第3章

类和对象

3.1 学习要求

3.1.1 基本要求

(1) 掌握类和对象的概念、相互关系和定义方法；
(2) 掌握构造函数和析构函数的定义及使用方法；
(3) 掌握静态数据成员和静态成员函数；
(4) 掌握对象指针和对象作为函数参数；
(5) 掌握对象成员的使用方法。

3.1.2 基本知识点

(1) 类的定义；
(2) 类的三种访问控制权限 private、protected 和 public 的作用；
(3) 类中成员函数的内联和外联两种定义方式；
(4) C++的多文件结构；
(5) 对象的定义和使用；
(6) 类中成员的访问；
(7) this 指针的用法；
(8) 带默认参数的成员函数和成员函数的重载；
(9) 构造函数和析构函数的特点、调用方法及调用顺序；
(10) 对象数组；
(11) 拷贝构造函数、深拷贝构造函数使用的情况及定义；
(12) 构造初始化表及其使用注意事项；
(13) 类类型转换函数的定义及使用方法；
(14) 包含对象成员的构造函数和析构函数的调用顺序；
(15) 静态数据成员的作用以及静态成员函数的特点。

3.1.3 重点和难点

【重点】
(1) 类的三种访问控制权限；

(2) 构造函数和析构函数的作用及特点;

(3) 拷贝构造函数;

(4) 静态成员;

(5) this 指针。

【难点】

(1) this 指针;

(2) 深拷贝构造函数;

(3) 静态成员。

3.2 内 容 概 要

3.2.1 类和成员函数的定义

1. 类的定义

类是对一组具有共同的属性特征和行为特征的对象的抽象。类定义的一般格式如下:

```
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:
   // 保护的数据成员和成员函数
};
```

类中的数据称为类的数据成员,类中对数据的操作称为类的成员函数。private 成员实现了数据的隐藏,public 成员实现了外界和这个类的对象的相互作用,protected 成员用于继承机制中。

注意:在类中说明的任何数据成员和成员函数均不能使用 extern、auto 和 register 关键字进行修饰;类的默认访问权限是 private;由于定义类时并没有分配内存空间,所以在定义类时不能对类中的数据成员进行初始化;在类定义时不要丢掉类定义的结束标志:分号。

2. 成员函数的定义

成员函数的定义有内联定义和外联定义两种方法。

外联定义成员函数的具体形式为:

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

内联函数有两种定义方法,一种方法是在类定义体内直接定义成员函数,另一种方法是在外联函数前使用 inline 关键字,如:

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

或

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

注意：类中简单的成员函数一般使用内联定义的方式，这样可以提高程序的运行效率；类的成员函数可以重载，也可以使用默认参数值，具体使用方法同普通函数。

3.2.2 C++的多文件结构

在C++中，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。

3.2.3 对象

对象是类的实例，通过对象可以访问类的公有成员。每个对象占用内存中的不同区域，在建立对象时被分配保存数据所需要的内存，而成员函数代码由该类的所有对象所共享。除了可以定义普通对象外，还可以定义对象指针和对象引用。普通对象和对象引用访问类的成员时使用“.”运算符，而对象指针访问类的成员时使用->运算符。

通过对象、对象引用和对象指针等访问类的成员函数时，系统自动向其隐含传递一个this指针，该指针指向当前访问的对象。

一个类的数据成员可以是另一个类的对象，这称为类的聚集。对象可以用作函数参数，函数的返回值也可以是类类型的对象。

注意：

(1) 对象、对象引用和对象指针不能访问类的private和protected成员。

(2) 用关键字const修饰的对象称为常对象，其数据成员的值在对象的整个生存期内不能改变。

3.2.4 构造函数和析构函数

1. 构造函数

构造函数是一个与类同名，没有返回值的特殊成员函数。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。

构造函数可以重载,也可以为参数提供默认值。

注意:定义构造函数时不能指定返回类型,即使是 void 也不可以指定。

2. 默认构造函数

C++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++提供一个默认的构造函数,也叫缺省构造函数,该默认构造函数是个无参构造函数,它仅负责创建对象,而不做任何初始化工作。默认构造函数主要用来构造无参对象和对象数组。

注意:只要一个类定义了一个构造函数,C++就不再提供默认的构造函数。如果使用默认构造函数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。一般情况下,程序员要定义一个无参构造函数。

