第3章 类和对象

类是C++语言支持面向对象思想的重要机制，是C++语言实现数据隐藏和封装的基本单元，它将一个数据结构与一个操作紧密地结合起来，是实现面向对象其他特性的基础。类对象是类的实例，用类对象模拟现实世界中的事物比用一般的数据变量更加确切。

3.1 类

类是C++语言的数据抽象和封装机制，它描述了一组具有相同属性（数据成员）和行为特征（成员函数）的对象。在系统实现中，类是一种共享机制，它提供了本类对象共享的操作实现。类是代码复用的基本单位，它可以实现抽象数据类型、创建对象、实现属性和行为的封装。例如，在学生中，有小学生、中学生、大学生等不同类型，但在描述时，可找出各种类型学生的共性，将其归为一类，即学生类。

对象是类的实例。类是对一组具有相同特征的对象的抽象描述，所有这些对象都是这个类的实例。例如，对于学籍管理系统，学生是一个类，而每个具体的学生则是学生类的一个实例。在程序设计语言中，类是一种数据类型，而对象是该类型的变量，变量名即是某个具体对象的标识。类和对象的关系相当于普通数据类型与其变量的关系。类是一种逻辑抽象概念。声明一个类只是定义了一种新的数据类型，声明对象才真正创建了这种数据类型的物理实体。由同一个类创建的各个对象具有完全相同的数据结构，但它们的数据值可能不同。

类提供了完整的解决特定问题的能力，因为类描述了数据结构（对象属性）、算法（对象行为）和外部接口（消息协议）。

在C++语言中，一个类的定义包含数据成员和成员函数两部分内容。数据成员定义该类对象的属性，不同对象的属性值可以不同；成员函数定义了该类对象的操作，即行为。

3.1.1 类的定义

类由三部分组成：类名、数据成员和成员函数。类定义的一般格式如下所示。

```
class 类名
{
private:
    //私有数据成员和成员函数
public:
    //公有数据成员和成员函数
protected:
    //受保护的数据成员和成员函数
};
```

下面是有关类定义的几点说明。

(1) class 是定义类的关键字,类名是一种标识符,必须符合 C++语言标识符的命名规则。一般情况下,类名的首字母大写,以区别于普通的变量和对象。花括号内是类的定义体部分,说明该类的成员,类的成员包括数据成员和成员函数。

(2) 类成员的三种访问控制权限。

类成员有三种访问控制权限,分别是 private(私有成员)、public(公有成员)、protected(受保护成员),在每一种访问控制权限下,均可以定义数据成员和成员函数。

① 私有成员 private:私有成员是在类中被隐藏的部分,它往往是用来描述该类对象属性的一些数据成员,私有成员只能由本类的成员函数或某些特殊说明的函数(如第 4 章中的友元函数)访问,而类的外部函数无法访问私有成员,实现了访问权限的有效控制,使数据得到有效的保护,有利于数据的隐藏;使内部数据不被任意地访问和修改,也不会对该类以外的其余部分造成影响;使模块之间耦合程度降到最低。私有成员若处于类声明中的第一部分,可省略关键字 private。

② 公有成员 public:公有成员对外是完全开放的,公有成员一般是成员函数,它提供了外部程序与类的接口功能,用户通过公有成员访问该类中的数据。

③ 受保护成员 protected:只能由该类的成员函数、友元、公有派生类成员函数访问的成员。受保护成员与私有成员在一般情况下含义相同,它们的区别体现在类的继承中对产生的新类的影响不同,具体内容将在第 5 章中介绍。

默认访问控制(未指定 private、protected、public 访问权限)时,系统默认为私有成员。

类具有封装性,C++语言中的数据封装通过类来实现,外部不能随意访问权限说明为 protected 和 private 的成员。

(3) 由于类的公有成员提供了一个类的接口,所以一般情况下,先定义公有成员,再定义保护成员和私有成员,这样可以在阅读时首先了解这个类的接口。当然,类声明中的三种访问控制权限说明可以按任意顺序出现任意次。

(4) 数据成员可以是任何数据类型,但是不能用自动(auto)、寄存器(register)或外部(extern)进行说明。

(5) 注意在定义类时,不允许初始化数据成员,下面的定义是错误的。

```
class A
{
private:
    int n = 0;                  //错误
    int m = 5;                  //错误
…
};
```

(6) 结构体和类的区别。

C 语言中的结构体只有数据成员,无成员函数。C++语言中的结构体可有数据成员和成员函数。在默认情况下,结构体中的数据成员和成员函数都是公有的,而在类中是私有的。从外部可以随意修改结构体变量中的数据,对数据的这种操作是很不安全的,程序员不能通过结构体对数据进行保护和控制;在结构体中,数据和其相应的操作是分离的,使得程序的复杂性难以控制,而且程序的可重用性不好,严重影响了软件的生产效率。所以,一般

仅有数据成员时使用结构体，当既有数据成员又有成员函数时使用类。

注意：在类定义时不要丢掉类定义的结束标志“;”。

例如定义日期类：

```
class Tdate                              //定义日期类
{
public:                                  //定义公有成员函数
    void set(int m,int d,int y);         //设置日期值
    int isLeapYear();                    //判断是否是闰年
    void print();                        //输出日期值
private:                                 //定义私有数据成员
    int month;
    int day;
    int year;
};                                       //类定义体的结束
```

3.1.2 类中成员函数的定义

类的数据成员说明对象的特征，而成员函数决定对象的操作行为。成员函数是程序算法实现的部分，是对封装的数据进行操作的唯一途径。类的成员函数有两种定义方法：外联定义和内联定义。

1. 外联成员函数(外联函数)

外联定义成员函数是指在类定义体中声明成员函数，而在类定义体外定义成员函数。在类中声明成员函数时，它所带的函数参数可以只指出其类型，而省略参数名；在类外定义成员函数时必须在函数名之前缀上类名，在函数名和类名之间加上作用域区分符“::”，作用域区分符“::”指明一个成员函数或数据成员所在的类。作用域区分符::前若不加类名，则成为全局数据或全局函数(非成员函数)。

在类外定义成员函数的具体形式如下。

```
返回值类型 类名::成员函数名(形式参数表)
{
    //函数体
}
```

如3.1.1节提到的日期类中的三个成员函数分别定义如下。

```
void Tdate::set(int m,int d,int y)                    //设置日期值
{
    month = m; day = d; year = y;
}
int Tdate::isLeapYear()                               //判断是否是闰年
{
    return (year % 4 == 0&&year % 100!= 0)||(year % 400 == 0);
}
void Tdate::print()                                   //输出日期值
{
    cout << month <<"/"<< day <<"/"<< year << endl;
}
```

2. 内联成员函数(内联函数、内部函数、内置函数)

函数调用有一定的时间和空间方面的开销,时间开销影响了程序的执行效率,使用内联函数可以避免函数调用机制所带来的时间开销,提高程序的执行效率。程序在编译时将内联成员函数的代码插入在函数的每个调用处,作为函数体的内部扩展。由于在编译时函数体中的代码被替代到程序中,因此会增加目标程序代码量,进而增加空间开销,而在时间开销上不像函数调用时那么大,所以提高了程序的执行效率。可以将这些仅由少数几条简单语句组成,却在程序中被频繁调用的函数定义为内联成员函数。一般情况下,内联函数体中不包含循环语句和 switch 语句。

内联成员函数有两种定义方法,一种方法是在类定义体内定义,另一种方法是使用 inline 关键字。

(1) 在类定义体内定义内联成员函数(隐式声明)。

```
class Tdate
{
public:
    void set(int m,int d,int y)                    //设置日期值
    {
        month = m; day = d; year = y;
    }
    int isLeapYear()                               //判断是否是闰年
    {
        return(year % 4 == 0&&year % 100!= 0)||(year % 400 == 0);
    }
    void print()                                   //输出日期值
    {
        cout << month <<"/"<< day <<"/"<< year << endl;
    }
private:
    int month;
    int day;
    int year;
};
```

(2) 使用关键字 inline 定义内联成员函数(显式声明)。

如果在类定义体外定义函数时使用关键字 inline,则可将定义在类定义体外的函数声明为内联成员函数,这时可在类定义体内相应函数的前面增加关键字 inline,将该函数声明为内联成员函数。

例如,Tdate 类中 Set()内联函数的定义体为:

```
inline void Tdate::Set(int m,int d,int y)          //设置日期值
{
    month = m; day = d; year = y;
}
```

或

```
void inline Tdate::Set(int m,int d,int y)          //设置日期值
{
```

```
    month = m; day = d; year = y;
}
```

3. C++程序的多文件结构

C++语言是混合型语言,它既支持结构化程序设计,又支持面向对象程序设计。C++语言支持面向对象程序设计,主要体现在类的定义和应用上。一般情况下,一个模块由规范说明和实现两部分组成。规范说明部分描述一个模块与其他模块的接口,而实现部分则是模块的实现细节。模块中的规范说明部分作为一个单独的文件存放起来,这个文件被称为头文件,其扩展名为".h";而实现部分可能由多个扩展名为".cpp"的文件组成。一般一个较大的程序可以分为三种文件来保存。

(1) 类的定义。将不同类的定义分别作为一个头文件来保存(主文件名一般为类名),成员函数一般采用外联定义方式。若是内联函数,则其原型和定义一般归入头文件。

(2) 类的实现。不同类的实现部分分别作为一个文件(.cpp 文件),用来保存类中成员函数的定义。

(3) 类的使用。类的使用放在一个单独的.cpp 文件中,该文件使用#include 编译预处理命令包含类定义的头文件,在 main()函数中使用不同的类。

模块化是信息隐蔽的重要思想,信息隐蔽对开发大的程序非常有用,可以在极大程度上保证程序的质量。类的用户只需了解类的外部接口,而无须了解类的内部实现。类的用户只能应用类的外部接口,不能修改类的内部结构。

【例 3.1】 在头文件中定义类,在程序文件中定义成员函数示例。

```
// tdate.h 这个头文件只存放有关 Tdate 类的定义说明
#ifndef Tdate_H                                    //用来避免重复定义
#define Tdate_H                                    //不是类的一部分
class Tdate
{
public:
    void set(int,int,int);                         //成员函数原型
    int isLeapYear();
    void print();
private:
    int month;
    int day;
    int year;
};                                                 //Tdate 类定义的结束
#endif                                             //Tdate_H

// tdate.cpp 类 Tdate 的实现部分
#include <iostream>    /* 因为 tdate.cpp 文件要访问运算符<< 和 ostream 类对象 cout,而这二
                          者都是定义在 iostream 类中的,所以包含 iostream 头文件 */

#include "tdate.h"     //包含用户自定义的头文件,该文件中提供了 Tdate 类的定义
using namespace std;
```

```
void Tdate::set(int m,int d,int y)
{
    month = m; day = d; year = y;
}

int Tdate:: isLeapYear()
{
    return ((year % 4 == 0 && year % 100 != 0)||(year%400==0));
}

void Tdate::print()
{
    cout << month << "/" << day << "/" << year << endl;
}
```

类的应用在此例中没有给出,具体参见例 3.2。

说明:头文件 tdate.h 中前两行

```
#ifndef Tdate_H                    //用来避免重复定义
#define Tdate_H                    //不是类的一部分
```

的作用是,如果一个程序系统中的多个文件均包含 Tdate 类,则在编译时可以避免 Tdate 类中标识符的重复定义。类定义前的这些行可使编译器跳过文件中最后一行 #endif//Tdate_H 之前的所有行。

