第3章　类 与 对 象

学习目标

（1）掌握类的概念。
（2）理解对象与类的关系，掌握对象的创建和使用方法。
（3）掌握构造函数、析构函数的概念和使用方法。
（4）掌握内存动态分配的概念和使用方法。
（5）掌握对象数组和对象指针。

3.1　类的定义

面向对象程序设计有三个主要特性：封装、继承和多态。封装是指将数据和代码捆绑在一起。C++中的封装是通过类来实现的。类是一种新的数据类型，也是实现抽象类型的工具。在C语言中，结构体将相关数据组合在一起；在 C++中，类是对一组具有共同属性特征和行为特征的对象的抽象，它可将相关数据和对这些数据的操作（函数）组合在一起。因此，类可以看作对结构体的扩展。

3.1.1　类定义格式

类定义格式为：

```
class 类名
{
public:
    数据成员或成员函数
protected:
    数据成员或成员函数
private:
    数据成员或成员函数
};
```

1．类名

class 是声明类的关键字，类名是标识符，且在其作用域内必须是唯一的，不能重名。在选择类名时应尽可能准确地描述该类所代表的概念。C++规定，标识符以字母（大、小写均可，但区分大、小写）或下画线开头，后面跟 0 或多个由字母、数字字符或下画线组成的字符串。类名的首字符通常采用大写字母。

2．成员说明

类包括两类成员：一类是代表对象属性的数据成员；另一类是实现对象行为的成员函数。成员函数的定义不仅可以与声明同时在类内完成，也可以在类外完成。如果在类外完成，则必须用作用域“::”符号告诉编译器该函数所属的类。

3．访问权符

访问权符也称访问权限符或访问控制符，它规定类中说明成员的访问属性，是 C++实现封装的基本手段。C++规定，在一个访问权符后面说明的所有成员都具有由这个访问权符所规定的访问属性，直到另一个不同的访问权符出现为止。

C++共提供了三种不同的访问权符：public、protected、private。

① public（公有类型）：声明该成员为公有成员。它表示该成员可以被和该类对象处在同一作用域内的任何函数使用。一般将成员函数声明为公有的访问控制。

② protected（保护类型）：声明该成员为保护成员。它表示该成员只能被所在类及从该类派生的子类的成员函数及友元函数使用。

③ private（私有类型）：声明该成员为私有成员。它表示该成员只能被所在类中的成员函数及该类的友元函数使用。

在具体使用时应根据成员的使用特点决定对其封装的程度。通常的做法是，将数据成员声明为私有类型或保护类型；将对象向外界提供的接口或服务声明为公有类型的成员函数。如果某些数据成员在子类中也需要经常使用，则应该把这些数据的访问属性声明为保护类型。

[例 3-1] 声明一个图书类。

分析：图书都有书名、作者、出版社和价格。对于图书的基本操作有输入和输出。因此先抽象出所有图书都具有的属性：书名、作者、出版社和价格，然后用成员函数实现对图书信息的输入和输出。

```
class Book{
public:                    //访问权限：公有成员
    void Input();          //行为，成员函数的原型声明，表示输入图书信息
    void Print();          //行为，成员函数的原型声明，表示输出图书信息
private:                   //访问权限：私有成员
    char title[20];        //属性，数据成员，书名
    char author[10];       //属性，数据成员，作者
    char publish[30];      //属性，数据成员，出版社
    float price;           //属性，数据成员，图书价格
};
```

说明：① 类声明中的 public、protected 和 private 关键字可以按任意顺序出现。为了使程序更加清晰，应将公有成员、保护成员和私有成员归类存放。默认访问权限为私有类型。

② 对于一个具体的类，类声明中的 public、protected 和 private 不一定都要有，但至少应该有其中的一个部分。

③ 数据成员可以是任何数据类型，但不能用自动（auto）、寄存器（register）或外部（extern）类型进行说明。

④ 类是一种广义的数据类型，系统并不会为其分配内存空间，所以不能在类声明中给数据成员赋初值。

⑤ 类的主体是包含在一对花括号中的，它的定义必须以“;”结束。

[例 3-2] 声明一个长方形类。

分析：长方形有长和宽就可以计算其面积和周长。因此，先抽象出所有长方形都具有的属性长和宽，然后用成员函数实现求面积和周长的运算。

```
class Rectangle    {
public:
    double Area();             //计算面积
    double Perimeter();        //计算周长
private:
    double length=3.5;         //错误，不能在类定义时，给数据成员赋值
    double width=4.6;          //错误，不能在类定义时，给数据成员赋值
};
```

C++规定只有在对象定义之后，才能通过对象名访问相应的成员函数，给数据成员赋初值，

或者通过构造函数对数据成员进行初始化（见 3.3 节）。

3.1.2 成员函数的定义

为了实现对象的行为，将一些相关的语句组织在一起，并给它们注明相应的名称，从而形成一些相对独立且便于管理和阅读的小块程序，每个小块程序都能描述一个完整的行为，这个小块程序就构成了成员函数。

成员函数的定义有两种形式：第一种是对于代码较少的成员函数，可以直接在类中定义；第二种是对于代码较多的成员函数，通常只在类中进行函数原型说明，在类外对函数进行定义，这种成员函数定义的格式如下：

```
返回类型 类名::函数名(参数表)
{
    //函数体
}
```

例如，例 3-1 中的成员函数的定义：

```
void Book::Input(){                          //定义成员函数 Input()
    cin>>title>>author>>publish>>price;      //输入书名、作者、出版社和价格
}
void Book::Print (){                         //定义成员函数 Print()
    //输出书名、作者、出版社和价格
    cout<<title<<"   "<<author<<"   "<<publish<<"    "<<price<<endl;
}
```

说明：① 如果在类外定义成员函数，则应在所定义的成员函数名前加上类名，在类名和函数名之间应加上作用域运算符“::”，它可说明成员函数从属于哪个类。例如，上例中的 void Book::Input()表示成员函数 Input()是类 Book 中的函数。