3. 拷贝构造函数

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,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拷贝到新创建的对象中。

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缺省拷贝构造函数,这个缺省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,是一种浅拷贝构造函数。拷贝构造函数的声明形式为:

```
类名(const 类名 & 对象名);
```

注意:拷贝构造函数的形参为当前类的引用,在拷贝构造函数体内可以访问形参的私有成员。

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 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 cat1;
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。

(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。

注意:在同时满足以下两个条件时,程序员必须要定义深拷贝构造函数。

(1) 肯定要调用拷贝构造函数;

(2) 数据成员包含指向堆内存的指针变量。

这时要在深拷贝构造函数体内为指针变量重新使用 new 运算符分配内存空间。

4. 构造初始化表

可以使用构造初始化表对数据成员进行初始化,其格式为:

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

注意：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。类中的常量和引用数据成员的初始化必须使用构造初始化表。

5. 类类型和基本数据类型的转换

(1) 构造函数用作类型转换(基本数据类型转换为类类型)

如果构造函数只有一个基本数据类型的形参，则该构造函数除了可以创建对象外，还可以实现基本数据类型转换为类类型。

(2) 类类型转换函数

类类型转换函数用来将类类型转换为基本数据类型。类类型转换函数的语法为：

```
类名::opertor primitiveDataType()
{
  …
  return primitiveDataType 类型的值;
}
```

注意：类类型转换函数既没有参数，又没有返回值类型，但在函数体中必须返回具有primitiveDataType 类型的一个值。

6. 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。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。析构函数的函数名为类名前加~。

C++编译器在类没有析构函数时为其产生的析构函数称为默认析构函数，其函数体为空。

注意：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；析构函数没有参数，不可以重载，每个类只能有一个析构函数；构造函数和析构函数的调用顺序刚好相反，先构造的后析构。

在下述三种情况下，系统会自动调用析构函数：

(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) 程序运行结束时；

(3) 一个编译器生成的临时对象不再需要时。

3.2.5 类的聚集——对象成员

对象成员也称为类的聚集，是指在类的定义中数据成员可以为其他类对象，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。含有对象成员的类的构造函数定义方式如下：

```
X::X(参数表 0):成员 1(参数表 1), 成员 2(参数表 2), …,成员 n(参数表 n)
{
   // 构造函数体
}
```

其中,参数表1、参数表2、……、参数表 n 一般均来自参数表0,用来为调用相应对象成员所在类的构造函数提供参数。

注意:对象成员的构造函数的调用顺序取决于它们在类中的说明顺序。

3.2.6 静态成员

静态成员分为静态数据成员和静态成员函数,静态成员使用 static 关键字进行修饰。

1. 静态数据成员

用关键字 static 修饰的数据成员不属于任何一个具体对象,而是整个类所有对象共有的。无论建立多少个该类的对象,都只有一个静态数据的拷贝。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。

