C++

Programming

# 类和对象 III
# Classes and Objects III

2025年3月17日

学而不厌 诲人不倦

# Chapter 3 怎样使用类和对象

# 3.4 对象动态建立、释放、赋值与复制

## 1. 对象的动态建立和释放

**格式：　new　类名；**

功能：在堆里分配内存，建立指定类的一个对象。如果分配成功，将返回动态对象的起始地址（指针）；如不成功，返回0。为了保存这个指针，必须事先建立以类名为类型的指针变量。

**格式：　类名　* 指针变量名；**

**例：　Box　*pt;**

**pt = new Box;**

如分配内存成功，就可以用指针变量pt访问动态对象的数据成员。

```
cout <<  pt -> height;
cout <<  pt -> volume();
```

**格式：　delete　指针变量；　　delete [] 指针变量；**

指针变量里存放的是用new运算返回的指针

# 3.4 对象动态建立、释放、赋值与复制

## 2. 对象的赋值

如果一个类定义了两个或多个对象，则这些同类对象之间可以互相赋值。这里所指的对象的值含义是对象中所有数据成员的值。

**格式　对象1 = 对象2;**

功能：将对象2值赋予对象1。对象1、对象2都是已建立好的同类对象

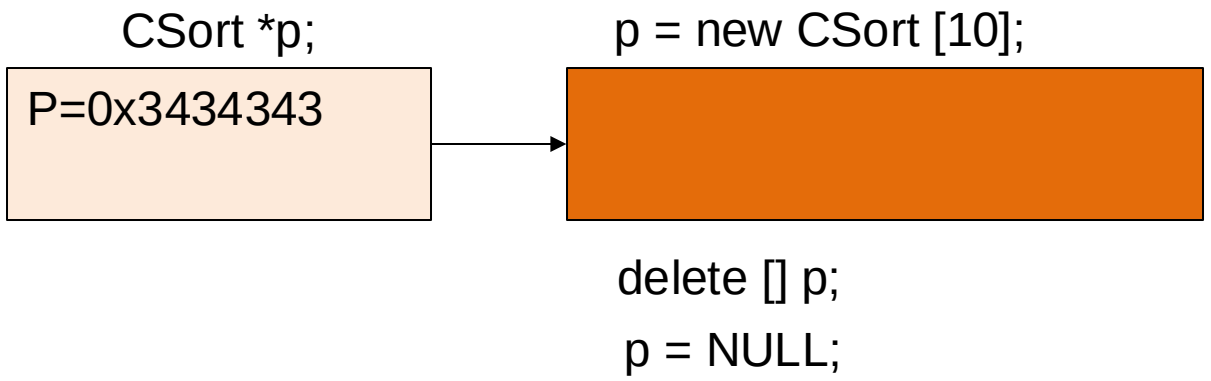
```
#include <iostream>
using namespace std;

class Box
{ public:
Box(int=10,int=10,int=10);
int volume();
private:
int height;
int width;
int length;
};
```

```
int main()
{
Box box1(15,30,25),box2;
cout<<"box1 体积= "<<box1.volume()<<endl;
box2=box1;
cout<<"box2 体积= "<<box2.volume()<<endl;
return 0;
}
```

## 2. 对象的赋值

对象的赋值只对数据成员操作。数据成员中不能含有动态分配的数据，否则在赋值时可能出现严重。

```
class A
{
public:
A(){ p = new int [100];}
~A(){delete []p;}
private:
int *p;
}
int main()
{
A a1, a2;//定义两个对象
a2 = a1;//对象赋值错误
return 0;
}
```




此时a2=a1这一句，a2原来的指针p丢失，内存泄漏。最后a1、a2析构的时候，原来a1.p被delete[]操作符删两次，可能出错。

## 3. 对象的复制

创建对象必须调用构造函数，复制对象要调用复制构造函数。
复制对象有两种格式：

**类名 对象2(对象1);** //按对象1复制对象2。
**类名 对象2=对象1,对象3=对象1,…;** //按对象1复制对象2、对象3。

```
Box ::Box ( const Box & b )
{ height = b.height;
width = b.width;
length = b.length; }
```

```
Box box1(15,30,25);
cout<<"box1的体积= "<<box1.volume()<<endl;
//Box box2=box1,box3=box2;
Box box2(box1),box3(box2);
```

**复制构造函数只有一个参数，这个参数是本类的对象，且采用引用对象形式**，为了防止修改数据，加const限制。
未定义复制构造函数，编译系统将提供默认的复制构造函数。
当函数参数是对象，调用函数时，调用复制构造函数将实参对象复制给形参对象。

# Chapter 3 怎样使用类和对象

- 3.1 构造与析构函数
- 3.2 对象数组与对象指针
- 3.3 共用数据的保护
- 3.4 对象动态建立、释放、赋值与复制
- 3.5 静态成员和友元
- 3.6 类模板

