



第3章：类和对象 再讨论



本章内容

1

对象指针和对象引用

2

对象数组和对象指针数组

3

常类型

4

子对象和堆对象

5

类型转换

6

类与对象小结



1.1 指向类的成员的指针

通过指向成员的指针只能访问公有成员

- 声明指向公有数据成员的指针
 - 类型说明符 类名::*指针名;
- 声明指向公有成员函数的指针
 - 类型说明符 (类名::*指针名)(参数表);



```
class A
{
    public:
        int fun(int b) { return a*c+b;}
        A(int i) { a=i;}
        int c; // 公有数据成员
    private:
        int a;
}
```



指向数据成员的指针

- 说明指针应该指向哪个成员
 - 指针名=&类名::数据成员名;
 - `int A::*pc=&A::c;`
- 通过对象名（或对象指针）与成员指针结合来访问数据成员
 - 对象名.* 类成员指针名
 - `A a; a.*pc=8;`
 - 对象指针名—>*类成员指针名
 - `A *p; p->*pc=8;`



指向函数成员的指针

- 初始化
 - 指针名=类名::函数成员名;
 - `int (A::*pfun) (int) = A::fun;`
- 通过对象名（或对象指针）与成员指针结合来访问函数成员
 - (对象名.* 类成员指针名)(参数表)
 - `A a; a.*pfun(9);`
 - (对象指针名—>*类成员指针名)(参数表)
 - `A *p; p->*pfun(9);`



1.2 对象指针和对象引用作函数参数

对象指针作为函数参数

- 实现传址调用
- 使用对象指针作为形参仅将对象的地址值传给形参，提高运行效率。

对象引用作函数参数

- 实现传址调用
- 比指针更简单、更直接



```
#include <iostream.h>
class M
{
public:
    M() {x=y=0;}
    M(int i, int j) {x=i; y=j;}
    void copy(M &m);
    void setxy(int i, int j) {x=i; y=j;}
    void print() {cout<<x<< " ," <<y<<endl;}
private:
    int x,y;
};
```




```
void M::copy(M &m)
{
    x = m.x;  y = m.y;
}
void fun(M m1, M &m2)
{
    m1.setxy(12,15);  m2.setxy(22,25);
}
void main()
{
    M p(5,7),q;
    q.copy(p);
    fun(p,q);
    p.print();
    q.print();
}
```



1.3 this 指针

- 隐含于每一个类的成员函数中的特殊指针。
- 明确地指出了成员函数当前所操作的数据所属的对象。
- 当通过一个对象调用成员函数时，系统先将该对象的地址赋给this指针，然后调用成员函数，成员函数对对象的数据成员进行操作时，就隐含使用了this指针。



例如: Point类的构造函数体
中的语句:

- `X=xx;`
- `Y=yy;`

相当于:

- `this->X=xx;`
- `this->Y=yy;`

```
class Rectangle
```

```
{
```

```
public:
```

```
    Rectangle (double i , double j)
```

```
    { height = i ; width= j ;}
```

```
    double Area()
```

```
    { return height * width;}
```

```
private:
```

```
    double height, width;
```

```
};
```

```
Rectangle r(5,6);
```

```
r.Area();
```

```
double Area (Rectangle *this)
```

```
{ return (this->height)*(this->width);}
```

```
r.Area(&r);
```



```
#include <iostream.h>
class A
{
public:
    A(int i,int j)
    { a=i;b=j; }
    A()
    { a=b=0; }
    void Copy(A &a);
    void Print()
    { cout<<a<<','<<b<<endl; }
private:
    int a,b;
};
```

```
void A::Copy(A &a)
{
    if(this==&a)
        return;
    *this=a;
}
void main()
{
    A a1,a2(1,5);
    a1.Copy(a2);
    a1.Print();
}
```



2.1 对象数组

➤ 相同类的若干个对象的集合构成一个对象数组。

➤ 声明

类名 数组名[元素个数];

