

C++程序设计

第三篇 面向对象编程

第10章 类与对象

第11章 类的静态成员与类的友元

第12章 运算符的重载

第13章 类的继承性

第14章 类的多态性

第15章 C++输入/输出系统

第16章 异常

第10章 类与对象的定义

面向过程程序设计：以功能为中心，数据为功能服务

面向对象程序设计：以数据为中心，功能为数据服务

10.1 类与对象的定义

10.1.1 结构的演变

结构类型的改进 {

- 加进成员函数
- 增加隐藏机制

例B10.1 结构的扩展1。 MB10_1A.cpp

struct triangle 定义带成员函数的结构类型

```
{ double a,b,c;  
    double area( ){double s=(a+b+c)/2; return sqrt(s*(s-a)*(s-b)*(s-c));}  
};  
int main( )  
{ triangle m;  
  cin>>m.a>>m.b>>m.c;  
  cout<< "三角形面积为: "<<m.area( )<<endl; return 0;  
}
```

成员函数定义

成员函数调用

例B10.1 结构的扩展2。 MB10_1B.cpp

struct triangle 定义具有隐藏机制的结构类型

```
{  
    private: 隐藏下面数据  
        double a,b,c;  
    public: 公开下面成员  
        void set(double ia,double ib,double ic) { a=ia; b=ib; c=ic; }  
        double area( ){double s=(a+b+c)/2; return sqrt(s*(s-a)*(s-b)*(s-c));}  
};  
int main( )  
{ triangle m;    double aa,bb,cc;  
  cin>>aa>>bb>>cc;    数据m.a,m.b,m.c已隐藏, 不能使用  
  m.set(aa,bb,cc);      将数据aa,bb,cc存入对象m  
  cout<< "三角形面积为: "<<m.area( )<<endl; return 0;  
}
```

注1: 程序中的private和public称为访问权限, private表示该权限说明之下的成员都是被隐藏的, 外界不能访问, 而public权限说明之下的成员是向外公开的, 外界可以无限制地使用。

10.1.2 类的定义与使用

附加了功能并有所隐藏的数据称为**对象**，对象的类型称为**类**。

类通常使用**class**引导，而非**struct**引导。

struct通常用于无成员函数的纯粹复合数据。

注2：类是一种特别的复合数据类型，含成员数据和成员函数，成员可用**private**和**public**来隐藏和公开，在定义时用**class**引导。类的变量称为对象。

注3：习惯上，**struct**依然只封装纯粹的数据，如第5章所使用的那样。

注4：类的成员函数的函数定义可以在类定义体内，也可以放在类定义体外。在类外定义的函数，函数名需要用作用域运算符“**::**”添上所属的类，如**triangle::set**。另外，类定义体内必须有类外定义的成员函数的函数说明。

例A10.2 类的定义。

类定义体

class triangle 定义类

```
{  
private:  
    double a,b,c;  
public:  
    void set(double ia,double ib,double ic); 成员函数说明  
    double area(); 成员函数说明  
};
```

类的范围

```
int main( )  
{   triangle m;      double aa,bb,cc;  
    cin>>aa>>bb>>cc; m.set(aa,bb,cc);  
    cout<< "三角形面积为: "<<m.area()<<endl; return 0;  
}
```

表示set属于triangle类

类的范围

```
void triangle::set(double ia,double ib,double ic) 成员函数类外定义  
{   a=ia; b=ib; c=ic; }  
double triangle::area() 成员函数类外定义  
{   double s=(a+b+c)/2; return sqrt(s*(s-a)*(s-b)*(s-c)); }
```

○ class 与 struct 的差异

class中的默认访问权限

```
class A
{
    int a;
public:
    void set(int x) { a=x; }
    int get( ) { return a; }
};
```



```
class A
{
    private:
    int a;
public:
    void set(int x) { a=x; }
    int get( ) { return a; }
};
```

struct中的默认访问权限

```
struct A
{
    int a;
public:
    void set(int x) { a=x; }
    int get( ) { return a; }
};
```



```
struct A
{
    public:
    int a;
public:
    void set(int x) { a=x; }
    int get( ) { return a; }
};
```

注5: struct类型成员的默认访问权限是public, class类型成员的默认访问权限是private。

类作用域：类中成员的作用域，范围为整个类的定义及类成员函数的定义

类作用域

```
class A
```

```
{
```

```
public:
```

```
    void set(int x) { a=x; }
```

```
    int get( ) ;
```

```
private:
```

```
    int a;
```

```
};
```

```
.....
```

```
int A::get( ) { return a; }
```

类作用域

函数体内的a在a的作用域内

类作用域

此处的a也在a的作用域内

类中成员函数可以访问类的**private**成员。

```
class A
{
private:
    int a;
public:
    void set(int x) { a=x; }
    int get() { return a; }
    void add(A x) { a+=x.a; } 也能访问x.a
};

.....
A oa,ob; oa.set(10); ob.set(8); oa.add(ob);
cout<<oa.get()<<endl;
.....
```

注6：在上述类A的成员函数add内可以使用成员a，也可以使用同类的其它对象的成员，例如x.a。

10.1.3 ○ 内联成员函数

内联成员函数 { • 类内定义
• 类外用inline定义

```
class triangle
```

```
{
```

```
private:
```

```
    double a,b,c;
```

```
public:
```

```
    void set(double ia,double ib,double ic) 内联成员函数
```

```
    {a=ia;b=ib;c=ic;}
```

```
    double area( ) ;
```

```
};
```

```
inline double triangle::area( ) 内联成员函数
```

```
{ double s=(a+b+c)/2; return sqrt(s*(s-a)*(s-b)*(s-c)); }
```

注7：类的成员函数可以是重载函数，也可以是内联函数，并可以有默认参数。

10.1.4 this指针

```
class A
{
private:
    int a;
public:
    void set(int x) { a=x; }
    int get() { return a; }
    void add(A x) { a+=x.a; }
};
```

A oa,ob;
oa.set(10); oa.set内部的this就是&oa
ob.set(8); ob.set内部的this就是&ob
oa.add(ob); oa.add为oa.a+=ob.a

类成员函数有一个内部指针：
this，就是对象自己的指针。



```
class A
{
private:
    int a;
public:
    void set(int x) { this->a=x; }
    int get() { return this->a; }
    void add(A x) { this->a+=x.a; }
};
```

注8：类的this指针是一个指向当前对象的内部指针，不能改动。

10.1.5 对象的使用

定义了类之后，就可以用来定义对象来处理数据了。

对象使用就像结构变量一样，但是可以使用成员函数来进行数据处理。

例**C10.3** 用对象处理一批学生成绩。 类的定义

```
class student
```

```
{ private:
```

```
    char name[10]; char ID[15]; int score;
```

```
public:
```

```
    void input( ) { cin>>name>>ID>>score; } 输入数据
```

```
    void output( ) const 输出数据, 常成员函数, 禁止改成员数据
```

```
{ cout<<name<<"\t"<<ID<<"\t"<<score<<endl;}
```

```
    int compare(student b) const
```

```
{ return score-b.score; } 可使用b的成员
```

```
};
```

例C10.3 用对象处理一批学生成绩。 类的使用

```
void sort(student s[ ],int n)
{  int i,j,k; student t;
   for(i=0;i<n;i++)
   {   for(k=i,j=i+1;j<n;j++) if(s[j].compare(s[k])>0) k=j;
       if(k!=i) {  t=s[k]; s[k]=s[i]; s[i]=t; } 对象可以直接赋值
   } }

int main( )
{  student st[200]; int n,i ; .....
   for(i=0;i<n;i++) st[i].input( );
   sort(st,n); .....
   for(i=0;i<n;i++) st[i].output( ); .....
}
```

注9：成员函数如output()用const说明用以禁止改动成员数据，从而加强对成员数据的保护。由于面向对象程序设计强调数据保护，本书后面的程序例子将大量使用const说明来增加对数据的保护。

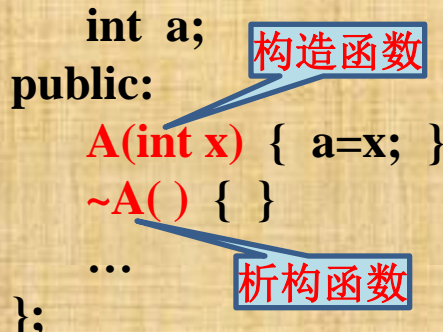
10.2 类的构造与析构

10.2.1 构造函数与析构函数

对象中隐藏的成员数据的初始化不能直接使用初值，必须使用专门的成员函数：**构造函数**。

构造函数在**对象定义时自动调用**，对应的，**对象撤销时**有个自动调用的函数：**析构函数**。

```
class A
{
private:
    int a;
public:
    A(int x) { a=x; }
    ~A() { }
    ...
};
```



或者

```
class A
{
private:
    int a;
public:
    A(int x);
    ~A();
    ...
};
A::A(int x) { a=x; }
A::~~A() { }
```

注1：对象定义时通过构造函数赋初值，构造函数名就是类名，构造函数在对象定义时由计算机自动调用。对象在撤销时，计算机会自动调用析构函数，析构函数名为类名前加~。

例A10.4 用构造函数初始化对象的成员数据。

class Point 定义了构造函数，省略析构函数

```
{  
private:  
    double x,y;  
public:  
    Point(double px,double py) { x=px; y=py; } 构造函数  
    double distance(Point b) const; 函数内无数据改动，用const说明  
    void show( ) const { cout<< "("<<x<< ","<<y<< ")"\n"; }  
};  
double Point::distance(Point b) const  
{ return sqrt((x-b.x)*(x-b.x)+(y-b.y)*(y-b.y)); }  
int main( )  
{ Point a(10,10),b(20,20); 自动调用构造函数，参数为10,10和20,20  
  a.show(); b.show();  
  cout<< "两点距离为: "<<a.distance(b)<<endl ;  
  return 0;  
}
```

构造函数和析构函数在对象使用动态空间时尤其有用。

例**C10.5** 构造函数与析构函数用于对象使用动态空间资源。

```
class String
```

```
{ private: char *p;
```

```
public:
```

```
String(const char *s); 构造函数说明
```

```
~String() { delete[] p; }
```

```
void show( ) const { cout<<p<<endl; }
```

```
};
```

```
String::String(const char *s) 构造函数定义, 可用于const char *型初值
```

```
{ p=new char[strlen(s)+1]; strcpy(p,s); }
```

```
int main( )
```

```
{ String s1("Hello"),s2("Byby");
```

```
s1.show(); s2.show(); return 0; }
```

注2: 上述程序构造函数的形参可以是“const char *s”, 也可以是“char *s”。使用“const char*s”有两点好处: 一是保护了作为初始值的字符串不被改动, 二是const char*类型字符串也可以作为构造对象的初始值。

构造对象过程分两步：

1、开辟空间并按定义中成员变量次序生成变量

2、执行构造函数函数体

```
class A
```

```
{
```

```
private:
```

```
    int a;
```

```
public:
```

```
    A(int x): a(x) { }    对象构造时成员数据a赋初值
```

```
// 也可以写成 A(int x) { a=x; } 成员数据在构造函数体内赋值
```

```
    ~A() { }
```

```
    .....
```

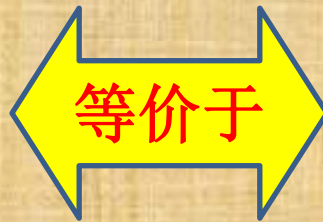
```
};    成员数据赋初值和赋值的差异就象: int a=x; 和 int a; a=x;
```

构造函数初始化列表

注3：构造函数的函数首部后面的冒号之后是成员变量的初始化列表，初始化列表用来提供成员的初始值，即在成员生成时赋初值。

注4：按照C++的语法，对象都必须由构造函数构造生成，由析构函数撤销对象。当没有给出构造函数时，计算机会自动生成一个空白的无参构造函数作为该类的构造函数，如果没有给出析构函数，计算机也会自动生成一个空白的析构函数，称为默认析构函数。


```
class A
{
private:
    int a;
public:
    void set(int x) { a=x; }
    int get( ) const {return a; }
};
```



```
class A
{
private:
    int a;
public:
    A() { }
    ~A() { }
    void set(int x) { a=x; }
    int get( ) const { return a; }
};
```

可以无参调用的构造函数统称为：**默认构造函数**
默认构造函数包括：

- 计算机自动生成的构造函数
- 无参构造函数
- 全部参数有默认值的构造函数

```
class A
{
private:  int a;
public:
    A(int x=10 ) { a=x; }
    void set(int x) { a=x; }
    int get( ) const { return a; }
};
```

注5: 若定义了类A, 要无参地调用构造函数必须用 “A oa;” 而不能用 “A oa();”, 因为后者表示一个函数说明, 函数名为oa, 无参, 返回值类型为A。

注6: 所有可以无参调用的构造函数都称为默认构造函数, 并且默认构造函数只能有一个, 否则会发生冲突。

注7: 只要定义了一个构造函数, 则计算机就不会自动生成默认构造函数。如果只定义了带参数的构造函数, 那么该类就没有默认构造函数, 则定义对象时一定要给出参数, 否则会出错。