② 在定义成员函数时，对函数所带的参数，不但要说明其类型，还要指出参数名。

③ 在定义成员函数时，其返回类型一定要与函数原型声明的返回类型匹配。

3.1.3 类的作用域

类的作用域是指在类的定义中由一对花括号所括起来的部分，包括数据成员和成员函数。在类的作用域中声明的标识符，其作用域与标识符声明的顺序没有关系。类中的成员函数可以不受限制地访问本类的数据成员和其他成员函数。但在该类的作用域外，就不能直接访问类的数据成员和成员函数了，即使是公有成员，也只能通过本类的对象才能访问。

[例 3-3] 类作用域应用举例。

```
#include <iostream>
using namespace std;
class Rectangle{
public:
        void SetPeri();
        void PrintPeri();
        double Area();
private:
        double length;
        double width;
};
int main(){
    Rectangle r1;
```

```
    r1.SetPeri();
    cout<<"Area= "<<r1.Area()<<endl;
    return 0;
}
double Rectangle:: Area(){
    PrintPeri();                    //调用类中的成员函数 PrintPeri()
    return length*width;            //使用类中的数据成员 length 和 width
}
void Rectangle::SetPeri(){
    cout<<"Input length and width "<<endl;
    cin>>length>>width;
}
void Rectangle:: PrintPeri(){
    cout<<"length = "<<length<<" width = "<<width<<endl;
}
```

说明：本例中虽然成员函数 SetPeri()、Area()和 PrintPeri()的实现在花括号的类外，但是其函数原型已在类 Rectangle 中声明了，因此也在类的作用域范围之内，即类作用域覆盖了所有成员函数的作用域。因此成员函数 SetPeri()、Area()和 PrintPeri()可以不受限制地访问本类的数据成员和成员函数。

3.2 对象的定义与使用

在 C++中声明类表示定义了一种新的数据类型，只有定义了类的对象，才真正创建了这种数据类型的物理实体。对象是封装了数据结构及可以施加在这些数据结构上操作的封装体。对象是类的实际变量，一个具体的对象是类的一个实例。因此，类与对象的关系就类似于整型 int 和整型变量 i 的关系。类和整型 int 均表示为一般的概念，而对象和整型变量则代表具体的内容。与定义一般变量一样，也可以定义类的变量。C++把类的变量称为类的对象，一个具体的对象也称为类的实例。创建一个对象称为实例化一个对象或创建一个对象实例。

3.2.1 对象的定义

有两种方法可以定义对象。

① 在声明类的同时，直接定义对象。例如，以下语句定义 b1、b2 是类 Book 的对象：

```
class Book {
public:
    void Input();
    void Print();
private:
    char title[20];
    char author[10];
    char publish[30];
    float price;
}b1,b2;
```

② 先声明类，然后使用时再定义对象，定义格式与一般变量定义格式相同：

类名 对象名列表;

例如，语句“Book b1, b2;”定义 b1 和 b2 为类 Book 的两个对象。

说明：① 必须先声明类以后才能定义类的对象。多个对象之间用逗号分隔。

② 声明一个类就声明了一种类型，但它并不能接收和存储具体的值，只能作为生成具体对

象的一种“样板”，在定义了对象后，系统才为对象且只为对象分配存储空间。

③ 在声明类的同时定义的对象是一种全局对象，在其生命周期内任何函数都可以使用它，一直到整个程序运行结束。

3.2.2 对象的使用

使用对象就是向对象发送消息，请求执行其某个方法，从而向外界提供所要求的服务，它的格式为：

对象名.成员函数名(实参表);

例如，使用前面定义的对象 b1：

```
Book   b1;
b1.Input();  //通过对象 b1 执行输入操作
b1.Print();  //通过对象 b1 执行输出操作
```

[例 3-4] 图书类的完整程序。

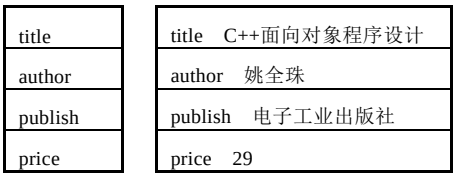
```
#include <iostream>
using namespace std;
class Book {
public:
    void Input();
    void Print();
private:
    char title[20];
char author[10];
char publish[30];
    float price;
};
int main(){
    Book   b1;
    b1.Input();
    cout<<"运行结果:"<<endl;
    b1.Print();
    return 0;
}
void Book::Input(){
    cout<<"请输入书名、作者、出版社和价格:"<<endl;
    cin>>title>>author>>publish>>price;
}
void Book::Print (){
    cout<<title<<"   "<<author<<"   "<<publish<<"     "<<price<<endl;
}
```

程序运行结果为：

```
请输入书名、作者、出版社和价格:
C++面向对象程序设计   姚全珠   电子工业出版社   29
运行结果:
C++面向对象程序设计   姚全珠   电子工业出版社     29
```

说明：本程序分为三部分，第一部分是类 Book 的声明；第二部分是主函数 main()；第三部分是图书类成员函数的具体实现。当在 main()中定义图书对象 b1 时，b1 有 4 个数据成员，如图3-1（a）所示，此时，每个数据成员的值都是随机值；当对象执行 Input()时，从键盘输入的数据就按输入顺序对应地存放在 b1 所对应的 title、author、publish、price 等数据成员中，如图 3-1（b）所示；

当对象执行 Print()时，数据成员 title、author、publish、price 的值就会在屏幕上输出。



(a) (b)

图 3-1 对象状态

注意：不能直接访问对象的私有成员。如果将该例的主程序改写成下面的形式，在编译时，程序就会给出语句错误的信息。

```
int main()
{
    Book  b1;
    cin>>b1.title >>b1.author >>b1.publish >>b1.price;
    //错误的访问，不能直接访问对象的私有数据成员
    cout<<"运行结果:"<<endl;
    b1.Print();
    return 0;
}
```

为了帮助读者理解上面的程序，给出以下 3 点说明。

① “//”表示注释。在 C++中用“//”实现单行的注释，它是将“//”开始一直到行尾的内容视为注释，称为行注释。“/*…*/”表示多行的注释，它是将由“/*”开头到“*/”结尾之间所有内容均视为注释，称为块注释。块注释的注解方式可以出现在程序的任何位置，包括在语句或表达式之间。

② #include <iostream>表示编译器对程序进行预处理时，将文件 iostream 中的代码嵌入程序中该指令所在的地方。文件 iostream 中声明了程序所需要输入和输出的操作信息。在 C++中，如果使用了系统中提供的一些功能，就必须嵌入相关的头文件。

