第一课 绪论

确定一个事物是一个类的步骤：

- 判断它是否有一个以上的实例，若有则可能是一个类
- 判断类的实例中有否绝对的不同点，若无则是一个类

> 注意：不能把一组函数组合在一起构成类！即类不是函数的集合。

抽象

封装

继承

- 代码重用
- 分为单继承和多继承

多态

- 不同对象收到相同消息是产生的多种不同的行为方式
- 分为编译时的多态性（重载）和运行时的多态性（虚函数）

续行符\

例1.5 操作符dec、hex和oct的使用：

```
#include<iostream.h>
void  main()
{
    int x=25;
    cout<<hex<<x<<' '<<dec<<x<<' '<<oct<<x<<'\n';
}
```

- 输出结果为：19 25 31

在C++中，结构名、联合名、枚举名都是类型名。在定义变量时，不必在结构名、联合名或枚举名前冠以struct、union或enum。例如：

```
# include<iostream>

using std::cout;
using std::cin;
using std::endl;

enum boole{FALSE = 1,TRUE = 0};
struct string
{
    char *string;
    int length;
};
union number
```

```
{
    int i;
    float f;
    double d;
};

int main(void)
{
    boole done;//在c中为enum boole done;
    string str;//
    number x;//
    done = FALSE;
    x.i = 2;
    x.f = 2.1;
    cout << sizeof(x) << endl << x.d << endl << x.f << endl;
    //cout << done;
}
```

c++中函数原型可以不包含参数的名字，只包含类型

```
long  Area(int ,int);
```

> 但在函数定义时必须给出参数名

未指定返回类型c++默认是int、未标注参数默认为空

const修饰符的用法：

```
const int a = 1;
```

- c中习惯用define来定义常量，c++中用const
- 用const定义的常量是类型化的，它有地址，可以用指针指向这个地址，但不能修改它
- 若省略数据类型，默认int
- 与指针一起用
    - 指向常量的指针

      是指：一个指向常量的指针变量

      ```
      const char* pc=“abcd”;//声明指向常量的指针
      pc[3]=‘x’;//出错
      pc=“efgh”;//合法
      ```

    - 常指针

      是指：把指针本身，而不是它指向的对象声明为常量

      ```
      char* const pc=“abcd”;//常指针
      pc[3]=‘x’;//合法
      pc=“efgh”;//出错
      ```

- 指向常量的常指针

  这个指针本身不能改变，它所指向的值也不能改变

```
const char* const pc=“abcd”;//指向常量的常指针
pc[3]=‘x’;//出错
pc=“efgh”;//出错
```

- 函数参数也可以用const说明

```
int  i_Max(const int* ptr);
```

void型指针

- 表示不确定的类型
- 任何类型指针值都可以幅给void类型的指针变量
- 但在处理时，必须进行强制类型转换

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

int main(void)
{
    int a = 1;
    void* p;
    p = &a;
    cout << *(int*)p;
}
```

内联函数

- 内联函数和c中的宏定义#define作用类似，但消除了#define的不安全性
- 内联函数一般不能有循环语句和开关语句
- 类结构中所有在类说明体内定义的函数都是内联函数
- 较短的函数才定义为内联函数

带有缺省参数值的函数

- 在函数原型中所有取缺省值的参数必须出现在不取缺省值的参数的右边
- 在函数调用时，若某个参数省略，其后参数皆应省略而采用缺省值

函数重载

- 统一作用域中，函数名相同，参数类型、参数个数不同
- 函数重载优先级
  - 寻找一个严格的匹配，即：调用与实参的数据类型、个数完全相同的那个函数。
  - 通过内部转换寻求一个匹配，即：通过（1）的方法没有找到相匹配的函数时，则由C++系统对实参的数据类型进行内部转换，转换完毕后，如果有匹配的函数存在，则执行该函数。
  - 通过用户定义的转换寻求一个匹配，若能查出有唯一的一组转换，就调用那个函数。即：在函数调用处由程序员对实参进行强制类型转换，以此作为查找相匹配的函数的依据。

```cpp
#include <iostream>
using namespace std;
void print(double d)
{ cout<<"this is a double "<<d<<"\n"; }
void print(int i)
{ cout<<"this is an integer "<<i<<"\n"; }
void main()
{
    int x=1,z=10;
    float y=1.0;
    char  c='a';
    print(x);//按规则（1）自动匹配函数void print(int i)
    print(y);//按规则（2）通过内部转换匹配函数   void print(double i)
                //因为系统能自动将float型转换成double型
    print(c); //按规则（2）通过内部转换匹配函数 void  print(int i)
              //因为系统能自动将char型转换成int型
    print(double(z));     //按规则（3）匹配void print(double i)
                          //因为程序中将实参z强制转换为double型。
}
```

函数模板

```cpp
template <class T>
template <typename T>
```

- class和typename可以通用
- 在编译时将函数名和其对应的模板函数匹配，将实参类型替代了函数模板中的虚拟类型
- 函数模板比重载方便，但只适用于参数个数相同而类型不同，且函数体相同的情况

作用域标识符

```cpp
#include <iostream.h>
int avar=10;
main()
{
    int avar;
    avar=25;
    cout<<"local avar ="<<avar<<endl;
    cout<<"global avar="<<::avar<<endl;
    return 0;
}
```

- ::前面什么都不加，代表全局变量

无名联合

```cpp
union
{
    int i;
    float f;
}
```

- 这是一个没有名字的联合，其中的变量可以直接调用

强制类型转换

```
int i=10;
float  x=(float)i;
float  x=float(i);
```

- 这两种写法c++都接受，推荐后者

动态内存分配

- 作为对c语言中malloc和free的替换

