C++

Programming

类和对象 II
Classes and Objects II

2025年3月10日

学而不厌 诲人不倦

Chapter 3 怎样使用类和对象

3.1 构造函数与析构函数

➢ 对象的初始化：构造函数

Constructor

构造函数是一种特殊的成员函数，在程序中不需要写调用语句，在系统建立对象时由系统自觉调用执行，用于为对象分配空间和进行初始化。

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

class Time //定义CPoint类
{
private:
int nHour;//小时
int nMinute;//分钟
int nSecond;//秒
public:
Time();//构造函数
//构造函数重载
Time(int h, int m, int s);
void ShowTime();
};

Box::Box( int h, int w, int len): height ( h),width(w), length( len ){ }
```

构造函数的特点：

(1) 构造函数的名字与它的类名必须相同。
(2) 没有类型，与不返回值
(3) 可以带参数，也可以不带参数。
(4) 构造函数可以重载
(5) 可以写成带参数初始化表的形式

3.1 构造函数与析构函数

➢ 对象的初始化：构造函数

Constructor

构造函数不需要显式地调用，构造函数是在建立对象时由系统自动执行的，且只执行一次。构造函数一般定义为public类型。

```
Time::Time()
{
nHour = 0;
nMinute = 0;
nSecond = 0;
}

Time::Time(int h, int m, int s)
{
nHour = h;
nMinute = m;
nSecond = s;
}
```

```
int main()
{
Time t1;
t1.ShowTime();
Time t2(12,20,15);
t2.ShowTime();
return 0;
}
```

用参数初始化表对数据成员初始化

```
void Time::ShowTime()
{
cout<<"Current Time:
"<<nHour<<":"<<nMinute<<":"<<nSecond<<endl;
}
```

3.1 构造函数与析构函数

➢ 使用默认参数值的构造函数

Constructor

```
#include <iostream>
using namespace std;
class Box
{
public:
Box(int,int,int);
Box(int w=10,int h=10,int len=10);
int volume();
private:
int height;
int width;
int length;
};

