殊的性质。

（1）构造函数的函数名与类名相同。

（2）不能定义构造函数的类型（即不能指明构造函数返回值的类型）。

（3）构造函数应声明为公有函数。

（4）构造函数不能在程序中调用，在对象创建时，构造函数被系统自动调用。

2. 构造函数的作用

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其他对象的特征。构造函数完成的是一个从一般到具体的过程，它在对象被创建的时候由系统自动调用。

下面为 CRect 类添加构造函数。

```
class CRect
{
private:
    char color[10];
……
public:
    CRect( );
    CRect(char *c, int t, int left, int len, int wid);
    void SetColor(char *c);
……
};
```

上面的 CRect 类中有两个构造函数，其中一个没有参数，另一个有 5 个参数，它们之间是重载关系，下面给出它们的定义：

```
CRect::CRect()
{
    strcpy(color, "Black");
    top = 0;
    left = 0;
    length = 0;
    width = 0;
}
CRect::CRect(char *c, int t, int lef, int len, int wid)
{
    strcpy(color, c);
    top = t;
    left = lef;
    length = len;
    width = wid;
}
```

在定义对象时，如果不给出参数，就自动调用第一个构造函数，如果给定 5 个参数，就自动调用第二个构造函数。

第一个构造函数没有参数，就将矩形的颜色置为 Black，左上角坐标及长和宽都置为 0。

下面将主函数改动如下：

```
void main()
{
```

```
    CRect r1;
    CRect r2("red", 10,10,100,100);
    CRect r3("green", 200,200,50,50);
    r1.Draw();
    r2.Draw();
    r3.Draw();
}
```

第一行定义对象 r1，由于没有提供参数，系统自动调用没有参数的构造函数，将 r1 的五个数据成员分别初始化为"Black"、0、0、0、0；第二行定义对象 r2，提供 5 个参数，系统自动调用有参数的构造函数，用提供的参数"red"、10、10、100、100 分别初始化 r2 的 5 个数据成员；第三行定义对象 r3，提供 5 个参数，系统自动调用有参数的构造函数，用提供的参数"green"、200、200、50、50 分别初始化 r3 的 5 个数据成员，后三行分别用三个对象调用成员函数 Draw()，输出三个对象中的数据成员的值。

如果类中的数据成员包含常量或引用，则在构造函数中要为它们初始化。C++是通过初始化表的方式为类中的常量和引用初始化的。下面通过实例介绍如何使用初始化表对类中的常量和引用初始化。

例 4.2　常量成员及引用成员的初始化。

```
#include <iostream>
using namespace std;
class A
{
private:
    const double PI;
    int b;
    int &c;
public:
    A(int x):PI(3.14),c(b)
    {
        b=x;
    }
    void Output()
    {
        cout << PI << "," << b   << "," << c << endl;
    }
};
void main()
{
    A x(10);
    x.Output();
}
```

构造函数冒号后面的部分是初始化表，PI(3.14)表示将 PI 初始化为 3.14，c(b)表示用 b 初始化 c，即 c 是 b 的引用（c 是 b 的别名）。

主函数中定义类 A 的对象 x，并为构造函数提供参数 10，因此在构造函数中将 b 赋值为 10，由于 c 是 b 的引用（别名），因此在 Output()函数中输出 c 的值应与 b 的值相同，也是 10。

程序运行结果如下：

3.14，10，10

4.2.2 析构函数

与构造函数对应的是析构函数。当一个对象消失，或用 new 创建的对象用 delete 删除时，系统自动调用类的析构函数。

1. 析构函数的特征

（1）析构函数名字为符号“~”加类名。

（2）析构函数没有参数，不能指定返回值类型。

（3）一个类中只能定义一个析构函数，所以析构函数不能重载。

（4）当一个对象作用域结束时，系统自动调用析构函数。

例如 CRect 类的析构函数的声明为：~CRect( );

定义为：

```
CRect ::~CRect( )
{
    ……
}
```

2. 析构函数的作用

与构造函数相反，析构函数的作用是：在删除一个对象前被调用，释放该对象成员的内存空间，以及其他一些清理工作。

例 4.3 设计一个简单的字符串类，类中有两个数据成员，分别表示字符串的长度和字符串的内容，有一个构造函数和一个析构函数，函数 GetLength()返回字符串长度，函数 GetContents()获得字符串的内容，重载函数 SetContents()给字符串设置值。

```
#include <iostream>
#include <string>
using namespace std;
class   CString
{
private:
     int   length;
     char *contents;
public:
    CString();            //构造函数
    ~CString();           //析构函数
    int   GetLength();
    void GetContents(char *str);
    void   SetContents(int len, char *cont);
    void   SetContents(char *cont);
};
CString::CString()
{
     length = 0;
     contents = NULL;
```

```
        cout << "字符串对象初始化" << endl;
}
```

在构造函数中，将字符串的长度置为 0，字符串内容置为空 NULL，并输出一个信息“字符串对象初始化”。

```
CString::~CString()
{
        cout << contents << "被销毁" << endl;
        if(contents != NULL)
                delete contents;
}
```

在析构函数中，首先输出字符串内容和“被销毁”的信息，然后判断字符串是否为空，如果不为空，用 delete 释放内存空间。

```
int    CString::GetLength()
{
        return    length;
}
void CString::GetContents(char *str)
{
        strcpy(str, contents);
}
```

函数 GetContents()将字符串内容复制到函数参数 str 中。

```
void CString::SetContents(int len, char *cont)
{
        length = len;
        if(contents != NULL)
                delete    contents;
        contents = new char[len+1];
        strcpy(contents,cont);
        cout << "两个参数的 SetContents 函数" << endl;
}
void CString::SetContents(char *cont)
{
        length = strlen(cont);
        if(contents != NULL)
                delete    contents;
        contents = new char[length+1];
        strcpy(contents,cont);
        cout << "一个参数的 SetContents 函数" << endl;
}
```

重载函数 SetContents()都是将要设置的字符串长度赋给数据成员 length，然后判断原来数据成员 contents 是否已经有数据（即已经不为空 NULL 了），如果已不为空，则先释放原来的内存，再根据新字符串的长度重新申请内存。

最后编写一个主函数，测试一下 CString 的工作情况。在主函数中先定义两个 CString 类的对象 str1 和 str2，然后分别用一个参数和两个参数调用 SetContents()函数，最后用成员函数