```
int *a,*b;
a=new int;
b=new int(10);//可以直接赋初值
delete a;
delete b;
```

- new和delete可以自动计算需要分配和释放类型的长度，省去了sizeof的步骤
- new后自动返回需分配类型的指针无需强制类型转换
- new能初始化所分配的类型变量
- new和delete都可以被重载，允许建立自定义的内存管理法
- 如果new分配失败，将返回NULL
- 用new分配数组空间

```
int  *pi=new  int[10];//类型后面加上方括号
int *pi=new int[2][3][4];//多维数组必须提供所有维的大小
int  i=3;
int  *pi=new  int[ i ][2][3];//第一维可以是任何合法的表达式
```

- delete释放数组空间（试了以下好像不行？？）

```
delete  []pi;//无需指出释放空间大小
delete  [10]pi;//老版本要求标出要释放多少个元素
```

引用

```
int &ra=a;   //定义引用ra,它是变量a的引用，即别名
```

- 除了用作函数的参数或返回类型外，在声明时必须立即初始化，不能声明完后再幅值
- 引用不能重新赋值，不能再把它作为别的变量的别名
- 可以声明一个指针变量的引用（即声明一个指针变量的别名）

```cpp
#include <iostream>
using namespace std;
int main(void)
{
    int *a;       //定义指针变量a
    int *&p=a;    //定义引用p，初始化为指针变量a，所以p是a的引用（别名）
    int b=10;
    p=&b;         //等价于a=&b，即将变量b的地址赋给a。
    cout<<*a<<endl;     //输出变量b的值
    cout<<*p<<endl;     //等价于cout<<*a;
    cout<<a<<endl; //输出a和p的地址
    cout<<p<<endl;
}
```

- 不能声明引用的引用，也不能定义引用的指针
- 不饿能建立数组的引用
- 不能建立空指针的引用
- 不能建立void的引用
- 引用仅在声明时带&，之后就像普通变量一样
- 用引用作为函数的参数

```cpp
#include <iostream>
using namespace std;
void swap(int& m,int& n)
{
    int temp;
    temp=m;
    m=n;
    n=temp;
}
main()
{
    int a=5,b=10;
    cout<<"a="<<a<<" b="<<b<<endl;
    swap(a,b);                              //运行结果:
    cout<<"a="<<a<<" b="<<b<<endl;          //a=5  b=10
    return 0;                               //a=10 b=5
}
```

- 引用作为函数的返回值

  例1：

```cpp
#include <iostream>
using namespace std;
int a[]={1, 3, 5, 7, 9};
int& index(int); // 声明返回引用的函数
void main()
{
    cout<<index(2)<<endl;
```

```
    index(2)=25;    // 将a[2]重新赋值为25
    cout<<index(2)<<endl;
}
int& index(int i)
{
    return a[i];
}
```

例2:

```
#include<iostream>
using namespace std;
int A[10];
int& array(int i);
main()
{
    int i,number;
    A[0]=0;
    A[1]=1;
    cin>>number;
    for (i=2;i<number;i++)
    {
        array(i)=array(i-2)+array(i-1);
        cout<<"array("<<i<<")="<<array(i)<<endl;
    }
}

int& array(int i)
{
    return A[i];
}
```

在定义返回引用的函数时，注意不要返回函数内的局部变量的引用

- 使用引用传递参数，可以在被调用函数中改变主调用函数中目标变量的值，实际上就相当于返回多个值
- 用引用返回一个函数值的最大好处是，在内存中不产生被返回值的副本。

  也就是说，上述程序段，如果不用引用，返回A[i]，会新开一个临时空间用于返回A[i]；而用了引用就相当于返回了A[i]本身。这也就是其可以作为左值的原因。
- 一个返回引用的函数值可作为赋值表达式的左值
- const限定引用

  用这种方式声明的引用，不能通过对引用对目标变量的值进行修改，增加安全性。
- 引用是直接对变量进行操作，指针是间接操作

第二课 类和对象1

类的构成

一般一个类的数据成员为私有成员，成员函数为公有成员

private处于类体中第一部分时，可以省略

不可以在类声明中给数据成员赋值

成员函数的声明

内联函数

- 指定义在类体内的数据成员（用内联函数的代码来代替），其效率与宏定义相同：
- 隐式声明：定义在类内
- 显式声明：定义在类外

外联函数

- 指说明在类体内，定义在类体外的函数
- 在函数头前面加上关键字inline就可以变成内联函数

注意

- 使用inline说明内联函数时，必须使函数体和inline说明结合在一起，否则会当作普通函数处理
- 短的成员函数定义为内联函数，长的最好作为一般函数

类对象的定义和使用

定义方法

- 在声明类的同时定义对象

```
class  Coord
{
public:
    void setCoord(int,int);
    int getx();
    int gety();
private:
    int  x,y;
}op1,op2;
```

- 在声明类之后在使用时再定义对象

注意：

- 在声明类的同时定义对象是一种全局对象
- 声明类只是声明了一种类型，不存储具体值；只有定义了对象后才分配空间

> 对象名.成员名是缩写
>
> 实际上表达的是对象名.类名::成员名

指向对象的指针在访问对象成员时，要用->操作符

对象的拷贝

- 两个同类型的对象可以进行赋值，将一个对象赋值给另一个对象时，所有数据成员都会逐位拷贝
- 两个对象的类型必须相同，否则会出错

- 拷贝仅仅是数据成员相同，两个对象任然分离
- 当类中存在指针时，用缺省赋值运算符进行对象赋值可能会出错

构造函数和析构函数

类对象的初始化

- 调用对yi外接口实现
- 应用构造函数实现

构造函数

- 在建立对象时自动执行
- 两个特殊性
  - 必须与类同名
  - 不返回任何值
- 其他特殊性质
  - 可以写在类体内也可以写在类体外
  - 可以重载！！！！
  - 虽然被声明为公有函数，但不能被显式地调用
- 没有定义构造函数，系统会自动生成一个默认形式的构造函数。
- new ClassA(参数，参数)可以直接初始化

成员函数初始化表的使用