除了第一次之外,以后编译器每遇到编译预处理命令 #include "tdate.h"则以 #ifndef(如果没有定义)开始的命令行测试标识符 Tdate_H 是否已经定义。如果没有定义,则第二行定义标识符 Tdate_H,且它的值为 NULL,然后编译器处理文件 tdate.h 中的其余行。如果以后再一次包含了 tdate.h 文件,则编译器要处理第一行 #ifndef Tdate_H,确定了标识符 Tdate_H 已经定义,则命令行 #endif 之前的所有行都被跳过,不进行编译,因此避免了类中标识符的重复定义。名字 Tdate_H 没有任何特殊的意义,只是在类名的末尾加了_H。

3.2 对　　象

1. 对象的基本概念

现实生活中,任何事物都可以称为对象,它是无所不在的。用面向对象方法开发一个系统时,对象的识别与描述只限定在待开发的软件系统中与系统目标相关的事物。

用面向对象方法开发的软件系统中,对象是类的实例,是属性和服务的封装体。一个对象就是一个实际问题域中的实体,它包含了数据结构和所需的相关操作功能,形成了一个基本程序模块。

对象的属性用于描述对象的静态数据特征。如人类有大脑、五官、四肢,鸟有翅膀、羽毛,树有根、茎、叶等,这些都是描述对象的静态数据特征。对象的属性可用系统的或用户自定义的数据类型来表示,也可以用抽象的数据类型表示。对象属性值的集合称为对象的状态(state)。

对象的服务用于描述对象的动态特征，也称为行为或性能，它是定义在对象属性基础上的一组操作方法(method)的集合。如人类有思维能力、有语言能力、可直立行走等，鸟类有翅膀可以飞行等。对象的服务是响应消息而完成的算法，它体现了对象的行为能力。对象的服务包括自操作和它操作，自操作是对象对其内部的数据属性进行的操作，它操作是对其他对象进行的操作。

当一个对象映射为软件实现时由以下三部分组成。

(1) 私有的数据：用于描述对象的内部状态。

(2) 处理：也称为操作或方法，对私有数据进行运算。

(3) 接口：这是对象可被共享的部分，消息通过接口调用相应的操作。接口规定哪些操作是允许的，但并不提供操作是如何实现的信息。

2. 对象的定义

对象的定义有两种方法，可以在定义类的同时直接定义，也可以在使用时通过类进行定义。

(1) 方法一：在定义类的同时直接定义。

```
class Location
{
public:
    void init(int x0, int y0);
    int getX( void );
    int getY( void );
private:
    int x, y;
}dot1,dot2;
```

(2) 方法二：在使用时定义对象。

格式如下：

```
类名 标识符,…,标识符;
```

如：

```
Location dot1,dot2;
```

3. 成员的访问

定义了类及其对象，就可以调用公有成员函数实现对对象内部属性的访问。当然，不论是数据成员，还是成员函数，只要是公有的(public)，在类的外部就可以通过类的对象进行访问。对公有成员的调用可以通过以下几种方法来实现。

(1) 通过对象调用成员。

格式如下：

```
对象名.公有成员
```

其中，“.”称为对象选择符，简称点运算符。

(2) 通过指向对象的指针调用成员。

格式如下:

```
指向对象的指针 ->成员
```

或

```
(*对象指针名).公有成员
```

(3) 通过对象的引用调用成员。

格式如下:

```
对象的引用.成员
```

需要注意,只有用 public 定义的公有成员才能使用点运算符访问。对象中的私有成员是类中隐藏的数据,类的外部不能访问对象的私有成员,只能通过该类的公有成员函数来访问它们。例如定义时钟类:

```
class Clock
{
public:
    void init();
    void update();
    void display();
private:
    int hour, minute, second;
};
int main()
{
    Clock myClock, *pclock;
    //定义对象 myClock 和指向 myClock 类对象的指针 pclock
    myClock.init();                          //通过对象访问公有成员函数
    pclock = &myClock;                       //指针 pclock 指向对象 myClock
    pclock->display();                       //通过指针访问成员函数
    //myClock.hour = 4;该语句错误,因为对象不能访问其私有成员
return 0;
}
```

【例 3.2】 对象成员的访问示例。

此例的项目文件中包含了例 3.1 中的 tdate.h 头文件和 tdate.cpp 源程序文件以及下述的 ch3_2.cpp 文件,共 3 个文件。在 Visual C++ 2010 环境中新建的项目中添加头文件 tdate.h,添加源文件 tdate.cpp 和 ch3_2.cpp,该项目的文件结构如图 3.1 所示。源程序代码如下:

```
// ch3_2.cpp
#include <iostream>
```

```
#include"tdate.h"
using namespace std;

void someFunc(Tdate& refs)
{
    refs.print();                          //通过对象的引用调用成员函数
    if(refs.isLeapYear())                  //通过对象的引用调用成员函数
        cout << "error\n";
    else
        cout << "right\n";
}

int main()                                 //类的应用部分
{
    Tdate s, * pTdate = &s;                //pTdate 为指向 Tdate 类对象的指针
    s.set(2,15,1998);                      //通过对象调用成员函数
    pTdate -> print();                     //通过指向对象的指针调用成员函数
    if(( * pTdate).isLeapYear())           //通过指向对象的指针调用成员函数
        cout << "error\n";
    else
        cout << "right\n";
    someFunc(s);                           //对象的地址传给引用

    return 0;
}
```

程序的运行结果如图 3.2 所示。

图 3.1 例 3.1 项目文件结构

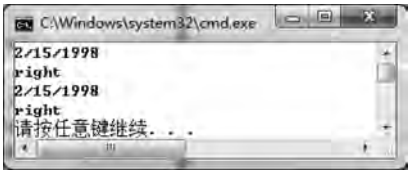

图 3.2 例 3.2 的运行结果

为了提高程序的可读性,类的定义和类的实现存放在不同的文件中,也可以采用例 3.3 的方式在项目中添加类,系统会自动创建对应的.h 头文件和.cpp 源文件以简化程序员操作。由于本书中的例题相对比较简单,为方便教材的编写,部分例题仍将类的定义和实现放在同一个文件中。如果程序比较复杂,建议读者将类的定义和实现部分分别放在头文件和对应的源程序文件中。

【例 3.3】 堆栈 Cstack 类的实现,该类用于存储字符。

在Visual C++ 2010环境中新建“Win32控制台应用程序”项目ch3_3,在解决方案资源管理器中右击ch3_3项目名称,在弹出的快捷菜单中选择“添加”→“类”,弹出如图3.3所示的“添加类”对话框,在该对话框中选中“C++类”,单击“添加”按钮,弹出如图3.4所示的“一般C++类向导”对话框,在“类名”文本框中输入Cstack,项目中会自动添加Cstack.h和Cstack.cpp两个文件,单击“完成”按钮。

图3.3 “添加类”对话框

图3.4 “一般C++类向导”对话框

自动生成的 Cstack.h 头文件的内容如下：

```
#pragma once
class Cstack
{
public:
    Cstack(void);
    ~Cstack(void);
};
```

自动生成的 Cstack.cpp 源文件的内容如下：

```
#include "Cstack.h"


Cstack::Cstack(void)
{
}


Cstack::~Cstack(void)
{
}
```

其中，编译预处理命令“#pragma once”的作用与下述代码段的作用相同，是为了避免重复定义。

```
#ifndef Tdate_H                                    //用来避免重复定义
#define Tdate_H                                    //不是类的一部分
…
#endif
```

特殊成员函数 Cstack(void) 和 ~Cstack(void) 的概念在后续小节中加以讲解。用户在此基础上修改 Cstack.h 和 Cstack.cpp 文件即可完成类和成员函数的定义，之后再添加 ch3_3.cpp 源文件即可完成该程序。完整的源程序代码如下：

```
// Cstack.h
#pragma once

const int SIZE = 10;                               //存储的最多字符数

class Cstack
{
public:
    Cstack(void);
```

```
    ~Cstack(void);
    void init();
    char  push (char ch);
    char  pop();
private:
    char stk[SIZE];
    int  position;
};
```

```
// Cstack.cpp
#include "Cstack.h"
#include <iostream>
using namespace std;

Cstack::Cstack(void)
{
}

Cstack::~Cstack(void)
{
}

void Cstack::init()
{
    position = 0;
}

char Cstack::push(char ch)
{
    if (position == SIZE)
    {
        cout <<"\n 栈已满 \n";
        return 0;
    }
    stk [position++] = ch;
    return  ch;
}

char Cstack::pop()
{
    if (position == 0)
    {
        cout <<"\n 栈已空"<< endl;
        return 0;
    }
    return stk[ --position];
}
```

```
// ch3_3.cpp
#include <iostream>
#include "Cstack.h"
using namespace std;

int main()
{
    Cstack s;
    s.init();
    char ch;
    cout <<"请输入字符: "<< endl;
    cin >> ch;
    while(ch != '#'&& s.push(ch))
        cin >> ch;
    cout <<"\n 现在输出栈内数据\n";
    while(ch = s.pop())
        cout << ch <<" ";

    return 0;
}
```

程序的运行结果如图 3.5 所示。

图 3.5 例 3.3 的运行结果

4. 名字解析和 this 指针

1) 名字解析

在调用成员函数时,通常使用缩写形式,如 Tdate 例中的表达式 s. set(2,15,1998)就是 s. Tdate::set(2,15,1998)的缩写,因此可以定义两个或多个类的具有相同名字的成员而不会产生二义性。

2) this 指针

当一个成员函数被调用时,C++语言自动向它传递一个隐含的参数,该参数是一个指向接受该函数调用的对象的指针,在程序中可以使用关键字 this 来引用该指针,因此称该指针为 this 指针。this 指针是 C++语言实现封装的一种机制,它将成员和用于操作这些成员的成员函数连接在一起。例如 Tdate 类的成员函数 set()被定义为:

```
void Tdate::set(int m,int d,int y)                    //设置日期值
{
    month = m; day = d; year = y;
}
```

其中,对 month、day 和 year 的引用,表示的是在该成员函数被调用时,引用接受该函数调用的对象中的数据成员 month、day 和 year。例如,对于下面的语句:

```
Tdate dd;
dd.set (5,16,1990);
```

当调用成员函数 set 时,该成员函数的 this 指针指向类 Tdate 的对象 dd。成员函数 set 中

对 month、day 和 year 的引用表示引用对象 dd 的数据成员。C++语言编译器所认识的成员函数 set 的定义形式为：

```
void Tdate::set(int m,int d,int y)                    //设置日期值
{
    this->month=m; this->day=d; this->year=y;
}
```

即对于该成员函数中访问的类的任何数据成员，C++语言编译器都认为是访问 this 指针所指向对象的成员。由于不同的对象调用成员函数 set()时，this 指针指向不同的对象，因此，成员函数 set()可以为不同对象的 month、day 和 year 赋初值。使用 this 指针，保证了每个对象可以拥有不同的数据成员值，但处理这些数据成员的代码可以被所有的对象共享。