GetLength()和 GetContents()将字符串的长度和内容读出，显示出来。

```
void main()
{
    CString str1,str2;              //两次调用构造函数
    str1.SetContents("第一个字符串");                    //调用有一个参数的 SetContents 函数
    str2.SetContents(20, "第二个字符串两个参数");    //调用有两个参数的 SetContents 函数
     int   i = str1.GetLength();
    char string[100];
    str1.GetContents(string);
    cout << i << "    "<< string << endl;
     i = str2.GetLength();
    str2.GetContents(string);
    cout << i << "    " << string << endl;
}
```

程序运行结果如下：

```
字符串对象初始化
字符串对象初始化
一个参数的 SetContents 函数
两个参数的 SetContents 函数
12   第一个字符串
20   第二个字符串两个参数
第二个字符串两个参数被销毁
第一个字符串被销毁
```

程序的第一行定义两个 CString 类的对象，因此两次调用构造函数，输出两行“字符串对象初始化”，第二行调用函数 SetContents()，因为提供一个参数，调用的是一个参数的 SetContents()函数，第三行调用两个参数的 SetContents()函数，然后对象 str1 调用成员函数 GetLength()得到它的长度，调用成员函数 GetContents()得到它的内容，并输出，然后再用对象 str2 分别调用这两个成员函数得到 str2 的长度和内容，并输出出来，最后函数结束之前两个对象消失，分别调用析构函数。

我们注意到对象 str2 的定义在后，但销毁在 str1 之前，即对象销毁的顺序和创建的顺序正好相反。

4.2.3 拷贝构造函数

拷贝构造函数可以实现用一个已经存在的对象初始化新对象，拷贝构造函数的参数为该类对象的引用。

例 4.4 设计一个复数类，两个数据成员分别表示复数的实部（real）和虚部（imag），三个构造函数分别在不同的情况下初始化对象，函数 Set()设置复数实部和虚部的值，函数 Print()输出复数，函数 Add()和函数 Sub()分别实现复数的加减法运算。

```
#include "iostream"
using namespace std;
class   CComplex
{
private:
```

```
    double real;
    double imag;
public:
    CComplex();
    CComplex(double r, double i);
    CComplex(CComplex &c);
    void Set(double r, double i);
    void Print();
    CComplex Add(CComplex c);
    CComplex Sub(CComplex c);
};
CComplex::CComplex()
{
    real = 0.0;
    imag = 0.0;
}
CComplex::CComplex (double    r, double i)
{
    real = r;
    imag = i;
}
CComplex::CComplex (CComplex &c)
{
    real = c.real;
    imag = c.imag;
}
// set thc value of a CComplex
void CComplex::Set(double r, double i)
{
    real = r;
    imag = i;
}
// display a complex
void CComplex::Print()
{
    cout << "(" << real << "," << imag << ")" << endl;
}
// return the result of adding two CComplexes
CComplex CComplex::Add(CComplex c)
{
    CComplex temp;
    temp.real = real + c.real;
    temp.imag = imag + c.imag;
    return temp;
}
// return the result of substracting two CComplexes
```

```
CComplex CComplex::Sub(CComplex c)
{
    CComplex temp;
    temp.real = real - c.real;
    temp.imag = imag - c.imag;
    return temp;
}
// main program
void main(void)
{
    CComplex    a, b(3.0,4.0), c;
    CComplex    d(b);
    cout << "a = ";
    a.Print();
    cout << "b = ";
    b.Print();
    cout << "d = ";
    d.Print();
    c = b.Add(d);
    d = a.Sub(d);
    cout << "c = ";
    c.Print();
    cout << "d = ";
    d.Print();
}
```

在主函数中，首先定义 4 个复数类的对象，其中对象 a 和 c 没有参数，调用没有参数的构造函数，将实部和虚部都设置为 0，对象 b 有两个实型参数，因此调用具有两个实型参数的构造函数，对象 d 的初始化参数是复数类的对象，因此调用拷贝构造函数。

```
CComplex::CComplex (CComplex &c)
{
    real = c.real;
    imag = c.imag;
}
```

将对象 b 作为参数传递给拷贝构造函数，形参 c 就是对象 b 的引用（也就是 b 的别名），用其实部和虚部的值初始化新构造的对象。

程序运行结果如下：

```
a = (0, 0)
b = (3, 4)
d = (3, 4)
c = (6, 8)
d = (-3, -4)
```

4.3 类的组合

在面向对象的程序设计中，可以将复杂的问题分解，把复杂的对象分解为若干简单对象

的组合，例如线段对象可以由两个点对象组成（表示线段的两个端点）。

类的组合就是在一个类中内嵌其他类的对象作为成员，因为内嵌对象是该类对象的组成部分，当创建该对象时，其内嵌对象也被自动创建。因此会调用内嵌对象类的构造函数，如果内嵌对象所属类的构造函数有参数，就必须为其内嵌对象提供参数。

在 C++中是通过构造函数的初始化表为内嵌对象初始化的。组合类带有初始化表的构造函数的定义格式为：

```
类名::构造函数（参数表）：内嵌对象 1（参数表 1），内嵌对象 2（参数表 2），…
{
    构造函数体
}
```

有了初始化表后，在构造组合类的对象时就会用内嵌对象后面的参数表初始化该内嵌对象，组合类构造函数的执行顺序为：

（1）按内嵌对象的声明顺序依次调用内嵌对象的构造函数。

（2）然后执行组合类本身的构造函数。

例 4.5　点类 CPoint 和线段类 CLine。

```
#include <iostream>
#include <math>
using namespace std;
class CPoint
{
public:
    CPoint(int x=0, int y=0)
    {
        X=x;
        Y=y;
        cout << " CPoint  构造函数被调用" << endl;
    }
    CPoint(CPoint &p);
    int GetX()
    {
        return X;
    }
    int GetY()
    {
        return Y;
    }
private:
    int X,Y;
};

CPoint::CPoint(CPoint &p)
{
    X = p.X;
    Y = p.Y;
```

```
        cout << " CPoint  拷贝构造函数被调用" << endl;
        cout << "(" << X << "," << Y << ")" << endl;
}

class CLine
{
public:
        CLine(CPoint p1, CPoint p2);
        float GetDistance();
private:
        CPoint start;
        CPoint end;
};
CLine::CLine(CPoint ps, CPoint pe): start(ps), end(pe)
{
        cout << "CLine  构造函数被调用" << endl;
}
float CLine::GetDistance()
{
        double x = double (end.GetX() - start.GetX() );
        double y = double (end.GetY() - start.GetY() );
        return (float) sqrt(x*x + y*y );
}

void main()
{
        CPoint p1(1,1), p2(4,5);
        CLine l(p1, p2);
        cout << "The distance is :";
        cout << l.GetDistance() << endl;
}
```