静态数据成员的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);
```

注意:由于静态成员不属于任何一个对象,因此通过类名对其进行访问:“类名::静态成员”。

2. 静态成员函数

用 static 关键字修饰的成员函数称为静态成员函数。

注意:

(1) 静态成员函数无 this 指针,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。

(2) 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,必须缀上对象名或类名,经常用类名。

(3) 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需向它传递 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。

(4) 不能用 static 和 virtual 两个关键字同时修饰一个类的成员函数。

3.3 典型例题解析

【例 1-3-1】 每个类(　　)构造函数。

A. 只能有一个　　B. 只可有公有的　　C. 可以有多个　　D. 只可有默认的

【解析】 类的构造函数可以有多个,可以进行重载,可以用 private 进行修饰。

【答案】 C

【例 1-3-2】 假定 MyClass 为一个类,则执行 MyClass a, b(2), c[2], *p;语句时,自动调用该类构造函数(　　)次。

A. 2　　B. 3　　C. 4　　D. 5

【解析】 执行 MyClass a, b(2), c[2], *p;语句时,创建对象时系统自动调用 MyClass 类的构造函数。调用无参构造函数创建对象 a;调用带一个参数的构造函数创建

对象 b；创建了对象数组 c，该数组有 2 个元素，每个元素都是 MyClass 类的对象，所以会自动调用 2 次无参构造函数创建数组 c；创建了指向 MyClass 类型对象的指针变量 p，不调用构造函数。所以自动调用 4 次构造函数。

【答案】 C

【例 1-3-3】 下列静态数据成员的特性中，错误的是(　　)。

A. 说明静态数据成员时，前面要加 static 修饰符

B. 静态数据成员要在类体外进行初始化

C. 引用静态数据成员时，要在静态数据成员前加类名和作用域运算符

D. 静态数据成员不是所有对象所共有的

【解析】 静态数据成员是类的所有对象所共享的，创建该类对象前就已经为静态数据成员分配了内存空间。静态数据成员只能在类外进行初始化。

【答案】 D

【例 1-3-4】 下列有关类的静态成员函数的说法中，错误的是(　　)。

A. 不属于类的某一个对象，它具有 this 指针

B. 在静态成员函数的定义中不能直接访问类的非静态成员

C. 静态成员函数不能是虚函数

D. 可以通过由参数传递的类对象访问类的非静态成员

【解析】 静态成员函数不具有 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，不能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。静态成员函数不能是虚函数。

【答案】 A

【例 1-3-5】 假设在程序中已经定义了 Student 类，并建立了 Student 类对象 s1 和 s2。请说明以下几条语句有何区别？

(1) Student s1, s2;

(2) Student s2 = s1;

(3) Student s2(s1);

(4) s2 = s1;

【答】 (1) 使用无参构造函数或带默认参数的构造函数，创建了 Student 类的 2 个对象 s1 和 s2。

(2) 用已创建的对象 s1 创建对象 s2，调用了拷贝构造函数。

(3) 用已创建的对象 s1 创建对象 s2，调用了拷贝构造函数。该语句和语句 2 的作用相同，只是形式不同而已。

(4) 该语句是对象赋值语句，调用运算符重载函数，将对象 s1 数据成员的值逐域拷贝到对象 s2 中。

【例 1-3-6】 写出下列程序的运行结果。

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

class Test{
public:
```

```
    Test(int n){
        i = n;
        cout <<"call the constructor"<< endl;
    }
    ~Test(){
        cout <<"call the destructor"<< endl;
    }
    int getI(){
        return i;
    }
private:
    int i;
};

int sqrI(Test tt){
    return tt.getI() * tt.getI();
}

int main(){
    Test t(10);
    cout <<"the object's private member is "<< t.getI()<< endl;
    cout <<"the square of object's private member is "<< sqrI(t)<< endl;
    return 0;
}
```

【解析】 程序从 main()函数开始执行,首先调用构造函数创建对象 t,输出“call the constructor”;然后执行第二条语句,调用 getI()成员函数,输出“the object's private member is 10”;执行第三条语句时,首先进行函数调用 sqrI(t),把实参对象 t 传递给形参对象 tt,此时会自动调用默认拷贝构造函数把实参 t 的数据成员逐域复制给形参 tt 的数据成员,之后执行 sqrI()函数的函数体,该函数的返回值为 100,函数调用结束后,系统会自动调用析构函数释放局部于 sqrI()函数的形参 tt 所占的内存空间,输出“call the destructor”,然后输出“the square of object's private member is 100”;在执行语句“return 0;”之前调用析构函数释放局部于 main()函数中的对象 t。

【答】 程序的运行结果如下。

```
call the constructor
the object's private member is 10
call the destructor
the square of object's private member is 100
call the destructor
```

【例 1-3-7】 写出下列程序的运行结果。

```
#include<iostream.h>
class Sample
{
    int A;
    static int B;
public:
    Sample(int a){A = a,B += a;}
```

```
    static void func(Sample s);
};

void Sample::func(Sample s)
{
    cout<<"A = "<<s.A<<",B = "<<B<<endl;
}

int Sample::B = 0;

void main()
{
    Sample s1(2),s2(5);
    Sample::func(s1);
    Sample::func(s2);
}
```

【解析】 本题说明了静态成员函数的使用方法。其中的数据成员 B 是静态数据成员，求 B 的值是在构造函数中进行的。注意：静态成员函数与静态数据成员一样，也不是对象成员。静态成员函数的调用不同于普通的成员函数。在静态成员函数的实现中，引用类的非静态数据成员是通过对象进行的，如本题中 s.A，引用类的静态数据成员是直接进行的，如本题中的 B。