构造函数可以重载。

例A10.8 构造函数的重载。

```
class A
{
private:
    int x,y;
public:
    A( ) { x=y=0; }    重载构造函数: 无参 默认构造函数
    A(int ix) { x=ix; } 重载构造函数: 单参
    A(int ix,int iy) { x=ix; y=iy; } 重载构造函数: 双参
    void show( ) const { cout<<x<<" "<<y<<endl; }
};

int main( )
{
    A a0,a1(10),a2(-1,-2); 分别调用无参,单参,双参构造函数
    a0.show( ); a1.show( ); a2.show( ); return 0;
}
```

10.2.2 复制构造函数

特殊的构造函数：**复制构造函数**-----初始值为同类对象

```
class A
```

```
{ private: int a;
```

```
public:
```

```
    A(int x=10){a=x;}
```

```
    A(const A&ob){a=ob.a;} 复制构造函数,也可A(A &b){a=ob.a;}
```

```
    void set(int x){a=x;}
```

```
    int get( ) const {return a;}
```

```
} oa(10);
```

复制构造过程:

A oc(oa); 调用复制构造函数
或者等价地

A oc=oa; 调用复制构造函数

注8: 复制构造函数的参数必须是const引用或引用, 例如const A& 或 A&, 若参数是引用, 则初值不能为常量(**常对象**)。

例A10.9 复制构造函数。 复制构造函数的典型使用

```
class String
{ private:  char *p;
public:
    String(const char *s= ""); 默认为空串
    String(const String &b); 复制构造函数，涉及浅复制和深复制
    ~String( ) { delete[ ]p; }
    void show( ) const { cout<<p<<endl; }
}
String::String(const char *s)
{ p=new char[strlen(s)+1]; strcpy(p,s); }
String::String(const String &b)
{ p=new char[strlen(b.p)+1]; strcpy(p,b.p); }
int main( )
{ String s1("Hello"); s1.show( );
    String s2(s1),s3=s1; s2.show( ); s3.show( ); return 0;
}
```

复制构造对象

注9：上述程序中对象的定义形式，“String s1("Hello");”等价于“String s1=String("Hello");”，“String s2(s1);”等价于“String s2=s1;”，等价于“String s2=String(s1);”。

注10：当没有定义复制构造函数时，计算机会自动生成一个默认复制构造函数，默认复制构造函数进行纯粹的字节复制。不管什么情况，复制构造函数有且仅有一个。

```
class A
{
private:
    int a,b;
public:
    A(int x=0,int y=0) { a=x; b=y; }
    void show( ) const { ..... }
};
```

等价于

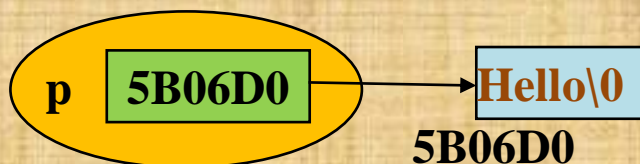
```
class A
{
private:
    int a,b;
public:
    A(int x=0,int y=0) { a=x; b=y; }
    A(const A&ob) {a=ob.a;b=ob.b;}
    void show( ) const { ..... }
};
```

注11：复制构造函数属于构造函数，所以只要定义了复制构造函数，计算机就不会自动生成默认构造函数。

浅复制与深复制问题

以下列程序为例讨论浅复制与深复制问题

```
class String                                     MA10_9.cpp
{ private:  char *p;
public:  String(const char *s= "");
        String(const String &b);
        ~String() { delete[] p; }
        void show() const { cout<<p<<endl; }
};
String::String(const char *s) { p=new char[strlen(s)+1]; strcpy(p,s); }
String::String(const String &b) { p=new char[strlen(b.p)+1];strcpy(p,b.p); }
int main()
{  String s1("Hello"); s1.show();
   String s2(s1),s3=s1; s2.show(); s3.show(); }
```

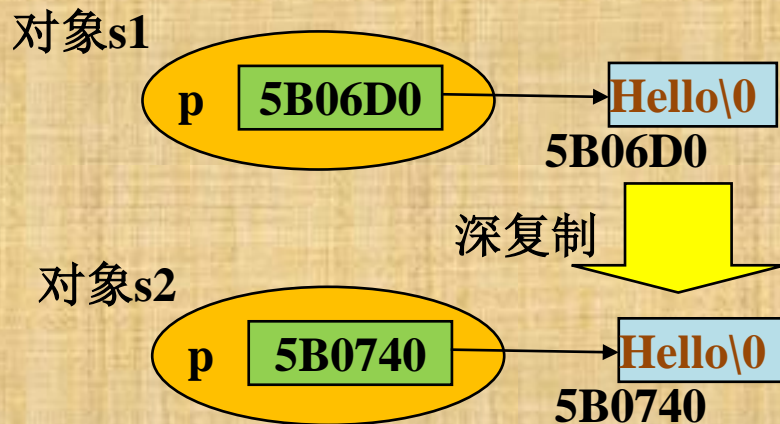


深复制型的对象

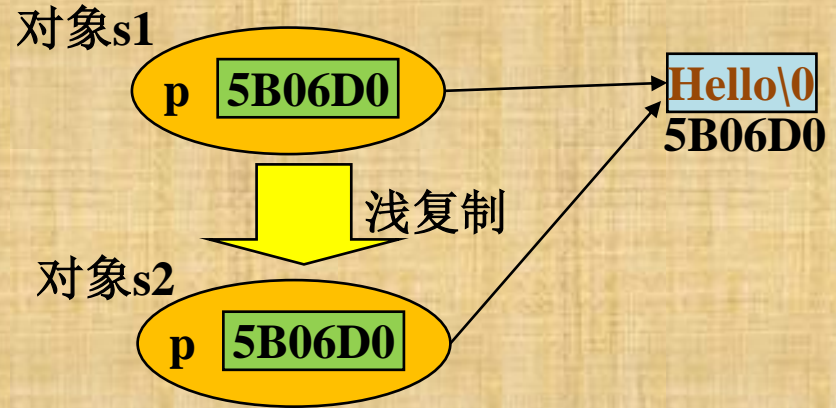


浅复制型的对象

浅复制过程与深复制过程图示



深复制过程



浅复制过程

若前面程序中删去复制构造函数，则复制构造过程调用默认复制构造函数进行指针复制，于是多个对象指向同一个动态数组。析构时将多次释放同一个动态数组，造成出错。

注12：深复制型对象(含深复制型数据)需要定义复制构造函数，用于进行深度复制，一般也要定义析构函数，以释放所占用的资源。非深复制型的对象一般不需要定义复制构造函数，可使用默认复制构造函数。

注13：对象的复制分复制构造对象和对象间的赋值。深复制型的对象既需要设计复制构造函数，也需要设计对象间的赋值(详见12.3节)。

10.2.3 临时对象

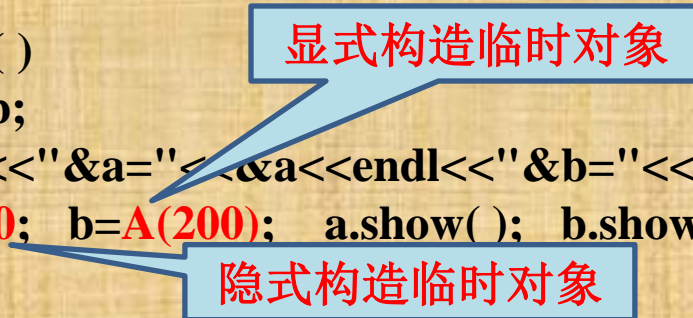
程序构造临时对象的情况：

- 用构造函数显式构造
- 隐式进行类型转换时构造
- 构造函数值对象

例B10.10 显式和隐式构造临时对象。

```
class A
{ private: int x;
public:
    A() { x=0; }
    A(int ix) { x=ix;cout<<"A("<<x
        <<") this="<<this<<endl; }
    ~A() { cout<<"~A():x="<<x
        <<"\t this="<<this<<endl; }
    void show() const { cout<<"x="<<x<<endl; }
};

int main()
{   A a,b;
    cout<<"&a="<<&a<<endl<<"&b="<<&b<<endl;
    a=100; b=A(200); a.show(); b.show(); return 0;
}
```



运行结果：

```
&a=003CFA80
&b=003CFA74
A(100) this=003CF990
~A(): x=100 this=003CF990
A(200) this=003CF99C
~A(): x=200 this=003CF99C
x=100
x=200
~A(): x=200 this=003CFA74
~A(): x=100 this=003CFA80
```

注14：临时对象在所在语句结束时析构。对象类型的函数值是临时对象。临时对象相当于一个表达式的值，严格来说不能做左值，可当成常量(常对象)对待。

10.2.4 调用函数时数据传递中的对象

函数数据传递时，对象一般复制构造，而引用则不调用任何构造函数，因为没有生成新的对象。

例B10.11 参数及函数值为对象时的对象构造。（定义类）

```
class A
{ private: int x;
public:  A() { x=0; }
        A(int ix) { x=ix; cout<<"A("<<x<<" ) this="<<this<<endl; }
        A(const A &b){ x=b.x;cout<<"copy A("<<x<<" ) this="<<this<<endl; }
        void add(int iy){ x+=iy; }
        ~A(){cout<<"~A( ):x="<<x<<"\tthis="<<this<<endl; }
        void show() const {cout<<"x="<<x<<endl;}
};
```

例B10.11 参数及函数值为对象时的对象构造。（函数）

```
A f1(int y)
{   cout<<"inside f1(int y)...\n"; return A(y);}
A f2(A t)
{   t.add(20);
    cout<<"inside f2(A t)... &t="<<&t<<endl;
    return t;}
A &g(A &s)
{   s.add(40);
    cout<<"inside g(A&s)... &s="<<&s<<endl;
    return s;}
int main( )
{   A a(10),b;
    cout<<"&a="<<&a<<endl
        <<"&b="<<&b<<endl;
    b=f1(100), cout<< "before end of the statement"
        <<endl;   b.show();
    b=f2(a); b.show(); b=g(a); b.show(); return 0;
}
```

运行结果：

A(10) this=007EFA5C 对象a

&a=007EFA5C

&b=007EFA50

inside f1(int y)...

A(100) this=007EF960 f1函数值

before end of the statement

~A():x=100 this=007EF960

x=100

copy A(10) this=007EF93C f2形参

inside f2(A t)... &t=007EF93C

copy A(30) this=007EF978 f2函数值

~A(): x=30 this=007EF93C

~A(): x=30 this=007EF978

x=30

inside g(A &s)... &s=007EFA5C

x=50

~A(): x=50 this=007EFA50

~A(): x=50 this=007EFA5C

注15：只要生成新的对象就要调用构造函数或者复制构造函数。当由对象生成新的对象时一定会调用复制构造函数。

10.3 动态对象与对象数组

10.3.1 ○对象数组的构造与析构

对象数组是逐个元素依次构造的。

例A10.12 对象数组的构造与析构。

```
class A
{ private: int x;
public: A( ) {x=0;cout<<"A( ):"<<this<<endl; }
      A(int ix){x=ix;cout<<"A("<<x<<"):"<<this<<endl; }
      ~A( ) { cout<<"~A( )"<<this<<endl; }
      void show() const {cout<<"x="<<x<<endl; }
};

int main( )
{ int i;  A  a[4];
  for(i=0;i<4;i++) cout<<&a[i]<< "\\t";cout<<endl;
  A  b[4]={A(1),A(2),A(3),A(4)};
  for(i=0;i<4;i++) b[i].show();
  for(i=0;i<4;i++) cout<<&b[i]<< "\\t"; cout<<endl;
}
```

注1: “A a[4];”表示定义对象数组a,

它含4个元素。“A a(4);”表示定义对象a, 初值为4, 两者不要混淆。

注2: 对象数组的初始化列表可以包含默认构造函数调用和复制构造函数调用, 例如“A b[4]={A(1),A(2),A(),A(a[0])};”, 甚至可以有更少的构造函数调用“A b[4]={A(1),a[3],5};”。Ch10 类与对象的定义

```
A(): 00E8FB38 默认构造a[0]
A(): 00E8FB3C 默认构造a[1]
A(): 00E8FB40 默认构造a[2]
A(): 00E8FB44 默认构造a[3]
00E8FB38 00E8FB3C 00E8FB40 00E8FB44
A(1): 00E8FB20 带参构造b[0]
A(2): 00E8FB24 带参构造b[1]
A(3): 00E8FB28 带参构造b[2]
A(4): 00E8FB2C 带参构造b[3]
x=1
x=2
x=3
x=4
00E8FB20 00E8FB24 00E8FB28 00E8FB2C
~A() 00E8FB2C 析构b[3]
~A() 00E8FB28 析构b[2]
~A() 00E8FB24 析构b[1]
~A() 00E8FB20 析构b[0]
~A() 00E8FB44 析构a[3]
~A() 00E8FB40 析构a[2]
~A() 00E8FB3C 析构a[1]
~A() 00E8FB38 析构a[0]
```


10.3.2 动态对象与动态对象数组

可以用new或new[]构造动态对象或动态对象数组，
用delete或delete[]析构动态对象和动态对象数组

若有类的定义

```
class A
{ private:    int a;
  public:
    A(int x=10) { a=x; }
    int get( ) const { return a; }
} *p1,*p2;
```