运行结果如下：

```
CPoint  构造函数被调用
CPoint  构造函数被调用
CPoint  拷贝构造函数被调用
（4，5）
CPoint  拷贝构造函数被调用
（1，1）
CPoint  拷贝构造函数被调用
（1，1）
CPoint  拷贝构造函数被调用
（4，5）
CLine  构造函数被调用
The distance is :5
```

在线段类 CLine 中，有两个点类的内嵌对象 start 和 end，分别表示线段的起点和终点，线段类构造函数有两个点类的参数，通过初始化表传递给两个内嵌对象，因为内嵌对象 start 在

end 之前声明，因此先构造 start 对象，再构造 end 对象，最后执行构造函数 CLine()中的语句。内嵌对象的构造顺序与初始化表中出现的先后顺序无关，即使将线段类的构造函数 CLine()改写成下面的样子，仍然是先构造 start 对象。

```
CLine::CLine(CPoint ps, CPoint pe): end(pe), start(ps)
{
    cout << "CLine  构造函数被调用" << endl;
}
```

下面分析一下运行结果。

主函数中的第一行，定义两个 CPoint 类对象 p1，p2，因此调用两次 CPoint 类的构造函数。接着定义一个 CLine 类的对象 l，并以 p1，p2 作为初始化参数，我们用图 4.1 来解释构造过程。

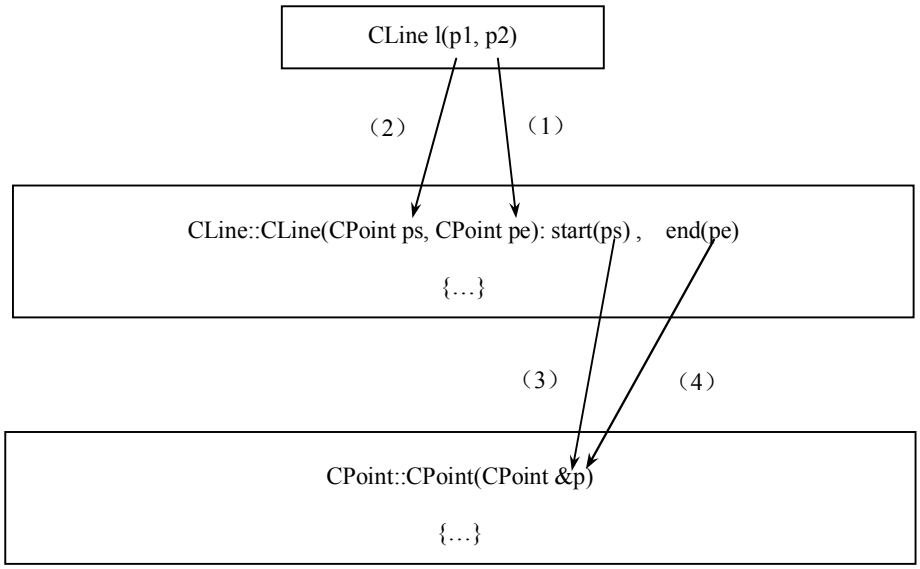

图 4.1　CLine 类的构造过程

在构造 CLine 类的对象 l 时，将实参 p1 和 p2 分别传递给形参 ps 和 pe，这时分别用 p1 和 p2 构造 ps 和 pe，在 C 或 C++中，函数参数的求解是从右至左的，因此先用 p2 构造 pe，该点的坐标为（4，5），然后用 p1 构造 ps，该点的坐标为（1，1）。ps 和 pe 构造完之后，再分别用 ps 和 pe 构造 start 和 end，因 start 的声明在 end 的声明之前，所以先构造 start，坐标为（1，1），再构造 end，坐标为（4，5）。最后执行 CLine 类自己的构造函数。

4.4　友元

类的主要特点是数据的封装与隐藏，该类之外的任何函数都不能对这个类的 private 部分进行存取。如果某个类外的函数需要访问该类的私有成员，只能通过类中的公有成员函数访问，非常不方便。为了解决这个问题，在 C++中提供了友元。

友元提供了在不同类的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制，友元分为友元函数和友元类两种。

在一个类中，可以利用关键字 friend 将其他函数或类声明为友元。如果友元是一般函数或类的成员函数，称为友元函数；如果友元是一个类，则称为友元类。友元类的所有成员函数都自动成为友元函数。

4.4.1 友元函数

定义友元函数时，只要在函数原型前加入关键字 friend，并将函数原型放在类中，格式为：

friend 类型标识符 友元函数名（参数列表）;

友元函数可以是一个普通函数，也可以是其他类的成员函数，在其函数体中可以通过对象名直接访问这个类的私有成员。

例 4.6 定义点类 CPoint，写一个函数计算两点之间的距离。

```
#include <iostream>
#include <math>
using namespace std;
class CPoint
{
public:
    CPoint(int x=0, int y=0);
    int GetX();
    int GetY();
private:
    int X,Y;
};

CPoint::CPoint(int x, int y)
{
    X=x;
    Y=y;
};
int CPoint::GetX()
{
    return X;
}
int CPoint::GetY()
{
    return Y;
}

double GetDistance(CPoint start, CPoint end)
{
    int x1,y1,x2,y2;
    double d;
    x1 = start.GetX();
    y1 = start.GetY();
    x2 = end.GetX();
    y2 = end.GetY();
```

```
    d = sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) );
    return d;
}

void main()
{
    CPoint p1(1,1), p2(4,5);
    double d;
    d = GetDistance(p1,p2);
    cout << "The distance is :" << d << endl;
}
```

由于 X，Y 是 CPoint 类的私有成员，在函数 GetDistance()中不能通过对象直接访问，只能通过类的公有函数 GetX()和 GetY()访问，如果需要经常计算两个点之间的距离，就要频繁地调用函数 GetX()和 GetY()，将降低程序的效率，为此可以将函数 GetDistance()声明为 CPoint 类的友元。将 CPoint 类修改如下：

```
class CPoint
{
public:
    CPoint(int x=0, int y=0);
    int GetX();
    int GetY();
    friend double GetDistance(CPoint start, CPoint end);
private:
    int X,Y;
};
```