③ using namespace std；表示针对命名空间的指令。

[例 3-5] 学生成绩管理。

```
#include <iostream>
using namespace std;
#define  N  5          //定义常量 N，表示有 N 个学生
struct Student{        //声明结构体类型，表示是学生
    long lNum;         //学号
    char sName[12];    //姓名
    float fGrade;      //成绩
};
class OurClass{        //声明一个班级类
public:
    void Input();      //输入学生信息
    void Print();      //输出学生信息
    void Sort();       //按学生成绩进行排序
private:
    char cName[20];    //定义班级名称
    Student stu[N];    //定义 N 个学生
};
int main(){
    OurClass cl;       //定义对象 cl
    cl.Input();        //输入 N 个学生的信息：学号、姓名、成绩
    cl.Sort();         //按学生成绩进行排序
    cout<<endl<<"排序结果:"<<endl<<endl;
    cl.Print();        //输出 N 个学生的信息：学号、姓名、成绩
    return 0;
```

```
}
void OurClass::Input(){
    int i;
    cout<<"输入班级名称:   ";
    cin>>cName;
    cout<<"输入"<<N<<"个学生的学号、姓名及成绩"<<endl;
    for(i=0;i<N;i++)
        cin>>stu[i].lNum >>stu[i].sName >>stu[i].fGrade;
}
void OurClass::Sort(){
    int i,j;
    Student Temp;
    for(i=0;i<N-1;i++){
        for(j=i+1;j<N;j++){
            if(stu[i].fGrade < stu[j].fGrade ) {
                //交换学生信息
                Temp= stu[i];
                stu[i]= stu[j];
                stu[j]=Temp;
            }
        }
    }
}
void OurClass::Print(){
    int i;
    cout<<"班级名称:   ";
    cout<<cName<<endl;
    cout<<"学号   姓名  成绩"<<endl;
    for(i=0;i<N;i++)
        cout<<stu[i].lNum <<"   "<<stu[i].sName <<"    "<<stu[i].fGrade <<endl;
}
```

3.2.3 对象的赋值

同类型的变量可以利用赋值运算符“=”进行赋值，如整型、实型、结构体类型等，对于同类型的对象也同样适用。也就是说，同类型的对象之间可以进行赋值，这种赋值默认通过成员复制进行。当对象进行赋值时，对象的每一个成员逐一复制（赋值）给另一个对象的同一个成员。

[例 3-6]　对象的赋值：平面上的点赋值。

分析：本程序实现的功能是将平面上的一个点坐标值赋给平面上的另一个点。平面上的点坐标有横坐标和纵坐标。确定类 Point，平面上的每个点都有相同的属性：*x* 坐标和 *y* 坐标。用成员函数 SetPoint()和 Print()实现点信息的输入和输出。

```
#include <iostream>
#include <iomanip>
using namespace std;
class Point{
public:
    void SetPoint(int a,int b);
    void Print(){
        cout<<"x="<<x<<setw(5)<<"y="<<y<<endl;
    }
```

```
private:
    int x;
    int y;
};
int main(){
    Point p1,p2;        //定义对象 p1 和 p2
    p1.SetPoint(3,5);
    cout<<"p1:"<<endl;
    p1.Print();
    cout<<"p2:"<<endl;
    p2.Print();
    p2=p1;              //对象赋值
    cout<<"p2=p1:"<<endl;
    p2.Print();
    return 0;
}
void Point::SetPoint (int a,int b) {
    x=a;
    y=b;
}
```

程序运行结果为：

```
p1:
x=3    y=5
p2:
x=-858993460    y=-858993460
p2=p1:
x=3    y=5
```

说明：本程序建立了 Point 类的两个对象，对象 p1 通过调用成员函数 SetPoint()将数据 3 和 5 对应地赋给 p1 的数据成员 x 和 y；而对象 p2 的数据成员 x 和 y 因为没有给赋值，因此输出的 x 和 y 的值是随机值，但是通过赋值语句 p2 = p1 后，p1 将自己的数据成员 x 和 y 的值对应地赋给 p2 的数据成员 x 和 y，因此再输出 p2 的数据成员值时，就输出了 3 和 5。成员函数 Print()中用到了 setw(5)。setw(int n)是 C++提供的标准运算符函数，表示设置输入、输出的宽度，该函数定义在 iomanip.h 头文件中。

注意：① 在使用对象赋值语句进行赋值时，两个对象的类型必须相同，但赋值兼容规则除外。

② 两个对象间的赋值，仅使对象中的数据相同，而两个对象仍然是彼此独立的，各自有自己的内存空间。

③ 如果类中存在指针，则不能简单地将一个对象的值赋给另一个对象，否则会产生错误。第 6 章将介绍这些问题及解决办法。

3.2.4 对象的生命周期

对象的生命周期是指对象从被创建开始到生命周期结束为止的时间，对象在定义时被创建，在释放时被终止。类中成员变量的生命周期是在类的对象定义时创建的，在对象的生命周期结束后停止。局部对象的生命周期在一个程序块或函数体内，而全局对象的生命周期从定义时开始到程序结束时停止。

[例 3-7] 对象生命周期应用举例 1。

```
#include <iostream>
```

```
using namespace std;
class Point{
public:
    void SetPoint(int a,int b);
private:
    int x;
    int y;
};
Point p1;                    //定义全局对象 p1
int main(){
    cout << "inside main()" << endl;
    Point p2;                //定义局部对象 p2
    {
        Point p3;            //定义局部对象 p3
        p3.SetPoint(3,5);
    }
    p2.SetPoint(1,2);
    p1.SetPoint(6,9);
    cout << "outside main()" << endl;
    return 0;
}
void Point::SetPoint (int a,int b) {
    x=a;
    y=b;
    cout<<"x="<<x<<", y="<<y<<endl;
}
```

说明：对象 p1 是全局对象，其生命周期从定义时开始到程序结束时停止，因此在 main()中可以对 p1 进行访问。对象 p2 是局部对象，其生命周期在 main()内；对象 p3 是局部对象，其生命周期在复合语句内，当在复合语句外面时，p3 生命周期就会结束，不能被访问。

