

第4章 类与对象

```
class Clock{
public:
    void setTime(int newH,int newM,int newS);
    void showTime();
private:
    int hour, minute,second;
};
```

4.2.4 类的成员函数

a. 成员函数的实现

函数的原型要写在类体中，原型说明了函数的参数表和返回值类型。但函数的具体事项是写在类定义之外。

```
void Clock::setTime(int newH,int newM,int newS){
    hour=newH;
    minute=newM;
    second=newS;
}
void Clock::showTime(){
    cout << hour << ":" << minute << ":" << second << endl;
}
```

b. 成员函数调用中的目的对象

调用一个成员函数需要"."操作符指出调用所针对的对象

```
Clock myClock; //创建对象
myClock.showTime(); //myClock调用showTime()函数，myClock就是这一调用过程的目的对象
```

在成员函数中可以不适用"."操作符而直接引用目的对象的数据成员。比如showTime()中所引用的hour, minute, second都是目的对象的数据成员，以myClock.showTime()调用该函数时，被输出的是myClock对象的hour, minute, second属性。在成员函数中调用当前类的成员函数时，如果不使用"."，那么这一次调用所针对的仍然是目的对象。

类的成员函数中，既可以访问目的对象的私有成员，又可以访问当前类其他对象的私有成员。

c. 带默认形参的成员函数

类成员函数的默认值一般写在类定义中，不能写在类定义之外的函数实现中。比如时钟类的setTime()函数，就可以使用默认值实现：

```
class Clock{
public:
    void setTime(int newH=0,int newM=0,int newS=0);
    ...
};
```

d. 内联成员函数

函数的调用要消耗一些内存资源和运行时间来传递参数和返回值，要记录调用时的状态，以便保证调用完成后能够返回并继续执行。**内联函数**：函数成员需要被频繁调用，并且代码比较简单。

内联函数的声明：隐式和显式

- 隐式：直接写在类内

```
class Clock{
public:
    void setTime(int newH,int newM,int newS);
    void showTime()
    {
        cout << hour << ":" << minute << ":" << second << endl;
    }
private:
    int hour,minute,second;
};
```

- 显式：函数体实现，关键字inline

```
inline void Clock::showTime(){
    cout << hour << ":" << minute << ":" << second << endl;
}
```

4.3 构造函数和析构函数

类与对象之间的关系就像是基本数据类型与其变量的关系，即一般和特殊。不同对象之间有所区别的地方在于两个，第一个就是对象的名称，另一个就是对象自身的属性值，也就是数据成员的值。定义对象时也可以同时对它的数据成员赋初值，这称之为**对象的初始化**

4.3.1 构造函数

首先要理解对象的建立过程。为此先看基本数据类型变量(int型)的初始化过程：每一个变量在程序运行时都要占据一定的内存空间，在声明一个变量时对变量进行初始化，就意味着在为这个变量分配内存单元的同时，在其中写入了变量的初始值。

对象的建立过程也是类似：在程序的执行过程中，当遇到对象声明语句时，程序会向OS申请一定的内存空间用于存放新建的对象。不幸的是，我们不能像对待普通变量一样，在分配内存空间的同时将数据成员的初始值写入。编译系统严格规定了初始化程序的接口形式，并有一套自动的调用机制，即构造函数

构造函数的作用就是在对象被创建时利用特定的值构造对象，将对象初始化为一个特定的状态。

只要类中有了构造函数，编译器就会在建立新对象的地方自动插入对构造函数调用的代码。**构造函数在对象被创建的时候将被自动调用。**

调用时无需提供参数的构造函数称为默认构造函数。类中没有写构造函数，编译器会自动生成一个隐含的默认构造函数，其参数列表和函数体皆为空。如果我们自己声明了构造函数，无论我们自己声明的构造函数是否有参数，编译器都不会再生成隐含的构造函数。

```
class Clock{
public:
    Clock(int newH,int newM,int newS);
    void setTime(int newH,int newM,int newS);
    void showTime();
};
```

```

private:
    int hour, minute, second;
};
Clock::Clock(int newH, int newM, int newS){
    hour = newH;
    minute = newM;
    second = newS;
}
void Clock::setTime(int newH, int newM, int newS){
    hour=newH;
    minute=newM;
    second=newS;
}
void Clock::showTime(){
    cout << hour << ":" << minute << ":" << second << endl;
}
int main()
{
    Clock c(0,0,0);
    c.showTime();
    c.setTime(0,0,0);
    Clock c1(8,30,30);
    c1.showTime();
    c1.setTime(8,30,30);
    system("Pause");
    return 0;
}

```

4.3.2 拷贝构造函数

拷贝构造函数是一种特殊的构造函数，具有一般构造函数的所有特性，其形参是**本类的对象的引用**。其作用是使用一个已经存在的对象(由拷贝构造函数的参数指定)，去初始化同类的一个新对象。

程序员可以根据实际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。如果没有定义类的拷贝构造函数，系统就会自动生成一个隐含的拷贝构造函数，其功能是：把初始值对象的每个数据成员的值复制到子新建立的对象中。

```

class Point{
public:
    Point(int xx=0, int yy=0)    /*内联构造函数*/
    {
        x=xx;
        y=yy;
    }
    Point(Point &p);            /*拷贝构造函数*/
    int getX() {return x;}
    int getY() {return y;}
private:
    int x;
    int y;
};
Point::Point(Point &p)          /*拷贝构造函数的实现*/
{
    x=p.x;
    y=p.y;
    cout<<"Calling the copy constructor"<<endl;
}

```

```

void f(Point p){
    cout << p.getX() << endl;
}
Point g(){
    Point a(1,2);    /*表面上函数g将a返回给主函数，但a是g()的局部对象*/
    return a;         /*离开建立它的函数g后就消亡了，不可能在主函数继续生存*/
                     /*解决方法：在主函数中创建一个临时对象，其生存期只在b=g()中*/
                     /*执行return a;后实际上调用拷贝构造函数将a的值复制到临时对象中*/
                     /*函数g运行结束时a对象a已消失，但临时对象会存在于表达式b=g()中*/
                     /*计算完b=g()后，临时对象使命完成，然后消失*/
}
int main()
{
    Point a(1,2);
    Point b(a);
    Point c=a;    //对b和c的初始化都能够调用复制构造函数
    f(a);         /*只有把对象用值传递时，才会调用拷贝构造函数，如果是传递引用，则不会调用*/
    Point d;
    d=g();
    system("Pause");
    return 0;
}