// 长方体构造函数
Box::Box(int h,int w,int len)
{
height=h;
width=w;
length=len;
}
```

```
// 计算长方体的体积
int Box::volume()
{
return(height*width*length);
}

int main()
{
Box box1;
cout<<"box1 体积= "<<box1.volume()<<endl;
Box box2(15);
cout<<"box2 体积 "<<box2.volume()<<endl;
Box box3(15,30);
cout<<"box3 体积 "<<box3.volume()<<endl;
Box box4(15,30,20);
cout<<"box4 体积"<<box4.volume()<<endl;
return 0;
}
```

➢ 对象的销毁：析构函数 Destructor

析构函数也是个特殊的成员函数，作用与构造函数相反，当对象的生命周期结束时，系统自动调用析构函数，收回对象占用的内存空间。

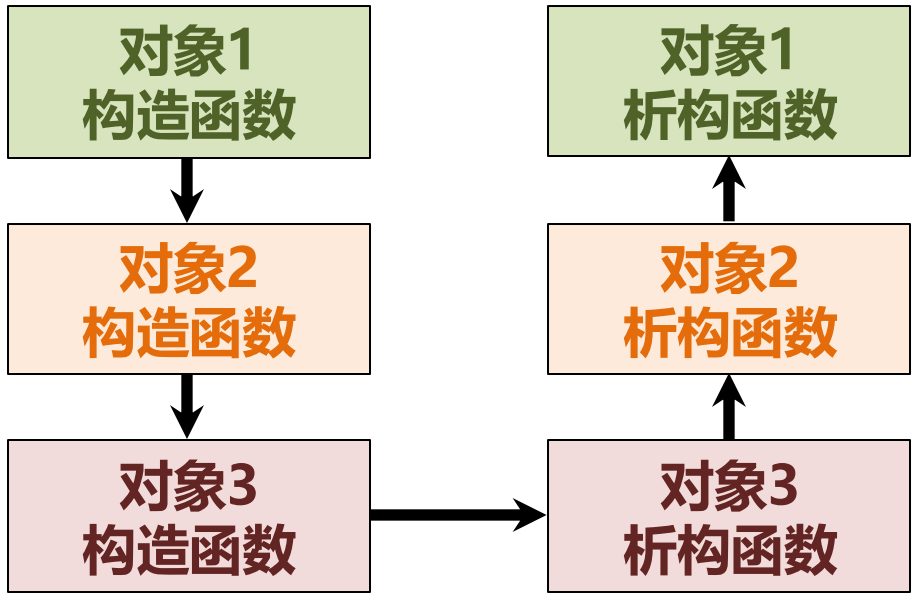
析构函数的特点：

①子函数结束、main函数结束、exit命令执行、delete对象时自动执行析构函数

②先构造的后析构，后构造的先析构

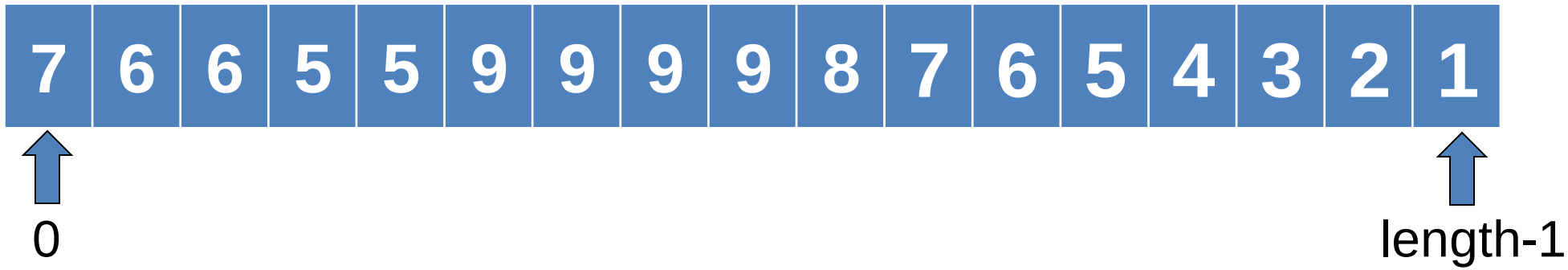
③以~符号开始后跟类名，没有数据类型、返回值、形参

```
~CBigInt();
```



- 例题：定义一个存储大整数的数据类型CBigInt

Data = 12345678999955667



整型数组data来存储大整数的各位数字，
大整数的低位存储在数组低下标处，高位存储在数组高下标处。

通过字符串来给数组初始化

```
string s ="12345678999955667";
```

3.1 构造函数与析构函数

➢ 例题：定义一个存储大整数的数据类型CBigInt

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

class CBigInt //定义CBigInt类
{
private:
int *data;//数据
int length;//BigInt数据长度

public:
CBigInt();//构造函数
CBigInt(string);//带参构造函数
CBigInt(const CBigInt &b);//复制构造函数
~CBigInt();
void Print();
};
```

```
CBigInt::CBigInt()
{
length =0;
data =NULL;
}

//由一个包含数字的字符串构造一个大整数
CBigInt::CBigInt(string str)
{
length = str.length();
data = new int [length];
for (int i=length-1;i>=0;i--)
data[length-i-1] = (int) (str[i]-'0');
}
//析构函数，使用delete释放存储空间
CBigInt::~CBigInt()
{
length = 0;
delete []data;
}
```

3.1 构造函数与析构函数

➢ 例题：定义一个存储大整数的数据类型CBigInt

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

class CBigInt //定义CBigInt类
{
private:
int *data;//数据
int length;//BigInt数据长度

public:
CBigInt();//构造函数
CBigInt(string);//带参构造函数
CBigInt(const CBigInt &b);//复制构造函数
~CBigInt();
void Print();
};
```