```
#include<iostream>
using namespace std;
class A
{
public:
    A(int x1):x(x1),rx(x),pi(3.14){} // rx(x)相当于rx=x ,
                                     // pi(3.14)相当于pi=3.14
    void print()
    {
        cout<<"x="<<x<<"  "<<"rx="<<rx<<"  "<<"pi="<<pi;
    }
private:
    int x; int& rx; const float pi;
};
main()
{
    A a(10);
    a.print();
    return 0;
}
```

- 如果需要将数据成员存放在堆中或数组中，则应在构造函数中使用赋值语句，即使构造函数有成员初始化表也应如此
- 类成员是按照在类里被声明的顺序初始化的，与初始化表中列出的顺序无关

```cpp
#include<iostream>
using namespace std;
class D
{
public:
    D(int i):mem2(i),mem1(mem2+1)
    {
        cout<<"mem1: "<<mem1<<endl;//mem1=-858993459
        cout<<"mem2: "<<mem2<<endl;//mem2=15
    }
private:
    int mem1;
    int mem2;
};
void main()
{
    D d(15);
}
```

构造函数的重载

拷贝构造函数

- 只有一个参数，是本类对象的引用
- 通过等于号复制对象时，系统自动调用拷贝构造函数
- 拷贝构造函数与原来的构造函数实现了重载

```cpp
class  Coord{
    int x,y;
public:
    Coord(int a, int b)              // 构造函数
    {
        x=a;
        y=b;
        cout<<"Using normal constructor\n";
    }
    Coord(const Coord& p)           // 拷贝构造函数
    {
        x=2*p.x
        y=2*p.y;
        cout<<"Using copy constructor\n";
    }
};

main()
{
    Coord p1(30,40);    // 定义对象p1,调用了普通的构造函数
    Coord p2(p1);       // 以“代入”法调用拷贝构造函数，用对象p1初始化对象p2
    p2 = p1;             //赋值法调用拷贝构造函数
    p1.print();  p2.print();
    return 0;
}
```

- 调用拷贝构造函数的三种情况
  - 代入法/赋值法调用一个对象**初始化**另一个对象

    > 只有初始化的时候才会调用自定义的拷贝构造函数！！！！！！！！！！！！！！！

  - 调用函数，形参和实参结合时

```
fun1(Coord p)       // 函数的形参是类的对象
{
    p.print();
}
main()
{
    Coord p1(10,20);
    fun1(p1);           // 当调用函数,进行形参和实参结合时,调用拷贝构造函数
    return 0;
}
```

  - 返回值是对象，函数执行完成，返回调用者时

```
Coord fun2()
{
    Coord p1(10,30);
    return p1;
}// 函数的返回值是对象

main()
{
    Coord p2;
    P2=fun2();// 函数执行完成,返回调用者时,调用拷贝构造函数

    return 0;
}
```

- 浅拷贝

  由缺省的拷贝构造函数所实现的数据逐一赋值，若类中含有指针数据则会产生错误

- 深拷贝

  显示地定义一个自己的拷贝构造函数，使之不但拷贝数据成员，而且为对象1和对象2分配各自的内存空间

析构函数

- 三个特点
  - 名字与构造函数相同，但前面需要加上波浪号
  - 没有参数，没有返回值，不能重载
  - 默认析构函数，默认析构函数只能释放对象的数据成员所占用的空间，但不包括堆内存空间
- 调用析构函数的两种情况
  - 对象被定义在一个函数体内，当函数结束时，析构函数被自动调用
  - 若对象时使用new运算符动态创建的，在使用delete释放时自动调用析构函数

调用构造函数和析构函数的顺序

- 一般顺序

  先调用的构造函数，其对应的析构函数最后被调用
- 全局对象

  在所有函数开始前构造

  在main函数结束或调用exit函数时析构
- auto局部对象

  调用函数时构造

  结束时析构
- static局部对象

  只在第一次构造

  main函数结束的时候析构

第三课 类和对象2

自引用指针this

this指针的类型就是成员函数所属的类的类型

当调用成员函数时，它被初始化为被调用函数所在类的对象的地址

通常情况下this是隐含地存在的，也可以显式地表示出来

this的应用

- 参数与数据成员同名

```
class Circle
{
private:
    double radius;
public:
    Circle(double radius)  // 参数与数据成员同名时
    {
         this->radius = radius;  // 去掉 this 如何理解?
    }                                          // P272.例9.5中形式参数为何取 nam ?
    double get_area()
    {
        return 3.14 * radius * radius;
    }
};
int main()
{
    Circle c = Circle(1);
    double a = c.get_area();
    cout << a << endl;
    return 0;
}
```

注意

- this指针是一个const指针，不能在程序中修改或给它赋值
- this指针作用域仅在一个对象的内部
- 几个特点
  - 隐式存在
  - 自动实现
  - *this

对象数组与对象指针

指每一数组元素都是对象的数组

构造函数的注意点

- 当各个元素的初值要求为相同的值时，应当在类中定义出不带参数的构造函数或带缺省参数值的构造函数
- 当各个元素初值不同时需要定义带形参的构造函数
- 定义对象数组可以通过初始化表进行赋值

  eg1：exam ob1[4]={11,22,33,44};

  eg2：exam ob3[4]={exam(11),exam(22),exam(33),exam(44)};（构造函数有多个参数时只能用这个）