5. 带默认参数的成员函数和重载成员函数

同普通函数一样，类的成员函数也可以是带默认值的函数，其调用规则与普通函数相同。成员函数也可以是重载函数，类的成员函数的重载与全局函数的重载方法相同。

【例 3.4】 带默认参数的成员函数。

该程序共包括 2 个源文件 Tdate.h 和 ch3_4.cpp，源程序代码如下：

```
// Tdate.h
#include <iostream>
using namespace std;

class Tdate
{
public:
    void set(int m=5,int d=16,int y=1990)                    //设置日期值
    {
        month=m; day=d; year=y;
    }
    void print()                                             //输出日期值
    {
        cout<<month<<"/"<<day<<"/"<<year<<endl;
    }
private:
    int month;
    int day;
    int year;
};
```

```
// ch3_4.cpp
#include "Tdate.h"

int main()
{
    Tdate a,b,c,d;
    a.set(4,12,1996);
    b.set(3);
```

```
    c.set(8,10);
    d.set();
    a.print();
    b.print();
    c.print();
    d.print();

    return 0;
}
```

程序的运行结果如图 3.6 所示。

图 3.6 例 3.4 的运行结果

【例 3.5】 重载成员函数示例。

```
// ch3_5.cpp
#include <iostream>
using namespace std;

class Cube
{
public:
    int volume(int  ht,int wd)
    {
        return ht * wd;
    }
    int volume(int ht,int wd,int dp)
    {
        height = ht;
        width = wd;
        depth = dp;
        return height * width * depth;
    }
private:
    int height,width,depth;
};

int main()
{
    Cube cube1;
    cout << cube1.volume(10,20) << endl;                    //调用带 2 个参数的成员函数
    cout << cube1.volume(10,20,30) << endl;                 //调用带 3 个参数的成员函数

    return 0;
}
```

程序的运行结果如图 3.7 所示。

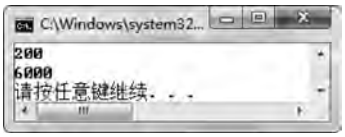

图 3.7　例 3.5 的运行结果

3.3　构造函数和析构函数

当声明一个对象时,对象的状态(数据成员的取值)是不确定的。但对象表达了现实世界的实体,因此,一旦声明对象,必须有一个有意义的初始值。C++语言中有一个称为构造函数的特殊成员函数,它可自动进行对象的初始化,还有一个析构函数在对象撤销时执行清理任务,进行善后处理。

构造函数和析构函数是类中的两个特殊的成员函数,具有普通成员函数的许多共同特性,但还具有一些独特的特性,可以归纳成以下几点。

(1) 它们都没有返回值说明,也就是说定义构造函数和析构函数时不能指出函数返回值的类型,即使是 void 也不能有。

(2) 它们不能被继承。

(3) 和大多数 C++ 函数一样,构造函数可以有默认参数。

(4) 析构函数可以是虚的(virtual),但构造函数不可以是虚的。

(5) 不可取它们的地址。

(6) 不能用常规调用方法调用构造函数,当使用完全的限定名(带对象名、类名和函数名)时可以调用析构函数。比较特殊的是,在对象数组中,初始化数组元素时可以显式调用构造函数,参见例 3.11。

(7) 当定义对象时,编译程序自动调用构造函数;当删除对象时,编译程序自动调用析构函数。

3.3.1　构造函数

对象的初始化是指对象数据成员的初始化,在使用对象前,一定要进行初始化。由于数据成员一般为私有的(private),所以不能直接赋值。对象初始化有以下两种方法。

一种方法是类中提供一个普通成员函数来初始化,但是会造成使用上的不便(使用对象前必须显式调用该函数)和不安全(未调用初始化函数就使用对象)。

另外一种方法是使用构造函数对对象进行初始化。下面具体介绍构造函数及其使用方法。

1. 构造函数(constructor)

构造函数是一个与类同名,没有返回值(即使是 void 也不可以有,但在函数体内可有无值的 return 语句)的特殊成员函数。一般用于初始化类的数据成员,每当创建一个对象时(包括使用 new 动态创建对象),编译系统就自动调用构造函数。构造函数既可在类外定义,也可作为内联函数在类内定义。

构造函数定义了创建对象的方法，提供了初始化对象的一种简便手段。在类外定义构造函数时，其声明格式为：

```
<类名>::构造函数名(<形式参数表>)
```

定义了构造函数后，在定义该类对象时可以将参数传递给构造函数来初始化该对象。

一个类可以有多个构造函数，但它们的形式参数的类型和个数不能完全相同，编译器在编译时可以根据参数的不同选择不同的构造函数。

【例 3.6】 构造函数的定义和调用示例。

项目中包含 MyQueue.h、MyQueue.cpp 和 ch3_6.cpp 三个文件。

```
// MyQueue.h
#pragma once
class MyQueue
{
public:
    MyQueue(void);
    ~MyQueue(void);
    void qput(int i);
    int qget();
private:
    int q[100];
    int sloc,rloc;
};

// MyQueue.cpp
#include "MyQueue.h"
#include <iostream>
using namespace std;

MyQueue::MyQueue(void)
{
    sloc = rloc = 0;
    cout << "queue initialized\n";
}

MyQueue::~MyQueue(void)
{
}

void MyQueue::qput(int i)
{
    if(sloc == 100)
    {
        cout << "queue is full\n";
        return;
    }
    sloc++;
```

```
    q[sloc] = i;
}

int MyQueue::qget()
{
    if(rloc == sloc)
    {
        cout << "queue is empty\n";
        return 0;
    }
    rloc++;
    return q[rloc];
}

// ch3_6.cpp
#include "MyQueue.h"
#include <iostream>
using namespace std;

int main()
{
    MyQueue a,b;
    a.qput(10);
    b.qput(20);
    a.qput(20);
    b.qput(19);
    cout << a.qget()<< "    ";
    cout << a.qget()<< "\n";
    cout << b.qget()<< "    ";
    cout << b.qget()<< "\n";
    return 0;
}
```

程序的运行结果如图 3.8 所示。

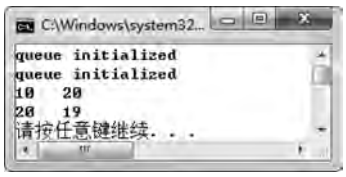

图 3.8 例 3.6 的运行结果

【程序解析】

MyQueue 类为队列类，队列是一种先进先出的线性表。从此例可以看出，在 main() 函数中，构造函数 MyQueue() 没有被显式调用。正如前面提到的，构造函数是在定义对象时被系统自动调用的，也就是说在定义对象 a、b 的同时 a.MyQueue::MyQueue() 和 b.MyQueue::MyQueue() 被自动调用执行。

【例 3.7】 构造函数的重载。

项目中包含 TestOverloadConstructor.h、TestOverloadConstructor.cpp 和 ch3_7.cpp 三个文件。

```
// TestOverloadConstructor.h
#pragma once
class TestOverloadConstructor
```

```
{
public:
    TestOverloadConstructor(void);
    ~TestOverloadConstructor(void);
    TestOverloadConstructor(int n, float f);                    //参数化的构造函数
private:
    int num;
    float f1;
};

// TestOverloadConstructor.cpp
#include "TestOverloadConstructor.h"
#include <iostream>
using namespace std;


TestOverloadConstructor::TestOverloadConstructor(void)
{
    num = 0;
    f1 = 0.0;
    cout <<"Initializing default "<< num <<", "<< f1 << endl;
    cout <<" ------------------------------ "<< endl;
}

TestOverloadConstructor::TestOverloadConstructor(int n, float f)
{
    num = n;
    f1 = f;
    cout <<"Initializing "<< num <<", "<< f1 << endl;
    cout <<" ------------------------------ "<< endl;
}


TestOverloadConstructor::~TestOverloadConstructor(void)
{
}

// ch3_7.cpp
#include "TestOverloadConstructor.h"

int main()
{
    TestOverloadConstructor x;                                  //调用无参的构造函数
    TestOverloadConstructor y(10, 21.5f);                       //调用带两个参数的构造函数
    TestOverloadConstructor * px = new TestOverloadConstructor;
    //调用无参的构造函数
    TestOverloadConstructor * py = new TestOverloadConstructor(100,36.6f);
    //调用带两个参数的构造函数

    return 0;
}
```

程序的运行结果如图 3.9 所示。

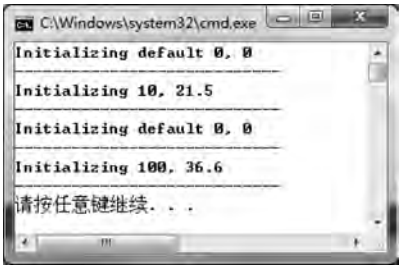

图 3.9　例 3.7 的运行结果

类的构造函数一般是公有的(public),但有时也声明为私有的,其作用是限制创建该类对象的范围,即只能在本类和友元中创建该类对象。

2. 带默认参数的构造函数

构造函数也可以使用默认参数,但要注意,必须保证参数默认后,函数形式不能与其他构造函数完全相同。即在使用带默认参数的构造函数时,要注意避免二义性。所带的参数个数或参数类型必须有所不同,否则系统调用时会出现二义性。

【例 3.8】 带默认参数的构造函数示例。

项目中包含 Tdate.h、Tdate.cpp 和 ch3_8.cpp 三个文件。

```
// Tdate.h
#pragma once
class Tdate
{
public:
    Tdate(int m = 5, int d = 16, int y = 1990);
    ~Tdate(void);
private:
    int month;
    int day;
    int year;
};

// Tdate.cpp
#include "Tdate.h"
#include <iostream>
using namespace std;

Tdate::Tdate(int m, int d, int y)
{
    month = m;   day = d;   year = y;
    cout << month <<"/" << day <<"/" << year << endl;
}

Tdate::~Tdate(void)
{
}

// ch3_8.cpp
#include "Tdate.h"

int main()
{
    Tdate aday;
    Tdate bday(2);
```

```
    Tdate cday(3,12);
    Tdate dday(1,2,1998);
    return 0;
}
```

程序的运行结果如图 3.10 所示。

图 3.10 例 3.8 的运行结果

【程序解析】

(1) 如果在 Tdate 类增加无参的构造函数 Tdate::Tdate(void),则编译时会出现如图 3.11 所示的警告和错误信息。警告信息表示 Tdate 类指定了多个默认构造函数。错误信息指编译语句"Tdate aday;"时,对重载函数的调用不明确。

图 3.11 重载函数调用不明确错误的提示信息

(2) 函数声明时加默认参数,定义时去掉默认参数,如果在定义函数时的说明部分加默认参数,如下所示:

```
Tdate::Tdate(int m = 5,int d = 16,int y = 1990)
{
    month = m;   day = d;   year = y;
    cout << month <<"/" << day <<"/" << year << endl;
}
```

编译时则会出现如图 3.12 所示的错误信息。

图 3.12 重定义默认参数错误的提示信息

3. 默认构造函数

C++语言规定,每个类必须有一个构造函数,没有构造函数,就不能创建任何对象。若用户未显式定义一个类的构造函数,则 C++语言提供一个默认构造函数,该默认构造函数是一个无参构造函数,其函数体为空,它仅负责创建对象,不做任何初始化工作。

只要一个类定义了一个构造函数(不一定是无参构造函数),C++语言就不再提供默认的构造函数。如果为类定义了一个带参数的构造函数,还想要使用无参构造函数,则必须自己定义。

与变量定义类似,在用默认构造函数创建对象时,如果创建的是全局对象或静态对象,则对象的位模式全为 0;否则,对象值是随机的。

【例 3.9】 默认构造函数示例。

```
// ch3_9.cpp
#include <iostream>
using namespace std;

class Student
{
public:
    Student(char * pName)
    {
        cout <<"call one parameter constructor"<< endl;
        strncpy_s(name,pName,sizeof(name));
        name[sizeof(name) - 1] = '\0';
        cout <<"the name is "<< name << endl;
    }
    Student(){cout <<"call no parameter constructor"<< endl;}
    void display()
    {
        cout <<"the name of the student is "<< name << endl;
    }
protected:
    char name[20];
};

int main()
{
    static Student noName1;
    Student noName2;
    Student ss("Jenny");
    noName1.display();
    noName2.display();
    return 0;
}
```