将函数 GetDistance()声明为 CPoint 类的友元函数之后，就可以在函数 GetDistance()中直接通过 CPoint 类的对象访问它的私有成员 X，Y。函数 GetDistance()现在可以写成如下的形式：

```
double GetDistance(CPoint start, CPoint end)
{
    double d;
    d = sqrt( (end.X-start.X)*(end.X-start.X) + (end.Y-start.Y)*(end.Y-start.Y) );
    return d;
}
```

运行结果如下：

```
The distance is : 5
```

4.4.2　友元类

如果一个类（如类 A）的很多成员函数都需要经常访问另一个类（如类 B）的私有成员，就需要逐个将类 A 中的这些成员函数声明为类 B 的友元，这样做起来比较麻烦，为此可以将类 A 声明为类 B 的友元。

若类 A 为类 B 的友元类，则类 A 的所有成员函数都是类 B 的友元函数，都可以访问 B 类的私有成员。声明友元类的语法形式为：

```
Class   B
{
```

```
    ……
    friend class A;     // 声明 A 为 B 的友元类
    ……
};
```

例 4.7 友元类的使用。

```
#include <iostream>
using namespace std;
class A
{
private:
    int x;
public:
    void Display()
    {
        cout << x << endl;
    }
    int Getx()
    {
        return x;
    }
    friend class B;
};

class B
{
private:
    A a;
public:
    void Set(int i);
    void Display();
};

void B::Set(int i)
{
    a.x=i;
}
void B::Display()
{
    cout << a.x << endl;
}

void main()
{
    B b;
    b.Set(10);
    b.Display();
}
```

运行结果如下：

10

类 B 中，有一个成员是类 A 的对象，因此在构造 B 类的对象时，也同时构造了这个成员对象，为了能够在 B 类的成员函数中直接访问 A 类的私有数据成员，将 B 类声明为 A 类的友元类。

因为将 B 类声明为 A 类的友元，所以在 B 类的成员函数 Set()和 Display()中，可以访问 A 类的私有成员 x。

友元关系是单向的，在上面的例子中，B 类是 A 类的友元，所以 B 类的成员函数可以访问 A 类的私有成员，但 A 类不是 B 类的友元，A 类的成员函数不能访问 B 类的私有成员。

友元关系是不能传递的，即如果 A 类是 B 类的友元，B 类是 C 类的友元，并不能推断 A 类是 C 类的友元。

4.5　静态成员

类相当于一个数据类型，当创建一个类的对象时，系统就为该对象分配一块内存单元来存放类中的所有数据成员，所以类的每个对象都有自己独立的存储空间。若在类中增加一个数据成员用以存放创建该类对象的总数，必然在每一个对象中都存储一副本，不仅冗余，而且每个对象分别维护这样一个“总数”，容易造成数据的不一致性。因此，比较理想的方法是类的所有对象共同拥有一个用于存放总数的数据成员，这就是下面要介绍的静态数据成员。

4.5.1　静态数据成员

类的普通数据成员在类的每一个对象中都拥有一个拷贝，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他对象的特征的需要。但是静态数据成员则不同，每个类只有一个拷贝，由该类的所有对象共同维护和使用，从而实现了同一类的不同对象之间的数据共享。

在定义类时，若在某个数据成员前加了关键字 static，则这个数据成员就是静态数据成员，静态数据成员的定义格式如下：

static　类型标识符　静态数据成员名；

在类的声明中仅仅对静态数据成员进行引用性说明，必须在文件作用域的某个地方用类名限定进行定义，这时也可以进行初始化，格式如下：

类型标识符　类名::静态数据成员名 = 初始值；

静态数据成员不属于任何一个对象，可以通过类名直接对它进行访问，一般的用法是：

类名::静态数据成员名

例 4.8　在 CStudent 类中添加静态数据成员，保存 CStudent 类的对象总数。

```
#include <iostream>
#include <string>
using namespace std;
class CStudent
{
private:
    int number;
```

```
    char name[10];
    static int total;
public:
    CStudent(int xh, char *xm);
    ~CStudent();
    int GetTotal();
    int GetNumber();

};

CStudent::CStudent(int xh, char *xm)
{
    number = xh;
    strcpy(name, xm);
    total++;
}

CStudent::~CStudent()
{
    total--;
}

int CStudent::GetTotal()
{
    return total;
}

int CStudent::GetNumber()
{
    return number;
}

int CStudent::total = 0;
void func();

void main()
{
    CStudent s1(10001, "AAAAAA" );
    cout << s1.GetNumber() << endl;
    cout << s1.GetTotal() << endl;
    CStudent s2(10002, "BBBBBB" );
    cout << s2.GetNumber() << endl;
    cout << s1.GetTotal() << endl;
    cout << s2.GetTotal() << endl;
    func();
    cout << s1.GetNumber() << endl;
    cout << s1.GetTotal() << endl;
```

```
}

void func()
{
    CStudent s3(10003, "CCCCCC" );
    cout << s3.GetNumber() << endl;
    cout << s3.GetTotal() << endl;
}
```

程序运行结果为：

```
10001
1
10002
2
2
10003
3
10001
2
```

在 CStudent 类中将 total 声明为静态成员，要在类的外面进行定义并初始化：

```
int CStudent::total = 0;
```

在构造函数中使 total 值加 1，即当创建一个 CStudent 类的对象时，total 值就加 1，在析构函数中使 total 值减 1，即当销毁一个 CStudent 类的对象时，total 值就减 1，因此 total 始终保存的是当前 CStudent 类的对象个数。

在主函数中的第一行定义了一个 CStudent 类对象 s1，学号和姓名分别是 10001 和“AAAAAA”，第二行程序输出该对象的学号，第三行输出 total 的值为 1。在主函数中的第四行又定义了一个 CStudent 类对象 s2，学号和姓名分别是 10002 和“BBBBBB”，第五行程序输出该对象的学号，第六行用对象 s1 调用函数 GetTotal()得到 total 的值为 2，第七行用对象 s2 调用函数 GetTotal()得到 total 的值也是 2，表明静态数据成员 total 是 s1 和 s2 共同维护的。然后程序进入函数 func()，在该函数中又定义了一个 CStudent 类对象 s3，这时 total 的值为 3。当程序离开函数 func()时，对象 s3 销毁，调用析构函数，total 值减 1，变为 2，回到主函数后，用对象 s1 调用函数 GetTotal()得到 total 的值为 2。