则可以如下构造动态对象和动态对象数组

p1=new A(10); 构造动态对象

p2=new A[15]; 构造动态对象数组

最后可以如下释放动态对象与动态对象数组

delete p1; delete[]p2; 析构动态对象和动态对象数组

注3：用new构造对象或new[]构造对象数组时都会自动调用构造函数。用delete析构对象或delete[]析构对象数组时都会自动调用析构函数。

注4：new构造的对象一定要用delete析构。new[]构造的对象数组一定要用delete[]析构。

例A10.13 动态对象的构造与析构。

class A

{ private: int x;

public:

A() { x=0;cout<<"A():"<<this<<endl; }

A(int ix) { x=ix;cout<<"A("<<x<<"):"<<this<<endl;}

A(const A &b) {x=b.x;cout<<"copy A("<<x<<"):"<<this<<endl;}

~A() { cout<<"~A()"<<this<<endl;}

void show() const {cout<<"x="<<x<<endl;}

};

int main()

{ A a(100),*p1,*p2,*p3;

a.show();

p1=new A; p1->show();

p2=new A(200); p2->show();

p3=new A(a); p3->show();

delete p1; delete p2; delete p3;

return 0;

}

运行结果:

A(100): 00DDFB18 构造对象a

x=100

输出a

A(): 01258350

默认构造动态对象, p1指向

x=0

输出动态对象*p1

A(200): 01258680

带参构造动态对象, p2指向

x=200

输出动态对象*p2

copy A(100): 012584D0 a复制构造动态对象, p3指向

x=100

输出动态对象*p3

~A() 01258350

析构动态对象*p1

~A() 01258680

析构动态对象*p2

~A() 012584D0

析构动态对象*p3

~A() 00DDFB18

析构对象a

10.3.3 ○对象的生命周期

对象构造和析构的时刻随对象类型而不同。规则如下：

- 全局对象在主函数运行之前构造，在主函数全部执行完成并撤消了所有普通局部变量、普通和静态局部对象后析构
- 普通局部对象(包括形参对象)在执行到定义时构造，在出了对象名的作用域时析构
- 静态局部对象在第一次执行到定义时构造，在主函数全部执行完并撤消了所有普通局部变量和普通局部对象后析构
- 动态对象执行到相应的new操作时构造，执行到相应的delete时析构
- 临时对象由转换时或显式调用构造函数或执行return语句时构造，在所在语句结束时析构
- 栈区中对象的构造次序与析构次序正好相反

例B10.15简化 各种对象的构造与析构时间点。

```
class A
{ private: char s[80];
public:
    A(const char *ss)
    { printf("%s 构造\n",strcpy(s,ss) ); }
    ~A() { printf("%s 析构\n",s); }
    void show() const { printf( "s=%s\n",s); }
};

void f()
{ A "f中局部对象fa");
  static A fb("f中静态局部对象fb");
}

A a("全局对象a");

int main()
{ printf( "begin...\n"); A b("main中局部对象b");
  static A c("main中静态局部对象c"),*p;
  p=new A("动态对象");
  f(); delete p; f();
  printf("end...\n");
  return 0; }
```

运行结果:

全局对象a 构造
begin...
main中局部对象b 构造
main中静态局部对象c 构造
动态对象 构造
f中局部对象fa 构造
f中静态局部对象fb 构造
f中局部对象fa 析构
动态对象 析构
f中局部对象fa 构造
f中局部对象fa 析构
end...
main中局部对象b 析构
f中静态局部对象fb 析构
main中静态局部对象c 析构
全局对象a 析构

注5: 全局对象用cin、cout输入/输出会有隐患, 可能执行不了。

10.4 常成员和对象类成员

- **常成员**：const说明的成员，分**常成员数据**和**常成员函数**
- **常对象**：const说明的对象

例A10.16 常成员常对象的使用。

```
class A
{ private:
    const int A1; 常成员数据
    int x;
public:
    const int A2; 常成员数据
    A(int c1,int c2,int ix):A1(c1),A2(c2),x(ix) { }
    void show( ) const ; 常成员函数，*this冻结为常量，不能改动
};
void A::show( ) const  show为void ( ) const类型，不同于void ( )类型
{  cout<<"A1="<<A1<<" , A2="<<A2<<" , x="<<x<<endl;  }
int main( )
{  A b(1,2,30); cout<<"b.A2="<<b.A2<<endl; b.show( );
    const A c(10,20,30); 定义常对象c，常对象只能使用常成员
    cout<<"c.A2="<<c.A2<<endl; c.show( ); return 0;
}
```

注1：对象的常成员数据必须在构造函数的初始化列表中赋初值。

注2：常成员函数的const说明冻结了*this，故不能改动成员数据。

注3：常对象只能使用常成员。常对象与普通对象的生命周期一样。

注4：不能用常成员数据作为对象定义的数组成员的数组大小。

如下定义是错误的，会造成**sizeof(B)**不确定

```
class B
{ private:
    const int n; 常成员数据
    double x[n];
public:
    B(int k):n(k) { }
    .....
};
```

例A10.17 类的对象类成员。

```
class A
{ private: int x;
public: A(int ix=0): x(ix) { }
      void show( ) const { cout<<"A::x="<<x<<endl;}
};
class B
{
private:
    int y;
    A oa,ob; 对象类成员
public:
    B(int x1):y(x1) { }
    B(int x1,int x2,int x3):y(x1),oa(x2),ob(x3) { }
    void show( ) const {cout<<"B::y="<<y<<endl;oa.show();ob.show();}
};
int main( ) { B b(1,2,3),c(10); b.show(); c.show(); return 0; }
```

对象类成员进行默认构造

对象类成员必须在构造函数初始化列表中进行构造

注5: 对象类的成员必须在构造函数的初始化列表中构造, 或者默认构造。

注6: 对象的成员在构造时是按照成员定义的次序依次构造, 与构造函数初始化列表中成员构造的次序无关。成员的析构与构造次序正好相反。

注7: 对象的成员在内存中存放的次序与定义的次序一致。

第11章 类的静态成员与类的友元

11.1 静态成员

11.1.1 静态成员变量

静态成员变量就是为所有对象所共享的变量，只有一份。静态成员变量**必须在类外定义**，没赋初值时默认初值为0。

例A11.1 静态成员变量的使用。

```
class A
{
private:  int x;
        static int y;  静态成员变量说明
public:
    A(int ix): x(ix) { }
    void show( ) const {cout<<"x="<<x<<"", y="<<y<<endl; }
    void setX(int ix) { x=ix; }
    void setY(int iy) { y=iy; }
};
int A::y=100;  静态成员变量定义，本质上是全局变量
int main( )
{  A a(10),b(20);    a.show( ); b.show( );
   a.setY(90);  b.show( );           A::y为a、b所共享
   a.setX(30);  b.setY(-50);  a.show( ); b.show( ); return 0;
}
```

注1: 静态成员变量必须在类外定义并赋初值。若静态成员变量定义时没有赋初值，则默认初值为0。

注2: 静态成员变量只有一份，在对象构造前就存在。构造对象时只构造非静态类的成员数据，不构造静态成员变量。

静态成员变量可以用来统计对象个数或数据之和。

例C11.2 用静态成员变量统计对象数据。

```
class A
{
private:
    int x;
    static int total;  静态成员变量说明
public:
    A(int ix): x(ix) { total+=x; }
    A(const A &b): x(b.x) { total+=x; }  复制构造也需要统计
    ~A( ) { total-=x; }
    void show( ) const { cout<<"x="<<x<<endl; }
    void showTotal( ) const { cout<<"total="<<total<<endl; }
};
int A::total=0;  静态成员变量定义
int main( )
{
    A a(10),b(20);    a.showTotal( );
    {
        A c1(15),c2(-100),c3(137);    c3.show( ); c3.showTotal( ); }
    A d(b); b.showTotal( ); return 0;
}
```

11.1.2 静态成员函数

静态成员变量与对象无关，故用对象使用静态成员变量不合适，

静态成员函数正好满足这个要求。

例C11.3 静态成员函数使用静态成员变量。

```
class A
```

```
{ private: int x; static int total;
```

```
public: A(int ix): x(ix) { total+=x; }
```

```
    A(const A&b):x(b.x){total+=x;} ~A( ) { total-=x; }
```

```
    void show( ) { cout<<"x="<<x<<endl; }
```

```
    static void showTotal( ) { cout<<"total="<<total<<endl; } }; 静态成员函数
```

```
int A::total=0;
```

静态成员函数不能说明成常成员函数

```
int main( ) { A::showTotal(); 静态成员函数使用与对象无关，本质上是普通函数
```

```
    A a(10),b(20);    A::showTotal( );
```

```
    {    A c1(15),c2(-100),c3(137);    A::showTotal( ); a.showTotal( ); }
```

```
    A d(b); A::showTotal( ); return 0; }
```

注3: 静态成员函数可以通过类名使用，也可以通过对象使用。

注4: 静态成员变量是从属于类的全局变量，而静态成员函数则是从属于类的普通函数，故调用时不传递对象地址(无this指针)，从而不需要用对象来调用。但是静态成员函数从属于类，成为类的成员，故可以访问类中用private隐藏的数据。

注5: 不同于普通成员，静态成员函数不能是常成员函数。静态成员数据在常成员函数内可改动。

11.2 类的友元

11.2.1 友元函数

类的**友元**就是可以访问类中**private**成员的函数或类。

例A11.6 普通函数做友元函数。

```
class A
```

```
{ private: int x;
```

```
public:
```

```
    A(int ix=0): x(ix) { }
```

```
    void show( ) const { cout<< "x="<<x<<endl; }
```

```
    friend int sum(A a,A b) { return a.x+b.x; } 友元函数定义,仍是普通函数
```

```
    friend int difference(A a,A b); 友元函数说明
```

```
};
```

```
int main( )
```

```
{    A a(10),b(20);    const A c(20);    cout<<sum(a,b)<<endl;
```

```
    cout<<difference(a,c)<<endl;    return 0;
```

```
}
```

```
int difference(A a,A b) { return a.x-b.x; } 普通函数
```

友元函数可以直接访问
private成员

友元函数可以直接访问
private成员

注1: 友元函数一定不是该类的成员函数, 就是定义在类中也是普通函数。

注2: 在类外定义的友元函数的函数定义, 函数首部不需要再用**friend**说明。

11.2.2 ○友元类

友元类的成员函数都是相应类的友元函数。

```
class A
{
    .....
    friend class B;  B为A的友元类
};
class B  B的成员函数都能访问A的private成员
{
    .....
};
```

若有定义:

```
class A
{   private:   int x;
    public:
        A(int ix=0){x=ix;}
        void show( )
        { cout<<"x="<<x<<endl;}
    friend class B;
};
```

可以如下定义A的友元类B:

```
class B
{   private:   int y;
    public:
        B(int iy=0) { y=iy; }
        void show( ) { cout<<"y="<<y<<endl; }
        B add(A b)
        { B c; c.y=y+b.x; return c; } 可访问A类成员变量
};
```

第12章 运算符的重载

C++中运算符是一种特殊的内联函数，也可以重载，重载时必须有一个操作数是对象，重载的函数名为 `operator+`, `operator*`,。

绝大部分运算符可以重载，见下表

+	-	*	/	%	^	&		~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	>>=	<<=	==	!=
<=	>=	&&		++	--	,	->*	->
()	[]	new	delete	new[]	delete[]			

有些运算符不能重载，如： `?:` `.` `.*` `::` `sizeof`

注1：不可以重载的运算符有 `?:`、`.`、`.*`、`::`、`sizeof`。

注2：运算符可以重载，但是运算符的操作数个数、运算符的优先级和结合方向是不能改变的。

注3：重载运算符必须至少有一个操作数为对象。

12.1 类成员的运算符重载

运算符重载形式：**类成员形式**和**非成员形式**

例**C12.1** 作为类成员的运算符重载：+、-、*、/、取负“-”。
类定义

```
class Complex
{ private: double x,y;
public:
    Complex(double ix=0,double iy=0): x(ix),y(iy) { }
    void show( ) const { cout<< "("<<x<< ", "<<y<< ")"<<endl; }
    Complex operator+(Complex b) const 重载+
    { return Complex(x+b.x,y+b.y);}
    Complex operator-( ) const { return Complex(-x,-y); } 重载一元-
    Complex operator-(Complex b) const {return Complex(x-b.x,y-b.y);} 重载-
    Complex operator*(Complex b) const; 重载*
    Complex operator/(Complex b) const; 重载/
};
```