# 3.5 静态数据成员和友元

## 1. 静态数据成员　定义格式：　static　类型　数据成员名

所有对象共享静态数据成员，不能用构造函数初始化。

**数据类型　类名::静态数据成员名 = 初值;**

```
class Box
{
public:
Box(int=10,int=10,int=10);
int volume();
private:
static int height;//定义
int width;
int length;
};
```

```
int Box::height=10;
int main()
{
Box a(15,20),b(20,30);
cout<<a.height<<endl;//通过对象引用
cout<<b.height<<endl; //通过对象引用
cout<<Box::height<<endl; //通过类引用
cout<<a.volume()<<endl;
return 0;
}
```

设Box有 n个对象 box1..boxn。这n个对象的height 成员在内存中共享一个整型数据空间。如果某个对象修改了height 成员的值，其他 n-1个对象的 height 成员值也被改变。从而达到n个对象共享height成员值的目的。

# 3.5 静态数据成员和友元

## 2. 静态成员函数

定义格式： **static 类型　成员函数( 形参表 ){…}**

调用格式： **类名::成员函数( 实参表 )**

静态成员函数不带this指针，必须用**对象名**和**成员运算符.**访问非静态成员；
而普通成员函数有this指针，可以在函数中直接引用成员名。

```
class Student
{private:
int num;
int age;
float score;
static float sum;
static int count;
public:
Student(int,int,int);
void total();
static float average();
};
```

```
Student::Student(int m,int a,int s)
{ num=m;
age=a;
score=s; }
void Student::total()
{ sum+=score;
count++; }
float Student::average()
{ return(sum/count); }

float Student::sum =0;
int Student::count =0;
```

## 2. 静态成员函数

```
int main()
{ Student stud[3]={
Student(1001,18,70),
Student(1002,19,79),
Student(1005,20,98) };
int n;
cout<<"请输入学生的人数： ";
cin>>n;
for(int i=0;i<n;i++)
stud[i].total();
cout << n <<"个学生的平均成绩是";
cout <<Student :: average() << endl;
return 0;
}
```

# 3.5 静态数据成员和友元

## 2. 静态成员函数

```
#include <iostream>
using namespace std;
class Point
{private:
    int X,Y;
    static int countP;
public:
    Point(int xx=0, int yy=0) {X=xx; Y=yy; countP++; }
   Point(Point &p);//  复制构造函数
    int GetX() {return X;}
    int GetY() {return Y;}
   void GetC() {cout<<" Object id="<<countP<<endl;}
};
```

```
Point::Point(Point &p)
{   X=p.X;
    Y=p.Y;
    countP++;
}
int Point::countP=0;
void main()
{   Point A(4,5);
    cout<<"Point A,"<<A.GetX()
        <<","<<A.GetY();
    A.GetC();
    Point B(A);
    cout<<"Point B,"<<B.GetX()
        <<","<<B.GetY();
    B.GetC();
}
```

## 2. 静态成员函数

```
#include<iostream>
using namespace std;
class Application
{private:
    static int global;
 public:
    static void f();
    static void g();
};
int Application::global=0;
```

```
void Application::f()
{  global=5;}
void Application::g()
{  cout<<global<<endl;}

int main()
{
 Application::f();
 Application::g();
 return 0;
}
```

## 3. 友元： 友元函数

**C++ 通过友元实现从类的外部访问类的私有成员这一特殊要求。**
友元可以是不属于任何类的一般函数，也可以是另一个类的成员函数，还可以是整个的一个类。友元是C++提供的一种**破坏数据封装和信息隐藏**的机制。

**友元函数声明格式：**

**friend 类型 类1::成员函数x(类2 &对象);**
**friend 类型 函数y( 类2 &对象 );**
类1是另一个类的类名。类2是本类的类名。

**功能：** 第一种形式在类2中声明类1的成员函数 x为友元函数。
第二种形式在类2 中声明一个普通函数 y 是友元函数。

# 3.5 静态数据成员和友元

## 3. 友元： 友元函数

```
//将普通函数声明为友元函数 。

class Time
{  public:
   Time(int,int,int);
   friend void display(Time &);
  private:
    int hour;
    int minute;
    int sec;
};
```

```
Time::Time(int h,int m,int s)
{  hour = h;
   minute = m;
   sec = s; }
void display(Time &t)
{
cout<<t.hour<<endl;
}
int main()
{  Time t1(10,13,56);
   display(t1);
   return 0;
}
```

# 3.5 静态数据成员和友元

## 3. 友元： 友元成员函数

```
class Date;
class Time
 {private:
   int hour;
   int minute;
   int sec;
public:
   Time(int,int,int);
   void display(const Date&);
 };
```

```
class Date
 {private:
   int month;
   int day;
   int year;
public:
   Date(int,int,int);
   friend void Time::display(const Date &);
 };
 Time::Time(int h,int m,int s)
 {hour=h;
  minute=m;
  sec=s;  }
```

## 3. 友元： 友元成员函数

注意：友元是单向的，此例中声明Time的成员函数display是Date类的友元，允许它访问Date类的所有成员。但不等于说Date类的成员函数也是Time类的友元。

```
void Time::display(const Date &da)
 {cout<<da.month<<"/"<<da.day<<"/"<<da.year
  cout<<endl;
  cout<<hour<<":"<<minute<<":"<<sec<<endl;
 }

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