程序的运行结果如图 3.13 所示。

【程序解析】

(1) 函数 strncpy_s(s1,s2,n)的作用是将字符串 s2 复制到字符串 s1 中,但最多复制 n 个字符。

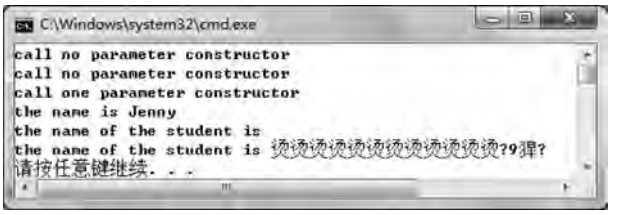

图 3.13 例 3.9 的运行结果

(2) 创建 noName1 对象和 noName2 对象时,调用了无参构造函数;而创建 ss 对象时,提供了一个实际参数"Jenny",所以调用的是带一个参数的构造函数。

(3) noName1 对象为 static 对象,其位模式为 0,输出其姓名为空;noName2 为非 static 的局部变量,其姓名为随机值。

(4) 该例中无参构造函数的定义不能省略,在 main()函数中创建对象 noName1 和 noName2 时会调用此函数,因为用户一旦定义了构造函数,系统不再提供默认构造函数。如果省略无参构造函数的定义则编译时会出现如图 3.14 所示的错误信息。

图 3.14 省略无参构造函数错误的提示信息

4. 对象数组

对象数组是指每个数组元素都是对象的数组,也就是说,若某一个类有若干个对象,这一系列被创建的对象可用一个数组来存放。

若要说明一个带有构造函数的类的对象数组,这个类一般情况下会含有一个不带参数的构造函数或带有默认参数的构造函数。因为当声明一个对象数组时,编译程序会为这个对象数组的每个元素调用一次无参构造函数来创建对象,如例 3.10 所示。

若类中没有无参构造函数,则在说明对象数组时必须提供初始值,如例 3.11 所示。

【例 3.10】 对象数组示例 1。

项目中包含 Test.h、Test.cpp 和 ch3_10.cpp 三个文件。

```
// Test.h
#pragma once
class Test
{
public:
    Test(void);
    ~Test(void);
    Test(int n, float f);                                    //参数化的构造函数
    int getInt();
```

```
    float getFloat();
private:
    int num;
    float f1;
};

// Test.cpp
#include "Test.h"
#include <iostream>
using namespace std;

Test::Test(void)
{
    num = 0;
    f1 = 0.0;
    cout <<"Initializing default"<< endl;
}

Test::Test(int n, float f)
{
    num = n;
    f1 = f;
    cout <<"Initializing "<< num <<", "<< f1 << endl;
}

int Test::getInt()
{return num;}

float Test::getFloat()
{return f1;}

Test::~Test(void)
{
}

// ch3_10.cpp
#include "Test.h"
#include <iostream>
using namespace std;

int main()
{
    cout <<"the main function:"<< endl;
    Test array[5];
    cout <<"the second element of array is "<< array[1].getInt()
        <<"    "<< array[1].getFloat()<< endl;

    return 0;
}
```

程序的运行结果如图 3.15 所示。

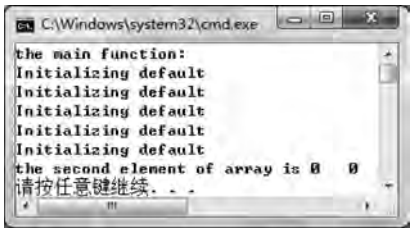

图 3.15 例 3.10 的运行结果

【程序解析】

(1) 此程序中若去掉无参构造函数的声明和定义,编译时会出现以下的错误信息:

```
error C2512: "Test": 没有合适的默认构造函数可用
```

因为编译系统需使用无参构造函数创建数组元素对象。用户一旦定义了构造函数,则系统不再提供默认构造函数。

(2) 若把两个构造函数的声明和定义均去掉,则不会出现错误信息。因为编译系统使用默认构造函数创建数组元素对象,其对象的数据成员值为随机值。

【例 3.11】 对象数组示例 2。

```
// ch3_11.cpp
#include <iostream>
using namespace std;

class Test
{
private:
    int num;
    float f1;
public:
    Test (int n);
    Test (int n, float f);
};

Test::Test(int n)
{
    num = n;
    cout <<"Initializing "<< num << endl;
}
Test::Test(int n, float f)
{
    num = n;
    f1 = f;
    cout <<"Initializing "<< num <<", "<< f1 << endl;
}

int main()
{
    Test array1[3] = {1,2,3};
    cout <<" ------------------ "<< endl;
    Test array2[] = {Test(2,3.5),Test(4)};
    cout <<" ------------------ "<< endl;
    Test array3[] = {Test(5.5,6.5),Test(7,8.5)};
    //array3 有 2 个元素
```

```
    cout <<" ------------------ "<< endl;
    Test array4[] = {Test(5.5, 6.5), 7.5, 8.5};
    //array4 有 3 个元素

    return 0;
}
```

程序的运行结果如图 3.16 所示。

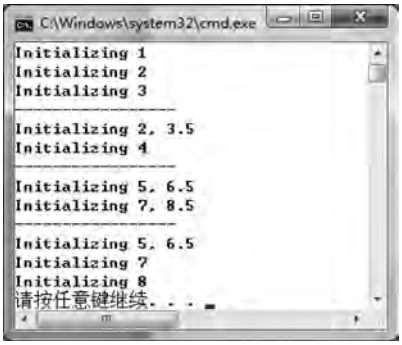

图 3.16　例 3.11 的运行结果

5. 拷贝构造函数

1) 拷贝构造函数的调用

拷贝构造函数的功能是用一个已有的对象来初始化一个被创建的同类对象,是一种特殊的构造函数,具有一般构造函数的所有特性; 其形参是本类对象的引用,它的特殊功能是将参数代表的对象逐域复制到新创建的对象中。

用户可以根据实际问题的需要定义特定的拷贝构造函数,以实现同类对象之间数据成员的传递。如果用户没有声明类的拷贝构造函数,系统就会自动生成一个默认拷贝构造函数,这个默认拷贝构造函数的功能是把初始对象的每个数据成员的值都复制到新建立的对象中。拷贝构造函数的声明形式为如下:

```
类名(类名 & 对象名);
```

下面定义了一个 Cat 类和 Cat 类的拷贝构造函数:

```
class Cat
{
public:
    Cat();
    Cat(Cat &);                          //拷贝构造函数的声明
    void play();
    void hunt();
private:
    int age;
    float weight;
    char * color;
};

Cat::Cat(Cat &other)
{
    age = other.age;
    weight = other.weight;
    color = other.color;
}
```

在以下四种情况下系统会自动调用拷贝构造函数:

(1) 用类的一个对象去初始化另一个对象。

```
Cat cat1;
Cat cat2(cat1);
// 创建 cat2 时系统自动调用拷贝构造函数,用 cat1 的数据成员初始化 cat2
```

(2) 用类的一个对象去初始化另一个对象时的另外一种形式

```
Cat cat2 = cat1;                    //注意并非 Cat cat1,cat2; cat2 = cat1;
```

(3) 对象作为函数参数传递时,调用拷贝构造函数

```
f(Cat a){ }                         //定义 f()函数,形参为 Cat 类对象
Cat b;                              //定义对象 b
f(b);                               //进行 f()函数调用时,系统自动调用拷贝构造函数
```

(4) 如果函数的返回值是类的对象,函数调用返回时,调用拷贝构造函数

```
Cat f()                             //定义 f()函数,函数的返回值为 Cat 类的对象
{
    Cat a;
    …
    return a;
}
Cat b;                              //定义对象 b
b = f();                            //调用 f()函数,系统自动调用拷贝构造函数
```

2) 深拷贝构造函数和浅拷贝构造函数

拷贝构造函数分为深拷贝和浅拷贝两种构造函数。由 C++语言提供的默认拷贝构造函数只是对对象进行浅拷贝(逐个成员依次拷贝),即只复制对象空间而不复制资源。图 3.17 所示为浅拷贝的两种情况。

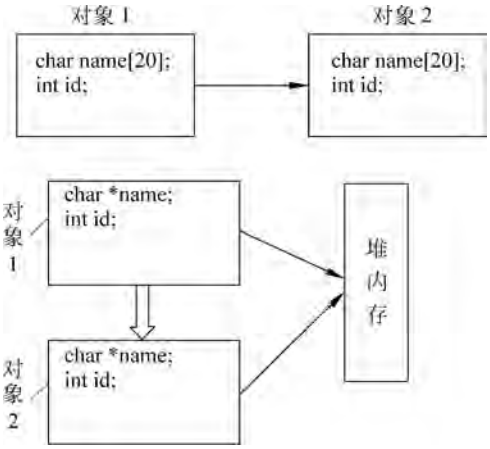

图 3.17 浅拷贝的两种情况示意图

一般情况下,只需使用系统提供的浅拷贝构造函数即可,但是如果对象的数据成员包括指向堆空间的指针,就不能使用这种拷贝方式,因为两个对象都拥有同一个资源,对象析构时,该资源将经历两次资源返还,此时必须自定义深拷贝构造函数,为创建的对象分配堆空

间,否则会出现动态分配的指针变量悬空的情况。深拷贝需要同时复制对象空间和资源,如图 3.18 所示。

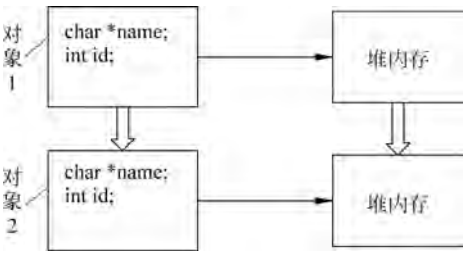

图 3.18 深拷贝示意图

说明:在同时满足以下两个条件时,必须要定义深拷贝构造函数。

① 满足调用拷贝构造函数的四种情况之一。

② 数据成员包括指向堆内存的指针变量。

【例 3.12】 深拷贝构造函数示例。

```
// ch3_12.cpp
#include <iostream>
using namespace std;

class Person
{
public:
    Person(char * na)                                    //构造函数
    {
        cout <<"call constructor"<< endl;
        name = new char[strlen(na) + 1];                 //使用 new 进行动态内存分配
        if(name!= 0)
        {strcpy_s(name,strlen(na) + 1,na);}
    }

    Person(Person&p)                                     //深拷贝构造函数
    {
        cout <<"call copy constructor"<< endl;
        name = new char[strlen(p.name) + 1];             //复制资源
        if(name!= 0)
            strcpy_s(name,strlen(p.name) + 1,p.name);    //复制对象空间
    }

    void printName()
    {
        cout << name << endl;
    }

    ~Person()                                            //析构函数的定义,参见 3.3.2 节
    {
        delete name;
```

```
    }
private:
    char *name;
};                                                    //类定义的结束

int main()
{
    Person wang("wang");
    Person li(wang);
    wang.printName();
    li.printName();

    return 0;
}
```