➤ 访问方法：通过下标访问

数组名[下标].成员名



对象数组初始化

数组中每一个元素对象被创建时，系统都会调用类构造函数初始化该对象。

通过初始化列表赋值。

- 例：

`Point A[2]={Point(1,2), Point(3,4)};`

如果没有为数组元素指定显式初始值，数组元素便使用默认值初始化（调用默认构造函数）。



数组元素所属类的构造函数

- 不声明构造函数，则采用默认构造函数。
- 各元素对象的初值要求为相同的值时，可以声明具有**默认形参值的构造函数**。
- 各元素对象的初值要求为不同的值时，需要声明带形参的构造函数。
- 当数组中每一个对象被删除时，系统都要调用一次析构函数。



```
#include <string.h>
```

```
class Student
```

```
{
```

```
    char name[20];
```

```
    long int stuno;
```

```
    int score;
```

```
public:
```

```
    Student(char name1[]="",long int no=0,int sco=0)
```

```
    { // 默认形参值的构造函数
```

```
        strcpy(name,name1);
```

```
        stuno=no;
```

```
        score=sco;
```

```
    }
```

```
    void Setscore(int n)
```

```
    { score=n; }
```

```
    void Print()
```

```
    { cout<<stuno<<'\t'<<name<<'\t'<<score<<endl; }
```

```
};
```




```
void main()
{
    Student stu[5]={ Student("Ma",5019001,94),
                     Student("Hu",5019002,95),
                     Student("Li",5019003,88)};
    stu[3] = Student("Zhu",5019004,85);
    stu[4] = Student("Lu",5019005,90);
    stu[1].Setscore(98);
    for(int i(0);i<5;i++)
        stu[i].Print();
}
```



2.2 指向数组的指针和指针数组

1) 指向数组的指针

➤ 指向数组的指针可以指向一维对象数组，也可以指向二维数组。

➤ 指向一维数组的一级指针的声明：

类名 (*指针名)[大小]



```
#include <iostream.h>
class A
{
public:
    A(int i,int j)
    { x=i;y=j; }
    A()
    { x=y=0; }
    void Print()
    { cout<<x<<','<<y<<'\t'; }
private:
    int x,y;
};
```

```
void main()
{
    A aa[2][3];
    int a(3),b(5);
    for(int i(0);i<2;i++)
    for(int j(0);j<3;j++)
        aa[i][j]=A(a++,b+=2);
    A (*paa)[3]=aa;
    for(i=0;i<2;i++)
    {
        for(int j=0;j<3;j++)
            (*(paa+i+j)).Print();
        cout<<endl;
    }
}
```



2.2 指向数组的指针和指针数组

2) 指针数组是指数组的元素是指向对象的指针，并要求所有数组元素都是指向相同类的对象的指针。

➤ 声明：

类名 *对象指针数组名[大小];

➤ 对象指针数组可以被初始化，也可以被赋值。



```
#include <iostream.h>
class A
{
public:
    A(int i=0,int j=0)
    { x=i;y=j; }

    void Print()
    { cout<<x<<','<<y<<endl;}

private:
    int x,y;
};
```

```
void main()
{
    A a1,a2(5,8),a3(2,5),a4(8,4);
    A *array1[4]={&a4,&a3,&a2,&a1};
    for(int i(0);i<4;i++)
        array1[i]->Print();
    cout<<endl;
    A *array2[4];
    array2[0]=&a1;
    array2[1]=&a2;
    array2[2]=&a3;
    array2[3]=&a4;
    for(i=0;i<4;i++)
        array2[i]->Print();
}
```



3.1 常类型——常对象

`const <类名> <对象名>(<初值>)`

`<类名> const <对象名>(<初值>)`



3.2 常指针和常引用

1. 常指针

- 地址值为常量的指针

`<类型> *const <指针名> = <初值>`

- 所指向的值为常量的指针

`const <类型> *<指针名> = <常量>`

2. 常引用

`const <类型> &<引用名> = <初值>`



```
#include<iostream.h>
```

```
void main( ) {
```

```
    int a=0, b=0;
```

```
    int &d = a; // 引用
```

```
    const int &c = a; // 常引用
```

```
    c = 1; // 非法
```

```
    a = 1; // 合法
```

```
    d = 1; // 合法
```

```
    int *const p1 = &a; // 地址值为常量的指针
```

```
    const int *p2 = &a; // 所指向值为常量的指针
```

```
    p1 = &c; // 非法
```

```
    p2 = &b;
```

```
    *p2 = 1; // 非法
```

```
    *p1 = 1; // 合法
```

```
    cout << a << b << c << d << *p2 << endl;
```

```
}
```