例C12.1 作为类成员的运算符重载:+、-、*、/、取负“-”。成员函数及主函数

```
Complex Complex::operator*(Complex b) const
{   return Complex(x*b.x-y*b.y, x*b.y+y*b.x); }
Complex Complex::operator/(Complex b) const
{   double t=b.x*b.x+b.y*b.y;
    if(fabs(t)<1e-14) { cout<< "Error: divided by 0"<<endl; exit(1); }
    return Complex((x*b.x+y*b.y)/t,(b.x*y-b.y*x)/t);
}

int main( )
{   Complex c1(3,4),c2(12,-7.5),c3,c4,c5,c6;
    c3=c1+c2;   等价于 c3= c1.operator+(c2)
    c4=c1-c2;   等价于 c4= c1.operator-(c2)
    c5=(-c1)*c2; 等价于 c5= (c1.operator-() ).operator*(c2)
    c6=c2/c1;    等价于 c6= c2.operator/(c1)
    c3.show(); c4.show(); c5.show(); c6.show(); return 0; }
```

exit(1)表示提前终止程序

注1: 上述程序的重载运算符可以写成函数形式, 如语句“c3=c1+c2;”可改写成“c3=c1.operator+(c2);”, 完全等价。

注2: 重载/的函数内调用exit(1)用于结束整个程序的运行。库函数exit在头文件iostream中说明。

注3: 运算符重载函数的功能必须与运算符的原意类似, 否则会造成程序语义的混乱。例如不要将+重载成实际的减法。

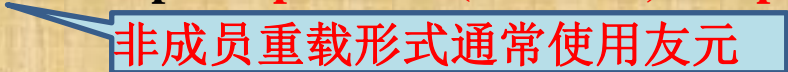
12.2 非成员的运算符重载

运算符也可以以非成员形式重载

注1：在第一操作数不是对象的情况下必须使用非成员形式的运算符重载。

例B12.2 非成员函数的运算符重载的必要性。

```
class Complex
{ private: double x,y;
public:
    Complex(double ix=0,double iy=0): x(ix),y(iy) { }
    void show() const { cout<< "("<<x<< ", "<<y<< ")"<<endl; }
    Complex operator+(double b) const { return Complex(x+b,y);} 成员形式重载
    friend Complex operator+(double a,Complex b);非成员形式重载
};
```

非成员重载形式通常使用友元

```
Complex operator+(double a,Complex b) 非成员形式运算符重载
{ return Complex(a+b.x,b.y); }
int main( )
{ Complex c1(3,4),c2,c3;
  c2=c1+15.5; 调用成员形式运算符，等价于 c2=c1.operator+(15.5)
  c3=15.5+c1; 调用非成员形式运算符，等价于 c3=operator+(15.5,c1)
  c2.show(); c3.show(); return 0; }
```

注2: 非成员形式的运算符重载可以不是友元函数。

例子程序的运算符重载可改写成:

```
class Complex
{ private: double x,y;
public:
    Complex(double ix=0,double iy=0): x(ix),y(iy) { }
    void show( ) const { cout<< "("<<x<< ", "<<y<< ")"<<endl; }
    Complex operator+(double b) const { return Complex(x+b,y);}
    double getX( ) const { return x; }
    double getY( ) const { return y; }
};
Complex operator+(double a,Complex b) 非成员形式运算符重载 (非友元)
{ return Complex(a+b.getX(), b.getY()); }
int main( )
{ Complex c1(3,4),c2,c3;
  c2=c1+15.5;   c3=15.5+c1;   ..... }
```

注3：在大部分情况下，运算符重载既可以定义为成员函数形式，也可以定义为非成员函数形式，二者任取其一，但不能同时定义，否则会出现二义性。

例B12.3 类成员与非类成员的运算符重载。

```
class A
{ private: double x,y;
public:
    A(double ix=0,double iy=0): x(ix),y(iy) { }
    void show( ) const { cout<< "("<<x<< ", "<<y<< ")"<<endl; }
    A operator+(A b) const { return A(x+b.x,y+b.y);} 成员形式运算符
};
class B
{ private: double x,y;
public:
    B(double ix=0,double iy=0): x(ix),y(iy) { }
    void show( ) const { cout<< "("<<x<< ", "<<y<< ")"<<endl; }
    friend B operator+(B a,B b) { return B(a.x+b.x,a.y+b.y);} 非成员形式运算符
};
int main( )
{ const A a1(3,4),a2(-7,-1); A a3; a3=a1+a2; a3.show();
  const B b1(3,4),b2(-7,-1); B b3; b3=b1+b2; b3.show(); return 0; }
```


注4: 运算符重载函数参数可以是引用, 但不能处理常对象参数。当常对象为第一操作数时, 成员形式的运算符重载函数必须是常成员函数, 此时*`this`为常对象。当常对象参数所占空间很大或深复制型时, 可用`const`引用参数, 以避免复制构造。

注5: 常对象参与重载运算, 重载说明(以+为例)应为 “`friend A operator+(A a,A b)`” 或 “`friend A operator+(const A&a,const A &b)`”, 或成员形式 “`A operator+(A b) const`” 或 “`A operator+(const A &b) const`”。

注6: 有些C++编译软件将临时对象等同于常对象, 不能进行引用。此时连续加操作如 “`oa+ob+oc`” 就要求第一操作数能处理常对象。

12.3 赋值运算符的重载

当对象进行赋值时，默认为字节复制，即浅复制。

注1：如果对象是深复制型的，数据在指针间接指向的其它空间中，此时对象进行赋值时需要进行深复制，这就需要进行赋值运算符的重载。

注2：在C++中规定赋值运算符的重载必须是类成员形式运算符。

```
class A
{ .....
public:
    .....
    A &operator=(const A &b); 重载赋值运算符,函数值一般为类的引用A&
    .....
};
A &A::operator=(const A &b)
{ .....      复制操作，一般进行深复制
  return *this;      返回对象本身
}
```

注3：在重载赋值运算符时，函数值一般为类的引用，这样赋值表达式的值就是左值。

例B12.4 演示深复制的赋值运算符重载的必要性。 有错程序

```
class String
{ private: char *p;
public:
    String(const char *s= "") { p=new char[strlen(s)+1]; strcpy(p,s); }
    String(const String &b) { p=new char[strlen(b.p)+1]; strcpy(p,b.p); }
    ~String( ) { delete[ ]p; }
    char &elem(int n) { return p[n]; }
    const char &elem(int n) const { return p[n]; } 用于常对象
    void show( ) const { cout<<p<<endl; }
// String &operator=(const String &b)
// {if(this!=&b){delete[ ]p; p=new char[strlen(b.p)+1]; strcpy(p,b.p);}return *this; }
};
String toUpper(const String &b)
{ String t(b); ..... return t; } 复制构造 深复制
int main( )
{ String s1("Hello"),s2("World"); s1.show(); s2.show();
  s2=toUpper(s1); 对象赋值 浅复制
  s1.show(); s2.show(); return 0;
}
```

赋值重载后，运行将正确

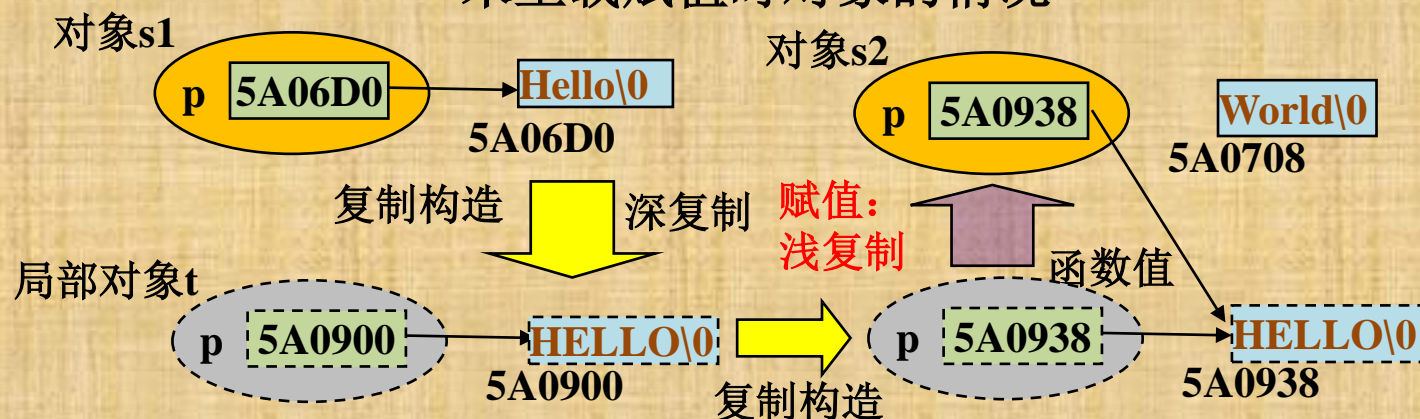
运行结果：

Hello
World
Hello
葦葦葦葦葦葦葦葦

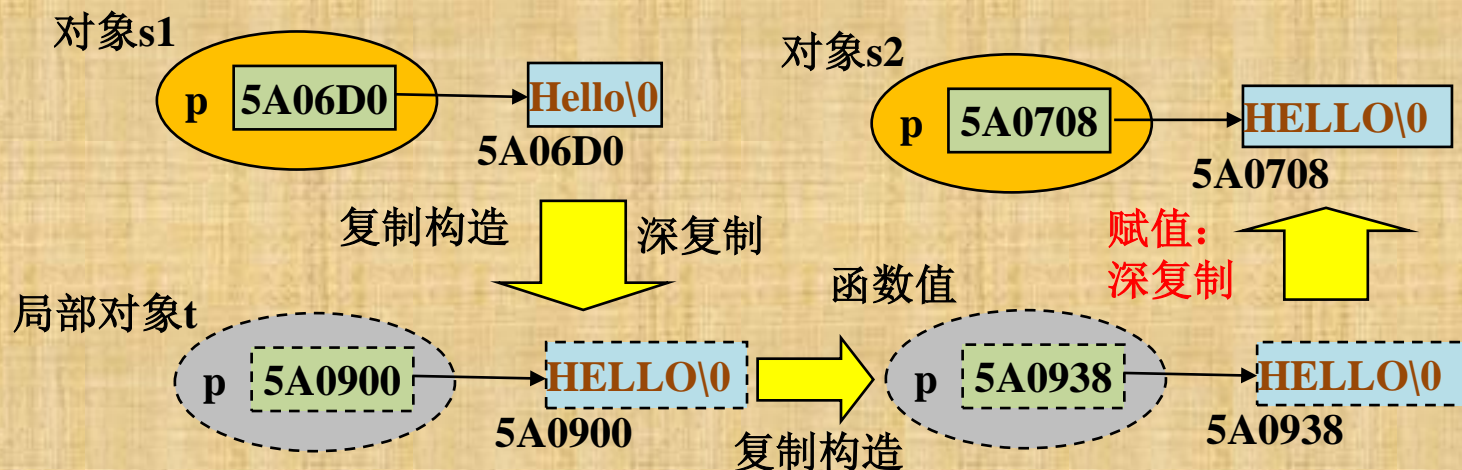
浅复制赋值产生的后果

例B12.4赋值重载前后运行时对象的变化情况图示。

未重载赋值时对象的情况



重载赋值后对象的情况



例C12.5 String类的设计。 类的定义

```
class String
```

```
{
```

```
private:
```

```
    char *p;
```

```
public:
```

```
    String(const char *s= "" ) { p=new char[strlen(s)+1]; strcpy(p,s); }    构造函数
```

```
    String(const String &b) { p=new char[strlen(b.p)+1]; strcpy(p,b.p); } 复制构造函数
```

```
    ~String( ) { delete[ ] p; }
```

```
    String &operator=(const String &b);    重载赋值
```

```
    friend String operator+(const String &a,const String &b);    重载+: 合并字符串
```

```
    friend String operator-(const String &a,const String &b);    重载-: 删去第一个子串
```

```
    friend bool operator==(const String &a,const String &b)    字符串==
```

```
        {return strcmp(a.p,b.p)==0; }
```

```
    friend bool operator!=(const String &a,const String &b)    字符串!=
```

```
        {return strcmp(a.p,b.p)!=0; }
```

```
    friend bool operator>(const String &a,const String &b) {return strcmp(a.p,b.p)>0;} >
```

```
    friend bool operator<(const String &a,const String &b) {return strcmp(a.p,b.p)<0;} <
```

```
    char &elem(int n) { return p[n]; }
```

```
    const char &elem(int n) const { return p[n]; }
```

```
    void show( ) const { cout<<p<<endl; }
```

```
};
```


例C12.5 String类的设计。 类成员函数的定义及主函数

```
String &String::operator=(const String &b)
{ if(this!=&b){delete[ ]p; p=new char[strlen(b.p)+1]; strcpy(p,b.p);} return *this; }
String operator+(const String &a,const String &b)
{ char s[200]; strcpy(s,a.p); strcat(s,b.p); return String(s); }
String operator-(const String &a,const String &b)
{ char s[200],*p; strcpy(s,a.p);
  if((p=strstr(s,b.p))!=0) strcpy(p,p+strlen(b.p));
  return String(s);
}
int main( )
{ const String s1("Hello"); String s2("World"),s3,s4;
  s1.show(); s2.show(); s3=s1+s2; s3.show();
  if(s3-s2==s1) cout<< "s3-s2==s1"<<endl;
  else cout<< "s3-s2!=s1"<<endl;
  s4=s3-String("He"); s4.show();
  for(int i=0;s1.elem(i)!='\0';i++) s4.elem(i)=s1.elem(i); s4.show(); return 0;
}
```

注4：赋值的重载必须是成员函数形式，但是复合赋值运算符的重载两种形式都可以。

注5：String类为深复制型，运算符重载函数参数用const引用较好。

12.4 ○转换函数

通过构造函数可以将其它类型数据转换成对象，
而通过**转换函数**则可以将**对象转换成其它类型**。

转换函数在类内定义，是成员函数，形式如下：

```
class A
{
    ... ..
    operator double( ) const { return sqrt(x*x+y*y); } 转换函数
};
```