【答】 程序的运行结果如下。

```
A = 2,B = 7
A = 5,B = 7
```

3.4　课后习题解答

1. 为什么要引入构造函数和析构函数？

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

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)。

当定义对象时，编译程序自动调用构造函数。

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。

2. 类的公有、私有和保护成员之间的区别是什么？

【解】

(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 章讲到的友元函数)访问，而类的外部根本就无法访问，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，使内部数据不能被任意的访问和修改，也不会对该类以外

的其余部分造成影响,使模块之间的相互作用被降低到最小。

private 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对象中的数据。

(3) 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。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。

缺省访问控制(未指定 private、protected、public 访问权限)时,系统认为是私有 private 成员。

3. 什么是拷贝构造函数,它何时被调用?

【解】 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,当创建一个新对象时系统自动调用它;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拷贝到新创建的对象中。

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 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。

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?

【解】

```
// counter.h 计数器类的定义文件
#ifndef counter_h                  // 为了避免重复定义
#define counter_h

class counter{
private:
    int count;
public:
    counter();                     // 构造函数,用来进行初始化
    void setCount(int i);          // 设置数据成员的值
    int getCount();                // 访问数据成员的值
    void displayCount();           // 显示数据成员的值
    void incrementCount();         // 计数器自增
    void decrementCount();         // 计数器自减
```

```
    ~counter(){}                    // 析构函数
};

#endif

// counter.cpp 计数器类的实现文件
#include "counter.h"
#include<iostream>
using namespace std;

counter::counter(){
    count = 0;
}

void counter::displayCount(){
    cout << count << endl;
}

int counter::getCount(){
    return count;
}

void counter::setCount(int i){
    count = i;
}

void counter::incrementCount(){
    count++;
}

void counter::decrementCount(){
    count--;
}

// main.cpp 计数器类的使用文件
#include "counter.h"
#include<iostream>
using namespace std;

int main(){
    counter c1;
    c1.displayCount();
    c1.setCount(4);
    c1.displayCount();
    for(int i=0; i<=10;i++){
        c1.incrementCount();
        c1.displayCount();
    }

    return 0;
}
```

程序的运行结果如下。

```
0
4
5
6
7
8
9
10
11
12
13
14
15
```

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。

【解】

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

class Time{
    int hour,minute,second;
public:
    Time(int h = 0,int m = 0, int s = 0){
        hour = h;
        minute = m;
        second = s;
    }

    void setHour(int h){
        hour = h;
    }

    void setMinute(int m){
        minute = m;
    }

    void setSecond(int s){
        second = s;
    }

    void display(){
        cout << hour << ": " << minute << ": " << second << endl;
    }
};

int  main(){
    Time t1,t2(10,15,30);     // 定义对象 t1 和 t2
```

```
    cout <<"t1 对象的时间为: ";
    t1.display();
    cout <<"t2 对象的时间为: ";
    t2.display();

    // 修改 t1 对象的时分秒;
    t1.setHour(13);
    t1.setMinute(20);
    t1.setSecond(20);
    cout <<"修改后的 t1 对象的时间为: ";
    t1.display();