```
//复制构造函数,重新申请内存空间将b中数据复制到当前对象
CBigInt::CBigInt(const CBigInt &b)
{
length = b.length;
data = new int [length];
for (int i=0;i<length;i++)
data[i]= b.data[i];
}

int main()
{
string s ="1234567899955667";
CBigInt b1(s);//自动调用有参构造函数
b1.Print();
CBigInt b2=b1;//自动调用复制构造函数
b2.Print();
//cout<<&b1<<" "<<&b2<<endl;
}
```

2023年3月22日

3.2 对象数组与对象指针

- **对象数组：一个对象数组中各个元素都是同类对象**

student std[50]; //一个班级有50个学生，
每个学生具有学号、年龄、成绩等属性

```
#include <iostream>
using namespace std;
class Box
{ public:
Box( int h=10, int w=12, int len=15 ):height(h),
width(w), length(len) { } // 带默认参数值和参数表
int volume();
private:
int height;
int width;
int length;
};
```

```
int Box::volume()
{ return(height*width*length); }

int main()
{
Box a[3]={ Box(10,12,15),
Box(15,18,20),
Box(16,20,26) };
cout<<"a[0]的体积是 "<<a[0] .volume()<<endl;
cout<<"a[1]的体积是 "<<a[1] .volume()<<endl;
cout<<"a[2]的体积是 "<<a[2] .volume()<<endl;
return 0;
}
```

3.2 对象数组与对象指针

- **对象指针：指向对象的指针**

C++中定义对象的指针变量与定义其他的指针变量相似，格式如下：

类名 * 变量名表

C++的对象也可以参加**取地址运算**。

&对象名

```
Time  *pt; // 定义pt是指向Time类对象的指针
Time  t1;  //  定义Time类对象t1
pt = &t1;   // 将对象t1的地址赋予pt
```

程序在此之后就可以用指针变量访问对象的成员。

```
(*pt).hour
pt->hour
(*pt). show_time()
pt->show_time()
```

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

class Time //定义CPoint类
{
private:
int nHour;//小时
int nMinute;//分钟
int nSecond;//秒
public:
Time();//构造函数
//构造函数重载
Time(int h, int m, int s);
void ShowTime();
};
```

3.2 对象数组与对象指针

- **对象指针：指向对象公有数据成员的指针**

  对象由成员组成。对象占据的内存区是各个数据成员占据的内存区的总和。

  指向对象成员的指针分指向数据成员的指针和指向成员函数的指针

  ①定义数据成员的指针变量： 数据类型 * 指针变量名

  ②计算公有数据成员的地址： &对象名.成员名

```
Time t1;
int * p1; //  定义一个指向整型数据的指针变量
p1 = & t1.hour; // 假定hour是公有成员
cout<< *p1 << endl;
```

3.2 对象数组与对象指针

- **对象指针：指向对象成员函数的指针**

  对象由成员组成。对象占据的内存区是各个数据成员占据的内存区的总和。

  指向对象成员的指针分指向数据成员的指针和指向成员函数的指针

①定义指向成员函数的指针变量

数据类型（**类名**::* **变量名**）(**形参表**)；

数据类型是成员函数的类型。

类名是对象所属的类

变量名按标识符取名

形参表：指定成员函数的形参表(形参个数、类型)

②取成员函数的地址

&类名::成员函数名

③给指针变量赋初值

指针变量名= & 类名::成员函数名；

④用指针变量调用成员函数

(对象名.*指针变量名)([实参表]);

3.2 对象数组与对象指针

- 对象指针：指向**对象成员函数**的指针

```
#include <iostream>
using namespace std;
class Time
{
public:
  Time(int,int,int);
  void get_time();
private:
  int hour;
  int minute;
  int sec;
};
```

```
int main()
{
Time t1(10,13,56);
int *p1=&t1.hour; // 定义指向成员的指针p1
cout<<*p1<<endl;
t1.get_time(); // 调用成员函数
Time *p2=&t1; // 定义指向对象t1的指针p2
p2->get_time(); // 用对象指针调用成员函数

void (Time::*p3)(); // 定义指向成员函数的指针
p3=&Time::get_time; // 给成员函数的指针赋值
(t1.*p3)(); // 用指向成员函数的指针调用成员函数
return 0;
}
```

```
void (Time::*p3) = &Time::get_time;
```