A转换成 double

注1：类中定义了转换函数，对象就可以进行对应的显式或隐式转换，如 (double)c1、double(c1)或static_cast<double>(c1)。

例A12.6 转换函数的作用。 提供了一种自动转换的途径

```
class A
{ private:
    double x,y;
public: A(double ix=0,double iy=0): x(ix),y(iy) { }
    void show( ) const { cout<< "("<<x<< ", "<<y<< ")"<<endl; }
    operator double( ) const { return sqrt(x*x+y*y); }    转换函数
};
int main( )
{ A a1(3,4),a2(-7,2),a3; double d1,d2,d3,s;
  d1=a1; d2=(double)a1; d3=double(a1);
  cout<< "d1="<<d1<< ",d2="<<d2<< ",d3="<<d3<<endl;
  s=a1+a2;    等价于 s=double(a1)+double(a2);
  a3=a1+a2;    等价于 a3=A(double(a1)+double(a2));
  cout<< "s="<<s<<endl; a3.show(); return 0; }
```

注2: 转换函数不加返回类型，其实返回类型就是转换的类型。

注3: 运算符的重载和转换函数都定义时，如果程序中出现了相应的运算，则首先选择调用运算符重载，若没有定义运算符重载，如果可以转换后再进行运算，则采取转换后运算方式进行。

12.5 特殊的运算符重载

本节只介绍特殊的运算符 `<<`、`>>` 的重载

12.5.2 重载`<<`和`>>`

我们也可以使用`cout<<a`和`cin>>a`这样简单的形式输入输出对象，这就需要重载`<<`和`>>`。

<code>cout<<a</code>	左移运算符 <code><<</code> 的重载，称为 插入运算符
<code>cin>>a</code>	右移运算符 <code>>></code> 的重载，称为 提取算符

`cout<<a`实际等价于 **`cout.operator<<(a)`** 返回`cout`本身

`cin>>a`实际等价于 **`cin.operator>>(a)`** 返回`cin`本身

其中

- `cout`是`ostream`类的对象，`ostream`是专门用于输出的类。
`cout.operator<<(a)`返回`cout`本身,故可以连续输出如:`cout<<a<<b`
即等价于 **`(cout.operator<<(a)).operator<<(b)`**
- `cin`则是`istream`类的对象，而`istream`是专门用于输入的类。
`cin`和`cout`在头文件`iostream`中定义。

<<和>>通常以友元的方式进行重载。
重载形式如下：

```
class A
{
    ... ..
    friend ostream &operator<<(ostream &os,A b);
    friend istream &operator>>(istream &is,A &b);
};

ostream &operator<<(ostream &os,A b) { os<<x<<y;..... return os; }
istream &operator>>(istream &is,A &b) { ..... return is; }
```

第二操作数
用A b 或者 const A &b
若用A &b则不能输出常对象

使用时：

A ob,oc;

cin>>ob; 等价于 operator>>(cin,ob) 函数内部is即为cin

cout<<ob<<oc;等价于 operator<<(operator<<(cout,ob),oc)

函数内部os即为cout, operator<<(cout,ob)返回操作完成后的cout, 故也可以看成

operator<<(cout,ob), operator<<(cout,oc);

注3：在重载<<和>>时，第一操作数和函数值都必须是引用；在重载>>时，第二操作数也必须是引用。

例A12.10 重载<<和>>。

```
class Point
{ private: double x,y;
public:
    Point(double px=0,double py=0): x(px),y(py) { }
    double distance(Point b) const;
    friend istream &operator>>(istream &is,Point &b); 重载>>
    friend ostream &operator<<(ostream &os,Point b); 重载<<
};

double Point::distance(Point b) const
{ return sqrt((x-b.x)*(x-b.x)+(y-b.y)*(y-b.y)); }

istream &operator>>(istream &is,Point &b) 重载>>
{ is>>b.x>>b.y; return is; }

ostream &operator<<(ostream &os,Point b) 重载<<
{ os<< "("<<b.x<< ","<<b.y<< ")"<<endl; return os; }

int main( )
{ Point a,b ;
  cin>>a>>b; 等价于 operator>>(operator>>(cin,a),b )
  cout<< a<<b <<"两点距离为: "<<a.distance(b)<<endl ; return 0;
}
```


12.6 string类的使用

C++中预先定义了**string**类来处理字符串，在头文件**string**中定义。

注1: **string**对象可以直接赋值、复制构造、合并(+和+=)和比较，可以与C风格的字符串相加和直接比较。**string**类定义在std名空间中。

例A12.15 使用**string**类处理字符串。

```
#include <iostream>
```

```
#include <string>
```

使用**string**类需要加上此行

```
using namespace std;
```

```
void sort(string s[ ],int n)
```

```
{ int i,j,k;
```

```
  for(i=0;i<n;i++)
```

```
  {for(k=i,j=i+1;j<n;j++)if(s[j]<s[k])k=j;if(k!=i)s[k].swap(s[i]); } s[k]与s[i]交换
```

```
int main( )
```

```
{ string str[100],mid; int n,i; 等价于 std::string str[100],mid;
```

```
  cin>>n; for(i=0;i<n;i++) cin>>str[i]; 输入单词,输入整行用getline函数
```

```
  sort(str,n); for(i=0;i<n;i++) cout<<str[i]<<endl;
```

```
  mid=str[n/2]; cout<< "中间的字符串为: "<<mid<<endl; return 0; }
```

一些string对象的操作

若有定义“`string s1,s2;`”，则有以下一些常用函数

- **getline(cin,s1)** 输入一行字符串，以回车为结束符
- **s1.empty()** 判定s1是否为空串
- **s1.size(), s1.length()** 求s1中字符串的长度
- **s1[n]** s1中字符串的n号(字符从0号开始)字符
- **s1.erase(pos,n)** 删除从下标pos开始的n个字符
- **s1.insert(pos,s2)** 在下标pos之前插入字符串s2
- **s1.substr(pos,n)** 从下标pos开始的n个字符构成的字符串
- **s1.find(s2,pos)** 从下标pos开始查找第一个子串s2，找到返回下标，找不到返回**string::npos** (即**0x7fffffff**)
- **s1.replace(pos, n,s2)** 将从下标pos处开始的n个字符用串s2替换
- **s1.c_str()** 将s1转换为C风格字符串，返回**const char***指针

相关名称: **string::size_type** 即**unsigned int** ; **string::npos**即**0x7fffffff**

例C12.16 string类中的子串处理。

```
# include <string>
bool checkBeginEnd(const string &s)
{  string wB("begin"),wE("end");  string::size_type kBegin,kEnd;
  for(kBegin=kEnd=0; ; kBegin++,kEnd++)
  {  kBegin=s.find (wB,kBegin);  kEnd=s.find (wE,kEnd);
    if(kBegin==string::npos&& kEnd==string::npos) return true;
    else if(kBegin==string::npos| |kEnd==string::npos) return false;
    else if(kBegin>kEnd) return false;  }
}

void replaceBrackets(string &s)
{  string wB("begin"),wE("end"),wLeft("{"),wRight("}");
  string::size_type kBegin,kEnd;  kBegin=kEnd=0;
  while(true) {  kBegin=s.find (wB,kBegin); if(kBegin==string::npos) break;
    s.replace(kBegin,wB.length(),wLeft);  }
  while(true) {  kEnd=s.find (wE,kEnd); if(kEnd==string::npos) break;
    s.replace(kEnd,wE.length(),wRight);  }
}

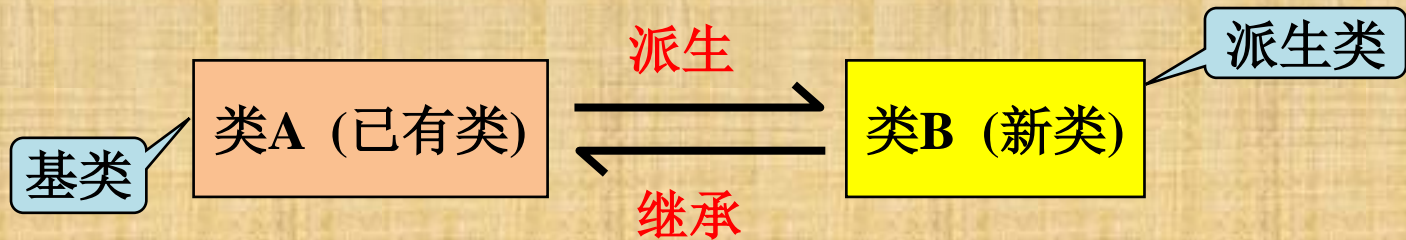
int main( )
{  string str;  getline(cin,str) ;
  if(checkBeginEnd(str)) { replaceBrackets(str); cout<<str<<endl;} .....
}
```

string::size_type即unsigned int

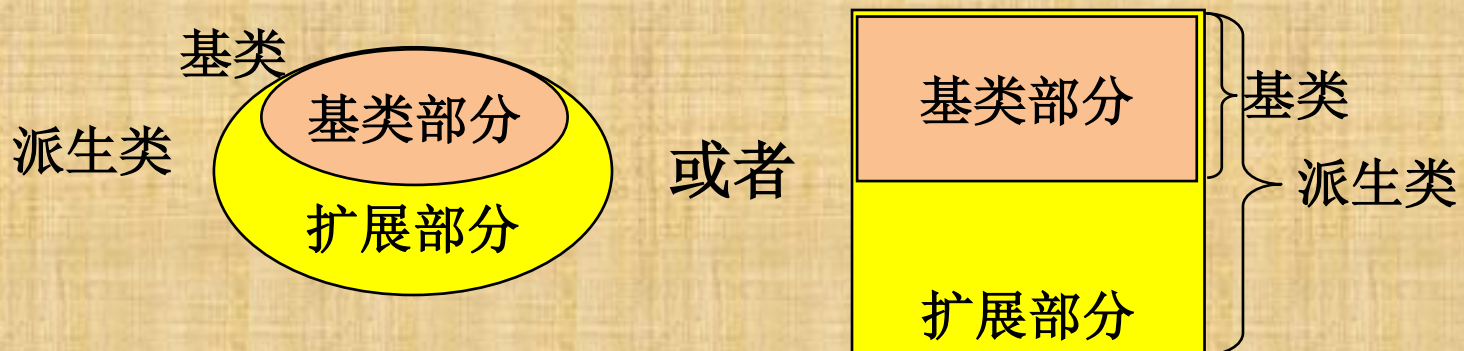
第13章 类的继承性

- **派生**：从已有类产生新类
- **继承**：新类包含了已有类

注1：派生和继承指的是同一件事不同的角度。

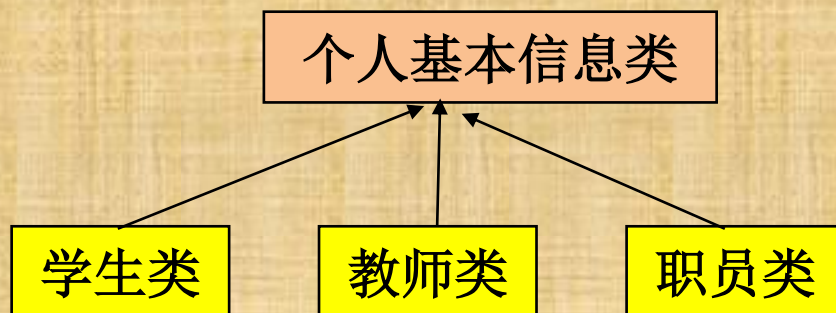


派生类的结构图示：

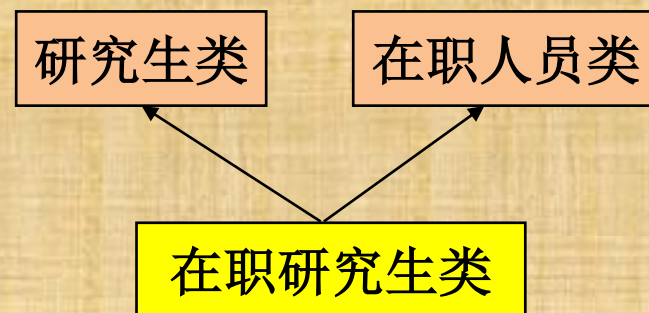


继承分为 {
• 单一继承：即单一派生
• 多重继承：即多重派生

单一继承：单个基类



多重继承：多个基类



13.1 单一派生

13.1.1 派生类的定义

基类A派生出派生类B的形式:

```
class B: public A    派生类B的定义, B继承了A
{
    ... ..
};
```


例A13.1 派生类的使用与派生类的成员。

```
class A
{ private: int x;
public:
    A(int ix=0):x(ix) { }
    void show() const {cout<<"x="<<x<<endl;}
};

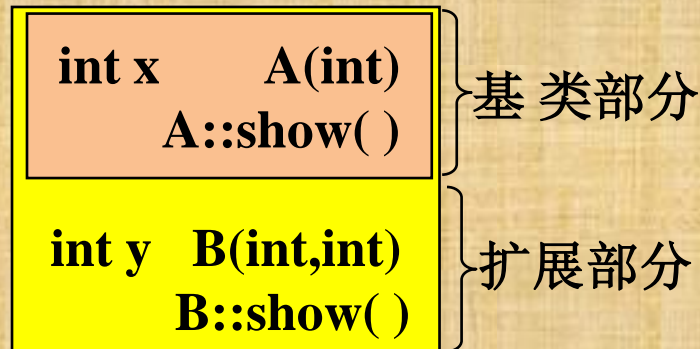
class B: public A    派生类定义
{ private: int y;
public:
    B(int ix=0,int iy=0):A(ix),y(iy) { }
    void show() const { A::show(); cout<<"y="<<y<<endl; }
};

int main( )
{ A a(10); B b(100,200);
  a.show(); b.show(); return 0;
}
```