用图 4.2 表示 CStudent 对象的存储空间，每个对象都有自己的学号和姓名，但静态数据成员 total 只有一个，它不属于某个对象，而是所有对象共享的。

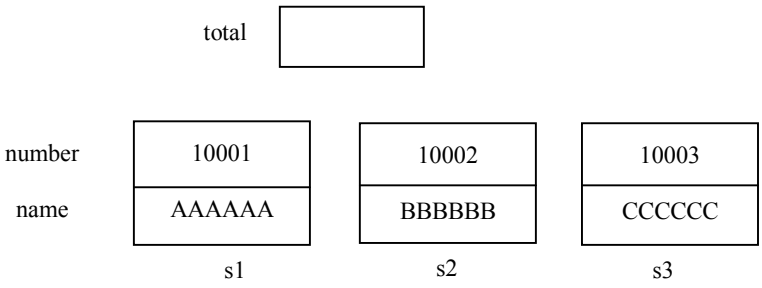

图 4.2　CStudent 对象的存储空间

虽然可以使用 **int CStudent::total = 0;**语句对类的静态数据成员进行定义和初始化，但如果在程序中使用 **CStudent::total** 来访问静态成员 **total**，就会出现错误。例如在主函数中使用下面的语句：

```
cout << CStudent::total << endl;
```

在编译时就会产生错误信息“不能访问 CStudebt 类的私有成员”，如果改写成下面的语句，通过类调用公有函数 GetTotal()得到 total 的值，仍然有语法错误“GetTotal()不是静态成员函数，调用非法”。

```
cout << CStudent::GetTotal() << endl;
```

不能用类名调用非静态成员函数。

也就是说如果不通过对象，没有办法得到私有静态成员变量的值。虽然在未创建对象时，静态变量就已经存在，但不能取得该变量的值。为了解决这个问题，可以使用静态成员函数。

4.5.2 静态成员函数

在定义类时，若在某个成员函数声明的前面加上关键字 static，这个函数就成为静态成员函数。

静态成员函数有以下几个特点：

对于公有的静态成员函数，可以通过类名或对象名来调用，而一般的非静态成员函数只能通过对象名来调用。

静态成员函数可以直接访问该类的静态数据成员和静态成员函数，不能直接访问非静态数据成员和非静态成员函数。

静态成员函数可以由类名通过符号“**::**”直接调用。

例 4.9 在 CStudent 类中添加静态成员函数。

```
#include <iostream>
#include <string>
using namespace std;
class CStudent
{
private:
    int number;
    char name[10];
    static int total;
public:
    CStudent(int xh, char *xm);
    ~CStudent();
    static int GetTotal();
    int GetNumber();
};

CStudent::CStudent(int xh, char *xm)
{
    number = xh;
    strcpy(name, xm);
```

```
    total++;
}

CStudent::~CStudent()
{
    total--;
}

int CStudent::GetTotal()
{
    return total;
}

int CStudent::GetNumber()
{
    return number;
}

int CStudent::total = 0;
void func();

void main()
{
    cout << CStudent::GetTotal() << endl;
    CStudent s1(10001, "AAAAAA" );
    cout << CStudent::GetTotal() << endl;
    CStudent s2(10002, "BBBBBB" );
    cout << CStudent::GetTotal() << endl;
    func();
    cout << CStudent::GetTotal() << endl;
}

void func()
{
    CStudent s3(10003, "CCCCCC" );
    cout << CStudent::GetTotal() << endl;
}
```

程序运行结果为：

```
0
1
2
3
2
```

将函数 GetTotal()声明为静态成员函数，就可以使用 **CStudent::GetTotal()**的方式直接调用了，这样即使未定义 CStudent 类的对象，也可以访问静态数据成员 **total** 了。

需要注意的是，将类的成员函数声明为静态的，是在声明成员函数时加上 static 关键字，

而在类外函数的定义处不能加 static 关键字。

另外，如果试图在静态成员函数中访问非静态数据成员，或非静态成员函数，都会产生语法错误。

例如将函数 GetTotal()改写为下面的形式就会出错。

```
int CStudent::GetTotal()
{
    cout << number;      // 语法错误，不能在静态成员函数中访问非静态数据成员
    return total;
}
```

因为非静态数据成员是属于某个对象的，但在没有创建对象时，就可以使用类名调用静态成员函数，而此时还不存在非静态数据成员，因此 C++禁止在静态成员函数中访问非静态数据成员。

4.6 常对象与常成员函数

在 C++中，可以定义基本数据类型的常量，如整型常量、实型常量等，也可以定义常对象，常对象的属性不能被修改。

为了确保常对象的属性不被修改，C++禁止用常对象调用普通的成员函数（因为普通的成员函数可能会修改对象的属性），为了能够对常对象进行访问，C++引入了常成员函数。

4.6.1 常对象

定义常对象格式是：

```
const  类名  对象名;
```

例 4.10 常对象调用普通成员函数产生的错误。

```
#include <iostream>
using namespace std;
class A
{
public:
    A( float a, float b)
    {
        x = a;
        y = b;
    }
    void Output()
    {
        cout << x << "," << y << endl;
    }
private:
    float x, y;
};

void main()
```

```
{
    const A   a(20, 20);
    a.Output();   //   错误，常对象不能调用普通成员函数
}
```

在主函数中定义常对象 a，然后试图调用成员函数 Output()输出成员 x，y 的值，编译时产生语法错误，因为常对象不能调用普通成员函数。

为了能够让常对象访问其数据成员，C++引入了常成员函数。

4.6.2　常成员函数

常成员函数是在类中声明或定义成员函数行后加 const 关键字，格式是：

```
函数类型　函数名(参数列表)　const；
```

例如：float area(float r) const；

例 4.11　使用常成员函数访问常对象中的数据成员。

```
#include <iostream>
using namespace std;
class A
{
public:
    A( float a, float b)
    {
        x = a;
        y = b;
    }
    void Output();
    void Output() const;
private:
    float x, y;
};

void A::Output()
{
    cout << "普通成员函数" << endl;
    cout << x << "," << y << endl;
}
void A::Output() const
{
    cout << "常成员函数" << endl;
    cout << x << "," << y << endl;
}

void main()
{
    A   a1(10, 20);
    const A   a2(30, 40);
    a1.Output();
```

```
    a2.Output();
}
```