    return 0;
}
```

程序的运行结果如下。

```
t1 对象的时间为: 0: 0: 0
t2 对象的时间为: 10: 15: 30
修改后的 t1 对象的时间为: 13: 20: 20
```

6. 设计一个学生类 student,它具有的私有数据成员是：注册号、姓名以及数学、英语、计算机成绩；具有的公有成员函数是：求 3 门课总成绩的函数 sum；求 3 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和平均成绩、全班学生的总成绩最高分、全班学生总平均分,并在输入一个注册号后,输出有关该学生的全部数据信息。

【解】

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

class Student{
   private:
     int num;
     char name[10];
     float math;
     float english;
     float computer;
   public:
      void set_stu_inf(int n,char *ch,float m,float e,float c)
      {
          num=n; strcpy(name,ch); math=m; english=e; computer=c;
      }

      float sum()
      {
          return (math+english+computer);
```

```
    }

    float average()
    {
        return (math + english + computer)/3;
    }

    int get_reg_num()
    {
        return num;
    }

    void print()
    {
        cout <<"学号: "<< num << endl
             <<"姓名: "<< name << endl
             <<"数学: "<< math << endl
             <<"英语: "<< english << endl
             <<"计算机: "<< computer << endl
             <<"总分: "<< sum()<< endl
             <<"平均分: "<< average()<< endl;
    }
};

int main()
{
    Student stu[50];
    int i,q,a,z,x,max = 0,aver = 0;          // i:循环变量;q: 学号; a: 数学成绩;
                                             // z: 英语成绩; x: 计算机成绩
    int count  =  0;                         // 表示学生人数
    char * we = new char[10];

    // 输入学生信息
    for(;;)
    {
        cout <<"请输入学生的学号、姓名、数学成绩、英语成绩、计算机成绩:(若输入的学号为 0 则表
示退出)" << endl;
        cin >> q >> we >> a >> z >> x;
        if (q  == 0 )
            break;

        stu[count ++ ].set_stu_inf(q,we,a,z,x);

        if(max > a + z + x);
        else max = a + z + x;

        aver + = (a + z + x);
    }

    // 输出所有学生信息
    cout <<"学生信息为: "<< endl << endl;
```

```
    for( i = 0; i < count; i++ ){
        stu[i].print();
        cout << endl;
    }

    cout <<"全班学生总成绩最高分为"<< max << endl
        <<"全班学生总平均分为"<< aver/3 << endl << endl;

    cout <<"请输入要查的学生的学号:"<< endl;
    cin >> q;
    for( i = 0; i < count; i++ ){
        if (q == stu[i].get_reg_num())
        {
            cout <<"此学生信息为: "<< endl;
            stu[i].print();
            break;
        }
    }

    if (i == count)
        cout <<"查无此人"<< endl;

    return 0;
}
```

程序的运行结果如下。

```
请输入学生的学号、姓名、数学成绩、英语成绩、计算机成绩:(若输入的学号为 0 则表示退出)
1 李强 98 78 90
请输入学生的学号、姓名、数学成绩、英语成绩、计算机成绩:(若输入的学号为 0 则表示退出)
3 王蒙 87 75 97
请输入学生的学号、姓名、数学成绩、英语成绩、计算机成绩:(若输入的学号为 0 则表示退出)
0 李丽 89 78 78
学生信息为:

学号: 1
姓名: 李强
数学: 98
英语: 78
计算机: 90
总分: 266
平均分: 88.6667

学号: 3
姓名: 王蒙
数学: 87
英语: 75
计算机: 97
总分: 259
平均分: 86.3333
```

```
全班学生总成绩最高分为 266
全班学生总平均分为 175

请输入要查的学生的学号:
3
此学生信息为:
学号: 3
姓名: 王蒙
数学: 87
英语: 75
计算机: 97
总分: 259
平均分: 86.3333
```

7. 模拟栈模型的操作,考虑顺序栈和链栈两种形式。

【解】 栈是先进后出的一种数据结构,有很广泛的应用。例如,编译器用栈处理函数调用。顺序栈采用一维数组来实现,如下所示:

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

class Stack{                              // 定义栈类
public:
   Stack();                               // 构造函数,创建一个空栈
   bool empty();                          // 如果栈空,则返回真
   int peek();                            // 返回栈顶元素
   void push(int value);                  // 将一个整数存入栈顶
   int pop();                             // 弹栈操作,删除栈顶元素,并将其返回
   int getSize();                         // 返回栈中元素的个数

private:
   int elements[100];                     // 用于保存栈中元素的数组
   int size;                              // 栈中元素的个数
};

Stack::Stack(){
   size = 0;
}

bool Stack::empty(){
   return (size == 0);
}

int Stack::peek(){
   return elements[size - 1];
}

void Stack::push(int value){
   elements[size++] = value;
}
```

```
int Stack::pop(){
   return elements[ -- size];
}

int Stack::getSize(){
   return size;
}

int main(){
   Stack s;
   for(int i = 0; i < 10; i++)
       s.push(i);

   while(!s.empty())
      cout << s.pop()<<" ";
   cout << endl;