先要构造基类部分: A(ix)

B::show不能访问基类的private成员

派生类结构图示



注1: 基类部分的构造只能在派生类的构造函数的初始化列表中构造。

注2: 派生类中扩展的成员函数不能访问基类的private成员, 但能访问基类的protected成员, 而外界不能访问protected的成员。

例A13.2 protected权限的用法。

```
class A
{ protected: int x;
public:
    A(int ix=0):x(ix) { }
    void show() const { cout<< "x="<<x<<endl; }
};

class B: public A
{
private:
    int y;
public:
    B(int ix=0,int iy=0):A(ix),y(iy) { }
    void show() const { cout<<"x="<<x<<"", y="<<y<<endl; }
};

int main( )
{ A a(10); B b(100,200);
  a.show(); b.show(); return 0;
}
```

B::show可以访问基类的**protected**成员

13.1.2 公有派生、私有派生和保护派生

前述例子派生类的继承说明为

`class B: public A` 为公有派生

继承说明也可以是`private`和`protected`的，即

`class B: private A` 私有派生

或者 `class B: protected A` 保护派生

不同派生形式下派生类继承成员的访问权限如下表

派生类权限	public派生	protected派生	private派生
基类private成员	不可访问	不可访问	不可访问
基类protected成员	protected	protected	private
基类public成员	public	protected	private

注3：派生方式可以是公有派生、私有派生和保护派生，若没有具体指出派生方式，则默认为私有派生。

例B13.3 公有派生中继承成员的访问权限。

class A

{ **private:** int x1;

protected: int x2;

public: int x3;

A(int ix1=0,int ix2=0,int ix3=0):x1(ix1),x2(ix2),x3(ix3) { }

int getx1() const { return x1; }

void show() const { cout<<x1<<","<<x2<<","<<x3<<endl; }

};

class B: **public A** 派生类B内: x1不可访问, x2为protected, x3为public

{ **public:**

B(int ix1=0,int ix2=0,int ix3=0):A(ix1,ix2,ix3) { }

int getx2() const { return x2; }

void showB() const;

};

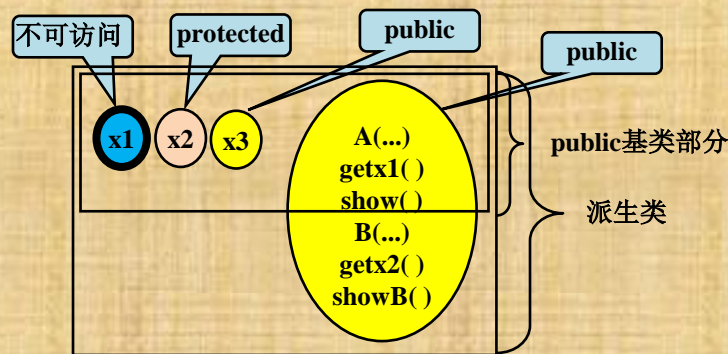
void B::showB() const { cout<<**getx1()**; cout<<","<<x2<<","<<x3<<endl; }

int main()

{ B b(100,200,300); b.show(); b.showB();

cout<<**b.getx1()**<< ","<<**b.getx2()**; //cout<<b.x1<<","<<b.x2;错误

cout<< ","<<b.x3<<endl; return 0; }



cout<<x1;错误, 因为x1不可访问

13.1.3 重名的优先级和隐藏

注5: 当派生类中有重名成员变量时, 内层(派生层次, 基类为外层)定义的变量名隐藏外层的变量名。当派生类中有重名成员函数时, 也是内层函数隐藏外层的函数, 不过此时函数指的是成员函数集(重载函数集)。

注6: 可以通过作用域运算符显式指定所属类来指定所用的成员变量或成员函数。

例A13.6 成员变量与成员函数的重名与标识。

```
class A
{ public:  int x;
      A(int ix=0):x(ix) { }
      void show() const { cout<<"x="<<x<<endl; }
};

class B: public A    成员有: A::x、B::x、A::show()、B::show()
{ public:  int x;    隐藏了基类的x
      B(int ix=0,int iy=0):A(ix),x(iy) { }
      void show( ) const { cout<<A::x<<","<<x<<endl; } 隐藏了基类的show
};

int main( )
{   B b(100,200);    b.A::show(); b.B::show(); b.show();
    cout<<b.A::x<< ","<<b.B::x<< ","<<b.x<<endl; return 0;
}
```

基类的x

派生类的x

基类的show()

派生类的show()

13.2 派生类的构造与析构

派生类由两部分组成，依次是基类部分和扩展部分。派生类对象在内存中，基类部分占据前面空间，扩展部分依次占据后面空间。故派生类对象构造和析构过程如下：

- 构造：先构造基类部分，再按定义依次构造扩展成员，最后执行构造函数函数体
- 析构：先执行析构函数函数体，再反向依次析构扩展成员，最后析构基类部分

例B13.8 复杂派生类的构造与析构。

class A

{ protected: int x;

public:

A(int ix=0):x(ix) { cout<< "A():x="<<x<<endl; }

~A() { cout<< "~A():x="<<x<<endl; }

int getx() const { return x; }

void show() const { cout<<"x="<<x<<endl; }

};

class B: public A

{ private: int y; A oa;

public:

B(int ix=0,int iy=0,int iz=0):A(ix),y(iy),oa(iz){cout<<"B():y="<<y<<endl;}

~B() { cout<< "~B():y="<<y<<endl; }

void show() const { cout<<x<<","<<y<<","<<oa.getx()<<endl; }

};

int main() { B b(100,200,300); b.A::show(); b.show(); return 0; }

依次构造A类、y、oa

运行结果:

A():x=100 构造基类部分

A():x=300 构造 oa

B():y=200 执行函数体

x=100

100,200,300

~B():y=200 执行函数体

~A():x=300 析构 oa

~A():x=100 析构基类部分

注1: 派生类的构造由构造函数的初始化列表给出, 基类部分用基类名构造、扩展的对象类成员用成员名构造, 没有列出的调用默认构造函数构造。实际构造的次序与初始化列表中的次序无关。

例B13.9 基类含默认构造函数的派生类的构造与析构。

class A

{ protected: int x;

public: A(): x(0) { cout<< "A()"<<endl; }

A(int ix):x(ix) { cout<< "A():x="<<x<<endl; }

~A() { cout<< "~A():x="<<x<<endl; }

void show() const { cout<<"x="<<x<<endl; } };

class B: public A

{ private: int y;

public:

B(): y(0) 默认构造基类部分,等价于 B():A(),y(0)

{ cout<< "B()"<<endl; }

B(int ix,int iy=0):A(ix),y(iy)

{cout<<"B():y="<<y<<endl;} 显式构造基类部分

~B() { cout<< "~B():y="<<y<<endl; }

void show() const { cout<<x<<" "<<y<<endl; }

};

class C: public A

{ public: void show() const { cout<<"C::show x="<<x<<endl; } };

int main() { B a,b(100,200); C c; a.show(); b.show(); c.show(); return 0; }

运行结果:

A() 默认构造a基类部分

B()

A():x=100 显式构造b基类部分

B():y=200

A() 默认构造c基类部分

0,0

100,200

C::show x=0

~A():x=0

析构c

~B():y=200

析构b

~A():x=100

~B():y=0

析构a

~A():x=0

13.3 多重派生

多重派生就是继承了多个基类的派生。多重派生类的对象的各个部分在内存中所占据的位置由定义确定。

派生类对象构造时以定义次序来依次构造各个基类部分和扩展部分，对象的各个组成部分确定后，开始执行构造函数函数体。具体过程如下：

- 按定义次序依次构造基类部分
- 按定义次序依次构造派生类的扩展部分
- 执行派生类的构造函数体

多重派生类对象析构时则以构造的相反次序进行析构。

例A13.10 多重派生及其构造和析构。

基类定义

```
class A1
{
protected: int x1;
public:
    A1(): x1(0) { cout<< "A1()"<<endl; }
    A1(int ix):x1(ix) { cout<< "A1():x1="<<x1<<endl; }
    ~A1() { cout<< "~A1():x1="<<x1<<endl; }
    void show1() const { cout<<"x1="<<x1<<endl; }
};

class A2
{
protected: int x2;
public:
    A2(): x2(0) { cout<< "A2()"<<endl; }
    A2(int ix):x2(ix) { cout<< "A2():x2="<<x2<<endl; }
    ~A2() { cout<< "~A2():x2="<<x2<<endl; }
    void show2() const { cout<<"x2="<<x2<<endl; }
};
```

例A13.10 多重派生及其构造和析构。

派生类定义及主函数

class B: public A1,public A2 多重派生

```
{
private:
    int y;
public:
    B(): y(0) { cout<< "B()"<<endl; }
    B(int ix1,int ix2,int iy):A1(ix1),A2(ix2),y(iy)
        { cout<<"B():y="<<y<<endl;}
    ~B() { cout<< "~B():y="<<y<<endl; }
    void show( ) const {cout<<x1<<","<<x2<<","<<y<<endl;}
};

int main( )
{   B a,b(100,200,300);
    a.show(); b.show(); b.show1(); b.show2();
    return 0;
}
```

构造次序:
A1部分、A2部分、y

运行结果:

```
A1()  构造a基类1
A2()  构造a基类2
B()
A1():x1=100 b基类1
A2():x2=200 b基类2
B():y=300
0,0,0      输出a
100,200,300 输出b
x1=100
x2=200
~B():y=300 析构b
~A2():x2=200
~A1():x1=100
~B():y=0    析构a
~A2():x2=0
~A1():x1=0
```

注1: 多重派生的派生类构造时是按照定义的次序依次构造各个基类和扩展的成员数据, 与构造函数的初始化列表中的次序无关。

13.4 重名的使用规则

多重派生时，派生类的重名成员情况更加复杂，可能出现重名冲突，此时需要利用所属基类进行区别。

各种情况下，派生类重名成员使用规则如下：

- 不同层次可以定义同名的成员，深层次的名字优先级高，浅层次的名字优先级低
- 优先级高的名字隐藏优先级低的名字
- 在同一个类内只可以定义多个同名的重载函数
- 同一层次的不同类可以定义同名的成员，它们有相同的优先级，故使用时必须加上所属类

例A13.11 各不同优先级的重名的使用。

```
class A1
{ protected: int x;
public: A1(int ix=0):x(ix) { } void show() const { cout<<"A1::x="<<x<<endl; }
};
class A2
{ protected: int x;
public: A2(int ix=0):x(ix) { } void show() const { cout<<"A2::x="<<x<<endl; }
};
class B: public A1, public A2  所含成员变量优先级 A1::x = A2::x
{ private: int y;           所含成员函数优先级 A1::show() = A2::show() < B::show()
public: B(int ix1=0,int ix2=0,int iy=0): A1(ix1),A2(ix2),y(iy){ }
void show( ) const { cout<<A1::x<<","<<A2::x<< ","<<y<<endl; }
};
class C: public A1  所含成员函数优先级 A1::show() < C::show(int)
{ public: C(int ix=0): A1(ix){ } void show(int iy) const {cout<<x<< "\t"<<iy<<endl; }};
int main( )
{ B b(100,200,300); C c(128);
  b.A1::show(); b.A2::show(); b.B::show(); b.show();
  c.show(64); c.A1::show(); // c.show(); 错误, 因为show()被show(int)隐藏
  ..... }
```

指明所属的类

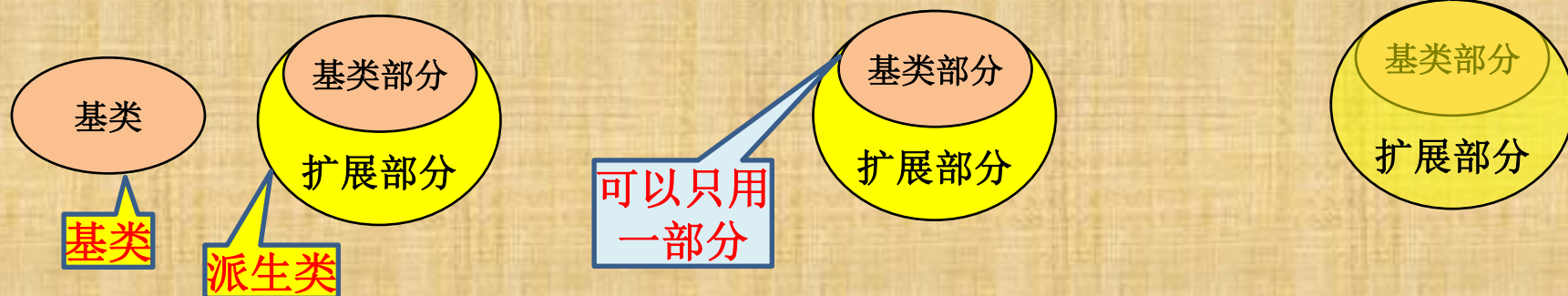
派生类B的show()