```
#include<iostream>
using namespace std;
class exam
{
public:
    exam()
        { x=123;}
    exam(int n)
        { x=n;}
    int get_x()
        { return x; }
private:
    int x;
};
main()
{
    exam ob1[4]={11,22,33,44};
    exam ob2[4]={55,66};
    exam ob3[4]={exam(11),exam(22),exam(33),exam(44)};
    exam ob4[4]={exam(55),exam(66)};
    ob4[2]=exam(77);
    ob4[3]=exam(88);
    int i;
    for (i=0;i<4;i++)
        cout<<ob1[i].get_x()<<' ';
    cout<<endl;
    for (i=0;i<4;i++)
        cout<<ob2[i].get_x()<<' ';
    cout<<endl;
```

```
    for (i=0;i<4;i++)
        cout<<ob3[i].get_x()<<' ';
        cout<<endl;
        for (i=0;i<4;i++)
            cout<<ob4[i].get_x()<<' ';
    cout<<endl;
    return 0;
}
```

对象指针

- 类名* 对象指针名

指向类的成员的指针

- 指向成员的指针只能访问公有数据成员和成员函数
- 先声明，再赋值，然后访问
  - 声明：类型说明符 类名::*数据成员指针名;
  - 赋值：数据成员指针名 = &类名::数据成员名
  - 使用：对象名. *数据成员指针名

    对象指针名-> *数据成员指针名

```
#include<iostream>
using namespace std;
class A
{
public:
    A(int i)    {  z=i;  }
    int z;
};
int main(void)
{
    A ob(5);
    A *pc1;          // 声明一个对象指针pc1
    pc1 = &ob;           // 给对象指针pc1赋值
    int A::*pc2;     // 声明一个数据成员指针pc2  // ①
    pc2 = &A::z;     // 给数据成员指针pc2赋值     // ②
    cout<<ob.*pc2<<endl;         // 用成员指针pc2访问数据成员z
    cout<<pc1->*pc2<<endl;    // 用成员指针pc2访问数据成员z
    cout<<ob.z<<endl;              // 使用对象名访问数据成员z
}
```

指向成员函数的指针

- 声明：类型说明符 (类名::*指针名)(参数表);
- 赋值：成员函数指针名 = 类名::成员函数名;
- 使用：(对象名.*成员函数指针名)(参数表)；

  (对象指针名 -> *成员函数指针名)(参数表)；

```
#include<iostream>
```

```
using namespace std;
class Coord {
public:
      Coord(int a=0,int b=0)  { x=a; y=b; }
     int getx()  { return x; }
     int gety()  { return y; }
private:
    int x,y;
};
int main(void)
{
    Coord op(5,6);
    Coord *pc1 = &op;
    int (Coord::*pc_getx)();
    pc_getx = Coord::getx;//函数指针这里不用加取地址符号
    cout<<pc1->getx()<<endl;
    cout<<op.getx()<<endl;
    cout<<(op.*pc_getx)()<<endl;
    cout<<(pc1->*pc_getx)()<<endl;
}
```

向函数传递对象

- 使用方法和向函数传递普通数据类似

使用对象指针作为函数的参数，提高运行效率

使用对象引用作为函数参数，具有用对象指针的有点，且更简单直接

静态成员

静态数据成员

- 用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

```
#include<iostream>
#include<cstring>
using namespace std;
class Student
{
public:
    Student(char *name1,char *stu_no1,float score1);
    ~Student();
    void show();                 // 输出姓名、学号和成绩
    void show_count_sum_ave();   // 输出学生人数

private:
    char *name;                      // 学生姓名
    char *stu_no;                    // 学生学号
    float score;                     // 学生成绩
```

```
    static int count;
    static float sum;
    static float ave;
};

Student::Student(char *name1,char *stu_no1,float score1)
{
    name=new char[strlen(name1)+1];
    strcpy(name,name1);
    stu_no=new char[strlen(stu_no1)+1];
    strcpy(stu_no,stu_no1);
    score=score1;
    ++count;                  // 累加学生人数
    sum = sum+score;            // 累加总成绩
    ave = sum/count;            // 计算平均成绩
}
Student::~Student()
{
    delete []name;
    delete []stu_no;
    --count;
    sum = sum-score;
}

void Student::show()
{   cout<<"\n name: "<<name;
     cout<<"\n stu_no: "<<stu_no;
     cout<<"\n score: "<<score; }
void Student::show_count_sum_ave()
{
     cout<<"\n count: "<<count;   // 输出静态数据成员count
     cout<<"\n sum: "<<sum;       // 输出静态数据成员sum
     cout<<"\n ave: "<<ave;       // 输出静态数据成员ave
}
int Student::count=0;             // 静态数员count初始化
float Student::sum=0.0;           // 静态数员sum初始化
float Student::ave=0.0;           // 静态数员ave初始化

void main()
{
    Student stu1("Liming","990201",90);
    stu1.show();
    stu1.show_count_sum_ave();
    Student stu2("Zhanghao","990202",85);
    stu2.show();
    stu2.show_count_sum_ave();
}
```

- 定义方式