```

4.3.3 析构函数

构造对象时，在构造函数中分配了资源(动态申请了一些内存单元)，对象消失时就要释放这些内存单元。析构函数就是用来完成对象被删除前的一些清理动作，它是在对象的生存期即将结束的时刻被自动调用的。

```

class Clock{
public:
    Clock();    //构造函数
    void setTime(int newH,int newM,int newS);
    void showtime();
    ~Clock() {}
private:
    int hour,minute,second;
};

```

4.3.4 程序实例

一圆形泳池，现需要在其周围建一圆形过道，并在其四周围上栅栏，栅栏价格35元/米，过道价格20元/平方米，过道宽度为3米，游泳池半径有键盘输入。编程计算并输出过道和栅栏的造价

```

#include <iostream>
using namespace std;
const float PI=3.141593;
const float FENCE_PRICE=35;
const float CONCRETE_PRICE=20;
class Circle{
public:
    Circle(float r);
    float circumference();    //计算周长
    float area();              //计算面积
private:
    float radius;
};

```

```

Circle::Circle(float r){
    radius=r;
}
//构造函数也可以改写成
Circle::Circle(float r):radius(r){}
float Circle::circumference(){
    return 2*PI*radius;
}
float Circle::area(){
    return PI*radius*radius;
}
int main()
{
    float radius;
    cout<<"Enter the radius of the pool: "<<endl;
    cin >> radius;
    Circle pool(radius);
    Circle poolRim(radius+3);

    float fenceCost=poolRim.circumference()*FENCE_PRICE;
    cout << "Fencing Cost is $ " << fenceCost << endl;

    float ConcreteCost = (poolRim.area()-pool.area())*CONCRETE_PRICE;
    cout << "Concrete Cost is $ " << ConcreteCost << endl;
    system("Pause");
    return 0;
}

```

4.4 类的组合

4.4.1 组合

类的组合描述的是一个类内嵌其他类的对象作为成员的情况，它们之间是一种包含与被包含的关系。当创建类的对象时，如果这个类具有内嵌对象成员，那么各个内嵌对象首先被自动创建。在创建对象时既要对本类的基本类型数据成员进行初始化，又要对内嵌对象成员进行初始化。

类名::类名(形参表):内嵌对象1(形参表),内嵌对象2(形参表),...
{类的初始化}

内嵌对象1(形参表),内嵌对象2(形参表)...称为初始化列表，作用是**对内嵌对象进行初始化**。

在创建一个组合类的对象时，不仅是它自身的构造函数被执行，而且还将调用其内嵌对象的构造函数，构造函数的顺序如下：

- 调用内嵌对象的构造函数，调用顺序按照内嵌对象在组合类的定义中出现的次序。要注意的是，内嵌对象在构造函数的初始化列表中出现的顺序内嵌对象构造函数的调用顺序无关
- 执行本类构造函数的函数体。

```

class Point{
public:
    Point(int xx=0,int yy=0){    /*构造函数*/
        x=xx;
        y=yy;
    }
}

```

```

    Point(Point &p);
    int getx(){return x;}
    int gety(){return y;}
private:
    int x,y;
};
Point::Point(Point &p){
    x=p.x;
    y=p.y;
    cout << "Calling the copy constructot of Point" << endl;
}
class Line{
public:
    Line(Point xp1,Point xp2);
    Line(Line &l);
    double getLen() {return len;}
private:
    Point p1,p2;
    double len;
};
Line::Line(Point xp1,Point xp2):p1(xp1),p2(xp2){
    cout<<"Calling constructor of Line"<<endl;
    double x=static_cast<double>(p1.getx()-p2.getx());
    double y=static_cast<double>(p1.gety()-p2.gety());
    len=sqrt(x*x+y*y);
}
Line::Line(Line &l):p1(l.p1),p2(l.p2){
    cout<<"Calling the copy constructor of Line"<<endl;
    len=l.len;
}
int main()
{
    Point myp1(1,1),myp2(4,5);
    Line line(myp1,myp2);
    Line line2(line);
    cout<<"The length of the line is: ";
    cout<<line.getLen()<<endl;
    cout<<"The length of the line2 is: ";
    cout<<line.getLen()<<endl;
    system("Pause");
    return 0;
}

```

程序分析：

主程序执行时，首先生成两个Point类对象，然后构造Line的对象line，接着通过拷贝构造函数建立Line的第二个对象line2，最后输出两点距离。

```

Calling the copy constructot of Point
Calling the copy constructot of Point
Calling the copy constructot of Point
Calling the copy constructot of Point
Calling constructor of Line
Calling the copy constructot of Point
Calling the copy constructot of Point
Calling the copy constructor of Line
The length of the line is: 5
The length of the line2 is: 5
请按任意键继续. . .

```

难点是这四个Point的拷贝构造函数是怎么输出的！

```
Point myp1(1,1),myp2(4,5); 只是调用Point的构造函数,建立好了两个点
Line line(myp1,myp2);现在建立线段对象
先进入到Point的拷贝构造函数实现,因为进行Line类的构造函数首先要把参数传进去, xp1、xp2参数传进去之后才来到线段类的构造函数, xp1初始化p1同样需要进入到Point类的拷贝构造函数中去, xp2同样, 之后计算长度。
之后的两次是因为:
1.p1初始化p1, 1.p2初始化p2, 而Line &l是引用, 并不需要调用拷贝构造函数。
```

4.4.2 前向引用声明

4.5 UML图形标识

4.5.1 UML简介

4.5.2 UML类图

4.6 结构体和联合体

4.6.1 结构体

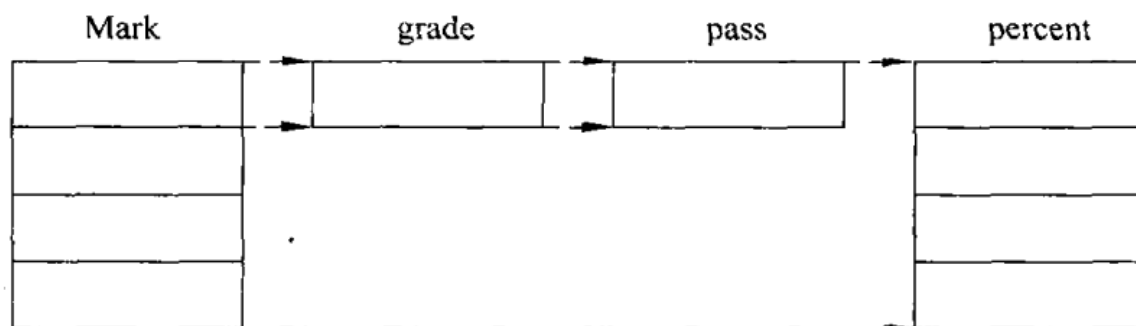
一种特殊的类，和类的区别就是具有不同的默认访问权限。类中对于未指定访问控制权限的成员，其访问控制属性是private；结构体中，对于未指定任何访问控制属性的属性，其访问控制属性为public。