程序的运行结果如图 3.19 所示。

图 3.19 例 3.12 的运行结果

【程序解析】

(1) 程序中使用了 strcpy_s 函数而非 strcpy 函数,这是因为从 Visual C++ 2005 版本开始,微软引入了一系列安全加强的函数来增强 CRT(C 运行时),_s 意为 safe,同样的道理,strcat 也是如此。用户仍然可以使用 strcpy 函数,只是在编译时会出现警告信息。注意并非所有的加强函数都是后面加_s,比如 stricmp 这个字符串比较函数的增强版名字是_stricmp。

(2) 在主函数 main()中,定义了 Person 类的对象 wang,在定义对象 li 时调用了深拷贝构造函数,如果用户没有定义深拷贝构造函数,则编译系统会调用默认的浅拷贝构造函数,这样,在程序运行时会出现如图 3.20 的错误信息。

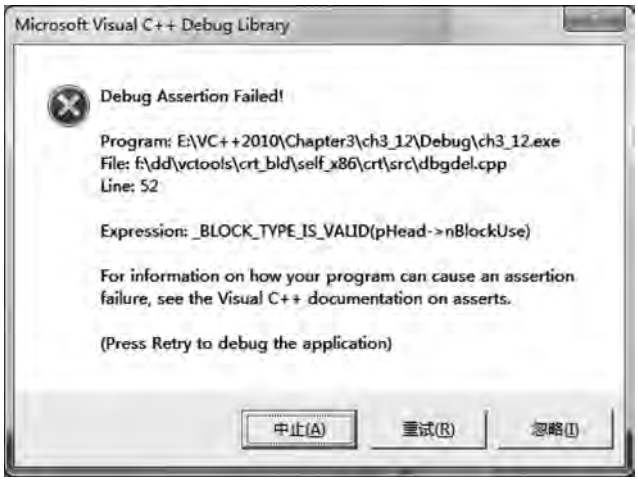

图 3.20 浅拷贝运行出错提示对话框

所以,使用拷贝构造函数时应考虑是使用深拷贝还是浅拷贝。当有使用 new 动态分配内存空间的数据成员,在析构函数中使用 delete 进行动态内存空间的释放以及对赋值运算符进行重载(参见 6.1.4 节的例 6.11)时,应该自定义深拷贝构造函数。

6. 数据成员的初始化

构造函数可以采用以下几种不同的形式初始化数据成员。

(1) 在构造函数的函数体中进行初始化,例如:

```
Circle::Circle(float r)
{
    radius = r;
}
```

(2) 使用构造函数初始化成员列表对数据成员进行初始化,其格式为:

```
类名::构造函数(形式参数表):变量 1(初值 1),…, 变量 n(初值 n)
{…}
```

如:

```
Circle::Circle(float r):radius(r)
{}
```

对于类的数据成员是一般变量的情况,放在构造函数初始化成员列表中与放在函数体中初始化的作用一样。

注意:① 在以下三种情况下需要使用初始化成员列表。

情况一:需要初始化 const 修饰的类成员或初始化引用成员数据,如例 3.13 所示。

情况二:需要初始化的数据成员是对象(这里包含了在继承情况下,通过显式调用父类的构造函数对父类数据成员进行初始化),并且这个对象所在类只有带参数的构造函数,没有无参构造函数(参见 3.4 节例 3.16)。

情况三:子类初始化父类的私有成员(参见第 5 章)。

② 数据成员初始化的次序取决于它们在类定义中的声明次序,与它们在成员初始化表中的次序无关。

【例 3.13】 对常量数据成员和引用数据成员初始化示例。

```
// ch3_13.cpp
class  SillyClass
{
public:
    SillyClass(int&i):TEN(10),refI(i)
    { }
protected:
    const int TEN;                                  //常量数据成员
    int &refI;                                      //引用数据成员
};

int main()
{
    int  i ;
    SillyClass sc(i);

    return 0;
}
```

【程序解析】

若把构造函数定义为下列方式：

```
SillyClass(int & i )
{
    TEN = 10;                                          //错误,常量不能重新赋值
    refI = i;                                          //错误,引用不能重新指派
};
```

则程序编译时会出现图 3.21 所示的错误提示信息。

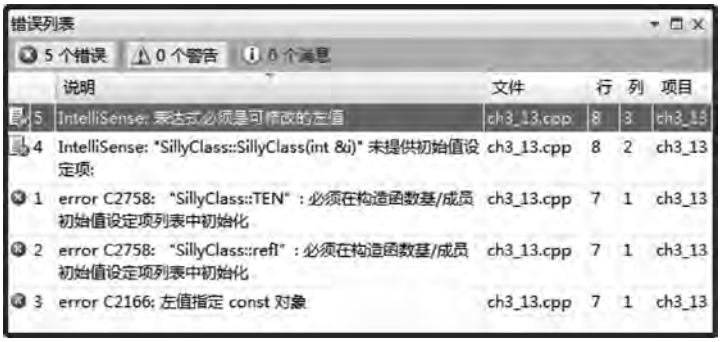

图 3.21 编译错误提示信息

(3) 混合初始化,如：

```
Student::Student(int n,int a,char * pname):number(n),age(a)
{
    strcpy(name,pname);
}
```

(4) 使用拷贝构造函数进行初始化(如上所述)。

7. 类类型和基本数据类型的转换

类型转换就是将一种类型的值转换为另一种类型的值。一般数据类型之间的转换分为隐式类型转换和显式类型转换,参见第 2.7 节,在此不再详述。在 C++语言中,类是用户自定义的数据类型,它可以像系统预定义的类型那样进行类型转换。类类型和基本数据类型之间的转换可以通过以下几种方法进行。

1) 使用转换构造函数进行类型转换(基本数据类型→类类型)

当一个构造函数只有一个参数,而且该参数又不是本类的 const 引用时,这种构造函数称为转换构造函数。通过转换构造函数可以将基本数据类型转换为类类型,并且这种转换是隐式的,即这个转换动作是由编译器来完成的,不需要程序员提供一个明确的操作。例如：

```
class A
{ …
public:
    A();
    A(int);
};                                  //A 类的定义
```

```
…
f(A a)                          //f()函数的定义,f()函数的形参为 A 类的对象
{…}
f(1);                           //f()函数的调用
```

上述程序代码段中语句“f(1);”进行了 f()函数的调用,进行 f()函数调用时首先通过转换构造函数 A(int)进行隐式类型转换,将 int 型实参 1 隐式转换成 A 类的对象,然后把 A 类的对象传递给函数 f 的形式参数 a。

注意:这种隐式转换的确为程序员提供了方便,但有时也会导致一些无法预料的错误,而这些错误往往细微得难以察觉。这时,宁愿关闭这种隐式类型转换动作,以保证程序的正确性。如果不想让转换构造函数生效,也就是拒绝其他基本数据类型通过转换构造函数转换为类类型,可以在转换构造函数前面加上 explicit 关键字。声明为 explicit 的构造函数不能在隐式转换中使用。使用 explicit 的好处是:将难以察觉的、后果严重的运行期错误变成了容易改正的编译期错误。

2) 类类型转换函数(类类型→基本数据类型)

通过构造函数进行类类型转换只能从参数类型向类类型转换,类类型转换函数用来将类类型向基本数据类型转换。类类型转换函数的定义和使用分为以下三个步骤。

(1) 在类定义体中声明:

```
operator type();
```

其中,type 为要转换的基本类型名。此函数既没有参数,又没有返回类型,但在函数体中必须返回具有 type 类型的一个值。

(2) 定义转换函数的函数体:

```
类名::operator type()
{
    …
    return type 类型的值;
}
```

(3) 使用类类型转换函数。

使用类类型转换函数与对基本类型进行强制转换一样,就像是一种函数调用过程。

【例 3.14】 类类型转换函数示例。

```
// ch3_14.cpp
#include <iostream>
using namespace std;

class RMB
{
public:
    RMB(double value = 0.0)                    //转换构造函数
    {
        yuan = value;
```

```
        jf = ( value - yuan ) * 100 + 0.5;
    }
    operator double()                                    //类类型转换函数
    {
        return yuan + jf / 100.0;
    }
    void display()
    {
        cout << (yuan + jf / 100.0) << endl;
    }
private:
    unsigned int yuan;
    unsigned int jf;
};

int main()
{
    RMB d1(2.0), d2(1.5), d3;
    d3 = RMB((double)d1 + (double)d2);                   //显式转换
    d3.display();
    d3 = d1 + d2;
    /* 隐式转换,系统首先会调用类类型转换函数把对象 d1 和 d2 隐式转换为 double 数据类型,
然后进行相加,加完的结果再调用构造函数隐式转换为 RMB 类类型,赋值给 d3 */
    d3.display();

    return 0;
}
```

程序的运行结果如图 3.22 所示。

图 3.22 例 3.14 的运行结果

【程序解析】

若在转换构造函数前面加上 explicit 关键字,如 explicit RMB(double value=0.0);,则转换构造函数不再进行隐式类型转换,当编译语句"d3=d1+d2;"时会出现如图 3.23 所示的编译错误提示信息。

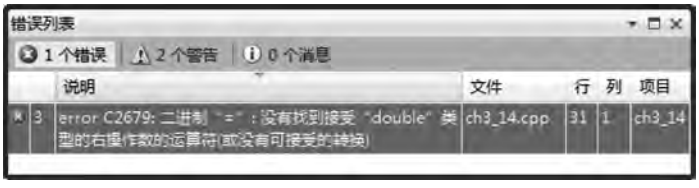

图 3.23 编译错误提示信息

3.3.2 析构函数

类的另一个特殊的成员函数是析构函数。析构函数的功能是当对象被撤销时,释放该对象占用的内存空间。析构函数的作用与构造函数正好相反,一般情况下,析构函数执行构造函数的逆操作。在对象消亡时,系统将自动调用析构函数,执行一些在对象撤销前必须执行的清理任务,例如将构造函数中动态分配的内存空间释放掉。

与构造函数相同的是在定义析构函数时,不能指定任何的返回类型,也不能使用 void。与构造函数不同的是构造函数可以带参数,可以重载,而析构函数没有参数,每个类只能有一个析构函数。若未显式编写自己的析构函数,编译器会提供一个默认析构函数,析构函数的函数名为类名前加~。

1. 析构函数被自动调用的三种情况

(1) 一个动态分配的对象被删除,即使用 delete 删除对象时,编译系统会自动调用析构函数。

(2) 某个对象的生命周期结束时。

(3) 一个编译器生成的临时对象不再需要时。

2. 析构函数的手工调用

除对象数组之外,构造函数只能由系统自动调用,而析构函数可以使用下述方法手工调用:

```
对象名.类名::析构函数名();
```

但一般情况下,不显式调用析构函数,而由系统自动调用。

3. 析构函数与构造函数的调用顺序

构造函数和析构函数的调用顺序刚好相反,在同一作用域中先构造后析构。

【例 3.15】 析构函数和构造函数的调用顺序示例。

```
// ch3_15.cpp
#include <iostream>
using namespace std;

class Student{
public:
    Student(char * pName = "no name", int ssId = 0)
    {
        strncpy_s(name,pName,40);
        name[39] = '\0';
        id = ssId;
        cout <<"Constructing new student " << pName << endl;
    }
    Student(Student& s)                              //拷贝构造函数
    {
        cout <<"Constructing copy of "<< s.name << endl;
        strcpy_s(name, "copy of ");
        strcat_s(name,s.name);
```

```
        id = s.id;
    }
    ~Student()
    {
        cout <<"Destructing " << name <<"\t"<< id << endl;
    }
private:
    char name[40];
    int id;
};

void fn(Student s)
{
    cout <<"In function fn()\n";              //fn()函数调用结束时,析构对象 s
    Student zhang("zhang",3);
    static Student zhao("zhao",4);
}

int main()
{
    Student randy("Randy",1);                 //调用构造函数,创建对象 Randy
    Student wang("wang",2);                   //调用构造函数,创建对象 Wang
    cout <<" --- Calling fn() --- \n";
    fn(randy);                                //调用 fn()函数,参数传递时调用拷贝构造函数
    cout <<" --- Returned from fn() --- \n";
    //主函数调用结束时,先析构对象 wang,再析构对象 Randy

    return 0;
}
```