```
static 数据类型 数据成员名
```

- 初始化

  数据类型 类名::静态数据成员 = 值

- 访问方式
  - 类名::静态数据成员
  - 对象名.静态数据成员
  - 对象指针->静态数据成员
- 私有静态成员不能被类外函数访问，也不能用对象进行访问
- 使用静态数据成员可以不必使用全局变量

静态成员函数

- 类外代码可以使用类名和作用域操作符来调用公有静态成员函数
- 静态成员函数只能引用属于改类的静态数据成员或静态成员函数，访问非静态数据成员。必须通过对象名访问
- 使用
  - 声明时加static
  - 类外定义时不加static
- 注意
  - 可以定义成内嵌的也可以在类外定义，在类外定义时不加static
  - 用于维护静态数据
  - 私有静态成员函数不能被类外函数和对象访问
  - ！！！！！可以在建立任何对象之前处理静态数据成员！！！
  - 静态成员函数中没有this指针，所以静态成员函数不能访问非静态数据陈冠，若需要则通过传入对象引用访问

用普通指针访问静态成员

友元

友元既可以是不属于任何一类的一般函数，也可以是另一个类的成员函数，还可以是一整个类

友元函数的声明

- 位置：当前类体中
- 格式：函数名前面加friend

友元函数的定义

- 类体外：同一般函数（不加类名::）
- 类体内：函数名前面加friend

> 友元函数没有this指针，它是通过入口参数（该类对象）来访问对象成员的

引入友元的原因

- 友元提供了不同类的成员函数之间，类的成员函数与一般函数之间进行数据共享的机制

注意：

- 友元函数可以是多个类的
- 友元函数不仅可以是一般函数还可以是另一个类的成员函数

```cpp
#include <iostream>
using namespace std;
class Date;                    //对Date类的提前引用声明
class Time                     //定义Time类
{
public:
    Time(int,int,int);
    void display(Date &);     //display是成员函数，形参是Date类对象的引用
    private:
    int hour;
    int minute;
    int sec;
};
class Date           //声明Date类
{
public:
    Date(int,int,int);
    friend void Time::display(Date &); //友元函数的声明
private:
    int month;
    int day;
    int year;
};
Time::Time(int h,int m,int s)    //类Time的构造函数
{
    hour=h;
    minute=m;
    sec=s;
}
void Time::display(Date &d)
{
    cout<<d.month<<"/"<<d.day<<"/"<<d.year<<endl;
    cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
Date::Date(int m,in  d,int y)           //类Date的构造函数
{
    month=m;
    day=d;
    year=y;
}
int main( )
{
    Time t1(10,13,56);              //定义Time类对象t1
    Date d1(12,25,2004);            //定义Date类对象d1
    t1.display(d1);                //调用t1中的display函数，实参是Date类对象d1
    return 0;
}
```

- 需要类的提前引用需要声明！！！

友元类

- 将B类声明为A类的友元类，则B类中所有函数都是A类的友元函数，可以访问A类中所有的成员

```
class  Y
{
    //…
};
class X
{
    //…
    friend  Y; // 声明类Y为类X的友元类
    //…
};
//类Y的所有成员函数都是类X的友元函数
//Y可以访问X
```

- 友元关系是单向的不是双向的

  Y类是X类的友元，X类不一定是Y类的友元

- 友元关系不能传递

  X类是Y类的友元，Y类是Z类的友元，但X类不一定是Z类的友元

对象成员

如果一个类的对象是另一个类的数据成员，称这样的数据成员为对象成员

```
class A
{
    //...
};
class B
{
    A a;       //类A的对象a为类B的对象成员
    public:
    //…
};
```

- B类的构造函数如何定义?

```
X::X(形参表0):对象成员名1(形参表1),…,
                对象成员名i(形参表i) ,…,对象成员名n(形参表n)
{
   // …构造函数体
}
```

法1:

```
//具体例子
#include<iostream>
using namespace std;
class A{
public:
    A(int x1, float y1)
```

```
    { x=x1; y=y1; }
    void show()
    { cout<<"\n x="<<x<<" y="<<y; }
private:
    int x;
    float y;
};
class B{
public:
    B(int x1,float y1,int z1):a(x1,y1)
    { z=z1; }
    void show()
    {
        a.show();
        cout<<" z="<<z;
     }
private:
    A a;
    int z;
};
main()
{
    B b(11,22,33);
    b.show();
    return 0;
}
```

法2:

```
B(A a1, int z1):a(a1)
 { z=z1; }

main()
 { A a(11,22);
   B b(a,33);
   b.show();
   return 0;
}
```

常类型

一般格式

- const 类型名 对象名(构造实参表)
- 类型名 const 对象名(构造实参列表)

限定作用

- 定义为const的对象的所有数据成员的值都不能被修改
- 凡出现非const的成员函数，都将出现编译错误

mutable成员

- 对数据成员声明为mutable时，即使是const对象，任然可以修改该数据成员值
- 在定义时，最前面加上mutable即可

常对象成员

- 常数据成员
- 常成员函数
  - const加载函数名和括号之后
  - 只能引用本类中的数据成员而不能修改它们(mutable除外)
  - 可以用常引用作为形参
- 常成员的使用
  - 若有些数据成员可以引用而有些不能
  - 若所有都不允许被改变
  - 若已经定义了一个const对象，则只能调用const成员函数!

指向常对象的指针变量

- const 类型 *指针变量名
- 指向const的指针可以指向非const，但不能通过指针来修改它的值
- 如果函数的形参是指向非const型变量的指针，实参只能用指向非const变量的指针，而不能用指向const变量的指针，这样，在执行函数的过程中可以改变形参指针变量所指向的变量(也就是实参指针所指向的变量)的值。
- 如果函数形参是指向const型变量的指针，允许实参是指向const变量的指针，或指向非const变量的指针。

> 基本规则：希望在调用函数时对象的值不被修改，就应当把形参定义为指向常对象的指针变量，同时用对象的地址作实参(对象可以是const或非const型)。如果要求该对象不仅在调用函数过程中不被改变，而且要求它在程序执行过程中都不改变，则应把它定义为const型 、

对象的常引用

- c++经常使用常指针和常引用作为函数的参数，这样既能保证数据安全，使数据不能被随意修改，在调用函数时又不必建立实参的拷贝，提高效率

非常对象和常对象的对比