C++引入结构体的目的是保持和C程序的兼容性

4.6.2 联合体

联合体的全部数据成员共享同一组内存单元

```
union Mark{
    char grade;
    bool pass;
    int percent;
};
```



正是由于联合体的成员共用相同的内存单元，联合体变量中的成员同时至多只有一个是具有意义的。除此以外，不同数据成员共用相同内存单元的特性，联合体有下面限制：

- 联合体的各个对象成员，不能有自定义的构造函数、自定义的析构函数和重载的拷贝复制运算符。
- 联合体不能继承、不支持包含多态

```

#include <iostream>
#include <string>
using namespace std;
class ExamInfo{
public:
    ExamInfo(string name,char grade):name(name),mode(GRADE),grade(grade){}
    ExamInfo(string name,bool pass):name(name),mode(PASS),pass(pass){}
    ExamInfo(string name,int
percent):name(name),mode(PERCENTAGE),percent(percent){}
    void show();
private:
    string name;
    enum{
        GRADE,
        PASS,
        PERCENTAGE,
    }mode;
    union {
        char grade;
        bool pass;
        int percent;
    };
};
void ExamInfo::show(){
    cout << name << ": ";
    switch (mode){
        case GRADE:cout<<grade;
        break;
        case PASS:cout<<(pass?"PASS":"FALL");
        break;
        case PERCENTAGE:cout<<percent;
        }
    cout<<endl;
}
int main()
{
    ExamInfo course1("English",'B');
    ExamInfo course2("Calculus",true);
    ExamInfo course3("C++",85);
    course1.show();
    course2.show();
    course3.show();
    system("Pause");
    return 0;
}

```

4.7 综合实例——一个人银行账户管理程序

```

class Savingsaccount{
private:
    int id;
    double balance;    //余额
    double rate;        //存款的年利率
    int lastDate;       //上次变更余额的日期

```



```

double accumulation;//余额按日累加之和
/*记一笔账，date为日期，amount为余额，desc为说明*/
void record(int date,double amount);
//获得指定日期为止的存款金额按日累计值
double accumulate(int date) const {
    return accumulation+balance*(date-lastDate);
}
public:
    Savingsaccount(int date,int id,double rate);
    int getId(){return id;}
    double getBalance(){return balance;}
    double getRate(){return rate;}
    void deposit(int date,double amount);    //存入现金
    void withdraw(int date,double amount);    //取出现金
    //结算利息，每年1月1日调用一次该函数
    void settle(int date);
    //显示账户信息
    void show();
};

Savingsaccount::Savingsaccount(int date,int id,double rate)
    :id(id),balance(0),rate(rate),lastDate(date),accumulation(0){
    cout<<date<<"\t#"<<id<<" is created"<<endl;
}

void Savingsaccount::record(int date,double amount){
    accumulation=accumulate(date);
    lastDate=date;
    amount=floor(amount*100+0.5)/100;
    balance+=amount;
    cout<<date<<"\t#"<<id<<"\t"<<amount<<"\t"<<balance<<endl;
}

void Savingsaccount::deposit(int date,double amount){
    record(date,amount);
}

void Savingsaccount::withdraw(int date,double amount){
    if(amount>getBalance())
        cout<<"Error:not enough money"<<endl;
    else
        record(date,-amount);
}

void Savingsaccount::settle(int date){
    double interest=accumulate(date)*rate/365;
    if(interest!=0)
        record(date,interest);
    accumulation=0;
}

void Savingsaccount::show(){
    cout<<"#"<<id<<"\tBalance:"<<balance<<endl;;
}

int main()
{
    Savingsaccount sa0(1,21325302,0.015);
    Savingsaccount sa1(1,58320212,0.015);
    sa0.deposit(5,5000);
    sa1.deposit(25,10000);
    sa0.deposit(45,5500);
    sa1.withdraw(60,4000);
    //开户后第90天到了银行的计息日，结算所有账户的年息
    sa0.settle(90);
}

```

```
sa1.settle(90);
sa0.show();
sa1.show();
system("Pause");
return 0;
}
```

```
1      #21325302 is created
1      #58320212 is created
5      #21325302      5000      5000
25     #58320212      10000     10000
45     #21325302      5500      10500
60     #58320212     -4000      6000
90     #21325302      27.64     10527.6
90     #58320212      21.78     6021.78
#21325302      Balance:10527.6
#58320212      Balance:6021.78
请按任意键继续. . .
```

4.8 深度探索

4.8.1 位域

允许将类中的多个数据成员打包，从而使不同成员可以共享相同的字节的机制。类定义中，位域的定义方式为：

```
数据类型说明符 成员名:位数;           //位数来指定一个位域所占用的二进制位数
```

note:

- 不同编译器，包含位域类所占用的空间也会有所不同
- 只有bool、int、char、enum的成员才能定义为位域
- 位域虽节省空间，但会增加时间

4.8.2 用构造函数定义类型转换

4.8.3 对象作为函数参数和返回值的传递方式

第 5 章 数据的保护与共享

5.1 标识符的作用域

5.1.1 作用域

一个标识符在程序正文中有效的区域

a. 函数原始作用域

在函数原型声明时形式参数的作用范围是函数原型作用域

```
double area(double radius);  
//标识符radius的作用范围就在函数area形参列表的左右括号之间  
//程序的其他地方都不能引用这个标识符
```

b. 局部作用域

```
void fun(int a) {  
    int b=a;  
    cin>>b;  
    if(b> 0) {  
        int c;  
        ...  
    }  
}
```

作用域图：
- **a 的作用域**：从 `fun` 开始到结束。
- **b 的作用域**：从 `int b=a;` 开始到 `}` 结束。
- **c 的作用域**：从 `int c;` 开始到 `}` 结束。