程序运行结果如下：

```
普通成员函数
10，20
常成员函数
30，40
```

在类 A 中，有两个 Output()函数，一个是普通成员函数，一个是常成员函数。从程序运行结果，可以发现使用普通对象调用的是普通成员函数 Output()，使用常对象调用的是常成员函数 Output()。

使用常成员函数，应注意以下几点：

（1）const 是函数的一部分，在说明部分和实现部分都要加上关键字 const。

（2）在常成员函数中，不能更新对象的数据成员，也不能调用该类的普通成员函数。

（3）常对象只能调用常成员函数，不能调用该类的普通成员函数。普通对象既可以调用普通成员函数，也可以调用常成员函数，但会优先调用普通成员函数。

4.7 对象数组与对象指针

4.7.1 对象数组

数组的元素不仅可以是基本数据类型，也可以是自定义类型。例如，要存储和处理全体学生的信息，就可以建立一个学生类的对象数组。对象数组的每个元素是一个对象。

声明一个一维对象数组的语法形式是：

类名 数组名[常量表达式]；

与基本类型数组一样，在使用对象数组时也只能引用单个数组元素。每个数组元素都是一个对象，通过这个对象，便可以访问到它的公有成员，一般形式是：

数组名[下标].成员名；

对象数组的初始化过程，实际上就是调用构造函数对每一个元素对象进行初始化的过程。如果在声明数组时给每一个数组元素指定初始值，在数组初始化过程中就会调用形参类型相匹配的构造函数，例如：

```
CStudent  s[3] = {CStudent(10001, "AAAAAA" ),
                  CStudent(10002, "BBBBBB" ),
                  CStudent(10003, "CCCCCC")};
```

在执行时会先后三次调用带形参的构造函数分别初始化 s[0]，s[1]和 s[2]。在上面是以 Cstudent(10001, "AAAAAA")的形式为 s[0]初始化，以 Cstudent(10002, "BBBBBB")的形式为 s[1]初始化，以 Cstudent(10003, "CCCCCC")的形式为 s[2]初始化。下面利用对象数组求学生的平均年龄。

例 4.12 对象数组的应用。

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

```
class CStudent
{
private:
    int number;
    char name[10];
    int age;
public:
    CStudent(int xh, char *xm, int a);
    int GetAge();
};

CStudent::CStudent(int xh, char *xm, int a)
{
    number = xh;
    strcpy(name, xm);
    age = a;
}

int CStudent::GetAge()
{
    return age;
}

void main()
{
    int sum=0;
    CStudent s[5] = { CStudent(10001, "AAAAAA", 20),
                      CStudent(10002, "BBBBBB",22 ),
                      CStudent(10003, "CCCCCC",24 ),
                      CStudent(10004, "DDDDDD",26 ),
                      CStudent(10005, "EEEEEE",28 )
                    };
    for(int i=0; i<5; i++)
    {
        sum += s[i].GetAge();
    }
    cout << sum/5 << endl;
}
```

程序运行结果为：

```
24
```

程序定义了具有 5 个元素的对象数组，并且所有元素都已初始化，在循环中使用成员函数 GetAge()取得每一个数组元素的年龄，并累加到变量 sum 中，最后输出 5 个学生的平均年龄 sum/5。

4.7.2　对象指针

和基本数据类型的变量一样，每一个对象在初始化之后都会在内存中占有一定的存储空

间。因此，既可以通过对象名，也可以通过对象的地址来访问一个对象。对象指针就是用于存放对象地址的变量。对象指针遵循一般变量指针的各种规则，声明对象指针的一般语法形式为：

类名 *对象指针名；

例如

```
CStudent  *p;
CStudent  s(1001,"AAAAAA", 20);
p = &s;
```

通过对象指针访问成员的方法为：

对象指针名->成员名

例 4.13 使用对象指针访问成员，将例 4.12 中的主函数改写如下：

```
void main()
{
    int sum=0;
    CStudent *p[5];
    p[0] = new CStudent(10001, "AAAAAA", 20);
    p[1] = new CStudent(10002, "BBBBBB", 22);
    p[2] = new CStudent(10003, "CCCCCC", 24);
    p[3] = new CStudent(10004, "DDDDDD", 26);
    p[4] = new CStudent(10005, "EEEEEE", 28);

    for(int i=0; i<5; i++)
    {
        sum += p[i]->GetAge();
    }
    cout << sum/5 << endl;
    for(i=0; i<5; i++)
    {
        delete p[i];
    }
}
```

程序运行结果和前例相同。

4.8 this 指针

类有两种类型的函数成员，即普通函数成员和静态函数成员，普通函数成员比静态函数成员多了一个隐含参数 this，隐含参数 this 是普通函数成员的第一个参数，该参数的类型为指向此类对象的 const 指针（常指针）。

当一个对象调用一普通函数成员时，对象的地址作为函数的第一个实参首先压栈，通过压栈将实参传递给函数的隐含参数 this。

普通函数成员的 this 指针参数是隐含的，不需要显式说明。使用 this 指针可以方便地访问调用该函数成员的对象、对象的地址和对象的引用。

回顾一下前面的 CPoint 类，如果它的数据成员声明为小写的 x，y，而构造函数的参数也是小写的 x，y，就会出现问题，如果没有隐含参数 this 指针，就没有办法将参数的值赋值给

类的数据成员 x，y。

```
class CPoint
{
public:
    CPoint(int x=0, int y=0);
    int GetX();
    int GetY();
private:
    int x, y;
};
CPoint::CPoint(int x, int y)
{
    x=x;
    y=y;
};
```

在上面的构造函数中，因为函数的参数是函数的局部变量，在函数中优先访问，因此没有办法访问到类的同名数据成员 x，y，在语句“x = x;”中，赋值运算符两边的 x 都是函数的参数。

通过 this 指针很容易解决这个问题，将构造函数修改如下：

```
CPoint::CPoint(int x, int y)
{
    this->x=x;
    this->y=y;
};
```

将赋值运算符的左边改写为 this->x，表示类中的数据成员 x。this 是指向 CPoint 类对象的，到底是指向哪个对象呢，谁调用这个函数，this 就指向谁。如下面的定义：

```
CPoint   p(10,20);
```