3.2 对象数组与对象指针

- **对象指针：this指针**

一个类的成员函数只有一个内存拷贝。类中不论哪个对象调用某个成员函数，调用的都是内存中同一个成员函数代码。

```
int Box :: volume()
{ return ( height * width * length ) ; }
```

C++编译成

```
int Box :: volume( * this )
{   //(*this) 就是this所指的对象，即：调用成员函数的对象
    return (this->height* this->width *  this->length);
    //或者
    return( (*this).height* (*this).width * (*this). length );
}
```

C++ 通过编译程序，在对象调用成员函数时，把对象的地址赋予this 指针，用this 指针指向对象，实现了用同一个成员函数访问不同对象的数据成员。

3.3 共用数据的保护

1. const 常对象

即希望数据在一定范围内共享，又不愿它被随意修改，从技术上可以把数据指定为只读型的。C++提供const手段，将对象、数据、成员函数指定为常量，从而实现了只读要求，达到保护数据的目的。

定义格式：const 类名 对象名(实参表);

或：　　类名 const 对象名(实参表);

const Time t1(10,15,36);

t1.get_time(); // 错误，不能调用

为了访问常对象中的数据成员，要定义常成员函数：

void get_time() const

把对象定义为常对象，对象中的数据成员就是常变量，在定义时必须带实参作为数据成员的初值，在**程序中不允许修改常对象的数据成员值**。

2. const 常数据成员

即希望数据在一定范围内共享，又不愿它被随意修改，从技术上可以把数据指定为只读型的。C++提供const手段，将对象、数据、成员函数指定为常量，从而实现了只读要求，达到保护数据的目的。

格式：　const　类型　数据成员名

将类中的数据成员定义为具有只读的性质。

例：

```
const   int  hour;
Time :: Time( int h)
{    hour = h; ...}   //      错误
```

应该写成：

```
Time :: Time( int h ) : hour (h) {}
```

注意只能通过带参数初始表的构造函数对常数据成员进行初始化。

3. const 常成员函数

任何不修改数据成员的函数，都应该声明为const类型，以提高程序的健壮性。

定义格式：

类型　函数名　(形参表)　const

const 是函数类型的一部分，在**声明函数原型**和**定义函数**时都要用const关键字。

```
float GetDistance() const;
void ShowPoint() const;
```

```
float d = p1.GetDistance();
```

常对象只能通过常成员函数读数据成员，常对象不能调用非const成员函数。

常成员函数不能调用非const成员函数。

```
//输出当前对象坐标
void CPoint::ShowPoint() const {
 cout<<"Point: ["<<x<<","<<y<<"]"<<endl;
}
```

```
//计算当前对象到原点距离
float CPoint::GetDistance() const {
x =5;//const成员函数中无法修改成员变量数值
return sqrt(x*x+y*y);
}
```

3. const 常成员函数

一般成员函数可以访问或修改本类中的非 const数据成员。而常成员函数只能读本类中的数据成员，而不能写它们。

表 3.1

数据成员	非 const 成员函数	const 成员函数
非 const 的数据成员	可以引用,也可以改变值	可以引用,但不可以改变值
const 数据成员	可以引用,但不可以改变值	可以引用,但不可以改变值
const 对象的数据成员	不允许引用和改变值	可以引用,但不可以改变值

4. 指向对象的常指针

如果在定义指向对象的指针时，使用了关键字 const ， 它就是一个常指针，必须在定义时对其初始化。并且在程序运行中不能再修改指针的值。

类名 * const 指针变量名 = 对象地址

例：Time t1(10,12,15)，t2;

Time * const p1 = & t1;