具有局部作用域的变量也称为局部变量

c. 类作用域

类可以看成是一组有名成员的集合，类X的成员m具有类作用域，对m的访问方式有以下三种

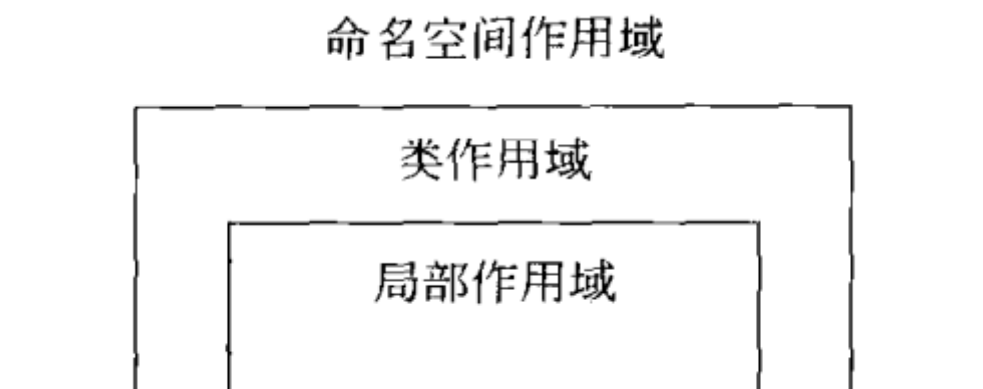
- 如果X的成员函数中没有声明同名的局部作用域标识符，那么在该函数内可以直接访问成员m
- 通过 `x.m` 或者 `X::m`
- 通过 `ptr->m`

d. 命名空间作用域

上海和武汉都有南京，直接说南京路会有歧义，所以命名空间类似“上海的南京路”和“武汉的南京路”的作用

5.1.2 可见性

作用域关系图：



作用域一般规则：

- 标识符要声明在前，引用在后
- 同一作用域，不能声明同名的标识符
- 在没有互相包含关系的不同的作用域中声明的同名标识符，互不影响
- 如果在两个或多个具有包含关系的作用域中声明了同名标识符，则外层标识符在内层不可见

5.2 对象的生存期

对象从诞生到结束的这段时间就是它的生存期，分为静态生存期和动态生存期

5.2.1 静态生存期

如果对象的生存期与程序的运行期相同，则称之为静态生存期。在函数内部的局部作用域声明具有静态生存期的对象，要使用关键字static

`static int i;` 其特点是，i并不会随着每次调用而产生一个副本，也不会随着函数返回而失效。即：当一个函数返回后，下一次再调用时，该变量还是会保持上一回的值。

5.2.2 动态生存期

局部作用域中声明的具有动态生存期的对象，习惯上也称为局部生存期对象。局部生存期对象诞生于声明点，结束于声明所在的块执行完毕之时。

```
int i=1;
void other(){
    static int a=2;
    static int b;
    //a,b是静态局部变量，具有全局寿命，局部可见
    //a,b只有第一次进入函数时被初始化
    int c=10;    //c为局部变量，具有动态生存期，每次进入函数时被初始化
    a+=2;
    i+=32;
    c+=5;
    cout << "----OTHER---" << endl;
    cout << "i: " <<i<<"a: " <<a<<"b: " <<b<<"c: " <<c<<endl;
    b=a;
}
int main()
{
    static int a;
    int b=-10;
    int c=0;
    cout<<"````MAIN````"<<endl;
    cout<<"i: " <<i<<"a: " <<a<<"b: " <<b<<"c: " <<c<<endl;
    c+=8;
    other();
    cout<<"````MAIN````"<<endl;
    cout<<"i: " <<i<<"a: " <<a<<"b: " <<b<<"c: " <<c<<endl;
    i+=10;
    other();
    system("Pause");
    return 0;
}
```

```
MAIN
i: 1a: 0b: -10c: 0
---OTHER---
i: 33a: 4b: 0c: 15
MAIN
i: 33a: 0b: -10c: 8
---OTHER---
i: 75a: 6b: 4c: 15
请按任意键继续. . .
```

```
class Clock{
public:
    Clock();
    void setTime(int newH,int newM,int newS);
    void showTime();
private:
    int hour,minute,second;
};
Clock::Clock():hour(0),minute(0),second(0){}
void Clock::setTime(int newH,int newM,int newS){
    hour=newH;
    minute=newM;
    second=newS;
}
void Clock::showTime(){
    cout<<hour<<":"<<minute<<":"<<second<<endl;
}
Clock globClock;
int main()
{
    cout<<"First time output: "<<endl;
    globClock.showTime();
    globClock.setTime(8,30,30);
    Clock myGlob(globClock);
    cout<<"Second time output: "<<endl;
    myGlob.showTime();
    system("Pause");
    return 0;
}
```

```
First time output:
0:0:0
Second time output:
8:30:30
请按任意键继续. . .
```

5.3 类的静态成员

静态成员时解决同一个类的不同对象之间数据和函数共享问题的。例如抽象出某公司全体雇员的共性，设计以下类：

```
class Employee{
private:
    int empNo;
    int id;
    string name;
    ...
};
```

如何统计雇员总数？这个数据应该放在何处，如果用类外的变量来存储总数，就不能实现数据的隐藏。若要在类中增加一个数据成员用于存放总数，必然在每个对象中都会存储一个副本，理想方案就是类的所有对象共同拥有一个用于存放总数的数据成员，这就是静态数据成员

5.3.1 静态数据成员

一个类的所有对象具有相同的属性，是指属性的个数、名称、数据类型相同，各个对象的属性值可以不同。

如果某个属性为整个类所共有，不属于任何一个具体对象，则采用static关键字来声明为静态成员。静态成员在每个类只有一个副本，由该类的所有对象共同维护和使用，从而实现了的同一类的不同对象之间的数据共享。

类属性是描述该类的所有对象共同特征的一个数据项，对于任何数据实例，它的属性值是相同的。举个例子，类是一个工厂，对象是工厂生产出的产品，那么静态成员就是存放于工厂中，属于工厂，但不属于每个产品。

静态数据成员具有静态生存期。可以用"类名::标识符"来访问

在类的定义中仅仅对静态数据成员进行引用型声明，必须在命名空间作用域的某个地方使用类名限定定义性声明，这时也可以进行初始化。**之所以类的静态数据成员需要在类定义之外再加以定义，是因为需要以这种方式专门为它们分配空间。非静态数据成员无须用这种方式定义是因为，它们的空间是与它们所属对象的空间同时分配的。**

```
class Point{
public:
    Point(int x,int y):x(x),y(y){
        count++;
    }
    Point(Point &p){
        x=p.x;
        y=p.y;
        count++;
    }
    ~Point(){count--;}
    int getX() const {return x;}
    int getY() const {return y;}
    void showCount(){
        cout<<" Object count="<<count<<endl;
    }
private:
    int x,y;
    static int count;
};
```

```
};
int Point::count=0;           //静态数据成员定义和初始化，使用类名限定
int main()
{
    Point a(4,5);
    cout<<"Point A: "<<a.getX()<<"", "<<a.getY();
    a.showCount();
    Point b(a);
    cout<<"Point B: "<<b.getX()<<"", "<<b.getY();
    b.showCount();
    system("Pause");
    return 0;
}
```

```
Point A: 4, 5 Object count=1
Point B: 4, 5 Object count=2
请按任意键继续. . .
```