```
int main()
{  Time t1(10,13,56);
   Date d1(12,25,2004);
   t1.display(d1);
   return 0; }
```

运行时输出：

12/25/2004

10:13:56

- **3. 友元： 友元类**

**C++允许将一个类声明为另一个类的友元。**假定A类是B类的友元类，A类中所有的成员函数都是B类的友元函数。

**在B类中声明A类为友元类的格式**：

```
friend A;
```

注意：

（1）友元关系是单向的，不是双向的。

（2）友元关系不能传递。

实际中一般并不把整个类声明友元类，而只是将确有需要的成员函数声明为友元函数。

# 3.5 静态数据成员和友元

- 3. 友元： 友元类

```
class B
{ public:
        void disp1(A temp)
        {  temp.x++; cout<<"disp1:x="<<temp.x <<endl;   };
        void disp2(A temp)
        {  temp.x--; cout<<"disp2:x="<<temp.x <<endl;    };
};
```

```
#include <iostream.h>
#include <math.h>
class A
{
private:
        int x;
public:
        A( ){x=3;}
        friend class B;
};
```

```
int main(int argc, char* argv[])
{
        A a;
        B b;
        b.disp1(a);
        b.disp2(a);
        return 0;
}
```

# 3.5 静态数据成员和友元

- **3. 友元：** **友元类**

```
class Student;   //前向声明，类名声明
class Teacher
{privated:
    int noOfStudents;
    Student * pList[100];
public:
    void assignGrades(Student& s);      //   赋成绩
    void adjustHours(Student& s);       //   调整学时数
};
class Student
{privated:
    int Hours;
    float gpa;
public:
    friend class Teacher;
};
void Teacher:: assignGrades(Student& s){…};
void Teacher:: adjustHours(Student& s){…};
```

通过传递参数可以实现对类Student所有成员的操作

函数定义必须在类Student定义之后

# 3.6 类模板

## 类模板的作用

对于功能相同而只是数据类型不同的函数，可以定义**函数模板**。
对于功能相同的类而数据类型不同，不必定义出所有类，只要定义一个可对任何类进行操作的**类模板**。

```
//比较两个整数的类
class Compare_int
{private:
    int x,y;
public:
   Compare_int(int a,int b)
    {x=a;y=b;}
   int max()
    {return (x>y)?x:y;}
   int min()
    {return (x<y)?x:y;}
};
```

```
//比较两个浮点数的类
class Compare_float
 {private:
    float x,y;
public:
   Compare_float(float a,float b)
    {x=a;y=b;}
   float max()
    {return (x>y)?x:y;}
   float min()
    {return (x<y)?x:y;}
};
```

## 类模板的定义格式

```
template<class numtype>
class Compare
 { private:
    numtype x,y;
 public:
  Compare(numtype a,numtype b) //  构造函数
   {  x=a;y=b;}
  numtype max()
   {  return (x>y)?x:y;}
  numtype min()
   {  return (x<y)?x:y;}
 };
```

# 3.6 类模板

## 类模板的定义格式

```
template<class numtype>
class Compare
 { private:
    numtype x,y;
 public:
    //构造函数
Compare(numtype a,numtype b)
    {  x=a;y=b;}
   numtype max()
    {  return (x>y)?x:y;}
   numtype min()
    {  return (x<y)?x:y;}
  };
```

## 类模板外定义max和min成员函数

```
numtype Compare< numtype > ::max()
{  return (x>y)?x:y;}
numtype Compare< numtype > ::min()
{  return (x<y)?x:y;}
```

## 类模板的使用

```
int main()
{  Compare<int> cmp1(3,7);
      Compare<float> cmp2(45.78,93.6);
      Compare<char> cmp3('a','A');
      return 0;
}
```

# 小结

- 3.1 构造与析构函数
- 3.2 对象数组与对象指针
- 3.3 共用数据的保护
- 3.4 对象动态建立、释放、赋值与复制
- 3.5 静态成员和友元
- 3.6 类模板

```
class Date;
class Time
 {private:
   int hour;
   int minute;
   int sec;
public:
   Time(int,int,int);
   void display(const Date&);
  };
```

```
class Date
 {private:
   int month;
   int day;
   int year;
public:
   Date(int,int,int);
   friend void Time::display(const Date &);
 };
```

**随堂练习任务**

1. 利用友元类实现
2. 不使用友元类，采用成员函数实现

```
void Time::display(const Date &da)
 {cout<<da.month<<"/"<<da.day<<"/"<<da.year<<endl;
  cout<<hour<<":"<<minute<<":"<<sec<<endl;
 }
Date::Date(int m,int d,int y)
 {  month=m;
    day=d;
    year=y;
 }
```

```
Time::Time(int h,int m,int s)
{hour=h;
 minute=m;
 sec=s;     }
```

```
int main()
{  Time t1(10,13,56);
   Date d1(12,25,2004);
    t1.display(d1);
    return 0; }
```

运行时输出：

12/25/2004

10:13:56


# Thank You !

## Q & A