13.5 派生类的类型兼容

派生类向自身及继承链上的基类类型兼容

派生类类型兼容是单向的。具体表现为：

- 派生类对象可以赋值给基类对象
- 派生类对象可以作为基类的引用
- 基类指针可以指向派生类对象



注1：派生类对象可以作为基类对象来用，反之则不行。

例A13.12 派生类对象与基类对象的兼容性。

```
class A
{ protected:  int x;
public:  A(int ix=0):x(ix){ }  void show() const { cout<<"x="<<x<<endl; }
};

class B: public A
{ private:  int y;
public:  B(int ix=0,int iy=0): A(ix),y(iy){ }
        void show( ) const { cout<<x<< ", "<<y<<endl; }
};

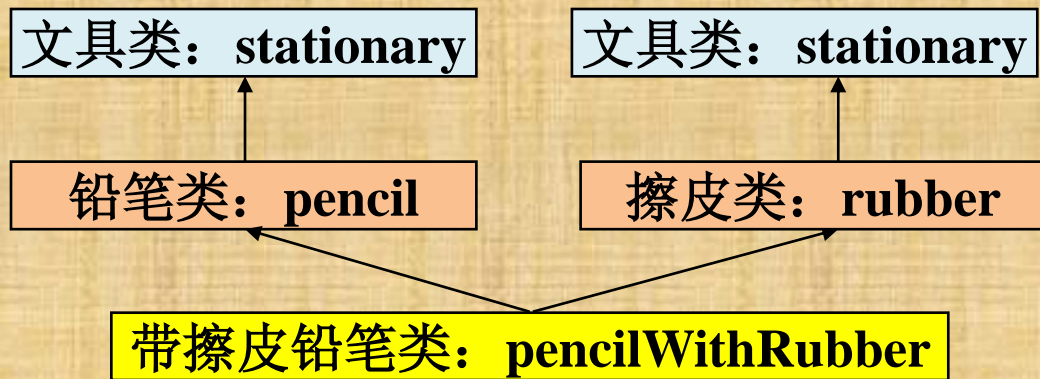
void f(A oa) { oa.show(); }
void g(A *p) { p->show(); }
void h(A &r) { r.show(); }

int main( )
{  A a(20);  B b(100,200); a.show(); b.show();
  a=b; a.show(); f(b);
  A c(50),*pc=&c; B d(10,20),*pd=&d; pc->show(); pd->show();
  pc=&d; pc->show(); g(&d);
  B u(-1,-2); A &w=u; u.show(); w.show(); h(u); return 0; }
```

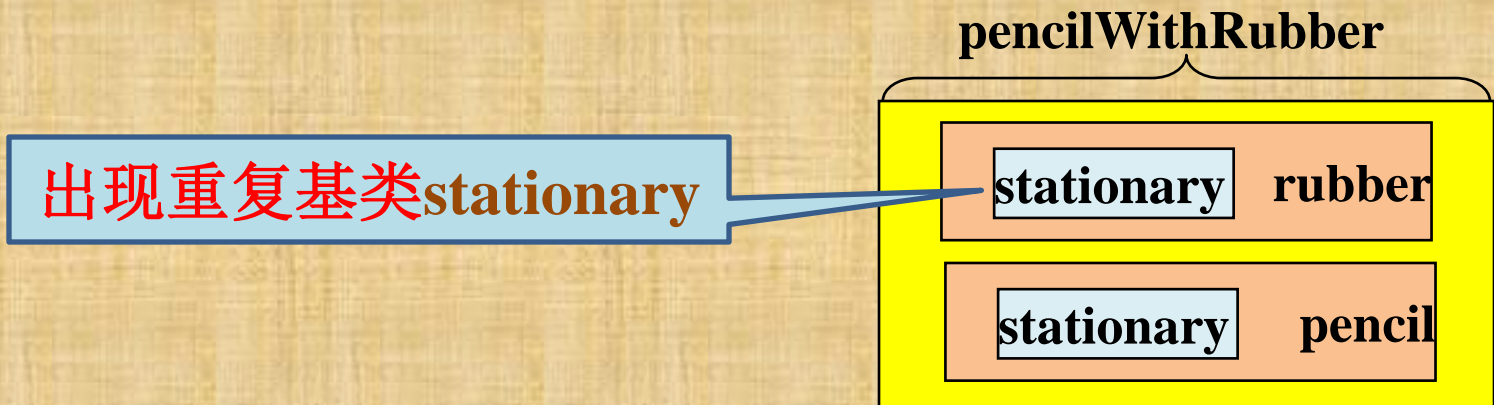

13.6 虚基类及复杂类的构造与析构

多重派生有可能导致基类重复问题，见如下例子：

带橡皮铅笔类的继承关系

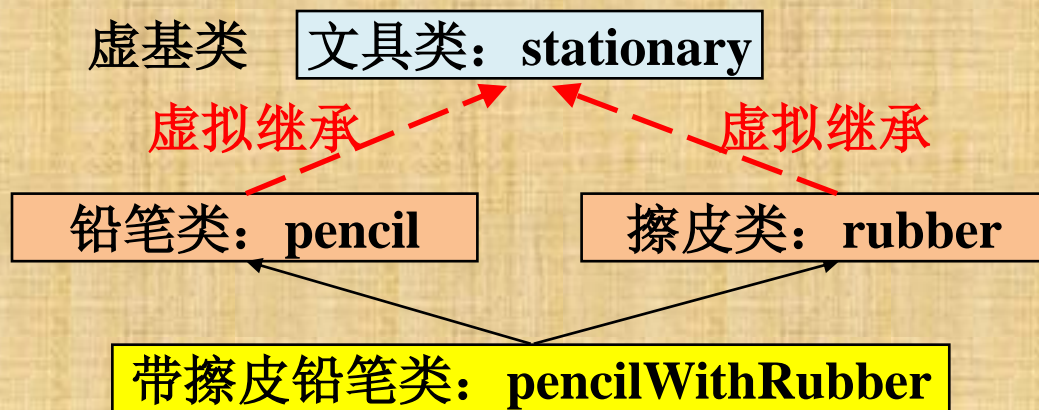


派生类对象的实际组成

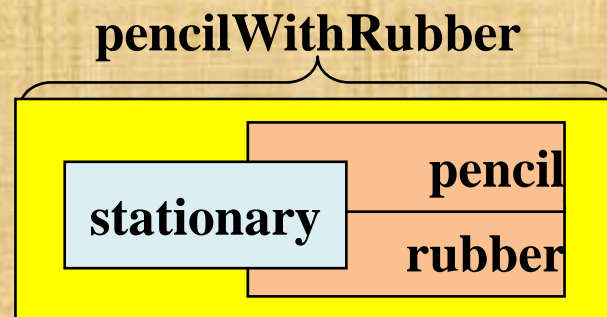


为了解决重复基类的问题，可以进行虚基类说明，如下图：

带橡皮铅笔类的继承关系2



派生类对象的实际组成



注1：继承列表中用virtual说明的基类是虚基类，任何类中的虚基类只有一份。

注2：派生类的虚基类在后续的派生中始终是虚基类。

注3：任何含虚基类的派生类对象在构造时都要先构造虚基类，就是间接继承的虚基类也要先构造，再依次构造基类。

例A13.15 虚基类的使用。

```
class stationery
{ protected:    float price;
public:    stationery(float p=0) : price(p) { }
    void setPrice(float p){ price=p; } float getPrice() const { return price; } };

class pencil : virtual public stationery    含虚基类的派生类
{ public:    pencil(float pri) : stationery(pri) { }
    void writing( ) const { cout<< "writing"<<endl; }
};

class rubber : virtual public stationery    含虚基类的派生类
{ public:    rubber(float pri) : stationery(pri) { }
    void erasing( ) const { cout<<"erasing"<<endl; }
};

class pencilWithRubber : public pencil, public rubber    间接含虚基类 stationery
{public:pencilWithRubber(float pri) :stationary(pri),pencil(pri), rubber(pri){ } };

int main( )
{    pencilWithRubber t(3);    t.writing(); t.erasing();
    t.setPrice(4.5);    cout<<t.getPrice()<<endl;
    t.setPrice(2.8);    cout<<t.getPrice()<<endl;    ..... }
```

只含一份 price

需构造虚基类

○包含虚基类、普通基类及扩展成员含对象的复杂派生类，构造时**最先构造虚基类**，然后如前面所讲的方式构造其它部分，最后执行函数体。具体见下面过程：

- 按定义次序依次构造虚基类部分 (包括直接和间接虚基类)
- 按定义次序依次构造普通基类部分
- 按定义次序依次构造派生类的扩展部分
- 执行构造函数体

○派生类对象析构时则是以构造的相反次序进行析构。

- 执行析构函数体
- 按定义的逆序依次析构派生类的扩展部分
- 按定义的逆序依次析构普通基类部分
- 按定义的逆序依次析构虚基类部分

第14章 类的多态性

类的多态性：运行时由实际的对象决定所调用的成员函数，
即基类对象调用的是基类的同名成员函数，
派生类对象调用的是派生类的同名成员函数。

类的多态性可以使我们对于派生链上的各种对象
用基类指针编写统一形式的代码。

14.1 虚函数

在C++中用虚函数实现类的多态性。虚函数体现的多态性见下面两个相似的程序例子。

例A14.1 基类指针和基类引用使用派生类。

```
class A
```

```
{ protected: int x;
```

```
public:
```

```
    A(int ix=0):x(ix) { }
```

```
    void show() const { cout<<"x="<<x<<endl; }
```

```
};
```

```
class B: public A
```

```
{ private: int y;
```

```
public:
```

```
    B(int ix=0,int iy=0): A(ix),y(iy){ }
```

```
    void show( ) const { cout<<"x="<<x<<" ,y="<<y<<endl; }
```

```
};
```

```
int main( )
```

```
{  A *pa; B b(100,200),*pb=&b;
```

```
    b.show(); pb->show();
```

```
    pa=&b; pa->show(); 等价于(*pa).show()，实际使用a的基类部分
```

```
    A &r=b; r.show(); r为a的基类部分的名字，使用r就是使用a的基类部分
```

```
    ..... }
```

运行结果:

x=100,y=200

x=100,y=200

x=100 由基类的show()输出

x=100 由基类的show()输出

例A14.2 虚函数的使用。

```
class A
{ protected: int x;
public:
    A(int ix=0):x(ix) { }
    virtual void show() const { cout<<"x="<<x<<endl; }
};

class B: public A
{ private: int y;
public:
    B(int ix=0,int iy=0): A(ix),y(iy){ }
    void show() const {cout<<"x="<<x<<"y="<<y<<endl; } 重写基类show()
};

int main( )
{  A *pa; B b(100,200),*pb=&b;
  b.show(); pb->show();
  pa=&b; pa->show(); 指针调用虚函数,实际执行的函数由对象确定
  A &r=b; r.show(); 引用调用虚函数,实际执行的函数由对象确定
  ..... }
```

运行结果:

x=100,y=200

x=100,y=200

x=100,y=200

x=100,y=200

由实际对象的show()输出
由实际对象的show()输出

show为虚函数

体现了多态性

体现了多态性

- 注1：指针或引用调用虚函数，则实际执行的函数由对象决定，什么类型的对象就执行什么类型的函数，这就是多态性。
- 注2：当A类中将show()说明成了虚函数后，派生类B中可以重新定义show()，此时前面可以不再加virtual说明，show()在派生链中永远是虚函数。当然，也可以在B的show()前面加上virtual，效果一样。若B类中未重新定义show()，则B类对象只有一个show()，即A::show()也是B::show()。
- 注3：一般来说在派生类中要对虚函数进行重新定义，定义的函数必须与基类中的函数的首部完全一样，但是有一个函数例外，那就是析构函数。析构函数可以是虚函数，而析构函数名是确定的，基类和派生类的析构函数名不会相同。但是基类的析构函数是虚函数，则基类指针对派生类对象来说析构时将调用派生类的析构函数。

例B14.3 析构造函数为虚函数。

```
class A
{ protected: int x;
public:
    A(int ix=0):x(ix) { }
    virtual ~A() { cout<< "virtual ~A()"<<endl; }    虚析构造函数
    virtual void show() const { cout<<"x="<<x<<endl; }
};

class B: public A
{ protected: int y;
public:
    B(int ix=0,int iy=0): A(ix),y(iy){ }
    ~B() { cout<< "~B()"<<endl; }
    void show() const { cout<<"x="<<x<< ",y="<<y<<endl; }
};

int main()
{ A *p; p=new B(100,200); p->show();
  delete p;    将自动调用虚析构造函数
  ..... }
```

运行结果:

体现了多态性

x=100,y=200

~B() 实际调用派生类的析构造函数
virtual ~A()

