面向对象及C++程序设计

类与对象

苏州大学计算机科学与技术学院
面向对象与C++程序设计课程组

类与对象

- 结构体
  - C的结构体：体现了变量的相关性
  - C++的结构体：扩展了函数
- 类：
  - 比C++的结构体严格
  - private
- 类与对象是抽象与具体的关系
- 类是面向对象程序设计的逻辑基础

c++中的类

- 类
  - 具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分。
  - 可以实现数据的封装、隐藏、继承与派生。
  - 易于编写大型复杂程序，其代码重用性比C中采用函数更高。

类的声明形式

类是一种用户自定义类型，声明形式：

```
class 类名称
{
    public:
        公有成员（外部接口）
    private:
        私有成员
    protected:
        保护型成员
};
```

公有类型成员

在关键字**public**后面声明，它们是类与外部的接口，任何外部函数都可以访问公有类型数据和函数。

私有类型成员

在关键字**private**后面声明，只允许本类中的函数访问，而类外部的任何函数都不能访问。

如果紧跟在类名称的后面声明私有成员，则关键字private可以省略。（即类的默认访问权限是私有）

保护类型

- protected
  - 为继承与派生服务
  - 不使用继承与派生则与private类似

类的成员

```
class  Clock
{
  public:
     void SetTime(int NewH, int NewM,
                                  int NewS);
     void ShowTime（）;
  private:
     int Hour, Minute, Second;
};
```

成员函数

成员数据

```
void Clock :: SetTime(int NewH, int NewM,
                                        int NewS)
{
      Hour=NewH;
      Minute=NewM;
      Second=NewS;
}
void Clock :: ShowTime（ ）
{
      cout<<Hour<<":"<<Minute<<":"<<Second;
}
```

成员数据

- 与一般的变量声明相同，但需要将它放在类的声明体中。
- 无论是**public**、**private**或**protected**，该类中所有函数都可以使用该成员
- 所以比**C**多出一个变量的存储级别

成员函数

- 在类中说明原形，可以在类外给出函数体实现，并在函数名前使用类名加以限定。也可以直接在类中给出函数体，形成内联成员函数。
- 允许声明重载函数和带缺省形参值的函数

内联成员函数

- 为了提高运行时的效率，对于较简单的函数可以声明为内联形式。
- 内联函数体中不要有复杂结构（如循环语句和**switch**语句）。
- 在类中声明内联成员函数的方式：
  - 将函数体放在类的声明中。
  - 使用**inline**关键字。

内联成员函数举例(一)

```
class Point
{
 public:
      void Init(int initX,int initY)
      {
        X=initX;
        Y=initY;
      }
      int GetX ( )  {return X;}
      int GetY ( )  {return Y;}
 private:
      int X,Y;
};
```

内联成员函数举例(二)

```
class Point
{
 public:
     void Init(int initX,int initY);
     int GetX( );
     int GetY( );
 private:
     int X,Y;
};
```

```
inline void Point::
            Init(int initX,int initY)
{
    X=initX;
    Y=initY;
}

inline int Point::GetX( )
{
    return X;
}

inline int Point::GetY( )
{
    return Y;
}
```

对象

- 类的对象是该类的某一特定实体，即类类型的变量。
- 声明形式：

  类名　对象名；
- 例：

  Clock myClock;

类中成员的访问方式

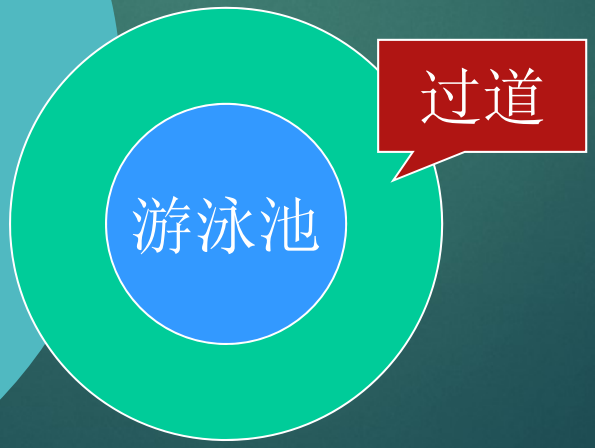
- 类中成员互访
  - 直接使用成员名
- 类外访问
  - 使用“对象名.成员名”方式访问 **public** 属性的成员

例子 类的应用举例

```
#include<iostream.h>
class Clock
{
        ......//类的声明略
}
//......类的实现略
void main(void)
{     Clock  myClock;
       myClock.SetTime(8,30,30);
       myClock.ShowTime（ ）;
}
```