```
#include<iostream.h>
class Sample
{
public:
    int m;
    Sample(int i,int j){  m=i; n=j;}
    void setvalue(int i){  n=i; }
    void disply(){ cout<<"m="<<m<<endl;
    cout<<"n="<<n<<endl;
}
private:
    int n;
};
void main()
{
    Sample a(10,20); //若为:const Sample a(10,20);
```

```
    a.setvalue(40);//不能修改常对象的数据成员
    a.m=30; //不能修改常对象的数据成员
    a.disply();//常对象不能调用普通成员函数
}
```

const型数据的小结

Time const t = Time(1,2,3);
const Time t = Time(1,2,3);
const int a = 10;
int const a = 10;
	t是常对象，其成员值在任何情况下都不能被改变。
	a 是常变量，其值不能被改变

void Time::fun() const;	fun 是Time类的常成员函数，可以调用该函数，但不能修改本类中的数据成员（非mutable）
Time * const pt; int * const pa;	pt 是指向Time对象的常指针， pa 是指向整数的常指针。指针值不能改变。
const Time *pt; const int *pa;	pt是指向Time类常对象的指针， pa是指向常整数的指针，不能通过指针来改变指向的对象（值）

多文件组成

好像没啥东西

第四课 派生类与继承

继承与派生类

种类

- 单继承
- 多继承

基本概念

- 基类（父类）
- 派生类（子类）

继承方式

继承内容

- 除构造函数、析构函数、私有成员外的其他成员

继承

- 保持已有类的特性而构造新的类

派生

- 在已有类的基础上新增自己的特性而产生新类
- 派生的实现方法
  - 修改基类成员

    在派生类中声明一个与基类成员同名的新成员
  - 描述新的特征或方法

    重载和重写的区别？？
- 从已有类派生出新类时，可以在派生类完成以下几种功能
  - 可以增加新的数据成员
  - 可以增加新的成员函数
  - 可以重新定义基类中已有的成员函数
  - 可以改变现有成员的属性

继承方式

- 系统默认为私有继承
- private成员在派生类中不可访问
- public成员在派生类中的访问属性和继承方式相同
- protected成员和继承方式取最小访问权限

派生类与基类的关系

- 派生类是基类的具体化
- 派生类是基类定义的延续
- 派生类是基类的组合（多继承）

多继承的例子

```
// multiple inheritance
#include <iostream>
using namespace std;
class Polygon
{
protected:
    int width, height;
public:
    Polygon (int a, int b) : width(a), height(b) {}
};
class Output
{
public:
    static void print (int i);
};
```

```
void Output::print (int i) { cout << i << '\n'; }

class Rectangle: public Polygon, public Output {
public:
    Rectangle (int a, int b) : Polygon(a,b) {}
    int area () { return width*height; }
};
class Triangle: public Polygon, public Output {
public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area () { return width*height/2; }
};
int main () {
    Rectangle rect (4,5);
    Triangle trgl (4,5);
    rect.print (rect.area());
    Triangle::print (trgl.area());
    return 0;
}
```

派生类的构造函数与析构函数

> 基类的构造函数和析构函数不能被继承，一般派生类要加入自己的构造函数

派生类构造函数和析构函数的执行顺序

- 先执行基类的构造函数，再执行派生类的构造函数

注意:

- 当基类构造函数不带参数时，派生类可以不定义构造函数，但基类构造函数带有参数，则派生类必须定义构造函数
- 若基类使用缺省构造函数或不带参数的构造函数，则在派生类中定义可以略去:基类构造函数名(参数表)
- 如果派生类的基类也是一个派生类，每个派生类只需要负责其直接基类的构造
- 由于析构函数是不带参数的，基类的析构函数不会因为派生类没有析构函数而不执行

多继承

多继承的声明

```
class 派生类名:继承方式1  基类名1,…,继承方式n  基类名n
{
    // 派生类新增的数据成员和成员函数
};
```

多继承时，对基类成员的访问必须无二义性

```
class X
{
public:
    int f();
};
```

```
class Y
{
public:
    int f();
    int g();
};
class Z: public X,public Y
{
public:
    int g();
    int h();
};
Z  obj;
obj.f(); //二义性错误，不知调用的是类X的f()，还是类Y的f()
```

构造函数的执行顺序

- 先执行基类构造函数，再执行对象成员的构造函数，最后执行派生类构造函数
- 必须负责所有基类的构造函数的调用
- 处于同一层次的各基类构造函数执行顺序，取决于声明派生类时所指定的各基类顺序，与初始化列表无关

虚基类

当一个类的多个基类是从另一个共同基类派生而来时，怎么办?

- 用作用域标识符来区分
- 定义虚基类，使派生类只保留一份拷贝

虚基类

- 声明位置：定义派生类时声明
- 语法形式

```
class  派生类名:virtual  继承方式  类名
{
    //…
}
```

- 举个例子

```
#include <iostream.h>
class base
{
public:
    base( ){ a=5; cout<<"base a="<<a<<endl;}
protected:
    int a;
};
class  base1: virtual public base
{
public:
    base1( ){ a=a+10; cout<<"base1 a="<<a<<endl;}
};
```