3.3 常成员函数

➤ 常成员函数格式如下：

```
<类型> <成员函数名> (<参数表>) const  
{ <函数体> }
```

➤ 常成员函数可以引用const数据成员，也可以引用非const的数据成员。

➤ 非常数据成员在常成员函数中可以引用，但不可改变。

➤ 常对象的数据成员都是常数据成员，其值不能改变。

➤ 常对象只能调用常成员函数，不能调用非常成员函数。



```
#include <iostream.h>
```

```
class B
```

```
{
```

```
public:
```

```
    B(int i, int j)
```

```
    { b1=i;b2=j; }
```

```
    void Print()
```

```
    { cout<<b1<<'<';'<<b2<<endl; }
```

```
    void Print() const
```

```
    { cout<<b2<<'<:'<<b1<<endl; }
```

```
private:
```

```
    int b1,b2;
```

```
};
```

```
void main()
```

```
{
```

```
    B b1(5,10);
```

```
    b1.Print();
```

```
    const B b2(2,8);
```

```
    b2.Print();
```

```
}
```



3.4 常数据成员

➤ 常数据成员格式如下：

`const <类型> <常数据成员名>`

➤ 常数据成员的值是不能改变的。

➤ 常数据成员初始化是通过构造函数的成员初始列表来实现的。

➤ 构造函数的成员初始化列表的格式：

`<构造函数名> (<参数表>): <成员初始化列表>`

`{ <函数体> }`



```
class A
{
public:
    A(int i, int j) : a(i)
    { b=j; }
    ...

private:
    const int a;
    int b;
};
```



```
#include <iostream.h>
class A
{
public:
    A(int i);
    void Print() {
        cout<<a<<', '<<b
        <<', '<<r<<endl;
    }
    const int &r;
private:
    const int a;
    static const int b;
};
```

```
const int A::b = 15;
//静态成员初始化
```

```
A::A( int i ) : a( i ), r( a ) { }
//常数据成员初始化
```

```
void main()
{
    A a1(10), a2(20);
    a1.Print();
    a2.Print();
}
```



4.1 子对象

- 在一个类中可以使用另一个类的对象作其数据成员，这种对象的数据成员称为子对象。
- 子对象反映两个类之间的包含关系。

```
class B  
{  
    ...  
};
```

```
class A  
{  
    ...  
    Private:  
        B b;  
};
```



子对象的初始化

➤ 子对象初始化应放在构造函数的成员初始化列表中。

➤ 具体格式如下：

```
<构造函数名> (<参数表>) : <成员初始化列表>  
{  
    <函数体>  
}
```



```
#include <iostream.h>
class B
{
public:
    B(int i,int j)
    { b1=i;b2=j; }
    void Print() {
        cout<<b1<<','<<b2<<endl;
    }
private:
    int b1,b2;
};
```

```
class A {
public:
    A(int i,int j,int k) : b(i , j)
    { a=k; } //子对象初始化
    void Print() {
        b.Print();
        cout<<a<<endl;
    }
private:
    B b;
    int a;
};

void main() {
    B b(7,8);
    b.Print();
    A a(4,5,6);
    a.Print();
}
```




```
Class Club {
```

```
Private:
```

```
    string name;
```

```
    Table members;
```

```
    Table officers;
```

```
    Date founded;
```

```
    //...
```

```
Public:
```

```
    Club(const string& n, Date fd);
```

```
};
```

```
Club::Club (const string& n, Date fd) // 构造函数
```

```
    : name(n), members(), officers(), founded( fd )
```

```
    // 子对象初始化表
```

```
{
```

```
    //...
```

```
}
```



4.2 堆对象 (动态对象)

1. 使用new运算符创建堆对象

① 使用new运算符创建一个对象或其他类型变量的格式如下:

new <类名> 或者 <类型说明符> (<初值>);

```
A * pa;
```

```
pa = new A(3,5);
```

```
int * p;
```

```
p = new int(8);
```