[例 3-8] 对象生命周期应用举例 2。

```
#include <iostream>
using namespace std;
class Point{
public:
    void SetPoint(int a,int b){
        x=a;
        y=b;
        cout<<"x="<<x<<", y="<<y<<endl;
    }
private:
    int x;
    int y;
};
namespace  A {
    char username[]="namespace A";
    void ShowName(){
        cout<<username<<endl;
    }
}
void Fun();
int main(){
```

```
    cout<<"inside main"<<endl;
    Point p1;            //定义局部对象 p1
    p1.SetPoint(1,2);
    A::ShowName();
    Fun();
    return 0;
}
void Fun(){
    strcpy_s(A::username,strlen("inside Fun")+1, "inside Fun");
    cout<<A::username<<endl;
}
```

说明：对象 p1 是局部对象，其生命周期在 main()内；命名空间 A 是在函数体外定义的，其生命周期与全局量一样，在声明时开始，并在程序运行结束时停止。因此，命名空间 A 中的成员既可以在 main()中被访问，也可在 Fun()中被访问。

[例 3-9] 对象生命周期应用举例 3。

```
#include <iostream>
using namespace std;
class Point {
public:
    void SetPoint(int a, int b);
    void Print(){
        cout << "x=" << x << ", y=" << y << endl;
    }
private:
    int x;
    int y;
};
Point & TestFun();
int main(){
    Point &p1 = TestFun();
    cout << "main() p1:   ";
    p1.Print();
    p1.SetPoint(7,8);
    cout << "main() p1:   ";
    p1.Print();
    return 0;
}
void Point::SetPoint(int a, int b) {
    x = a;
    y = b;
}
Point & TestFun(){
    Point p;
    p.SetPoint(5, 6);
    cout << "TestFun() p:   ";
    p.Print();
    return p;
}
```

程序运行结果错误。

说明：对象 p1 是对象引用，其生命周期在 main()内；对象 p 是局部对象，其生命周期在

TestFun()内。在 TestFun()中，对象 p 可以调用成员函数 Print()，输出相应的数据成员值；但在 TestFun()结束时，对象 p 的生命周期就结束了，其数据成员 x 和 y 的生命周期也结束了，这时返回的是一个空引用。因此在 main()中，对象 p1 的数据成员 x 和 y 的输出结果是错误的。

3.3 构造函数和析构函数

如果变量在使用之前没有进行正确的初始化或清除，将导致程序出错。例如，在例 3-6 中，p2 在没有对数据成员初始化时就调用输出函数进行输出，结果输出的数据成员的值是随机值。如果对这个没有正确初始化的对象进行使用，则很容易导致程序出错。因此要求对对象必须正确地初始化。对对象进行初始化的一种方法是编写初始化函数，然而很多用户在解决问题时，常常忽视这些函数，以至于给程序带来了隐患。为了方便对象的初始化工作，C++提供了两个特殊的成员函数，即构造函数和析构函数。构造函数的功能是在创建对象时，给数据成员赋初值，即对象的初始化。析构函数的功能是释放一个对象，在对象删除之前，用它来做一些内存释放等清理工作，它的功能与构造函数的功能正好相反。

3.3.1 构造函数

在类的定义中不能直接对数据成员进行初始化，要想对对象中的数据成员进行初始化，一种方法是手动调用成员函数来完成初始化工作，但这样做会加重程序员和编译器的负担。因为在每次创建对象时都要另外书写代码调用初始化函数，而且编译器也需要处理这些调用。另一种方法是使用构造函数。构造函数是一种特殊的成员函数，它的特点是为对象分配空间、初始化，并且在对象创建时会被系统自动执行。

定义构造函数原型的格式为：

```
类名(形参列表);
```

在类外定义构造函数的格式为：

```
类名::类名(形参列表)
{
    //函数语句;
}
```

构造函数的特点如下。

① 构造函数的名字必须与类名相同。

② 构造函数可以有任意类型的参数，但是没有返回值类型，也不能指定为 void 类型。

③ 定义对象时，系统会自动地调用构造函数。

④ 通常构造函数被定义在公有部分。

⑤ 如果没有定义构造函数，系统会自动生成一个默认的构造函数，它只负责对象的创建，不做任何初始化工作。

⑥ 构造函数可以重载。

[例 3-10] 构造函数应用举例：输出日期。

分析：本程序要求输出某日期，该日期由年、月、日构成。定义日期类 Date，确定属性为年、月、日。利用构造函数对对象进行初始化，调用成员函数输出日期。

```
#include <iostream>
using namespace std;
class Date {
public:
    Date(int y,int m,int d){
        year=y;
```

```
        month=m;
        day=d;
    }
    void Print();
private:
    int year;
    int month;
    int day;
};
int main(){
    Date today(2020,3,1);
    cout<<"today is    ";
    today.Print();
    return 0;
}
void Date::Print(){
    cout<<year<<"-"<<month<<"-"<<day<<endl;
}
```

程序运行结果为：

```
today is    2020-3-1
```

说明：main()没有显示调用构造函数 Date()，构造函数是在创建对象 today 时系统自动调用的，即在创建对象 today 时，系统自动调用构造函数 today.Date，并将数据成员 year、month、day 初始化为 2020、3 和 1。

注意：① 构造函数的名字必须与类名相同，否则系统会将它当作一般的成员函数来处理。

② 构造函数没有返回值，在声明和定义构造函数时，不能说明其类型，甚至 void 类型也不行。

③ 在实际应用中，通常需要给每个类定义构造函数，如果没有给类定义构造函数，则系统自动生成一个默认的构造函数。这个默认的构造函数不带任何参数，只能给对象开辟一个存储空间，也不能为对象中的数据成员赋初值，此时数据成员的值是随机的。系统自动生成的构造函数的格式为：

类名::构造函数名(){ }

例如，假设例 3-4 的类 Book 没有定义构造函数，而系统为类 Book 自动生成的构造函数为：

```
Book:: Book(){ }
```

但是，如果在类中自己定义了构造函数，则系统提供的默认构造函数不再起作用，如果还需要使用无参数的默认构造函数来初始化对象，则必须再定义一个无参数的构造函数，即构造函数重载（详细内容见第 4 章）。