```
class base2: virtual public base
{
public:
    base2( ){ a=a+20; cout<<"base2 a="<<a<<endl;}
};
class derived:public base1,public base2
{
public:
    derived( ){ cout<<"derived a="<<a<<endl;}
};
main( )
{
    derived  obj; return  0;
}
```

虚基类的初始化

- 若虚基类没有缺省构造函数，所有直接间接的派生类都必须再构造函数从成员初始化列表中列出对虚基类构造函数的调用
- 虚基类从成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的
- 同一层次中同时包含虚基类和非虚基类，先调用虚基类的构造函数，再调用非虚基类的构造函数，最后调用派生类的构造函数
- 对于多个虚基类，构造函数执行的顺序是先左后右，自上而下
- 多余非虚基类，构造函数执行的顺序是先左后右，自上而下
- 若虚基类由非虚基类派生而来，任然先调用基类构造函数

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

class base {
public:
         base(int sa)
      {   a=sa;
         cout<<"Constructing base"<<endl;   }
      private:
      int a;
};
class base1:virtual public base{
public:
      base1(int sa,int sb):base(sa)
      {    b=sb;
         cout<<"Constructing baes1"<<endl;      }
      private:
      int b;
};
class base2:virtual public base{
public:
      base2(int sa,int sc):base(sa)
      {   c=sc;
         cout<<"Constructing baes2"<<endl;   }
private:
      int c;
```

```
};

class derived:public base1,public base2 {
public:
      derived(int sa,int sb,int sc,int sd):
         base(sa),base1(sa,sb),base2(sa,sc)
      {
         d=sd;
         cout<<"Constructing derived"<<endl;
      }
   private:
      int d;
};
main()
{
   derived obj(2,4,6,8);
   return 0;
}
```

注意：

- virtual和继承方式先后无关紧要
- 一个基类作为一些派生类的虚基类和另一些派生类的非虚基类是允许的

赋值兼容规则

可以用派生类对象给基类对象赋值

可以用派生类对象来初始化基类的引用

可以把派生类对象的地址赋值给向基类的指针

可以把指向派生类对象的指针赋值给指向基类对象的指针

注意：

- 声明为指向基类对象的指针可以指向它的公有派生对象，但不允许指向私有派生对象
- 不能将声明为指向派生类对象的指针指向基类对象
- 声明为指向基类对象的指针，当其指向公有派生类对象时，只能用它来直接访问派生类中从基类继承来的成员，而不能直接访问公有派生类中定义的成员！！！！重要！！！

第五课 多态性与虚函数

多态性

多态性的实现

- 函数重载
- 运算符重载
- 虚函数

静态多态性

- 函数重载
- 运算符重载

运行时的多态

- 继承
- 虚函数

四类多态

- 参数多态：函数模板和类模板
- 包含多态：虚函数
- 重载多态：函数重载，运算符重载
- 强制多态：强制类型转换

联编

- 将函数名与函数体的代码连接在一起
- 静态联编是编译阶段完成的联编
- 动态联编是运行阶段完成的联编

虚函数

虚函数允许调用与函数体之间的联系在运行时才建立，即动态联编

若某类中的成员函数被声明为虚函数，说明该成员函数在派生类时可能有不同实现，在用指针或引用操作时将采用动态联编调用虚函数

派生类对基类虚函数重写时的要求

- 与基类虚函数有相同的参数个数
- 参数的类型与基类虚函数对应参数类型相同
- 返回值或者与基类虚函数相同或者都返回指针或引用

动态联编的三个条件

- 说明虚函数
- 用指针或引用调用虚函数
- 子类关系的建立

注意：

- 赋值兼容的成立条件是派生类从基类公有派生！！！！因此在使用虚函数时必须公有派生
- 必须首先在虚函数中定义虚函数
- 派生类virtual可省略
- 对象名和.运算符也可以调用虚函数，但是这是静态联编
- 一个函数无论被公有继承多少次，仍然保持虚函数的特性
- 虚函数必须是其所在类的成员函数，不能是友元函数，也不能是静态成员函数
- 内联函数不能是虚函数，因为内联函数不能再运行过程中动态确定

  > 即使虚函数在类内部定义编译时仍看作是非内联的

- 构造函数不能是虚函数

- 析构函数可以是虚函数

虚析构函数

- virtual ~类名();
- 在程序执行带指针参数的delete时，不能清理干净（只调用基类析构函数）

纯虚函数

virtual 函数类型 函数名 参数表 = 0

表面在基类中不用定义该函数

> 如果在一个类中声明了纯虚函数，而在其派生类中没有对该函数定义，则该虚函数在派生类中仍然为纯虚函数

抽象类

如果一个类至少有一个纯虚函数，那么就称改类为抽象类

使用抽象类的几点规定

- 不能建立抽象类的对象
- 不允许从具体类派生出抽象类
- 抽象类不能用作参数类型函数返回类型或显式转换类型
- 可以声明指向抽象类的指针或引用，此指针可以指向它的派生类，进而实现多态性
- 抽象类的析构函数可以被声明为纯虚函数，此时至少提供该析构函数的一个实现
- 若派生类没有重新定义纯虚函数，则派生类仍然是抽象类
- 抽象类中也可以定义普通成员函数或虚函数

运算符重载

运算符重载实际上是将运算对象转化为运算函数的实参，并根据实参的类型来确定重载的运算函数

运算符重载规则

- . :: * sizeof ?:不能重载
- 重载后优先级和结合性都不能改变

运算符重载形式

- 重载为类的成员函数
- 重载为类的友元函数

限制

- 不可臆造新的运算符
- 四个不能改变
    - 操作符个数
    - 优先级
    - 结合性
    - 语法结构

重载格式

- 类型 类名::operator要重载的运算符(形参表)
- friend 函数类型 operator运算符(形参表)

运算符重载为成员函数

- 双目运算：oprd1 B oprd2
  - oprd1.operator+(oprd2)
- 单目运算：
  - 前置单目运算：U oprd
    - 函数类型 operator++();
    - oprd.operator++()对它本身调用?
  - 后置单目运算oprd V
    - 函数类型 operator--(int)：
    - oprd.operator--(0)对传入参数?