② 使用new运算符创建一个对象数组或其他类型数组的格式如下:

new <类名> 或者 <类型说明符>[<大小>];

A * parray;

parray = new A[10];

10* sizeof (class A)



对象数组创建后可使用如下语句，判断创建是否成功：

```
if( parray==NULL )  
{  
    cout<<"数组创建失败! /n";  
    exit(1);  
}
```

使用new所创建的数组，可以给其元素赋值。



2. 使用delete运算符释放对象

delete运算符的功能是用来释放使用new运算符创建的堆对象和堆对象数组的。

① 使用delete运算符释放对象或变量的格式如下：

delete <指针名>;

② 使用delete运算符释放对象数组或其他类型数组的格式如下：

delete []<指针名>;



```
#include <string.h>
```

```
class B
```

```
{
```

```
public:
```

```
    B() {        strcpy(name, " "); b=0;  
           cout<<"Default constructor called.\n";
```

```
    }
```

```
    B(char *s, double d) {  
        strcpy(name,s); b=d;  
        cout<<"Constructor called.\n";
```

```
    }
```

```
    ~B() { cout<<"Destructor called."<<name<<endl; }
```

```
    void GetB( char *s, double &d) { strcpy(s, name); d=b; }
```

```
private:
```

```
    char name[20];
```

```
    double b;
```

```
};
```



```
void main()
{
    B *pb;
    double d;
    char s[20];
    pb = new B[4];
    pb[0]=B("Ma",3.5);
    pb[1]=B("Hu",5.8);
    pb[2]=B("Gao",7.2);
    pb[3]=B("Li",9.4);
    for(int i=0;i<4;i++)
    {
        pb[i].GetB(s,d);
        cout<<s<<','<<d<<endl;
    }
    delete []pb ;
}
```



注意:

1. 使用new运算符创建对象数组时，先使用默认构造函数对数组元素进行初始化；
2. 创建对象数组后，通常通过重载的赋值运算符使数组元素获得所需要的值；
3. 在使用重载赋值运算符给对象赋值时，如果右值表达式不是一个对象，应先转换成一个对象再进行赋值；
4. 对象数组删除，delete[]



5.1 类型的隐含转换

C++语言编译系统提供内部数据类型的自动隐含转换规则如下：

- 在执行算术运算时，低类型自动转换为高类型。
- 在赋值表达式中，赋值运算符右边表达式的类型自动转换为左边变量的类型。
- 在函数调用时，将调用函数的实参初始化形参，系统将实参转换为形参类型后，再进行传值。
- 在函数有返回值时，系统自动将返回的表达式的类型转换为该函数的类型后，再将表达式的值返回给调用函数。

如果数据精度受损失，系统会报错。



5.2 构造函数具有类型转换功能

```
#include <iostream.h>
class D
{
public:
    D() { d=0; }
    D(double i) { d=i; }

    void Print()
    { cout<<d<<endl; }

private:
    double d;
};
```

```
void main()
{
    D d;
    d=20;
    d.Print();
}
```



5.3 类型转换函数

➤ 类型转换函数:

实现由某种类类型转换为某种指定的数据类型
的操作。

➤ 类型转换函数的格式:

operator <数据类型说明符>()

{ <函数体> }



```
#include <iostream.h>
```

```
class E
```

```
{
```

```
public:
```

```
    E(int i,int j) { den=i; num=j; }
```

```
    operator double();
```

```
private:
```

```
    double den, num;
```

```
};
```

```
E::operator double() {
```

```
    return double(den)/double(num);
```

```
}
```

```
void main()
```

```
{
```

```
    E e(6,10);
```

```
    double a(3.5);
```

```
    a += e-2;
```

```
    cout<<a<<endl;
```

```
}
```

类和对象的主要知识点

类和对象

基本概念

什么是类
什么是对象

类定义的语法形式

类中的成员

数据成员+
成员函数+

友元

友元普通(全局)函数

友元成员函数

友元函数

其中的所有的成员函数是承认它为友元的类的友元函数

友元类

可以通过类的对象直接访问类的私有成员，友元不是类的成员

友元必须在承认它为友元的类的内部使用关键字**friend**声明，

成员的访问权限

public
private
protected

类和对象的主要知识点