例如，若将例 3-10 的 main()修改如下，则编译时会发生错误。

```
int main(){
    Date today1(2020,3,1);
    Date today2;
    //错误，创建对象时将调用构造函数，但类 Date 没有无参数的构造函数，无法创建 today2 对象
    //解决方法是构造函数重载
    cout<<"today is    ";
    today1.Print();
    return 0;
}
```

④ 构造函数可以不带参数，例如：

```
class Point{
public:
```

```
    Point();
    //…
private:
    int x, y;
};
Point:: Point(){
    x=0;
    y=0;
}
```

⑤ 构造函数也可采用构造初始化表对数据成员进行初始化，例如：

```
class Date{
public:
    Date(int y,int m,int d):year(y),month(m),day(d){}
    //构造函数初始化表对数据成员进行初始化
    //…
private:
    int year;
    int month;
    int day;
};
```

⑥ 如果数据成员是数组，则应在构造函数中使用相关语句进行初始化，例如：

```
class   Student{
public:
     Student(char na[],int a);
    //….
private:
    char name[10];
    int age;
};
Student:: Student(char na[],int a): age(a) {
    //name 是字符数组，所以用 strcpy_s()进行初始化
    strcpy_s(name, strlen(na)+1,na);
}
```

3.3.2 析构函数

在对象生命周期结束前，通常需要进行必要的清理工作。这些相关的清理工作由析构函数完成。析构函数是一种特殊的成员函数，当删除对象时就会被调用。也就是说，在对象的生命周期即将结束时，由系统自动调用，随后这个对象也就消失了。需要注意的是，析构函数的目的是在系统回收对象内存之前执行结束清理工作，以便内存可被重新用于保存新对象。

定义析构函数的格式为：

```
~类名();
```

例如：

```
class Date{
public:
    …
    ~ Date () {} //析构函数
    …
};
```

析构函数的特点：

① 析构函数名是由“~”和类名组成的；

② 析构函数没有参数，也没有返回值，而且不能重载；

③ 通常析构函数被定义在公有部分，并由系统自动调用；

④ 一个类中有且仅有一个析构函数，应为 public。

说明：① 与类的其他成员函数一样，析构函数不仅可以在类内定义，也可以在类外定义。如果不定义，系统会自动生成一个默认的析构函数为“类名::~ 类名(){ }”。

② 析构函数的功能是释放对象所占用的内存空间，它在对象生命周期结束前由系统自动调用。

③ 析构函数与构造函数两者的调用次序相反，即最先构造的对象最后被析构，最后构造的对象最先被析构。

[例 3-11] 构造函数与析构函数的执行顺序，即 Point 类的多个对象的创建与释放。

```
#include <iostream>
using namespace std;
class Point{
public:
    Point(int a,int b);
    ~Point();
private:
    int x;
    int y;
};
int main(){
    Point p1(1,2),p2(3,5);
    return 0;
}
Point::Point(int a,int b) {          //定义构造函数
    cout<<"inside constructor"<<endl;
    x=a;   y=b;
    cout<<'('<<x<<','<<y<<')'<<endl;
}
Point::~Point(){                     //定义析构函数
    cout<<" inside destructor"<<endl;
    cout<<'('<<x<<','<<y<<')'<<endl;
}
```

说明：调用构造函数的顺序与 main()中创建对象的顺序一致，先创建对象 p1，然后再创建对象 p2；调用析构函数的顺序与创建对象的顺序相反，先析构对象 p2，然后再析构对象 p1。

④ 除了在显式撤销对象时，系统会自动调用析构函数，在下列情况下，析构函数也会被调用，即如果一个对象被定义在一个函数体内，则当这个函数结束时，该对象的析构函数会被自动调用。

[例 3-12] 对象定义在函数体内的析构函数的执行情况。

```
#include <iostream>
using namespace std;
class Complex{
public:
    Complex(double r,double i);
    ~Complex();
private:
    double real;
    double imag;
```

```
};
void fun(Complex c);
int main(){
    cout<<"inside main"<<endl;
    Complex c1(1.1,2.2);
    fun(c1);
    cout<<"outside main"<<endl;
    return 0;
}
Complex::Complex (double r,double i){
    cout<<"inside constructor"<<endl;
    real=r;
    imag=i;
}
Complex::~Complex(){
    cout<<"inside destructor"<<endl;
}
void fun(Complex c){
    cout<<"inside fun"<<endl;
}
```

程序运行结果为：

```
inside main
inside constructor
inside fun
inside destructor
outside main
inside destructor
```

说明：在 main()中定义对象 c1 时，系统自动调用 c1 的构造函数；当调用 fun()时，实参 c1 将值对应地赋给形参 c；当函数 fun 执行完，系统自动调用对象 c 的析构函数；当 main()结束时，系统自动调用对象 c1 的析构函数。由此可见，只要对象超出其定义范围，系统就会自动调用析构函数。

[例 3-13] 复合语句中对象的析构函数的执行情况。

```
class Complex{
    …//与例 3-12 的类 Complex 声明相同
};
#include <iostream>
using namespace std;
int main(){
    cout<<"inside main"<<endl;
    Complex c1(1.1,2.2);
    cout<<"begin compound-statement:"<<endl;
    {
        Complex c2(3.4,5.8);
    }
    cout<<"end compound-statement:"<<endl;
    cout<<"outside main"<<endl;
    return 0;
}
…//成员函数的实现部分与例 3-12 相同
```

说明：main()中先自动调用 c1 的构造函数。当执行复合语句中的定义对象 c2 时，系统自动

调用 c2 的构造函数；当复合语句结束时，系统自动调用对象 c2 的析构函数；当 main()结束时，系统自动调用对象 c1 的析构函数。

如果一个对象使用 new 运算符动态创建，在使用 delete 运算符释放它时，delete 会自动调用析构函数（在程序中如果不显示撤销该对象，系统就不会自动调用析构函数），详细说明见 3.4 节。

3.4 内存的动态分配

用户存储区空间分为三部分：程序区（代码区）、静态存储区（数据区）和动态存储区（栈区和堆区）。代码区存放程序代码，程序运行前就可分配存储空间。数据区存放常量、静态变量、全局变量等。栈区存放局部变量、函数参数、函数返回值和临时变量等。堆区是程序空间中存在的一些空闲存储单元，这些空闲存储单元组成堆。在堆中创建的数据对象称为堆对象。当创建对象时，堆中的一些存储单元从未分配状态变为已分配状态；当删除所创建的堆对象时，这些存储单元从已分配状态又变为未分配状态。当堆对象不再使用时，应予以删除，回收其所占用的动态内存。