在此后，程序中不能修改p1。

例： Time * const p1 = & t2; **// 错误语句**

5. 指向常对象的指针变量

const 类名 * 指针变量名 = 对象地址

（1）如果一个对象已被声明为常对象，只能用指向常对象的指针变量指向它。

（2）如果定义了一个指向常对象的指针变量，并使它指向一个非const对象，则指向的对象是不能通过该指针变量来改变的。

（3）常用作函数的形参，保护形参指针所指向的对象不被修改。

（4）指针变量本身的值是可以改变的

6. 对象的常引用

如果用引用形参又不想让函数修改实参，可以使用常引用机制。

格式： const 类名 & 形参对象名

常引用作为函数形参，形参与实参引用同一块内存空间，既节省存储，同时又能保证数据不能被随意修改。

```
void CBigInt::Add(const CBigInt &a, const CBigInt &b)
{
}
```

6. 对象的常引用

```
void CBigInt::Add(const CBigInt &a, const CBigInt &b)
{
int flag = 0; //表示进位
int i = 0;
int m= a.length;
int n= b.length;
int len=m>n?m:n;//求长度较大者
data = new int [len+1];//申请空间，最高位可能有进位，因此多申请一位空间。
while (i<m && i<n)//逐位相加，直到一个大整数结束。
{
data[i] = (a.data[i]+b.data[i]+flag)%10;
flag = (a.data[i]+b.data[i]+flag)/10;
i++;
}
//待续
}
```

6. 对象的常引用

```
void CBigInt::Add(const CBigInt &a, const CBigInt &b)
{
//续
for (;i<m;i++)//如果大整数a没有结束，则剩余位数与进位相加
{
data[i] = (a.data[i]+flag)%10;
flag = (a.data[i]+flag)/10;
}
for (;i<n;i++)//如果大整数b没有结束，则剩余位数与进位相加
{
data[i] = (b.data[i]+flag)%10;
flag = (b.data[i]+flag)/10;
}
length = len +flag; //a+b的长度与最后的进位有关。
if(flag)
data[length-1]= flag;
}
```

6. 对象的常引用

```
int main()
{
string s ="1234567899955667";
CBigInt b1(s);//自动调用有参构造函数
b1.Print();
CBigInt b2=b1;//自动调用复制构造函数
b2.Print();
cout<<&b1<<" "<<&b2<<endl;
CBigInt b3;
b3.Add(b1,b2);
b3.Print();
}
```

```
BigInt:1234567899955667
BigInt:1234567899955667
0x16bc6f468   0x16bc6f430
BigInt:2469135799911334
```

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

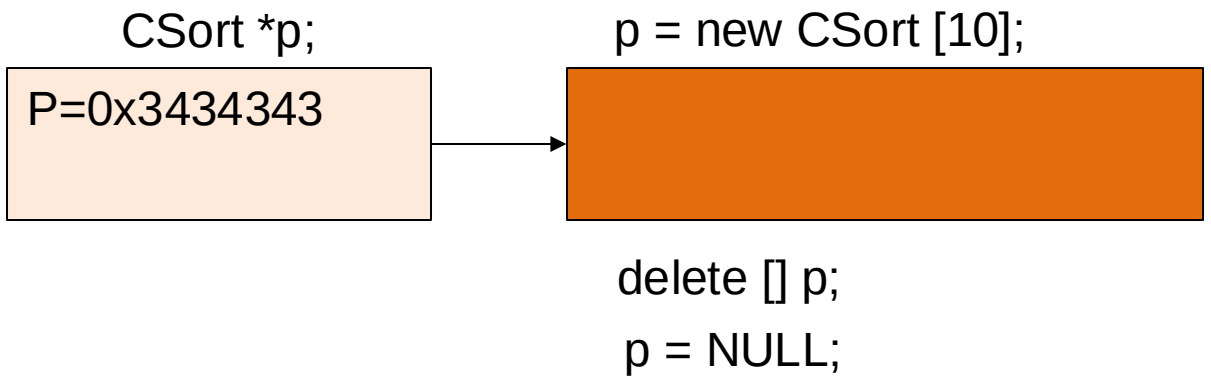
```
#include <iostream>
using namespace std;

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。

2023年4月12日

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. 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