   return 0;
}
```

程序的运行结果如下。

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

【说明】 上述顺序栈中的元素保存在一个固定大小(100 个元素)的数组中。因此，上述顺序栈不能容纳 100 个以上的元素，当然可以将 100 改为一个更大的数，但如果实际栈中元素较少的话，会造成空间浪费。解决的方法是：预先分配较少的空间，在需要时分配更多的空间。在 Stack 类中增加一个新的数据成员 capacity，表示保存元素的数组的当前大小。当向栈中增加一个新元素时，如果当前数组已满，就需要增加数组大小来保存新元素。如何增加数组的容量呢？数组一旦创建后，其大小是无法改变的。为突破这一局限，我们可以创建一个新的、更大的数组，将旧数组中的内容复制到新数组，并删除掉旧数组。

改进后的顺序栈如下所示：

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

class Stack{                          // 定义堆栈类
public:
   Stack();                           // 构造函数,创建一个空栈,保存元素的数组的当前大小为 8
   bool empty();                      // 如果栈空,则返回真
   int peek();                        // 返回栈顶元素
   void push(int value);              // 将一个整数存入栈顶
   int pop();                         // 弹栈操作,删除栈顶元素,并将其返回
   int getSize();                     // 返回栈中元素的个数

private:
   int *elements;                     // 用于指向保存栈中元素的数组
```

```
    int size;                       // 栈中实际元素的个数
    int capacity;                   // 保存元素的数组的当前大小
    void ensureCapacity();          // 确保元素的实际个数小于数组大小
};

Stack::Stack(){
    size = 0;
    capacity = 8;
    elements = new int[capacity];
}

bool Stack::empty(){
    return (size == 0);
}

int Stack::peek(){
    return elements[size - 1];
}

void Stack::push(int value){
    ensureCapacity();
    elements[size++] = value;
}

int Stack::pop(){
    return elements[--size];
}

int Stack::getSize(){
    return size;
}

void Stack::ensureCapacity(){
    if(size >= capacity){
        int *old = elements;
        capacity = 2 * size;
        elements = new int[size * 2];

        for(int i = 0; i < size; i++)
            elements[i] = old[i];

        delete old;
    }
}
```

```
int main(){
   Stack s;
   for(int i = 0; i < 10; i++)
       s.push(i);

   while(!s.empty())
      cout << s.pop()<<" ";
   cout << endl;

   return 0;
}
```

程序的运行结果如下。

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

【链栈的参考程序如下】

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

class Stack                          // 定义堆栈类
{
  struct Node
 {
     int content;
     Node * next;
 } * top;
 public:
   Stack() { top = NULL; }           // 构造函数的定义
   bool push(int i);                 // 压栈成员函数的声明
   bool pop(int& i);                 // 弹栈成员函数的声明
};

bool Stack::push(int i)              // 压栈成员函数的定义
{
   Node * p = new Node;
   if (p == NULL)
   {
     cout << "Stack is overflow.\n";
     return false;
   }
   else
   {
     p->content = i;
     p->next = top;
     top = p;
     return true;
   }
}
```

```
bool Stack::pop(int& i)          // 弹栈成员函数的定义
{
   if (top == NULL)
   {
      cout << "Stack is empty.\n";
      return false;
   }
   else
   {
      Node * p = top;
      top = top->next;
      i = p->content;
      delete p;
      return true;
   }
}

int main()
{
   Stack st1,st2;                // 定义对象 st1 和 st2
   int x;
   for(int i = 1;i <= 5;i++)
   {
      st1.push(i);               // 压栈成员函数的调用
      st2.push(i);               // 压栈成员函数的调用
   }
   cout <<"stack1:"<< endl;
   for(i = 1;i <= 3;i++)
   {
      st1.pop(x);                // 弹栈成员函数的调用
      cout << x <<"  ";
   }

   st1.push(20);
   for(i = 1;i <= 4;i++)
   {
      if(st1.pop(x))
         cout << x <<" ";
      else
         break;
   }
   cout <<"stack2:"<< endl;
   while(st2.pop(x))
      cout << x <<" ";
   return 0;
}
```

程序的运行结果如下。

```
stack1:
5  4  3  20 2 1 Stack is empty.
```

```
stack2:
5 4 3 2 1 Stack is empty.
```

8. 写出程序的运行结果。

【解】

```
Constructing 22   11
Constructing 20   10
display:22   11
display:20   10
Destructing20   10
Destructing22   11
```

【说明】 程序执行时，首先 2 次调用构造函数创建对象 t1 和 t2，创建 t2 时，使用了默认参数值 10。构造函数和析构函数的调用顺序刚好相反，先析构对象 t2，然后析构对象 t1。