5.3.2 静态函数成员

上例代码中，`showCount` 是专门用于输出静态数据成员 `count` 的。要输出 `count` 只能通过 `Point` 类的某个对象来调用函数 `showCount`。在所有对象声明之前 `count` 的值是初始值0。想要输出这个初始值，只能通过**静态成员函数**。静态成员函数也属于整个类，由同一个类的所有对象共同拥有，为这些对象所共享。

静态成员函数可以通过类名或者对象名来调用，而非静态成员函数只能通过对象名来调用

静态成员函数可以直接访问该类的静态数据和函数成员。而访问非静态成员，必须通过对象名。

```
//静态成员函数访问非静态成员
class A{
public:
    static void f(A a);
private:
    int x;
};
void A::f(A a){
    cout<<x;      //错误的
    cout<<a.x;    //正确的
}
```

可以发现，这种访问是很麻烦的，一般情况下，静态成员函数主要用来访问同一个类中的静态数据成员，维护对象之间共享的数据。

之所以静态函数成员访问非静态数据成员要指明对象，是因为对静态成员函数的调用是没有目的对象的

```
class Point{
public:
    Point(int x,int y):x(x),y(y){
        count++;
    }
    Point(Point &p){
        x=p.x;
        y=p.y;
        count++;
    }
}
```

```

    }
    ~Point(){count--;}
    int getX() const {return x;}
    int getY() const {return y;}
    static void showCount(){
        cout<<"Object count="<<count<<endl;
    }
private:
    int x,y;
    static int count;
};
int Point::count=0;           //静态数据成员定义和初始化，使用类名限定
int main()
{
    Point::showCount();
    Point a(4,5);
    cout<<"Point A: "<<a.getX()<<"", "<<a.getY();
    Point::showCount();
    Point b(a);
    cout<<"Point B: "<<b.getX()<<"", "<<b.getY();
    Point::showCount();
    system("Pause");
    return 0;
}

```

```

Object count=0
Point A: 4, 5Object count=1
Point B: 4, 5Object count=2
请按任意键继续. . .

```

5.5 共享数据的保护

5.5.1 常对象

常对象：它的数据成员在对象的整个生存期间内不能被改变。也就是说常对象必须先初始化，而且不能被更新。

```

const 类型说明符 对象名;
class A{
public:
    A(int i,int j):x(i),y(j){}
private:
    int x,y;
};
const A a(3,4);

```

在定义一个变量或常量时为它指定初值叫做初始化，而在定义一个变量或常量以后使用赋值运算符修改它的值叫做赋值。

第6章 数组、指针与字符串

6.1 数组

用于存储和处理大量同类型数据的数据结构。数组是具有一定顺序关系的若干对象的集合体，组成数组的对象称为该数组的元素。

6.1.1 数组的声明与使用

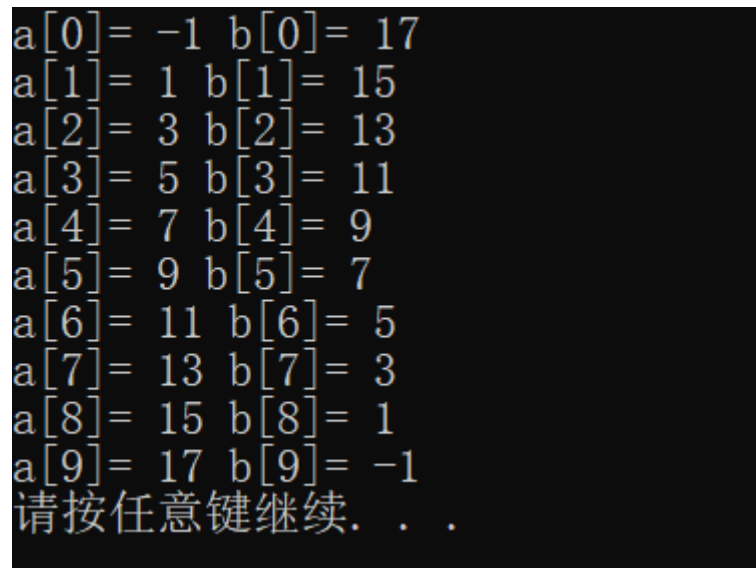
a. 数组的声明

```
数据类型 标识符[常量表达式1][常量表达式2]...
```

b. 数组的使用

只能分别对数组的各个元素进行操作

```
int main()
{
    int a[10], b[10];
    for(int i=0; i<10; i++){
        a[i]=i*2-1;
        b[10-i-1]=a[i];
    }
    for(int i=0; i<10; i++){
        cout<<"a["<<i<<"]= "<<a[i]<<" ";
        cout<<"b["<<i<<"]= "<<b[i]<<endl;
    }
    system("Pause");
    return 0;
}
```



```
a[0]= -1 b[0]= 17
a[1]= 1 b[1]= 15
a[2]= 3 b[2]= 13
a[3]= 5 b[3]= 11
a[4]= 7 b[4]= 9
a[5]= 9 b[5]= 7
a[6]= 11 b[6]= 5
a[7]= 13 b[7]= 3
a[8]= 15 b[8]= 1
a[9]= 17 b[9]= -1
请按任意键继续. . .
```

6.1.2 数组的存储与初始化

a. 数组的存储

数组元素在内存中顺序、连续存储的。

二维数组被当做一维数组的数组。例如，`int m[2][3]` 所定义的m，可以看作是这样一个数组，它的大小是2，每个元素是一个大小为3、int类型的数组。

b. 数组的初始化

数组的初始化就是在声明数组时给部分或全部元素赋初值。

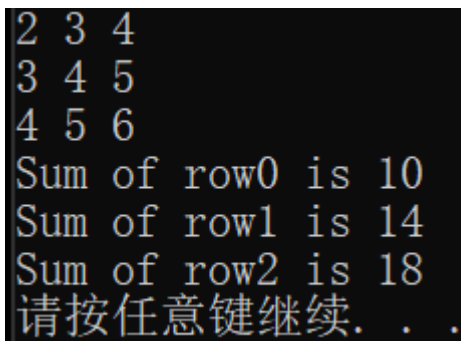
- 基本类型数组，初始化过程就是给数组元素赋值
- 对象元素，每个元素都是某个类的一个对象，初始化就是调用该对象的构造函数