程序的运行结果如图 3.24 所示。

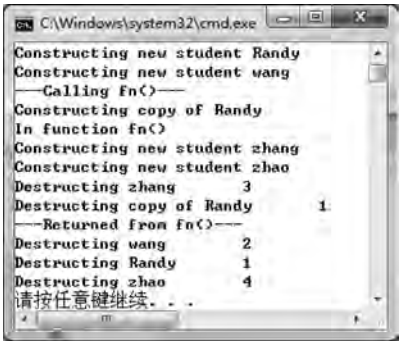

图 3.24 例 3.15 的运行结果

3.4 类的聚集——对象成员

类的聚集也称为对象成员,是指在类的定义中数据成员可以为其他类的对象,即类对象作为另一个类的数据成员。

如果在类定义中包含有对象成员,则在创建类对象时先调用对象成员的构造函数,再调

用类本身的构造函数。析构函数和构造函数的调用顺序正好相反。

从实现的角度讲,实际上是首先调用类本身的构造函数,在执行本身构造函数的函数体之前,调用对象成员的构造函数,然后再执行类本身构造函数的函数体。因此,在构造函数的编译结果中包含了对对象成员的构造函数的调用,至于调用对象成员的哪一个构造函数,是由成员初始化表指定的;当成员初始化表为空时,则调用对象成员的默认构造函数,这一点解释了当类没有提供任何构造函数时,为什么编译系统要为之产生一个默认构造函数的原因。

【例 3.16】 含有对象成员的类的构造函数和析构函数的调用顺序示例一。

```
// ch3_16.cpp
#include <iostream>
using namespace std;

class StudentID{
public:
    StudentID(int id = 0)                    //带默认参数的构造函数
    {
        value = id;

        cout <<"Assigning student id " << value << endl;
    }
    ~StudentID()
    {
        cout <<"Destructing id " << value << endl;
    }
private:
    int value;
};

class Student{
public:
    Student(char * pName = "no name", int ssID = 0):id(ssID)
    {
        cout <<"Constructing student " << pName << endl;
        strncpy_s(name, pName, sizeof(name));
        name[sizeof(name) - 1] = '\n';
    }
    ~Student()
    {
        cout <<"Deconstructing student  "<< name << endl;
    }
protected:
    char name[20];
    StudentID id;                            //对象成员
};

int main()
{
```

```
    Student s("wang",9901);
    Student t("li");
    cout <<" ----------------------- "<< endl;

    return 0;
}
```

程序的运行结果如图 3.25 所示。

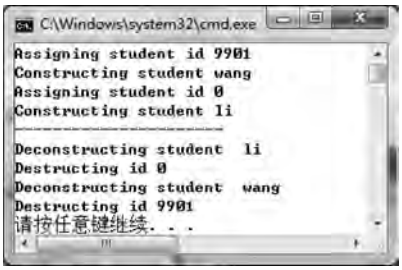

图 3.25 例 3.16 的运行结果

【例 3.17】 含有对象成员的类的析构函数和构造函数的调用顺序示例二。

```
// ch3_17.cpp
#include <iostream>
using namespace std;

class Student                                    //学生类的定义
{
public:
    Student()
    {
        cout <<"constructing student."<< endl;
        semesHours = 100;
        gpa = 3.5;
    }
    ~Student()
    {
        cout <<"destructing student."<< endl;
    }
protected:
    int semesHours;
    float gpa;
};

class Teacher                                    //教师类的定义
{
public:
    Teacher()
    {
        cout <<"constructing teacher.\n";
```

```
    }
    ~Teacher()
    {
        cout <<"destructing teacher.\n";
    }
};

class Tutorpair                                    //助教类的定义
{
public:
    Tutorpair()
    {
        cout <<"constructing tutorpair."<< endl;
        nomeeting = 0;
    }
    ~Tutorpair()
    {
        cout <<"destructing tutorpair."<< endl;
    }
protected:
    Student student;
    Teacher teacher;
    int nomeeting;                                 //会晤时间
};

int main()
{
    Tutorpair tp;
    cout <<" ------- back to main function. ------- "<< endl;

    return 0;
}
```

程序的运行结果如图 3.26 所示。

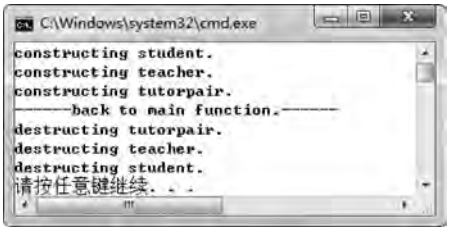

图 3.26　例 3.17 的运行结果

3.5　静 态 成 员

类是一种自定义的数据类型,类中定义了数据成员和成员函数。每个该类对象都有该类数据成员的副本。如果需要所有对象共享某个数据成员时,则可以使用关键字 static 将

需要共享的数据成员声明为类的静态数据成员。例如,将鼠标的位置、状态及其操作封装为一个类,不管该类有多少个对象,鼠标始终只有一个,所有的该类对象共享鼠标的位置、状态等数据成员的值。

静态成员的特征是不管这个类创建了多少个对象,其静态成员只有一个副本,此副本被这个类的所有对象共享。静态成员分为静态数据成员和静态成员函数。实际上,这种成员属于类本身,而不属于类的某一个对象。

1. 静态数据成员

静态数据成员被存放在内存的某一单元内,该类的所有对象都可以访问它。无论建立多少个该类的对象,都只有一个静态数据的副本。即使没有创建任何一个该类对象,类的静态成员在存储空间中也是存在的,可以通过名字解析运算符来直接访问。含有静态数据成员的类在创建对象时不为静态数据成员分配存储空间,例如下面的类。

```
class A
{
    static float x, y;
    int a, b;
};
```

在创建一个 A 类的对象时,编译器只分配存储两个整型数据的空间。

实际上可以将静态数据成员看成是一个全局变量,将其封装在某个类中通常有两个目的。

(1) 限制该变量的作用范围。例如,将其放在类的私有部分声明,则它只能由该类对象的成员函数直接访问。

(2) 将意义相关的全局变量和相关的操作放在一起,可以增加程序的可读性和可维护性。

由于静态数据成员仍是类成员,因而具有很好的安全性能。当这个类的第一个对象被建立时,所有 static 数据都被初始化,并且,以后再建立对象时,就无须再对其初始化。初始化在类体外进行,其格式如下。

```
数据类型 类名::静态数据成员名 = 初始值;
```

对上面的类 A 的静态成员 x、y 初始化的方法如下。

```
float A::x = 5.0f;
float A::y = 10.0f;
```

如果整个程序分为多个文件进行分块编译,则应该将类的声明放在头文件中,然后将静态成员的初始化放在某一个源文件中。这一点类似于 C 语言中全局变量的使用,类声明中静态变量的声明相当于外部变量声明,而静态变量的初始化相当于全局变量的定义。如果将静态变量的初始化放在头文件中被多个源程序文件包含,会导致变量的重复初始化。

【例 3.18】 静态数据成员的使用示例。

```
// ch3_18.cpp
#include <iostream>
using namespace std;
```

```
class  A
{
public:
    A()
    {
        numbers++;
    }
    int  getNumbers()
    {
        return  numbers;
    }
    static  int  numbers;                    //定义静态数据成员
};
int  A::numbers = 0;                         //静态数据成员的初始化

int  main()
{
    cout << A::numbers << endl;
    A  a1,a2,a3;
    cout << A::numbers <<'\t'
        << a1.getNumbers()<<'\t'
        << a2.getNumbers()<<'\t'
        << a3.getNumbers()<< endl;
    //显示均为 3(因为创建三个对象,三次使得静态数据成员自增 1)
    return 0;
}
```

程序的运行结果如图 3.27 所示。

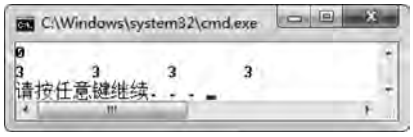

图 3.27　例 3.18 的运行结果

注意：静态数据成员初始化时，不能加 static 关键字。

2. 静态成员函数

C++语言中类的成员函数也可以定义为静态的，它的作用与静态数据成员类似，可以将它看成是全局函数，将其封装在某个类中的目的与静态数据成员相同。

静态成员函数具有如下的特点。

(1) 静态成员函数无 this 指针，它是同类的所有对象共享的资源，只有一个共用的副本，因此它不能直接访问非静态的数据成员，必须要通过某个该类对象才能访问。而一般的成员函数中都含有一个 this 指针，指向对象自身，可以直接访问非静态的数据成员。

请分析以下的代码段：

```
class X
{
public:
    static void func(int i, int j, X obj);
private:
    int member_int;
```

```
    static int static_int;
};

int X::static_int = 0;
void X::func(int i, int j, X obj)
{
    member_int = i;                          //编译时会出现如下的错误提示信息
    /* illegal reference to data member 'X::member_int' in a static member function */
    static_int = j;                          //正确,static_int 为静态成员
    obj.member_int = i;                      //正确,指定了被引用的对象
}
```

(2) 在静态成员函数中访问的是静态数据成员或全局变量。

(3) 由于静态成员函数属于类独占的成员函数,因此访问静态成员函数的消息接收者不是类对象,而是类自身。在调用静态成员函数的前面,必须缀上类名或对象名,经常用类名。

(4) 一个类的静态成员函数与非静态成员函数不同,调用静态成员函数时无须向它传递 this 指针,它不需要创建任何该类的对象就可以被调用。静态成员函数的使用虽然不针对某一个特定的对象,但在使用时系统中最好已经存在此类的对象,否则无意义。

(5) 静态成员函数不能是虚函数(虚函数的概念参见第 6 章),非静态成员函数和静态成员函数不能具有相同的名字和参数类型。

【例 3.19】 静态数据成员和静态成员函数使用示例。

```
// ch3_19.cpp
#include <iostream>
using namespace std;

class Student{
public:
    Student(char * pName = "no name")
    {
        cout <<"create one student\n";
        strncpy_s(name,pName,40);
        name[39] = '\0';
        numbersOfStudent++;                  //静态成员:每创建一个对象,学生人数增加 1
        cout <<"现有 "<< numbersOfStudent <<" 个学生"<< endl;
    }
    ~Student()
    {
        cout <<"destruct one student\n";
        numbersOfStudent--;                  //每析构一个对象,学生人数减 1
        cout <<"现有 "<< numbersOfStudent <<" 个学生"<< endl;
    }
    static int getNumbers()                  //静态成员函数
    {
        return numbersOfStudent;
    }
private:
```

```
    static int numbersOfStudent;                    //若写成 numbersOfStudent = 0;则非法
    char name[40];
};
int Student::numbersOfStudent  = 0;                 //静态数据成员在类外分配空间和初始化

void fn()
{
    cout <<" -------- In fn function -------- "<< endl;
    Student s1;
    Student s2;
}

int main()
{
    fn();
    cout <<" -------- Back to main function -------- "<< endl;
    //调用静态成员函数用类名引导
    cout <<"现有 "<< Student::getNumbers()<<" 个学生"<< endl;

    return 0;
}
```