例 类的应用举例

一圆型游泳池如图所示，现在需在其周围建一圆型过道，并在其四周围上栅栏。栅栏价格为35元/米，过道造价为20元/平方米。过道宽度为3米，游泳池半径由键盘输入。要求编程计算并输出过道和栅栏的造价。



```
#include <iostream.h>

const float PI = 3.14159;
const float FencePrice = 35;
const float ConcretePrice = 20;

//声明类Circle 及其数据和方法
class Circle
{
  private:
    float   radius;

  public:
    Circle(float r);  //构造函数

    float Circumference（ ） const; //圆周长
    float Area（ ） const;  //圆面积
};
```

```
// 类的实现
// 构造函数初始化数据成员radius
Circle::Circle(float r)
{radius=r}

// 计算圆的周长
float Circle::Circumference（ ） const
{
    return 2 * PI * radius;
}

// 计算圆的面积
float Circle::Area（ ） const
{
    return PI * radius * radius;
}
```

```
void main （ ）
{
   float radius;
   float FenceCost, ConcreteCost;

   // 提示用户输入半径
   cout<<"Enter the radius of the pool: ";
   cin>>radius;

   // 声明 Circle 对象
   Circle Pool(radius);
   Circle PoolRim(radius + 3);
```

```
// 计算栅栏造价并输出
    FenceCost = PoolRim.Circumference（ ） * FencePrice;
    cout << "Fencing Cost is ￥" << FenceCost << endl;

    // 计算过道造价并输出
    ConcreteCost = (PoolRim.Area（ ） - Pool.Area
    （ ）)*ConcretePrice;
    cout << "Concrete Cost is ￥" << ConcreteCost << endl;
}
```

运行结果

```
Enter the radius of the pool: 10
Fencing Cost is ￥2858.85
Concrete Cost is ￥4335.39
```

指针与对象

- new 与delete

  Ticket *a=new Ticket;

  delete a;

- 通过->访问成员
- 可以将一个对象直接赋值给同类型的对象，此时公有与私有成员数据都将传递过去
- this指针
  - 指向当前对象的指针
  - 程序中被隐藏
  - 可以用于判断两个对象是否相同

友元函数

- 允许在类外访问类中的所有成员
- 用**friend**修饰
- 友员函数并非类的成员函数，它不带有**this**指针，因此通常将对象名称或对象的引用作为友元函数的参数，需要用.访问对象的成员
- 友员函数其实只是一个声明
- 在类中指定友元函数的访问权限无效

友元函数

- 友元函数具有文件作用域
- 也可以将一个类的成员函数作为另一个类的友员函数
- 作用：增加灵活性，使程序员可以在封装和快速性方面做合理选择
- 破坏了封装性

例子 使用友元函数计算两点距离


```
#include <iostream.h>
#include <math.h>
class Point    //Point类声明
{public:   //外部接口
    Point(int xx=0, int yy=0) {X=xx;Y=yy;}
    int GetX（ ） {return X;}
    int GetY（ ） {return Y;}
    friend float fDist(Point &a, Point &b);
private:   //私有数据成员
    int X,Y;
};
```

```
double Distance( Point& a, Point& b)
{
    double dx=a.X-b.X;
    double dy=a.Y-b.Y;
    return sqrt(dx*dx+dy*dy);
}
int main（ ）
{  Point p1(3.0, 5.0), p2(4.0, 6.0);
   double d=Distance(p1, p2);
   cout<<"The distance is "<<d<<endl;
   return 0;
}
```

运算符重载

- 允许程序设计者重新定义已有的运算符，完成特定的操作
- 体现了多态性
- 通过编写运算符重载函数

运算符重载（续）

- 不可以重载的符号
  - .成员运算符
  - *成员指针运算符
  - ：：作用域操作符
  - ？：三目运算符
  - sizeof()
- 不可以改变运算符操作数的数目
- 运算符的优先级不可以改

通过类成员函数重载

- 一般形式

  <类型> operator @(<参数表>)

  {…

  }

  - **<类型>**为该函数返回值类型
  - @为要重载的运算符
  - **operator**与后面的运算符一起构成函数名

使用友元函数重载运算符

- 通常比用成员函数重载多一个参数
- 但是有些运算符不能用友元函数重载
  - =赋值运算符
  - []数组下标运算符
  - ()函数调用运算符
  - new内存分配运算符
  - Delete内存删除运算符

赋值运算符的重载

- 对于类中含有指针数据成员的类，常常需要重载=号运算符
- 只能利用成员函数重载，不能用友元重载
- 重载的赋值运算符不能被继承
- 不能将赋值运算符重载函数声明为虚函数

自增与自减运算符的重载

- 如何区分i++与++I

<<运算符的重载

- 返回引用
- 只能用友元重载

数组下标运算符[]重载

- **C和C++中数组访问元素不做下标越界检查**
- 下标运算符为双目运算符
- **<类型> operator[](<参数>)**
  - 参数通常为整型
- 下标运算符必须利用类的成员函数重载
- 数组越界时如何处理比较好？

构造函数

- 构造函数的作用是在对象被创建时使用特定的值构造对象，或者说将对象初始化为一个特定的状态。
- 在对象创建时由系统自动调用。
- 如果程序中未声明，则系统自动产生出一个缺省形式的构造函数
- 允许为内联函数、重载函数、带缺省形参值的函数