是对象 p 调用构造函数，因此此时 this 指向的是对象 p。

例 4.14　定义一个二叉搜索树，在树中查找指定的节点（本例只是介绍 this 指针的用途，不是作为一个真正二叉搜索树的实用程序）。

```
#include <iostream>
using namespace std;
class CTree
{
private:
    int value;
    CTree *left, *right;
public:
    CTree(int v);
    ~CTree();
    int GetValue();
    void add(int v);
    CTree *find(int v);
};
```

二叉树的每个节点数据由左节点指针、右节点指针和该点的值组成，函数 GetValue()获得该点的值，函数 add()向二叉树添加一个节点，函数 find()查找值为 v 的节点。

构造函数用参数 v 初始化节点的值，并将其左节点和右节点都置为空。

```
CTree::CTree(int v)
{
    value = v;
    left = NULL;
    right = NULL;
}
```

析构函数使用递归的方法，将该节点以下的节点全部删除。

```
CTree::~CTree()
{
    if(left)
    {
        delete left;
        left = NULL;
    }
    if(right)
    {
        delete right;
        right = NULL;
    }

}

int CTree::GetValue()
{
    return value;
}
```

函数 add()将某个值插入到二叉树的合适的位置，本例建立的是二叉搜索树，节点没有重复的值，且该节点左边节点的值都小于该节点的值，该节点右边节点的值都大于该节点的值，仍然使用递归的方法将指定的值插入到合适的位置。

```
void CTree::add(int v)
{
    if(v==value)
        return;
    else if(v < value)
    {
        if(left != NULL)
            left->add(v);
        else
            left = new CTree(v);
    }
    else
    {
```

```
            if(right != NULL)
                right->add(v);
            else
                right = new CTree(v);
        }
    }
```

函数 find()也是使用递归的方法在树中查找指定的值，如查到就返回该节点的指针，否则返回 NULL。因为本函数要返回该节点的指针，因此必须使用 this 指针，否则没有办法实现。

```
    CTree* CTree::find(int v)
    {
        if(v==value)
            return this;
        else if(v < value)
        {
            if(left != NULL)
                return left->find(v);
            else
                return NULL;
        }
        else
        {
            if(right != NULL)
                return right->find(v);
            else
                return NULL;
        }
    }

    void main()
    {
        CTree *root;
        root = new CTree(10);

        root->add(20);
        root->add(2);
        root->add(6);
        root->add(35);
        root->add(15);

        CTree *n1, *n2;
        n1=root->find(6);
        if(n1)
            cout << n1->GetValue() << endl;
        else
            cout << "not found!" << endl;
        n2=root->find(7);
```

```
    if(n2)
        cout << n2->GetValue() << endl;
    else
        cout << "not found!" << endl;
    delete root;
}
```

程序运行结果为：

```
6
not found!
```

图 4.3 为本例建立的二叉搜索树。

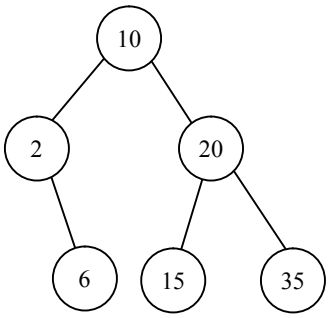

图 4.3　例 4.14 创建的二叉搜索树

4.9　程序实例

例 4.15　设计一个栈类。

栈是一种特殊的线性表，这种线性表只能在固定的一端进行插入和删除操作，允许插入和删除的一端称为栈顶，另一端称为栈底。一个新元素只能从栈顶一端进入，删除时，只能删除栈顶的元素。因此栈的运算规则是“先进后出”（或称“后进先出”）。

栈有三种基本操作：进栈（或入栈），退栈（或出栈）和获取栈顶元素。

对于顺序栈类，进栈操作的方法是先将进栈元素赋给栈顶，然后将栈顶元素下标加 1。出栈操作的方法是，将栈顶元素下标减 1。

```
#include<iostream>
using namespace std;
class Stack
{
private:
    int *data;          //存放栈元素的数组
    int top;            //栈顶元素的下标
    int max;            //栈中最多的数据个数
public:
    Stack(int size);
    ~Stack();
    bool push(int n);                       //入栈
```

```
    bool pop();                          //出栈并返回出栈元素
    int topElement() const;              //返回栈顶元素，但不出栈
    bool isEmpty() const;                //判断是否栈空
    bool isFull() const;                 //判断是否栈满
};

Stack::Stack(int size)
{
    max = size;
    data = new int[max];
    top = 0;
}
Stack::~Stack()
{
    delete []data;
}
bool Stack::push(int n)
{
    if(top<max)
    {
        data[top++]=n;
        return true;
    }
    else
    {
        cout << "栈满" << endl;
        return false;
    }
}
bool Stack::pop()
{
    if(top > 0)
    {
        top--;
        return true;
    }
    else
    {
        cout << "栈空" << endl;
        return false;
    }
}
int Stack::topElement()const
{
    if( top > 0)
        return data[top-1];
```

```
        else
        {
            cout << "栈为空" << endl;
            return 0;
        }
    }
    bool Stack::isEmpty()const
    {
        return top==0;
    }
    bool Stack::isFull()const
    {
        return top==max;
    }
    void main()
    {
        Stack st(10);
        for(int i=10; i<20; i++)
            st.push(i);
        cout << st.isFull() <<endl;
        cout << st.isEmpty() <<endl;
        for(i=0; i<10; i++)
        {
            cout << st.topElement() << "    ";
            st.pop();
        }
        cout << endl;
        cout << st.isFull() <<endl;
        cout << st.isEmpty() <<endl;
        st.pop();
        st.push(100);
        cout << st.isFull() <<endl;
        cout << st.isEmpty() <<endl;
    }
```

程序运行结果如下：

```
1
0
19   18   17   16   15   14   13   12   11   10
0
1
栈空
0
0
```

在主函数中首先定义顺序栈类对象 st，并将最大元素个数设置为 10，然后通过循环将 10 个元素压入栈。此时调用函数 isFull()判断是否栈满，返回真值（输出 1），调用函数 isEmpty()判断是否栈空，返回假值（输出 0）。再通过循环 10 次取栈顶元素输出并出栈，此时调用函数

isFull()判断是否栈满，返回假值（输出 0），调用函数 isEmpty()判断是否栈空，返回真值（输出 1），再调用函数 pop()试图出栈，给出提示信息“栈空”。调用函数 push()，再将一个数据压入栈，此时栈中有一个元素，既不满也不空，因此最后两行输出都是 0。