程序的运行结果如图 3.28 所示。

此例中,如果在 main()函数中 fn(); 语句之前增加如下语句。

```
Student wang;
```

则程序的运行结果如图 3.29 所示。

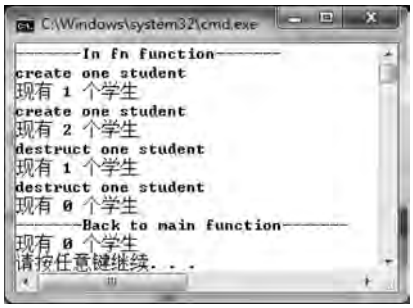

图 3.28　例 3.19 的运行结果

图 3.29　修改例 3.19 后的运行结果

请读者仔细分析此运行结果。

【例 3.20】 静态成员(学生链表的构建和使用)示例。

```
// Student.h
#pragma once
class Student
```

```
{
public:
    Student(void);
    ~Student(void);
    Student(char * pName);
protected:
    static Student * pFirst;
    Student * pNext;
    char name[40];
};

// Student.cpp
#include "Student.h"
#include <iostream>
using namespace std;

Student * Student::pFirst = 0;
Student::Student(void)
{
}

Student::Student(char * pName)
{
    cout <<"Construct "<< pName << endl;
    strncpy_s(name,pName,sizeof(name));
    name[sizeof(name) - 1] = '\0';
    pNext = pFirst;                          //每新建一个结点(对象),就将其挂在链首
    pFirst = this;
}

Student::~Student(void)
{
    cout << "Deconstruct "<< name << endl;
    if(pFirst == this)
    {
        //如果要删除链首结点,则只需将链首指针指向下一个结点
        pFirst = pNext;
        return;
    }
    for(Student * pS = pFirst; pS; pS = pS -> pNext)
        if(pS -> pNext == this){             //找到时,pS 指向当前结点的前一结点
            pS -> pNext = pNext;             //pNext 即 this -> pNext
            return;
        }
}

// ch3_20.cpp
#include <iostream>
#include "Student.h"
using namespace std;
```

```
Student * fn()
{
    cout <<" ------- In fn() ------- "<< endl;
    Student * p = new Student("Jenny");
    Student s3("Jone");
    return p;
}

int main()
{
    Student s1("Jamsa");
    Student * ps = fn();
    cout <<" ------- Back to main() ------- "<< endl;
    Student s2("Tracey");
    delete ps;

    return 0;
}
```

程序的运行结果如图 3.30 所示。

图 3.30　例 3.20 的运行结果

【程序解析】

(1) 当程序运行时,析构对象时会调用析构函数,输出该对象的 name 数据成员值。

(2) 程序首先创建对象 s1(其 name 成员值为 Jamsa),然后调用 fn()函数。在 fn()函数体中首先使用 new 运算符为 p 进行动态内存分配(此时,要调用 Student 类的构造函数,该内存中的 name 成员值为 Jenny),然后创建对象 s3(其 name 成员值为 Jone),之后把 p 返回给主函数中的 ps 指针变量(即 ps 指向动态内存单元),这时函数 fn()调用结束,系统会自动释放 fn()函数中的局部变量所占的内存单元,所以 p 不再指向分配的动态内存单元(但是该动态内存单元未释放,使用 delete 运算符或整个程序运行结束才会释放该动态内存单元),系统自动析构局部于 fn()函数的 s3 对象,输出 Jone。

(3) 接着,执行 main()函数中的第 4 条语句,创建对象 s2(其 name 成员值为 Tracey)。

(4) 然后执行 main()函数中的第 5 条语句,释放 ps 指向的动态内存单元(此时会调用 Student 类的析构函数),输出 Jenny。

(5) main()函数中所有语句执行完后,系统会释放局部变量所占的内存单元。先构造的后析构,所以首先析构 s2,输出 Tracey; 然后析构 s1,输出 Jamsa。整个程序运行结束。

学习了上面的两个实例后,再来分析下面的程序代码段:

```
// ch3_20_1.cpp
#include <iostream>
#include <string>
using namespace std;
```

```
class MyString                                  //定义一个字符串类
{
public:
    MyString(char * s)                          //定义构造函数
    {
        length = strlen(s);                     //取字符串长度
        contents = new char[length + 1];        //为字符串分配存储空间
        strcpy_s(contents,length + 1,s);        //字符串复制
    }

    static int set_total_length()
    {
        total_length += length;
        return total_length;
    }
    ~MyString()
    {
        delete[]contents;
    }
private:
    static int total_length;                    //定义一个静态变量,用来存放所有字符串的总长度
    int length;                                 //存放此字符串长度的变量
    char * contents;                            //存放字符串内容
};

int MyString::total_length = 0;                 //给静态变量赋初值

int main()
{
    MyString obj1("the first object");
    cout << obj1.set_total_length()<< endl;
    MyString obj2("the second object");
    cout << obj2.set_total_length()<< endl;

    return 0;
}
```

【程序解析】

上面的程序段中,static int set_total_length()是静态成员函数,不含有 this 指针。在此函数体中,下列语句存在问题:

```
total_length += length;
```

其中,total_length 是静态数据成员,有一个共用副本,在这里使用它没有任何问题。但 length 是一个一般的数据成员,每个对象都有自己的一个副本,但静态成员函数中无 this 指针,它无法判断当前是哪个对象,因而无法取值,所以在这里此种表达方式是不合适的。上面的静态成员函数有以下两种改法。

方法一：

```
static int set_total_length(MyString obj)
{
    total_length += obj.length;
    return total_length;
}
```

如果采用此种改法，则调用此静态成员时，实参对象要传递给形参对象，系统会调用默认拷贝构造函数(浅拷贝)，而在此例中构造函数使用 new 运算符进行了动态内存分配，析构函数使用 delete 运算符进行了动态内存释放。所以用户还必须自定义深拷贝构造函数如下：

```
MyString(MyString &str)
{
    length = str.length;
    contents = new char[length + 1];
    strcpy(contents,str.contents);
}
```

方法二：

```
static int set_total_length(MyString & obj)
{
    total_length += obj.length;
    return total_length;
}
```

相应地修改 main()函数如下：

```
int main()
{
    MyString obj1("the first object");
    cout << MyString::set_total_length(obj1)<< endl;
    MyString obj2("the second object");
    cout << MyString::set_total_length(obj2)<< endl;

    return 0;
}
```

程序的运行结果如图 3.31 所示。

图 3.31　程序的运行结果

【例 3.21】 编写程序：已有若干学生数据，包括学号、姓名、成绩，要求输出这些学生数据并计算所有学生的平均分。

```
// ch3_21.cpp
#include <iostream>
```

```
using namespace std;

class Student
{
public:
    Student(int id1,char name1[],int score1)
    {
        id = id1;
        score = score1;
        strcpy_s(name, name1);
        sum += score1;                    //求学生成绩总和
        number++;                         //学生人数自增
    }
    static double average()
    {
        return sum / number;              //求所有学生的平均分
    }
    void display()
    {
        cout << id <<'\t'<< name <<'\t'<< score << endl;
    }
private:
    int id;
    char name[10];
    int score;
    static double sum;
    static int number;
};

double Student::sum = 0.0;
int Student::number = 0;

int main()
{
    Student s1(1,"Li",89),s2(2,"Chen",78),s3(3,"Zheng",95);
    cout <<"学号    姓名    成绩"<< endl;
    s1.display();
    s2.display();
    s3.display();
    cout <<"所有学生的平均分 = "<< Student::average()<< endl;

    return 0;
}
```

程序的运行结果如图 3.32 所示。

【程序解析】

本例中设计了学生类 Student，除了包括 id(学号)、name(姓名)和 score(成绩)数据成员外，还有两个静态数据成员 sum 和 number，分别用来存放学生总分和总人数，另有一个构造函数、一个普通成员函数 display()和一个静态成员函数 average()，average()函数用于计算所有学生的平均分。

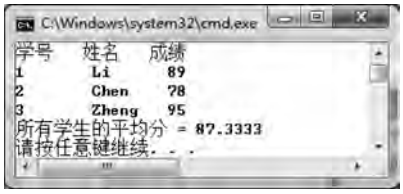

图 3.32 例 3.21 的运行结果

3.6 指向类成员的指针

C++语言提供一种特殊的指针类型,它指向类的成员,而不是指向该类的一个对象中该成员的一个实例,这种指针称为指向类成员的指针。

1. 由类外指向类内的指针变量

语法声明格式:

```
数据类型  类名::*指针名=类中数据成员地址描述;
```

【例 3.22】 类外指向类内的指针变量示例。

```
// ch3_22.cpp
#include <iostream>
using namespace std;

class  A
{
public:
    int  i,*p;
    A(){i=10;p=&i;}
};

int  A::*p=&A::i;                          //p是类外指针变量

int  main()
{
    A  aa,bb;                              //按默认构造函数的安排,aa、bb中的i初值都是10
    (bb.*p)++;                             //括号不可缺,否则编译器将理解为非法操作
    --*aa.p;
    cout<<"AA:"<<aa.*p<<"  BB:"<<bb.*p<<"\n";
    cout<<"AA:"<<*aa.p<<"  BB:"<<*bb.p<<endl;

    return 0;
}
```

程序的运行结果如图 3.33 所示。

例 3.22 中的指针变量 p 出现在两处,一处是类 A 的构造函数中(p=&i),另一处是全局数据区中。在本例中它们彼此并不冲突,各行其是。要特别注意由类外指向类内的指针数据在使用时的写法:

图 3.33 例 3.22 的运行结果

```
类所定义的对象名.指针名…
```

这种在类外定义的指向类内数据成员的指针在使用上由于受类的数据封装特性的限

制,其所指成员只能声明具有 public 属性。

2. 类外指向成员函数的指针数据

语法声明格式:

```
数据类型(类名::*指针名)(参数类型表) = &类名::函数名;
```

与指向数据成员的指针相同,这里的成员函数也只能声明具有 public 属性。

【例 3.23】 类外指向成员函数的指针变量的示例。

```
// ch3_23.cpp
#include <iostream>
using namespace std;

class  A
{
public:
    int set(int k)
    {
        i = ++k;
        return i;
    }
private:
    int i;
};

int main()
{
    int  (A::*f)(int) = &A::set;
    A  aa;
    cout <<(aa.*f)(10)<< endl;          //括号不能省略

    return 0;
}
```

程序的运行结果如图 3.34 所示。

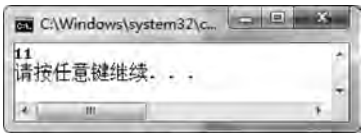

图 3.34 例 3.23 的运行结果

3. 指向类内静态成员的指针

静态成员由于可与对象分离,唯一地存放于公用的全局区中,所以指向静态成员的指针也就简化了。与之前介绍的指针的不同之处在于以下两点。

(1) C++语言不检查静态成员处于何处(实际上全放在全局数据区)。

(2) 必须在全局区用"数据类型 类名::静态变量名[=…];"的格式来指明其作用域或初值。

例如:

```
class  A
{
```

```
    static int i;
};
int A::i = 0;
```

这里可以对 private 区中的变量赋初值且并不会改变其封装特性,即只能由本类中的函数才能访问。在定义静态成员函数的指针时,可以省去类作用域的描述。

例如:

```
class A
{
public:
    static void set(int k){…}
};
void( * f)(int) = &A::set;                    //( * f)前省去了::
```