在 C++中使用运算符 new 和 delete 来实现在堆内存区中进行数据的动态分配和释放。

3.4.1 运算符 new

在 C++中，new 的功能是实现内存的动态分配。在程序运行过程中申请和释放的存储单元称为堆对象。申请和释放的过程称为建立和删除堆对象。

new 的使用格式有以下三种：

指针变量 = new T;
指针变量 = new T(初值列表);
指针变量 = new T [元素个数];

说明：① T 是数据类型名，表示在堆中建立一个 T 类型的数据。初值列表可以省略，例如：

```
int   *p;
float   *p1;
p=new int(100);  //让 p 指向一个类型为整型的堆地址，该地址中存放数值 100
p1=new float;    //让 p1 指向一个类型为实型的堆地址
```

② 用 new 创建堆对象的格式：

类名 *指针名= new 类名([构造函数参数]);

例 3-10 中创建一个类 Complex 的对象也可以采用下列语句：

```
Complex *c1=new Complex(1.1,2.2);
//创建对象*c1，并调用构造函数初始化数据成员 real、imag 为 1.1、2.2
```

类名后面是否带参数取决于类的构造函数，如果构造函数带参数，则 new 后面的类名需要带参数；反之，则不带参数。例如：

```
class Date{
public:
    Date();
    …
private:
    int year;
    int month;
    int day;
};
int main(){
    Date *today=new Date;
    …
```

```
}
Date::Date(){
    year=2001;
    month=12;
    day=3;
}
```

③ new 返回一个指定的合法数据类型内存空间的首地址（指针），若分配不成功，则返回一个空指针。

④ new 可以为数组动态分配内存空间，这时应该在类型名后面指明数组的大小，其中，元素个数是一个整型数值，可以是常数也可以是变量。指针类型应与数组类型一致，例如：

```
int n,*p;
cin>>n;
p=new int[n];     //表示 new 为有 n 个元素的整型数组分配内存空间，并将首地址赋给指针 p
```

⑤ new 不能对动态分配的数组存储区进行初始化，例如：

```
int *p;
p=new int[10](0);//错误，不能对动态分配的数组进行初始化
```

⑥ 用 new 分配的空间，使用结束后只能用 delete 显式释放，否则这部分空间将不能回收，从而造成内存泄漏。

3.4.2 运算符 delete

delete 用来释放动态变量或动态数组所占的内存空间，其应用格式为：

```
delete   指针变量名;
delete   []指针变量名;
```

① 释放动态变量所占的内存空间，例如：

```
int *p=new int;
//…
delete p;     //释放指针 p 所指向的动态内存空间
```

② 释放动态数组所占的内存空间，例如：

```
int *p;
p=new int[10];
//…
delete []p;   //释放为数组动态分配的内存
```

说明：① new 和 delete 需要配套使用，如果搭配错了，程序运行时就会发生不可预知的错误。

② 在用 delete 释放指针所指的空间时，必须保证该指针所指的空间是用 new 申请的，并且只能释放一次，否则将产生指针悬挂问题（见第 6 章）。

③ 如果在程序中用 new 申请了空间，就应该在结束程序前释放所有申请的空间，这样才能保证堆内存的有效利用。

④ 当 delete 用于释放由 new 创建的数组连续内存空间时，无论是一维数组还是多维数组，指针变量名前必须使用[]，且[]内没有数字。

[例 3-14] 动态创建类 Point 的对象。

```
#include <iostream>
using namespace std;
class Point{
public:
    Point(int a,int b);
```

```
    ~Point();
private:
    int x,y;
};
int main(){
    Point *p=new Point(1,3);      //动态创建对象，自动调用构造函数
    delete p;                     //删除对象，自动调用析构函数
    return 0;
}
Point::Point(int a,int b) {
    cout<<"inside constructor"<<endl;
    x=a;   y=b;
}
Point::~Point(){
    cout<<"inside destructor"<<endl;
}
```

说明：当 main()利用 new 动态创建对象 p 时，系统可自动调用构造函数。当执行 delete 运算符删除对象 p 时，系统可自动调用析构函数。

注意：读者可以将“delete p;”这条语句去掉，看一下会有怎样的结果，为什么？

3.5 对象数组和对象指针

3.5.1 对象数组

数组的元素既可以是基本数据类型的数据，也可以是用户自定义数据类型的数据。对象数组是指每一个数组元素都是对象的数组。对象数组的元素是对象，它不仅具有数据成员，而且还有成员函数。

说明对象数组的方法与说明基本类型的数组的方法相似，因为类实质上是一种数据类型。在执行对象数组说明语句时，系统不仅分配适当的内存空间以创建数组的每个对象（数组元素），还会自动调用适当的构造函数以完成数组内每个对象的初始化工作。

声明对象数组的格式为：

类名 数组名[下标表达式];

例如：Book b[10];

与基本类型的数组一样，在使用对象数组时也只能引用单个数组元素。通过对象可以访问其公有成员。对象数组的引用格式为：

数组名[下标].成员函数

例如：b[1].Print();

[例 3-15] 对象数组的应用：求圆的面积。

分析：假设仅计算圆的面积而不考虑圆所在的位置，则可抽象出所有圆都具有的属性——半径；定义一个 Circle 类，用构造函数实现对半径的初始化，并用成员函数计算圆的面积。

```
#include <iostream>
using namespace std;
class Circle{
public:
    Circle(double r);
    double Area();
    ~Circle();
private:
```

```
    double radius;
};
int main(){
    Circle c[3]={1,3,5};    int i;
    for(i=0;i<3;i++)
        cout<<"第 "<<i+1<<" 个圆的面积是:"<<c[i].Area()<<endl;
    return 0;
}
Circle::Circle (double r) {
    cout<<"inside constructor"<<endl;
    radius=r;
}
double Circle::Area(){
    return 3.14*radius*radius;
}
Circle::~Circle(){
    cout<<"inside destructor"<<endl;
}
```

说明：main()中定义对象数组 c[3]，并通过直接调用构造函数对每个元素即每个对象进行初始化，当程序结束时，系统自动调用析构函数释放每个对象，因此，需要执行三次析构函数。