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

class Complex
{
    private:
        float real,image;
    public:
        Complex(float r=0,float i=0);
        Complex Add(const Complex &c);   //定义一个Add函数
        Complex operator+(const Complex &c); //重载+运算符
        Complex operator-(const Complex &c); //重载-运算符
        Complex& operator+=(const Complex &c); //重载+=运算符，复合赋值操作符必须返回左操作数的引用，必须定义为成员函数
        Complex& operator=(const Complex &other); //重载=运算符，赋值运算符必须返回对*this的引用，
        void Show(int i);
};

Complex::Complex(float r,float i)
{
    real=r;  image=i;
}
void Complex::Show(int i)
{  //一般情况下，这里不应该有参数i，本例的目的是为了区分不同的复数，便于观看结果
    cout<<"复数: c"<<i<<"="<<real;
    if(image>0)
        cout<<"+"<<image<<"i"<<endl;
    if(image<0)
        cout<<image<<"i"<<endl;
}

Complex  Complex::Add(const Complex &c)
{
    Complex t;
    t.real=this->real+c.real;
```

```
    t.image=this->image+c.image;
    return t;
}
Complex  Complex::operator+(const Complex &c)
{
    Complex t;
    t.real=this->real+c.real;
    t.image=this->image+c.image;
    return t;
}

Complex  Complex::operator-(const Complex &c)
{
    Complex t;
    t.real=this->real-c.real;
    t.image=this->image-c.image;
    return t;
}
Complex&  Complex::operator+=(const Complex &c)
{
    real=real+c.real;
    image=image+c.image;
    return *this;
}
Complex& Complex::operator=(const Complex & other)
{
    if(this == &other)
        return *this;
    this->real=other.real;
    this->image=other.image;
    return *this;
}

int main()
{
    Complex c1(12,35),c2(20,46),c3,c4,c5,c6;
    c1.Show(1);
    c2.Show(2);
    c3=c1.Add(c2);
    c3.Show(3);
    c4=c1+c2;
    c4.Show(4);
    c2+=c1;
    c2.Show(2);
    c5=c1-c2;
    c5.Show(5);
    return 0;
}
```

注意

- 对于二元运算符，函数的参数为一个右操作数，左操作数为调用重载函数的对象
- 对于一元运算符运算符的左操作数或右操作数为调用重载函数的对象

- 运算符重载函数的返回类型：若对象进行运算后的结果仍未原类型，则运算符重载函数的返回类型仍为原类型

赋值运算符重载的步骤

- 检查是否自赋值，若是则立即返回
- 释放原因的内存资源（若无指针则不需要）
- 分配新的内存资源，并复制内容
- 返回本对象的引用

赋值运算符与拷贝构造函数

- 拷贝构造函数在对象建立的时候执行
- 赋值运算符在对象建立之后执行

运算符重载为友元函数

由于友元函数没有this指针，传入参数比重载为成员函数多了一个

总结

友元函数or成员函数?

- 赋值=、下标[]、调用()、和成员访问->必须定义为成员函数（要返回自引用？）
- 符合赋值也要定义为成员函数
- 自增自减等定义为成员函数
- 对称的操作符如算术操作符、相等操作符、关系操作符和位操作符定义为友元函数

重载流插入和流提取运算符

输出操作符重载：ostream& operator << (ostream&,const 自定义类&);

第一个参数是ostream&类型，第二个参数可以是const

```
#include <iostream>
using namespace std;
class Date
{
    public:
        Date(/int y,int m,int d);
        bool isLeapYear();
        void print();
        friend ostream &operator<<(ostream &output,Date &d);
    private:
        int  year;
        int  month;
        int  day;
};

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

```
}
bool Date::isLeapYear()
{
    return (year%4==0 && year%100!=0)||(year%400==0);
}

void Date::print()
{
     cout<< year<<"-"<< month<<"-"<< day<<endl;
}
ostream &operator<<(ostream &output,Date &d)
{
    output<<d.year<<"-"<<d.month<<"-"<<d.day;
    return output;
}

int main()
{
    Date d(1999,06,28);
    cout << d;
}
```

输入操作符重载：istream &operator >> (istream &, 自定义类&);

第二个参数必须为nonconst，和输入不同

```
istream& operator>>(istream &input,Date &d)
{
    input>>d.year>>d.month>>d.day;
    if(!input)
    {
        d=Date();
    }
    return input;
}
```

类型重载

```
class <类名1>
{
    public:
    operator <类名2>( );
    ……
};
<类名1>::operator <类名2>()
{     函数体;  }
```

```
#include <iostream>
using namespace std;
class Time
{
```

```
private:
    int hour,minute,second;
public:
    Time(int h=0,int m=0,int s=0);
    void Show();//显示时：分：秒的成员函数
    operator float();
};
Time::Time(int h,int m,int s)
{
    hour=h;
    minute=m;
    second=s;
}

void Time::Show()
{
    cout<<hour<<":"<<minute<<":"<<second<<endl;
}
Time::operator float()
{
    float sec;
    sec=hour*3600+minute*60+second;
    //cout<<"second="<<sec<<endl;
    return sec;
}

int main()
{
    float s1,s2,s3;
    Time t(10,15,20);
    s1=t;
    s2=float(t);
    t.Show();
    s3=(float)t;
    cout<<"s1="<<s1<<'\t'<<"s2="<<s2<<'\t'<<"s3= "<<s3<<endl;
}
```

第六课 流类库与输入输出

第六课 流类库与输入输出

第七课 模板

第八课 类库和标准模板库STL

第九课 异常处理