构造函数举例

```
class Clock
{
public:
    Clock (int NewH, int NewM, int NewS);//构造函数
    void SetTime(int NewH, int NewM, int NewS);
    void ShowTime（ ）;
private:
    int Hour,Minute,Second;
};
```

构造函数的实现：

```
Clock::Clock(int NewH, int NewM, int NewS)
{
    Hour=H;
    Minute=M;
    Second=S;
}
```

建立对象时构造函数的作用：

```
void main（ ）
{
    Clock  c (0,0,0); //隐含调用构造函数，将初始值作为实参。
    c.ShowTime（ ）;
}
```

析构函数

- 完成对象被删除前的一些清理工作。
- 在对象的生存期结束的时刻系统自动调用它，然后再释放此对象所属的空间。
- 如果程序中未声明析构函数，编译器将自动产生一个缺省的析构函数。

析构函数

- 函数名与类相同，前面加~
- 不允许指定返回值
- 不可以带参数
- 不能重载
- 系统自动调用

构造函数和析构函数举例

```
#include<iostream.h>
class Point
{
   public:
      Point(int xx,int yy);
      ~Point（ ）;
      //...其它函数原形
   private:
      int X,int Y;
};
```

```
Point::Point(int xx,int yy)
{      X=xx;   Y=yy;
}
Point::~Point（ ）
{
}
//...其它函数的实现略
```

复制构造函数

- 一种特殊的有参数的构造函数
- 其形参为本类的对象引用
- 有效解决类中含有指针的问题

```
class 类名
{  public :
        类名（形参）；  //构造函数
        类名（类名 &对象名）；//复制构造函数
            ...
};
类名:: 类名（类名 &对象名）//复制构造函数的实现
{    函数体    }
```

复制构造函数

- 如果程序员没有为类声明拷贝初始化构造函数，则编译器自己生成一个拷贝构造函数。
- 这个构造函数执行的功能是：用作为初始值的对象的每个数据成员的值，初始化将要建立的对象的对应数据成员。

常成员函数

- 只对对象执行读操作的函数
- **const** 写在函数定义的后面
- 一旦该函数中修改数据，则编译器报错
- 对象的常引用只能调用常成员函数，不能调用非常成员函数（为什么？）

静态数据成员

- 静态数据成员
  - 用关键字**static**声明
  - 该类的所有对象维护该成员的同一个拷贝
  - 必须在类外定义和初始化，用**(::)**来指明所属的类。
  - 在编译期分配存储空间，其它成员数据在运行期分配存储空间
  - 可以通过类名限定直接引用**public**静态成员数据，无需对象

例子 具有静态数据成员的 Point类


```
#include <iostream.h>
class Point
{public:
    Point(int xx=0, int yy=0) {X=xx; Y=yy; countP++; }
   Point(Point &p);
    int GetX（ ） {return X;}
    int GetY（ ） {return Y;}
    void GetC（ ） {cout<<" Object id="<<countP<<endl;}
private:
    int X,Y;
    static int countP;
};
```

```
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（ ）;
}
```

静态成员函数

- 静态成员函数可以通过类名限定直接调用或通过对象调用
- 静态成员函数可以直接使用类的静态成员数据与静态成员函数
- 静态成员函数不可以直接使用类的非静态成员数据与非静态成员函数
- 静态成员函数没有this指针

静态成员函数举例

```
#include<iostream.h>
class Application
{ public:
    static void f（）;
    static void g（）;
  private:
    static int global;
};
int Application::global=0;
void Application::f（）
{  global=5;}
void Application::g（）
{  cout<<global<<endl;}
```

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

静态成员函数举例

```
class A
{
    public:
        static void f(A a);
    private:
        int x;
};
void A::f(A a)
{
    cout<<x; //对x的引用是错误的
    cout<<a.x;  //正确
}
```

例 具有静态数据、函数成员的 Point类

```
#include <iostream.h>
class Point      //Point类声明
{public:    //外部接口
    Point(int xx=0, int yy=0) {X=xx;Y=yy;countP++;}
    Point(Point &p);    //拷贝构造函数
    int GetX（ ） {return X;}
    int GetY（ ） {return Y;}
    static void GetC（ ）
       {cout<<" Object id="<<countP<<endl;}
private:    //私有数据成员
    int X,Y;
    static int countP;
}
```

```
Point::Point(Point &p)
{   X=p.X;
    Y=p.Y;
    countP++;
}
int Point::countP=0;
void main（ ）  //主函数实现
{   Point A(4,5);  //声明对象A
    cout<<"Point A,"<<A.GetX（ ）<<","<<A.GetY（ ）;
    A.GetC（ ）; //输出对象号，对象名引用
    Point B(A);    //声明对象B
    cout<<"Point B,"<<B.GetX（ ）<<","<<B.GetY（ ）;
    Point::GetC（ ）;    //输出对象号，类名引用
}
```