注意：构造函数不只有一个参数，在定义对象并对对象进行初始化时，通常采用直接调用构造函数的方法。

[例 3-16] 输出若干个平面上的点。

```
#include <iostream>
using namespace std;
class Point{
public:
    Point(int a,int b);
    void Print();
private:
    int x, y;
};
int main(){
    Point ob[3]={Point(1,2),Point(3,4),Point(5,6)};
    int i;
    for(i=0;i<3;i++){
        cout<<"第 "<<i+1<<" 个点的坐标为:";
        ob[i].Print();
    }
    return 0;
}
Point::Point(int a,int b) {
    x=a;    y=b;
}
void Point::Print(){
    cout<<'('<<x<<','<<y<<')'<<endl;
}
```

程序运行结果为：

```
第 1 个点的坐标为(1,2)
第 2 个点的坐标为(3,4)
```

第 3 个点的坐标为(5,6)

说明：在定义对象数组 ob 时系统会自动调用构造函数进行初始化，然而此时构造函数的参数是两个，因此就需要通过直接调用构造函数给对象数组赋值：

```
Point ob[3]={Point(1,2),Point(3,4),Point(5,6)};
```

从而实现对象数组的初始化操作。

3.5.2 对象指针

访问一个对象既可以通过对象名访问，也可以通过对象地址访问。对象指针就是用于存放对象地址的变量，它遵循一般变量指针的各种规则。声明对象指针的格式为：

类名 *对象指针名;

例如，例 3-15 的 Circle 类：

```
Circle *c;          //定义 Circle 类的对象指针变量 c
```

与用对象名来访问对象成员一样，使用对象指针也可以访问对象的成员，其格式为：

对象指针名->成员名

例如：

```
Circle *c;
…                   //对指针 c 进行初始化
c-> Area();
```

与一般变量指针一样，对象指针在使用之前必须先进行初始化，让它既可以指向一个已经声明过的对象，也可以用 new 运算符动态建立堆对象。

例如：

```
Circle *c1,c(3);
c1=&c;;
c1-> Area ();            //正确，c1 在使用之前已指向一个已经声明过的对象
Circle *c2=new Circle(3);
c2-> Area ();            //正确，c2 在使用之前已利用 new 动态建立堆对象 c2
Circle   *c3;
c3-> Area ();            //错误，不能使用没有初始化的对象指针
```

[例 3-17] 用对象指针访问 Circle 类的成员函数。

修改例 3-15 中的 main()，代码如下：

```
int main(){
    Circle *c=new Circle(3);
    cout<<"圆的面积是:"<<c->Area()<<endl;
    delete c;
    return 0;
}
```

程序运行结果为：

```
inside constructor
圆的面积是:28.26
inside destructor
```

说明：main()动态建立对象 c 时，系统自动调用构造函数将对象 c 的数据成员 radius 初始化为 3，初始化后的对象指针调用成员函数 Area 计算面积并输出结果。当删除对象时自动调用析构函数。

[例 3-18] 用对象指针引用 Circle 类的对象数组。

修改例 3-15 中的 main()，代码如下：

```
int main(){
    Circle c[3]={1,3,5};
    Circle *p=c;
```

```
    for(;p<c+3;p++)
        cout<<p->Area()<<endl;
    return 0;
}
```

程序运行结果与例 3-15 类似。

说明：main()中定义对象数组 c[3]，并将对象数组 c 的首地址赋给指针变量 p，通过指针变量 p 的移动，计算并输出每个圆的面积。

3.5.3 自引用指针 this

当定义了一个类的若干对象后，每个对象都有属于自己的数据成员，而同一类的不同对象将共同拥有一份成员函数的副本，那么在执行不同对象所对应的成员函数时，各成员函数是如何分辨出当前调用自己的是哪个对象呢？

[例 3-19] 输出不同正方形的面积。

分析：所有正方形都具有共同的属性——边，对于正方形可以进行求面积的运算。定义一个 Square 类，数据成员是边，用构造函数对边进行初始化，并用成员函数计算正方形的面积。

```
#include <iostream>
using namespace std;
class Square{
public:
    Square(double len);
    double Area();
private:
    double length;
};
int main(){
    Square s1(3),s2(5);
    cout<<"s1 area is   "<<s1.Area ()<<endl;
    cout<<"s2 area is   "<<s2.Area ()<<endl;
    return 0;
}
Square::Square (double len) {
    length=len;
}
double Square::Area(){
    return length* length;
}
```

程序运行结果为：

```
s1 area is   9
s2 area is   25
```

说明：运行这个程序输出 s1 和 s2 所对应的面积，但是在执行 s1.Area()时，成员函数 Area()怎么知道是对象 s1 在调用自己，从而输出 s1 所对应的值呢？类似地，在执行 s2.Area()时，成员函数 Area()怎么知道是对象 s2 在调用自己，从而输出 s2 所对应的值呢？

这是因为 C++为非静态成员函数提供了一个名为 this 的指针，即自引用指针。每当对象调用成员函数时，系统就将该对象的地址赋给 this 指针，这时 C++编译器将根据 this 指针所指向的对象来确定应该引用哪一个对象的数据成员。

通常，this 指针在系统中是隐含地存在的。在使用时可以将其显式地表示出来。上例中语句“length=len;”就可以写成“this->length=1en;”。

当执行 s1.Area()时，系统将 this 指针指向对象 s1，这样所读取的 length 是对象 s1 所对应的 length，从而计算出 s1 所对应的面积；同理，当执行 s2.Area()时，系统将对象 s2 的地址赋给 this 指针，读取对象 s2 所对应的 length，计算 s2 所对应的面积。

一般来说，this 指针主要用于运算符重载（见第 6 章）和自引用等场合。

[例 3-20]　this 应用举例：通过成员函数 copy()实现 Square 类对象的赋值。

```
#include <iostream>
using namespace std;
class Square{
public:
    Square(double len);
    double Area();
    void copy(Square &s);
private:
    double length;
};
int main(){
    Square s1(3),s2(5);
    cout<<"before copy"<<endl;
    cout<<"s1 area is    "<<s1.Area ()<<endl;
    cout<<"after copy"<<endl;
    s1.copy(s2);
    cout<<"s1 area is    "<<s1.Area ()<<endl;
    return 0;
}
Square::Square (double len) {
    length=len;
}
double Square::Area(){
    return length* length;
}
void Square::copy (Square &s) {
    if(this==&s)
        return;
    *this=s;
}
```