【例 3.24】 指向静态成员的指针示例。

```
// ch3_24.cpp
# include <iostream>
using namespace std;

class A
{
public:
    static void set(int k)
    {
        i = k;i++;
    }
private:
    static int i;
    friend class B;
};

class B
{
public:
    static void ds(int l)
    {
        int  * p = &A::i;
        cout << * p << endl;
         * p = l;cout << * p << endl;
    }
};

int  A::i = 0;
void  ( * f1)(int) = &A::set;
void  ( * f2)(int) = &B::ds;

int main()
{
```

```
    f1(10);
    f2(20);

    return 0;
}
```

程序的运行结果如图 3.35 所示。

图 3.35 例 3.24 的运行结果

请特别注意例中两种指针使用的格式。由于不需要对象,所以使用时指针前的对象名可以省略。

3.7 综 合 示 例

【例 3.25】 实现一个大小固定的整型数据元素集合及其相应操作。

实现该功能的项目中包含 Set.h 头文件,Set.cpp 和 ch3_25.cpp 两个源程序文件。程序代码如下:

```
// Set.h
#pragma once

const int maxCard = 16;                    //集合中元素个数的最大值,默认为 int 型
enum ErrCode {noErr, overflow};            //错误代码
enum Bool {False, True};                   //Bool 类型定义

class Set                                  //定义集合类
{
public:
    Set(void);
    ~Set(void);
    void EmptySet(){card = 0;}
    Bool Member(int);                      //判断一个数是否为集合中的元素
    ErrCode AddElem(int);                  //向集合中添加元素
    void RmvElem(int);                     //删除集合中的元素
    void Copy(Set *);                      //把当前集合复制到形参指针指向的集合中
    Bool Equal(Set *);                     //判断两个集合是否相等
    void Print();
    void Intersect(Set *, Set *);          //交集
    ErrCode Union(Set *, Set *);           //并集
private:
    int elems[maxCard];                    //存储元素的数组
    int card;                              //集合中元素的个数
```

```
};

// Set.cpp
#include "Set.h"
#include <iostream>
using namespace std;

Set::Set(void)
{
}

Set::~Set(void)
{
}

Bool Set::Member(int elem)
{
    for(int i = 0;i < card;++i)
        if(elems[i] == elem)
            return True;
    return False;
}

ErrCode Set::AddElem(int elem)
{
    if(Member(elem))
        return noErr;
    if(card < maxCard)
    {
        elems[card++] = elem;
        return noErr;
    }
    return overflow;
}

void Set::RmvElem(int elem)
{
    for(int i = 0; i<card; ++i)
        if(elems[i] == elem)
        {
            for(;i<card-1;++i)
                elems[i] = elems[i+1];
            --card;
            return;
        }
}

void Set::Copy(Set *set)
{
    for(int i = 0; i<card; ++i)
        set->elems[i] = elems[i];
```

```
    set->card = card;
}

Bool Set::Equal(Set *set)
{
    if(card != set->card)
        return False;
    for(int i = 0; i < card;++i)
        //判断当前集合的某元素是否是 Set 所指集合中的元素
        if(!set->Member(elems[i]))
            return False;
    return True;
}

void Set::Print()
{
    cout <<"{";
    for(int i = 0; i < card; ++i)
        cout << elems[i]<< ";";
    cout <<"}\n";
}

void Set::Intersect(Set *set, Set *res) //交集: *this∩*set->*res
{
    res->card = 0;
    for(int i = 0; i < card; ++i)
        for(int j = 0; j < set->card; ++j)
            if(elems[i] == set->elems[j]){
                res->elems[res->card++] = elems[i];
                break;
            }
}

ErrCode Set::Union(Set *set,Set *res)  //并集: *set∪*this->*res
{
    set->Copy(res);
    for(int i = 0; i < card; ++i)
        if(res->AddElem(elems[i]) == overflow)
            return overflow;
    return noErr;
}

// ch3_25.cpp
#include "Set.h"
#include <iostream>
using namespace std;

// 下面是测试用的程序代码
int main()
{
    Set s1, s2, s3;
```

```
    s1.EmptySet();
    s2.EmptySet();
    s3.EmptySet();
    s1.AddElem(10);
    s1.AddElem(20);
    s1.AddElem(30);
    s1.AddElem(40);
    s2.AddElem(30);
    s2.AddElem(50);
    s2.AddElem(10);
    s2.AddElem(60);
    cout <<"s1 = "; s1.Print();
    cout <<"s2 = "; s2.Print();
    s2.RmvElem(50);
    cout <<"s2 - {50} = "; s2.Print();
    if(s1.Member(20))
        cout <<"20 is in s1\n";
    s1.Intersect(&s2,&s3);
    cout <<"s1 intsec s2  = "; s3.Print();
    s1.Union(&s2,&s3);
    cout <<"s1 union s2  = "; s3.Print();
    if(!s1.Equal(&s2))
        cout <<"s1!= s2\n";

    return 0;
}
```

程序的运行结果如图 3.36 所示。

图 3.36　例 3.25 的运行结果

【例 3.26】 实现一个大小可变的整型数据元素集合，集合可存储的数据元素个数在对象构造时给定，由构造函数为数据元素分配存储空间，在对象被释放时由析构函数释放存储空间。

实现该功能的项目中包含 Set.h 头文件，Set.cpp 和 ch3_26.cpp 两个源程序文件。程序代码如下：

```
// Set.h
#pragma once

const int maxCard = 16;                    //集合中元素个数的默认最大值
enum ErrCode {noErr, overflow};            //错误代码
enum Bool {False, True};                   //Bool 类型定义
class Set
{
public:
    Set(int sz = maxCard);
    ~Set();
    Bool Member(int);                      //判断一个数是否为集合中的元素
    ErrCode AddElem(int);                  //向集合中添加元素
```

```
    void RmvElem(int);                  //删除集合中的元素
    void Copy(Set *);                   //把当前集合默认到形参指针指向的集合中
    Bool Equal(Set *);                  //判断两个集合是否相等
    void Print();
    void Intersect(Set *, Set *);       //交集
    ErrCode Union(Set *, Set *);        //并集
private:
    int size;                           //元素的最大个数
    int *elems;                         //存储元素的数组
    int card;                           //集合中元素的个数
};

// Set.cpp
#include "Set.h"
#include <iostream>
using namespace std;

Set::Set(int sz)
{
    card = 0;
    size = sz;
    elems = new int[size];
}

Set::~Set(void)
{
    delete []elems;
}

Bool Set::Member(int elem)
{
    for(int i = 0;i < card;++i)
        if(elems[i] == elem)
            return True;
    return False;
}

ErrCode Set::AddElem(int elem)
{
    if(Member(elem))
        return noErr;
    if(card < size)
    {
        elems[card++] = elem;
        return noErr;
    }
    return overflow;
}

void Set::RmvElem(int elem)
{
```

```
    for(int i = 0;i < card;++i)
        if(elems[i] == elem)
        {
            for(;i < card - 1;++i)
                elems[i] = elems[i + 1];
            --card;
            return;
        }
}

void Set::Copy(Set * set)
{
    if(set -> size < size)
    {
        delete [ ]set -> elems;
        set -> elems = new int [size];
        set -> size = size;
    }
    for(int i = 0;i < card;++i)
        set -> elems[i] = elems[i];
    set -> card = card;
}

Bool Set::Equal(Set * set)
{
    if(card!= set -> card)
        return False;
    for(int i = 0;i < card;++i)
        if(!set -> Member(elems[i]))
            return False;
    return True;
}

void Set::Print()
{
    cout <<"{";
    for(int i = 0;i < card - 1;++i)
        cout << elems[i]<< ";";
    if(card > 0)
        cout << elems[card - 1];
    cout <<"}\n";
}

void Set::Intersect(Set * set, Set * res)
{
    if(res -> size < size)
    {
        delete [ ]res -> elems;
        res -> elems = new int[size];
        res -> size = size;
    }
```

```
    res->card = 0;
    for(int i = 0;i < card;++i)
        for(int j = 0;j < set->card;++j)
            if(elems[i] == set->elems[j])
            {
                res->elems[res->card++] = elems[i];
                break;
            }
}

ErrCode Set::Union(Set *set, Set *res)
{
    if(res->size < size + set->size)
    {
        delete []res->elems;
        res->elems = new int[size + set->size];
        res->size = size + set->size;
    }
    set->Copy(res);
    for(int i = 0;i < card;++i)
        if(res->AddElem(elems[i]) == overflow)
            return overflow;
    return noErr;
}

// ch3_26.cpp
#include"Set.h"
#include <iostream>
using namespace std;

int main()
{
    Set s1, s2, s3;
    s1.AddElem(10);
    s1.AddElem(20);
    s1.AddElem(30);
    s1.AddElem(40);
    s2.AddElem(30);
    s2.AddElem(50);
    s2.AddElem(10);
    s2.AddElem(60);
    cout <<"s1 = "; s1.Print();
    cout <<"s2 = "; s2.Print();
    s2.RmvElem(50);
    cout <<"s2 - {50} = "; s2.Print();
    if(s1.Member(20))
        cout <<"20 is in s1\n";
    s1.Intersect(&s2,&s3);
    cout <<"s1 intsec s2 = "; s3.Print();
    s1.Union(&s2,&s3);
    cout <<"s1 union s2 = "; s3.Print();
```

```
    if(!s1.Equal(&s2))
        cout <<"s1!= s2\n";
    return 0;
}
```

程序的运行结果与例 3.25 相同。

习　　题

1. 为什么要引入构造函数和析构函数?

2. 类的公有、私有和保护成员之间的区别是什么?

3. 什么是拷贝构造函数,它何时被调用?

4. 设计一个计数器类,当建立该类的对象时其初始状态为 0,考虑为计数器定义哪些成员?

5. 定义一个时间类,能提供和设置由时、分、秒组成的时间,并编写出应用程序,定义时间对象,设置时间,输出该对象提供的时间。

6. 设计一个学生类 Student,它具有的私有数据成员是:注册号、姓名、数学成绩、英语成绩、计算机成绩;具有的公有成员函数是:求三门课总成绩的函数 sum();求三门课平均成绩的函数 average();显示学生数据信息的函数 print();获取学生注册号的函数 get_reg_num();设置学生数据信息的函数 set_stu_inf()。

编制主函数,说明一个 Student 类对象的数组并进行全班学生信息的输入与设置,而后求出每一学生的总成绩、平均成绩、全班学生总成绩最高分、全班学生平均分,并在输入一个注册号后,输出该学生有关的全部数据信息。

7. 模拟栈模型的操作,考虑顺序栈和链栈两种形式。

8. 写出下列程序的运行结果:

```
// ex3_8.cpp
#include <iostream>
using namespace std;

class Tx
{
public:
    Tx(int i,int j);
    ~Tx();
    void display();
private:
    int num1,num2;
};

Tx::Tx(int i,int j = 10)
{
    num1 = i;
    num2 = j;
```

```
    cout <<"Constructing "<< num1 <<"    "<< num2 << endl;
}

void Tx::display()
{
    cout <<"display: "<< num1 <<"    "<< num2 << endl;
}

Tx::～Tx()
{
    cout <<"Destructing "<< num1 <<"    "<< num2 << endl;
}

int main()
{
    Tx t1(22,11);
    Tx t2(20);
    t1.display();
    t2.display();

    return 0;
}
```