6.1.3 数组作为函数参数

- 数组元素和数组名都可以作为函数的参数以实现数据间数据的传递和共享
- 使用数组名作为函数的参数，则实参和形参都应该是数组名称，且类型要相同
- 使用数组名传递数据时，传递的是地址

```
void rowSum(int a[][4],int nRow){
    for(int i=0;i<nRow;i++){
        for(int j=1;j<4;j++){
            a[i][0]+=a[i][j];
        }
    }
}

int main()
{
    int table[3][4]={1,2,3,4},{2,3,4,5},{3,4,5,6};
    for(int i=0;i<3;i++){
        for(int j=1;j<4;j++){
            {cout<<table[i][j]<<" ";}
        }
        cout<<endl;
    }
    rowSum(table,3);
    for(int i=0;i<3;i++){
        cout<<"Sum of row"<<i<<" is "<<table[i][0]<<endl;
        system("Pause");
    }
    return 0;
}
```



```
2 3 4
3 4 5
4 5 6
Sum of row0 is 10
Sum of row1 is 14
Sum of row2 is 18
请按任意键继续. . .
```

- 把数组作为参数时，一般不指定数组第一维的大小，即使制定，也会被忽略

6.1.4 对象数组

数组元素不仅可以是基本数据类型，也可以是自定义类型。对象数组的元素是对象，不仅是数据成员，而且还有函数成员。

```
//声明一个一维对象的语句形式
类名 数组名[常量表达式]
//访问对象数组的对象
数组名[下标表达式].成员名
```

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

6.2 指针

6.2.1 内存空间的访问方式

计算机的内存储器被划分为一个个内存单元，内存单元按一定的规则编号，编号就是存储单元的地址。

地址编码的基本单位是字节，每个字节由8个二进制组成。

如何利用内存单元存取数据？

- 通过变量名
- 通过地址

具有静态生存期的变量在程序开始运行之前就被分配了内存单元。具有动态生存期的变量，是在程序运行时遇到变量声明语句时被分配内存空间的。

在变量获得内存空间的同时，变量名也就成了相应内存空间的名称，在变量的整个生存期内都可以用这个名字访问该内存空间，表现在程序语句中就是通过变量名存取变量内容。

6.2.2 指针变量的声明

指针也是一种数据类型，具有指针类型的变量称之为指针变量。指针变量是用于存放内存单元地址。

```
数据类型 *标识符
//数据类型指的是指针所指向的对象
int * ptr;
//定义了一个指向int类型的指针变量，指针名称为ptr，专门用来存放int型数据的地址
```

6.2.3 与地址有关的运算 * 和 &

*称为指针运算符，也称解析，表示获取指针所指向的变量的值。

&称为取地址运算符，用来得到一个对象的地址。

6.2.4 指针的赋值

定义了一个指针，只是得到了一个用于存储地址的指针变量，但是变量中并没有确定的值，其中的地址值是一个不确定的数，不能确定这时候的指针变量中存放的是哪个内存单元的地址。

定义指针必须先赋值，再引用

指针的类型：

- 可以声明指向常量的指针，此时不能通过指针来改变所指对象的值，但指针本身可以改变，可以指向另外的对象。

```
int a;
const int * p1 = &a;    //p1是指向常量的指针
int b;
p1 = &b;                //正确，p1本身的值可以改变
*p1 = 1;                //错误，不能通过p1来改变所指的对象
```

使用指向常量的指针，可以确保指针所指向的常量可以不被意外更改。

- 可以声明指针类型的常量，这可确保指针不被改变

```
int * const p2 = &a;
p2 = &b;    //错误, p2是指针常量, 不能更改
```

- 一般情况下, 指针的值只能赋给相同类型的指针。有一种特殊的void类型指针, 可以存储任何类型的对象地址。

6.2.5 指针运算

指针是一种数据类型。

* (p1+n1) 表示p1当前所指位置后方第n1个数的内容, 也可写作 p1[n1]

0专用于表示空指针, 也就是一个不指向任何有效地址的指针

赋给指针变量的值必须是地址常量或地址变量, 不能是非0的整数

6.2.6 用指针处理数组元素

指针加减运算的特点使得指针特别适合于处理存储在一段连续内存空间中的同类数据(比如数组)

```
//三种方法输出数组10个元素
int main()
{
    int a[10];
    for(int i=0;i<10;i++)
        a[i]=i;
    //数组名+下标
    for(int i=0;i<10;i++)
        cout<<a[i]<<" ";
    //使用数组名和指针
    for(int i=0;i<10;i++)
        cout<<* (a+i)<<" ";
    //使用指针变量
    for(int * p=a;p<(a+10);p++)
        cout<<*p<<" ";
    cout<<endl;
    system("Pause");
    return 0;
}
```

6.2.7 指针数组

一个数组的元素全部都是指针变量, 称数组为指针数组。指针数组的每个元素必须都是同一类型的指针

int * pa[3] 声明了一个int类型的执行数组pa, 里面有3个元素, 每个元素都是指向int类型数据的指针。

指针数组的每个元素都是一个指针, 必须先赋值再引用。

```
int main()
{
    //利用指针数组输出单位矩阵
    //单位矩阵是主对角元素为1, 其余元素都为0的矩阵
    int line1[3]={1,0,0};
    int line2[3]={0,1,0};
    int line3[3]={0,0,1};
}
```

```

int *pLine[3]={line1,line2,line3};
cout<<"Matrix test: "<<endl;
for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        cout<<pLine[i][j]<<" ";
    }
    cout<<endl;
}
system("Pause");
return 0;
}

```

6.2.8 用指针作为函数参数

以指针作为形参，在调用时实参将值传递给形参，也就是形参与实参指针变量指向同一内存地址，这样在子函数运行过程中，通过形参指针对数据值的改变同样影响着实参指针所指向的数据值。

6.2.9 指针型函数

当一个函数返回值是指针类型时，这个函数就是指针型函数。使用指针类型函数的最主要目的是在函数结束时把大量的数据从被调函数返回到主调函数中。

```

数据类型 * 函数名(参数列表){
    函数体
}

```

6.2.10 指向函数的指针

每一个函数都有函数名，实际上这个函数名就表示函数的代码在内存中的起始地址。由此看来，调用函数的通常形式“函数名(参数表)”的实质就是“函数代码首地址(参数表)”。