例 4.16　设计一个日期类，包含数据成员 year、month、day；成员函数有判断某年是否为闰年，某天是当年的第几天，是星期几。

分析：计算某天是当年的第几天，只要知道每月有多少天就容易计算了，如 1、3、5、7、8、10、12 月有 31 天，4、6、9、11 月有 30 天，如果是闰年 2 月有 29 天，否则 2 月有 28 天。

计算某天是星期几，首先计算从原点日子（公元前 1 年 12 月 31 日为星期日）到当前日期的天数 TotalDays，这个天数应该等于这段时间经历多少年的总天数加上当年的天数，然后计算 TotalDays 除以 7 的余数，余数是 0 为星期日、余数是 1 为星期一、……、余数是 6 为星期六。

可用下面的公式计算：

（（year – 1）* 365 + year / 4 – year / 100 + year / 400 + days）% 7;

其中 year– 1 是经历的整年数，year / 4 –　year / 100 + year / 400 是经历的闰年数，闰年一次就要增加一天，days 是当年经历的天数。

程序如下：

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

class Date
{
private:
    int year;
    int month;
    int day;
public:
    Date(int yy=0, int mm=0, int dd=0);
    void Set(int yy, int mm, int dd);
    bool IsLeapYear();
    int DayOfYear();
    void DayOfWeek();
};

Date::Date(int yy, int mm, int dd)
{
    year = yy;
    month = mm;
    day = dd;
}

void Date::Set(int yy, int mm, int dd)
{
    year = yy;
```

```
    month = mm;
    day = dd;
}

bool Date::IsLeapYear()
{
    if((year%4==0 && year%100!=0)||(year%400==0))
        return true;
    else
        return false;
}

int Date::DayOfYear()
{
    int days=0;
    for(int i=1; i<month; i++)
    {
        if((i<=7 && i%2!=0)||(i>=8 && i%2==0))
        {
            days += 31;
        }
        else if(i==2)
        {
            if(IsLeapYear())
                days += 29;
            else
                days += 28;
        }
        else
        {
            days += 30;
        }
    }
    return days+day;
}

void Date::DayOfWeek()
{
    int days;
    days = (year-1) *365+ ((year-1)/4) - ((year-1)/100) + ((year-1)/400);
    days += DayOfYear();
    switch(days % 7)
    {
    case 0: cout << "星期日" << endl;
            break;
    case 1: cout << "星期一" << endl;
```

```
                break;
        case 2: cout << "星期二" << endl;
                break;
        case 3: cout << "星期三" << endl;
                break;
        case 4: cout << "星期四" << endl;
                break;
        case 5: cout << "星期五" << endl;
                break;
        case 6: cout << "星期六" << endl;
                break;
        }
}
void main()
{
        Date d(2009,6,4);
        if(d.IsLeapYear())
                cout << "该年是闰年" << endl;
        else
                cout << "该年不是闰年" << endl;
        cout << "该天是当年的第 " << d.DayOfYear() << " 天" << endl;
        cout << "该天是 " ;
        d.DayOfWeek();

        d.Set(2008, 5, 18);
        if(d.IsLeapYear())
                cout << "该年是闰年" << endl;
        else
                cout << "该年不是闰年" << endl;
        cout << "该天是当年的第 " << d.DayOfYear() << " 天" << endl;
        cout << "该天是 " ;
        d.DayOfWeek();
}
```

程序运行结果如下：

```
该年不是闰年
该天是当年的第 155 天
该天是 星期四
该年是闰年
该天是当年的第 139 天
该天是 星期日
```

习题

4.1　在面向对象程序设计中，有属性和方法两个重要的概念，分别对应 C++类中的什么成员？

4.2 在C++类中，成员的访问权限有哪几种？用什么关键字指定？

4.3 类的静态数据成员与静态成员函数各有什么特点？

4.4 友元函数有什么特点？它与类的成员函数有什么异同？

4.5 写出程序运行结果。

```
#include <iostream>
using namespace std;
class   A
{
private:
      int   a;
public:
      A(int x):a(x)
      {
            cout<<"a="<<a<<endl;
      }
      ~A()
      {
            cout<<"a="<<a<<endl;
      }
};

class   B
{
      int   b;
      A   a1;
      A   a2;
public:
      B(int x,int y):a2(x),a1(y)
      {
            b=y;
            cout<<"b="<<b<<endl;
      }
      ~B()
      {
            cout<<"b="<<b<<endl;
      }
};
void   main()
{
      B   b1(2,3);
}
```

4.6 写出程序运行结果。

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

```
{
private:
    static int count;
public:
    test(){count++;}
    ~test(){count--;}
    static int getcount(){return count;}
};

int test::count=0;
void main()
{
    cout<<test::getcount()<<" objects exist\n";
    test test1;
    cout<<test::getcount()<<" objects exist\n";
    test *test2=new test;
    cout<<test::getcount()<<" objects exist\n";
    delete test2;
    cout<<test::getcount()<<" objects exist\n";
}
```

4.7　参考例 4.1，定义一个圆类，数据成员有颜色、圆心坐标、半径，成员函数有构造函数（有 4 个参数）、拷贝构造函数、设置圆的各种参数的函数、显示圆的各种参数的函数、计算圆的面积函数、计算圆的周长函数。并编写一个主函数对所定义的圆类进行测试。

4.8　定义一个点类，数据成员有点的坐标；成员函数有构造函数（有 2 个参数），拷贝构造函数，取得点坐标的函数。再编写一个矩形类，包括两个内嵌点类对象，分别表示矩形左上角坐标和右下角坐标，还有一个静态数据成员，用来记录当前矩形类对象的个数；成员函数有构造函数（注意初始化表要为内嵌对象提供初始化参数），计算矩形的周长和面积，显示矩形的左上角坐标和右下角坐标，一个静态函数用于访问静态数据成员。编写一个主函数对所定义的类进行测试。

4.9　定义一个 CStudent 类，数据成员包括学号、姓名和成绩；成员函数有构造函数、设置数据成员值的函数、读取数据成员值的函数等。在主函数中定义对象数组，再编写一个 CStudent 类的友元函数，计算平均成绩，查找最高成绩和最低成绩。