说明：定义对象 s1 时通过构造函数将其数据成员初始化为 3，因此调用 Area()输出 9；当程序执行 s1.copy(s2)时，对象 s1 调用成员函数 copy()，因此 this 指针指向 s1。在 copy()中首先判断是不是对象在给自己赋值，如果是，就返回；否则，将形参 s 的值赋给 this 所指的对象 s1。本例中形参 s 是实参 s2 的引用，因为不是 s1 给自己赋值，所以执行语句“*this=s;”，即将 s 的值赋给 this 所指的对象 s1。

使用 this 指针时应该注意以下三点：

① this 指针是一个 const 指针，不能在程序中修改或给它赋值；

② this 指针是一个局部数据，其作用域仅在一个对象的内部；

③ 静态成员函数不属于任何一个对象，且没有 this 指针。

小结

1．类和对象

类是面向对象程序设计的核心，它不仅是一种新的数据类型，也是实现抽象类型的工具。类是对某一类对象的抽象，一个具体的对象是类的实例。在定义类后，才能定义类的对象。在定义对象后，系统才为对象并且只为对象分配存储空间。

同类型的对象之间可以进行赋值，当一个对象赋值给另一个对象时，所有的数据成员都会逐位复制。但是如果类中存在指针时，则不能简单地将一个对象的值赋给另一个对象。否则会产生错误。

局部对象是指定义在一个程序块或函数体内的对象。定义对象时，系统自动调用构造函数创建对象，程序运行结束时调用析构函数释放对象。

全局对象的生命周期从定义时开始到程序结束时停止。定义对象时，自动调用构造函数创建对象，程序运行结束时调用析构函数释放对象。

2．构造函数和析构函数

构造函数的功能是在创建对象时给数据成员赋初值，即对象的初始化。析构函数的功能是释放一个对象，在对象删除之前，用它来做一些内存释放等清理工作。

构造函数的名字必须与类名相同，它没有返回值，既可以带参数也可以不带参数。如果没有定义构造函数，则编译器自动生成一个默认的构造函数。

每个类只能有一个析构函数，应为 public，且不能被重载，不能有任何参数，不能有返回值。如果不定义，系统会自动生成一个默认的析构函数。

当系统声明了多个对象时，这些对象的析构函数与构造函数的调用次序相反，即先构造的对象最后被析构，后构造的对象最先被析构。

3．运算符 new 和运算符 delete

new 的功能是实现内存的动态分配。

delete 用来释放动态变量或动态数组所占的内存空间。在用 delete 释放指针所指的空间时，必须保证这个指针所指的空间是用 new 申请的，并且只能释放一次，否则将产生指针悬挂问题。

执行 new 时创建动态对象，执行 delete 时释放动态对象。

4．对象数组和对象指针

对象数组是指每一数组元素都是对象的数组。对象数组的元素是对象，不仅具有数据成员，而且还有函数成员。

访问一个对象既可以通过对象名访问，也可以通过对象地址访问。对象指针就是用于存放对象地址的变量。

5．自引用指针 this

C++为非静态成员函数提供了一个名为 this 的指针，这个指针称为自引用指针。每当对象调用成员函数时，系统就将 this 指针初始化为指向该对象，然后调用成员函数。当成员函数处理数据时，则隐含使用 this 指针。当不同对象调用同一个成员函数时，C++编译器将根据 this 指针所指向的对象来确定应该引用哪一个对象的数据成员。

习题 3

1．构造函数和析构函数的主要作用是什么？

2．关于构造函数的叙述正确的是（　　）。

A．构造函数可以有返回值　　B．构造函数的名字必须与类名完全相同

C．构造函数必须带有参数　　D．构造函数必须定义，不能默认

3．关于析构函数特征描述正确的是（　　）。

A．一个类中可以有多个析构函数　　B．析构函数名与类名完全相同

C．析构函数不能指定返回类型　　D．析构函数可以有一个或多个参数

4．构造函数是在（　　）时被执行的。

A．程序编译　　B．创建对象　　C．创建类　　D．程序装入内存

5．定义 A 是一个类，那么执行语句“A a, b(3),*p;”调用了（　　）次构造函数。

A．2　　B．3　　C．4　　D．5

6．在下列函数原型中，可以作为类 Base 的析构函数是（　　）。

A．void～Base　　B．～Base()　　C．～Base()const　　D．Base()

7．this 指针是 C++实现（　　）的一种机制。

A．抽象　　B．封装　　C．继承　　D．重载

8．写出程序运行结果。

```
#include <iostream>
using namespace std;
class Sample {
public:
    Sample(){}
    Sample(int m);
    void Addvalue(int m);
    void Disp();
private:
    int n;
};
int main(){
    Sample s(10);
    s.Addvalue(5);
    s.Disp();
    return 0;
}
Sample::Sample(int m) {
    n=m;
}
void Sample::Addvalue(int m) {
    Sample s;
    s.n=n+m;
    *this=s;
}
void Sample::Disp(){
    cout<<"n="<<n<<endl;
}
```

9．写出程序运行结果。

```
#include <iostream>
using namespace std;
class CExample{
public:
    CExample(int n);
    ~CExample();
    int Geti();
private:
    int i;
};
int Addi(CExample ob);
int main(){
    CExample x(10);
    cout<<x.Geti()<<endl;
    cout<<Addi(x)<<endl;
    return 0;
}
CExample::CExample(int n) {
    i=n;
    cout<<"Constructing"<<endl;
}
CExample::~CExample(){
    cout<<"Destructing"<<endl;
}
int CExample::Geti(){ return i; }
int Addi(CExample ob) {
    return ob.Geti()+ob.Geti();
}
```

思考题

1．设计学生类，学生信息有学号、姓名和成绩。成绩包括计算机、英语、数学和平均分。要求利用队列实现学生的入队、出队和显示等功能。

2．设计图书类，图书信息有图书名称、作者、出版社和价格。要求利用栈实现图书的入库、出库和显示等功能。