函数指针就是专门用来存放函数代码首地址的变量。函数指针一旦指向了某个函数，它与函数名便具有同样的作用，函数名在表示函数代码起始地址的同时，也包括函数的返回值类型和参数个数、类型、排列次序等信息。

```

数据类型 (* 函数指针名)(形参表)
//数据类型表示函数指针所指函数的返回值类型
//第一个括号的内容指明一个函数指针的名称
//形参表列出该指针所指函数的形参类型和个数

```

通常使用typedef来定义函数指针

```

typedef int (* DoubleIntFunction)(double)
//声明了DoubleIntFunction为“有一个double形参、返回值类型为int的函数的指针”
DoubleIntFunction funcPtr;

```

函数指针在使用前也要进行赋值： 函数指针名=函数名

```

void printStuff(float){
    cout<<"This is the print stuff function."<<endl;
}
void printMessage(float data){
    cout<<"The data to be listed is "<<data<<endl;
}
void printFloat(float data){

```

```

        cout<<"The data to be printed is "<<data<<endl;
    }
    const float PI=3.14159f;
    const float TWO_PI=PI*2.0f;
    int main()
    {
        void (* functionPointer)(float);    //函数指针
        printStuff(PI);
        functionPointer=printStuff;        //函数指针指向printStuff
        functionPointer(PI);
        functionPointer=printMessage;
        functionPointer(TWO_PI);
        functionPointer(13.0);
        functionPointer=printFloat;
        functionPointer(PI);
        printFloat(PI);
        system("Pause");
        return 0;
    }

```

```

This is the print stuff function.
This is the print stuff function.
The data to be listed is 6.28318
The data to be listed is 13
The data to be printed is 3.14159
The data to be printed is 3.14159
请按任意键继续. . .

```

6.2.11 对象指针

a. 对象指针的一般概念

与基本类型的变量一样，每一个对象在初始化之后都会在内存占据一定的空间。既然可以通过对象名，也可以通过对象地址来访问一个对象。虽然，对象同时包含了数据和函数两种成员，但是对象所占据的内存空间只是用于存放数据成员的，函数成员不在每一个对象中存储副本。

对象指针就是用于存放对象地址的变量。

```

类名 * 对象指针名;
Point * pointPtr;
Point p1;
pointPtr = &p1;
使用对象指针一样可以方便地访问对象的成员:
对象指针名->成员名

```

```

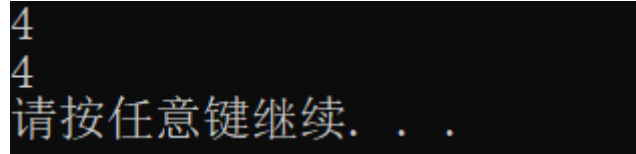
class Point{
public:
    Point(int x,int y):x(x),y(y){}
    int getX() const {return x;}
    int getY() const {return y;}
private:
    int x,y;
};
int main()

```

```

{
    Point a(4,5);
    Point * p1= &a;
    cout<<p1->getX()<<endl;
    cout<<a.getX()<<endl;
    system("Pause");
    return 0;
}

```



对象指针在使用之前，一定要先进性初始化，让它指向一个已经声明过的对象，然后再使用。

b. this指针

this指针是一个隐含于每一个类的非静态成员函数中的特殊指针(包括构造函数和析构函数)，它用于指向正在被成员函数操作的对象。

this指针实际上是类成员函数的一个隐含参数。在调用类的成员函数时，目的对象的地址会自动作为该参数的值，传递给被调用的函数，这样被调用函数就能够通过this指针来访问目的对象的数据成员。

this是一个指针常量，对于常成员函数，this又是一个指向常量的指针。在成员函数中，可以使用*this来表示正在调用该函数的对象。

c. 指向类的非静态成员的指针

类的成员函数自身也是变量、函数或者对象等等，因此可以直接将它们的地址存放到一个指针变量中，这样就可以使指针直接指向对象的成员，进而可以通过指针访问对象的成员。指向对象成员的指针也要先声明再赋值，然后引用。因此首先要声明指向该对象所在类的成员的指针。

```

类型说明符 类名:: * 指针名;           //声明指向数据成员的指针
类型说明符 (类名:: * 指针名)(参数表); //声明指向函数成员的指针

```

对数据成员指针赋值的一般语法：

```

指针名 = &类名::数据成员名;
//对类成员取地址时也要遵守访问权限的约定，即：在一个类的作用域之外不能够对它的私有成员取地址

```

与普通变量不同，类的定义只确定了各个数据成员的类型、所占内存大小以及它们的相对位置，在定义时并不为数据成员分配具体的地址。因此经过赋值之后，只是说明了被赋值的指针是专门用于指向哪个数据成员的，同时在指针中存放该数据成员在类中的数据成员(即相对于起始地址的地址偏移量)，此时这样的指针并不能访问什么。

由于类是通过对象来进行实例化的，在声明类的对象时才会为具体的对象分配具体内存空间，这时只要将对象在内存中的起始地址与成员指针中存放的相对偏移结合起来就可以访问到对象的数据成员了。如何结合？

```

对象名.* 类成员指针名
或
对象指针名->*类成员指针名

```

而成员函数指针的声明如下：

指针名 = & 类名::成员函数名

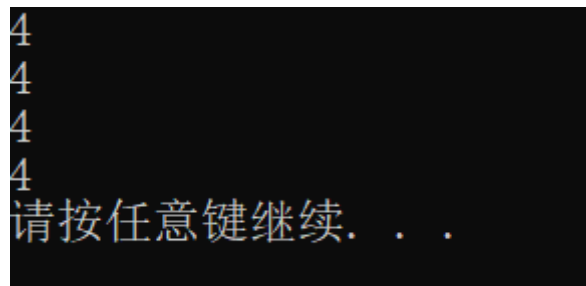
调用成员函数?

(对象名.* 类成员指针名)(参数表)

或

(对象指针名-> * 类成员指针名)(参数表)

```
class Point{
public:
    Point(int x,int y):x(x),y(y){
        count++;
    }
    Point(Point &p){
        x=p.x;
        y=p.y;
        count++;
    }
    ~Point(){count--;}
    int getX() const {return x;}
    int getY() const {return y;}
    static void showCount(){
        cout<<"Object count="<<count<<endl;
    }
private:
    int x,y;
    static int count;
};
int Point::count=0;           //静态数据成员定义和初始化，使用类名限定
int main()
{
    Point a(4,5);
    Point *p1 = &a;
    int (Point::* funcPtr)() const=&Point::getX;
    //定义了一个成员函数指针并初始化
    cout<<(a.*funcPtr)()<<endl;
    cout<<(p1->*funcPtr)()<<endl;
    cout<<a.getX()<<endl;
    cout<<p1->getX()<<endl;
    system("Pause");
    return 0;
}
```



```
4
4
4
4
请按任意键继续...
```

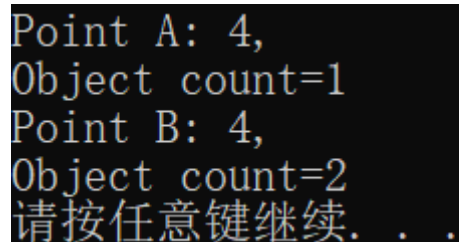

d. 指向类的静态成员的指针

对类的静态对象的访问是不依赖于对象的，因此可以用普通的指针来指向和访问静态成员

```
class Point{
public:
    Point(int x,int y):x(x),y(y){
        count++;
    }
    Point(const Point &p){
        x=p.x;
        y=p.y;
        count++;
    }
    ~Point(){count--;}
    int getX() const {return x;}
    int getY() const {return y;}
    static void showCount(){
        cout<<"Object count="<<count<<endl;
    }
    static int count;
private:
    int x,y;
};

int Point::count=0;           //静态数据成员定义和初始化，使用类名限定
int main()
{
    //定义一个int型指针，指向类的静态数据成员
    int* ptr= &Point::count;
    Point a(4,5);
    cout<<"Point A: "<<a.getX()<<"", "<<endl;
    cout<<"Object count="<<*ptr<<endl;

    Point b(a);
    cout<<"Point B: "<<b.getX()<<"", "<<endl;
    cout<<"Object count="<<*ptr<<endl;
    system("Pause");
    return 0;
}
```



```
Point A: 4,
Object count=1
Point B: 4,
Object count=2
请按任意键继续. . .
```

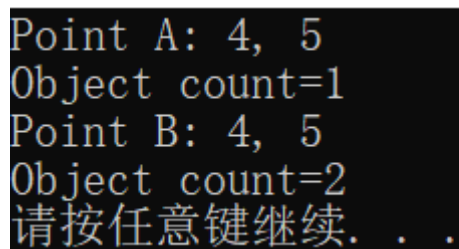
```
class Point{
public:
    Point(int x,int y):x(x),y(y){
        count++;
    }
    Point(const Point &p){
        x=p.x;
        y=p.y;
        count++;
    }
};
```

```

}
~Point(){count--;}
int getX() const {return x;}
int getY() const {return y;}
static void showCount(){
    cout<<"Object count="<<count<<endl;
}

private:
    int x,y;
    static int count;
};
int Point::count=0;           //静态数据成员定义和初始化，使用类名限定
int main()
{
    void (* funcPtr)()=Point::showCount;
    Point a(4,5);
    cout<<"Point A: "<<a.getX()<<"", "<<a.getY()<<endl;
    funcPtr();
    Point b(a);
    cout<<"Point B: "<<b.getX()<<"", "<<b.getY()<<endl;
    funcPtr();
    system("Pause");
    return 0;
}

```



```

Point A: 4, 5
Object count=1
Point B: 4, 5
Object count=2
请按任意键继续. . .

```

6.3 动态内存分配

在程序运行过程中申请和释放的存储单元称为堆对象

建立和删除堆对象使用两个运算符：**new**和**delete**

- 运算符new
 - 功能：动态分配内存，动态创建堆对象

new 数据类型 （初始化参数列表）

- 内存申请成功，new运算会返回一个指向新分配内存首地址的类型的指针，通过这个指针对堆对象进行访问，内存申请失败，会抛出异常

```

int * point;
point = new int(2);
//动态分配了用于存放int类型数据的内存空间，并将初值2存入该空间，然后将首地址赋给指针point

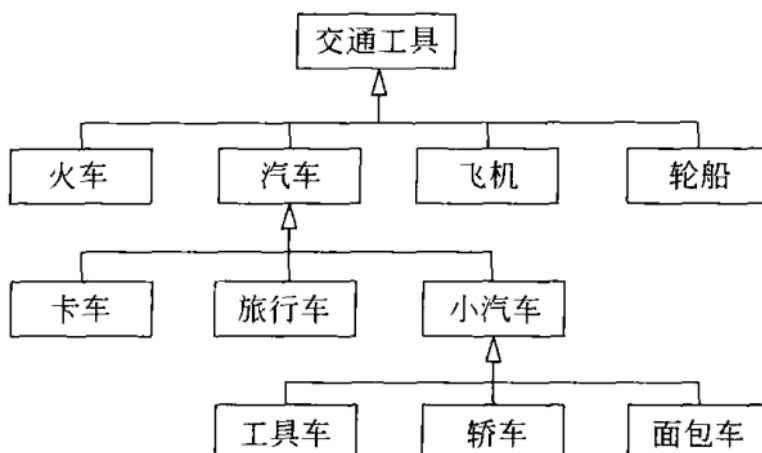
```

-

第 7 章 继承与派生

7.1 类的继承与派生

7.1.1 继承关系举例



类的继承，是新的类从已有类那里得到已有的特性。

原有的类称**基类或父类**，新的类称为**派生类或子类**。

7.1.2 派生类的定义

假设Base1和Base2是以应对类，派生类Derived的派生类

```
class Derived:public Base1,private Base2{
public:
    Derived();
    ~Derived();
};
```

一个派生类，可以同时有多个基类，这种情况称之为多继承。

一个派生类只有一个直接基类的情况，称为单继承。

7.1.3 派生类生成过程

a. 吸收基类成员

派生类包含基类中除构造和析构函数之外的所有成员，派生过程中构造函数和析构函数都不被继承。

b. 改造基类成员

改造包含两方面：

- 基类成员的访问控制问题，这部分靠派生类定义时的继承方式来控制
- 对基类数据或函数成员的覆盖或隐藏。隐藏是在派生类中声明一个和基类数据或函数同名的成员。如果派生类声明了一个和某基类成员同名的新成员，派生类的新成员就隐藏了外层同名成员。**如果声明了一个同名的成员函数，则参数表也要相同，参数相同的情况属于重载。**在派生类中或者通过派生类的对象，直接使用成员名就只能访问到派生类中声明的同名函数，这称为**同名隐藏**。

c. 添加新的成员

7.2 访问控制

基类的自身成员可以对基类中任何一个其他成员进行访问，但对于类的对象，就只能访问该类的公有成员。

7.3 类型兼容原则

类型兼容规则是指在需要基类对象的任何地方，都可以使用公有派生类的对象来替代。包括以下情况：

- 派生类的对象可以隐含转换为基类对象
- 派生类的对象可以初始化基类的引用
- 派生类的指针可以隐含转换为基类指针