可以利用虚函数体现的多态性用基类指针或基类引用来统一处理派生链上的各种对象。

例**C14.4** 虚函数用于统一处理不同层次的派生类。 类的设计

```
class triangle    三角形类
{ protected: double height,bottom;
public: triangle(double h=0,double b=0) : height(h),bottom(b) { }
    virtual ~triangle() { }
    virtual void show() const
        {cout<<"三角形:底="<<bottom<<","高="<<height<<endl;}
};

class trapezoid: public triangle    梯形类
{ private: double top;
public: trapezoid(double h=0,double t=0,double b=0):triangle(h,b),top(t) { }
    ~trapezoid() { }
    void show( ) const;
};

void trapezoid::show( ) const
{ cout<<"梯形:上底="<<top<<","下底="<<bottom<<","高="<<height<<endl; }
```

例C14.4 虚函数用于统一处理不同层次的派生类。 类的使用

```
void input(triangle **p)
{  string s; double h,t,b;
  for(cin>>s;s!="over"; cin>>s)
  {  if(s=="triangle")
    { cin>>h>>b; *p++=new triangle(h,b); }
    else if(s=="trapezoid")
    { cin>>h>>t>>b; *p++=new trapezoid(h,t,b); }
  }
```

基类指针指向基类对象

基类指针指向派生类对象

***p=0;** 用0指针结束

```
}
```

基类指针调用虚函数show:有多态性

```
void output(triangle **p) { while(*p) { (*p)->show(); p++; } }
```

```
void del(triangle **p) { while(*p) { delete *p; p++; } }
```

```
int main( )
```

```
{  triangle *pt[100];
```

基类指针调用虚析构函数:有多态性

```
  input(pt); output(pt); del(pt);
```

```
  return 0;
```

```
}
```

14.2 虚函数的使用规则和内部实现原理

14.2.1 虚函数的使用规则

多态性特点：

- 多态性就是实际调用的成员函数由实际对象类型决定，什么类型对象就调用什么类型下的成员函数
- 当通过指针或引用使用对象时，若所使用的成员函数是指针或引用类型的虚函数，则具有多态性
- 对象调用成员函数由对象类型决定，不体现多态性
- 非虚函数没有多态性
- 虚函数不改变重名隐藏规则

只有指针或引用使用虚函数才有多态性，即什么对象调用什么成员函数

例B14.5 虚函数的进一步使用1(各种基类形参)。

```
class A
```

```
{ protected: int x;
```

```
public:
```

```
    A(int ix=0):x(ix) { }
```

```
    virtual void show() const { cout<<"x="<<x<<endl; }
```

```
};
```

```
class B: public A
```

```
{ protected: int y;
```

```
public:
```

```
    B(int ix=0,int iy=0): A(ix),y(iy){ }
```

```
    void show( ) const { cout<<"x="<<x<< ",y="<<y<<endl; }
```

```
};
```

```
void f1(A t) { t.show(); } 对象使用虚函数，无多态性
```

```
void f2(A *p) { p->show(); } 指针使用虚函数，有多态性
```

```
void f3(A &r) { r.show(); } 引用使用虚函数，有多态性
```

```
int main( )
```

```
{ B b(100,200);
```

```
    f1(b); f2(&b); f3(b);
```

```
    return 0; }
```

运行结果:

x=100 无多态性

x=100,y=200 有多态性

x=100,y=200 有多态性

例B14.6 虚函数的进一步使用2 (派生类中说明虚函数)。

```
class A
```

```
{ protected: int x;
```

```
public: A(int ix=0):x(ix){} void show()const{cout<<"x="<<x<<" ";}
```

```
class B: public A
```

```
{ protected: int y;
```

```
public: B(int ix=0,int iy=0): A(ix),y(iy){ }
```

```
virtual void show( ) const { cout<<"x="<<x<<" ",y="<<y<<endl; } };
```

```
class C: public B
```

```
{ protected: int z;
```

```
public: C(int ix=0,int iy=0,int iz=0): B(ix,iy),z(iz) { }
```

```
void show( ) const { cout<<"x="<<x<<" ",y="<<y<<" ",z="<<z<<endl; } };
```

```
int main( )
```

```
{ A *pa; B b(100,200),*pb; C c(2,3,5);
```

```
pa=&b; pa->show(); pa=&c; pa->show(); show不是A类虚函数
```

```
pb=&b; pb->show(); pb=&c; pb->show(); show是B类虚函数
```

```
return 0; }
```

运行结果:

x=100 无多态性

x=2 无多态性

x=100,y=200 有多态性

x=2,y=3,z=5 有多态性

注1: 虚函数的实现需要对象信息, 而静态成员函数和友元函数没有对象信息, 构造函数开始执行时也没有对象, 执行完后才生成对象, 故它们都不能做虚函数。

14.3 纯虚函数与抽象类

为了能使用统一形式来处理不同的对象，可以把要处理的各种不同类的对象设计成有共同基类的派生类，把要统一处理的各种功能设计成基类的虚函数，再在各个不同的派生类中重写虚函数代码。



此时共同基类中的虚函数只是为了派生时重写代码，本身是无意义的。可以免写此类代码，以**0指针(函数指针)**代替。这就是**纯虚函数**。带纯虚函数的类就是**抽象类**。

例C14.9 抽象类的使用。

第1部分

class shape 形状类: 抽象类

```
{ public: shape() { }
```

```
    virtual ~shape() { }    不能是纯虚函数, 因为基类析构函数总是要执行的
```

```
    virtual void show() const =0;    纯虚函数
```

```
    virtual double area() const =0;    纯虚函数
```

```
};
```

class triangle: public shape 三角形类

```
{ private: double x1,y1,x2,y2,x3,y3;
```

```
public: triangle(double ix1=0,double iy1=0,double ix2=0,double iy2=0,  
               double ix3=0,double iy3=0) ;
```

```
    ~triangle() { }
```

```
    void show() const ;    重新定义虚函数
```

```
    double area() const ;    重新定义虚函数
```

```
};
```

```
triangle::triangle(double ix1,double iy1,double ix2,double iy2,  
                  double ix3,double iy3):x1(ix1),y1(iy1),x2(ix2),y2(iy2),x3(ix3),y3(iy3) { }
```

```
void triangle::show() const
```

```
{    cout<<"三角形:三顶点为 ("<<x1<<","<<y1<<"),("<<x2<< ","<<y2  
    <<"),("<<x3<< ","<<y3<< ")"<<endl;   }
```

```
double triangle::area() const
```

```
{    double s=((x2-x1)*(y3-y1)-(x3-x1)*(y2-y1))/2; return s<0?-s:s; }
```

例C14.9 抽象类的使用。

第2部分

class circle: public shape 圆形类

{ private: double x,y,r;

public: circle(double ix=0,double iy=0,double ir=0):x(ix),y(iy),r(ir) { }

~circle() { }

void show() const {cout<<"圆:x="<<x<<"",y="<<y<<"",r="<<r<<endl;}

double area() const { return pi*r*r; }

};

class polygon: public shape 多边形类，深复制型

{ private: double (*p)[2]; int n;

void copy(const double pp[][2],int nn);

public: polygon(const double pp[][2],int nn) { copy(pp,nn); }

polygon(const polygon &b) { copy(b.p,b.n); }

~polygon() { delete[]p; } 虚析构函数可以保证多边形类对象析构时释放空间

polygon &operator=(const polygon &b)

{ if(this!=&b) { delete[]p; copy(b.p,b.n); } return *this; }

void show() const ;

double area () const ;

};

void polygon::copy(const double pp[][2],int nn)

{ n=nn; p=new double[n][2]; memcpy(p,pp,2*n*sizeof(double)); }

例C14.9 抽象类的使用。

第3部分

```
void polygon::show( ) const
{   int i;   cout<< "多边形: \n";
    for(i=0;i<n;i++) {cout<<"("<<p[i][0]<<","<<p[i][1]<<")"<<(((i+1)%5)?'\t':'\n'); }
    if(i%5) cout<<endl; }

double polygon::area( ) const
{   double s=p[n-1][0]*p[0][1]-p[0][0]*p[n-1][1];
    for(int i=0;i<n-1;i++) s+=p[i][0]*p[i+1][1]-p[i+1][0]*p[i][1];
    if(s<0) s=-s;   s/=2;   return s; }

void showShape(shape **s)    显示一批几何形状: 三角形、圆、多边形
{   while(*s) { (*s)->show(); cout<<"面积: " <<(*s)->area()<<endl; s++; } }

int main( )
{   shape *sh[20];    基类指针数组, 用以处理一批派生类对象
    const double pp[6][2]={ {3,5},{1,2},{4,-2},{7,7},{5,8},{0,6}};
    sh[0]=new circle(0,0,10); sh[1]=new triangle(0,1,6,1,3,5); sh[2]=new circle(0,0,5);
    sh[3]=new polygon(pp,6); sh[4]=new triangle(-2,-1,7,-1,7,5); sh[5]=0; 0指针结束
    showShape(sh);
    for(int i=0;sh[i];i++) delete sh[i];    析构所有动态对象
    return 0;
}
```


注1: 不能构造抽象类的对象。

注2: 对于抽象类而言, 如果派生类中重新定义了所有的纯虚函数, 则派生类就不是抽象类。若纯虚函数在派生类中未全部重新定义, 此时派生类中仍然保留有纯虚函数, 那么该派生类仍然是抽象类。

注3: 析构函数不能说明为纯虚函数。

注4: 形为 “(type)表达式” 或 “type(表达式)” 的显式转换是从C语言继承过来的, 安全性较差。

注5: C++标准的显式类型转换有如下4种:

- **static_cast<type>(表达式)**: 针对可赋值转换的显式转换。
- **const_cast<type>(表达式)**: 用于解冻数据(常量转换成变量)。
- **dynamic_cast<type>(表达式)**: 从有虚函数的基类指针或引用转换成派生类的指针或引用。指针非法转换得到0指针, 引用非法转换产生一个bad_cast异常(见第16章)。
- **reinterpret_cast<type>(表达式)**: 不改变数据值, 只改变类型, 用于指针或引用。

第15章 C++输入/输出系统

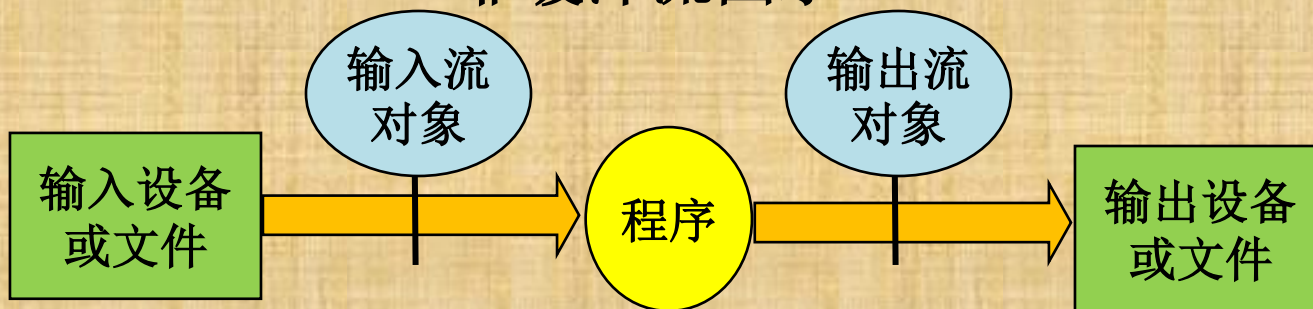
- **输入流**：C++将输入/输出看成一连串的字符流，输入的数据称为输入流
- **输出流**：C++中将输出的数据称为输出流
- **文本流**：ASCII字符流(文字流)
- **二进制流**：任意数值的字节序列构成的流

C++输入/输出有两种机制：一种是直接进行输入/输出，一种是不直接进行输入/输出，而是通过缓冲区进行输入/输出，即缓冲机制。

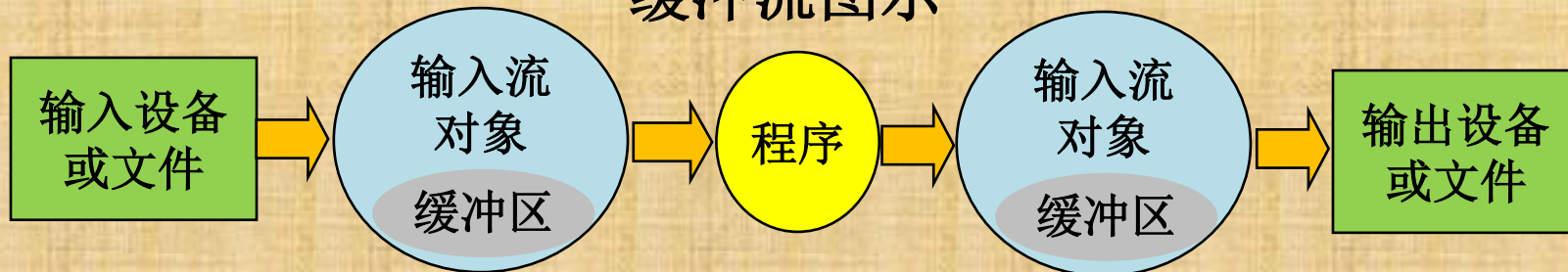
缓冲机制就像工厂进行生产，进货的原料和生产的產品不是直接进行进货和出货，而是通过仓库，在需要的时候才一次进一批货或者一次出一批产品。

缓冲机制下输入/输出流为缓冲流，否则称为非缓冲流。

非缓冲流图示

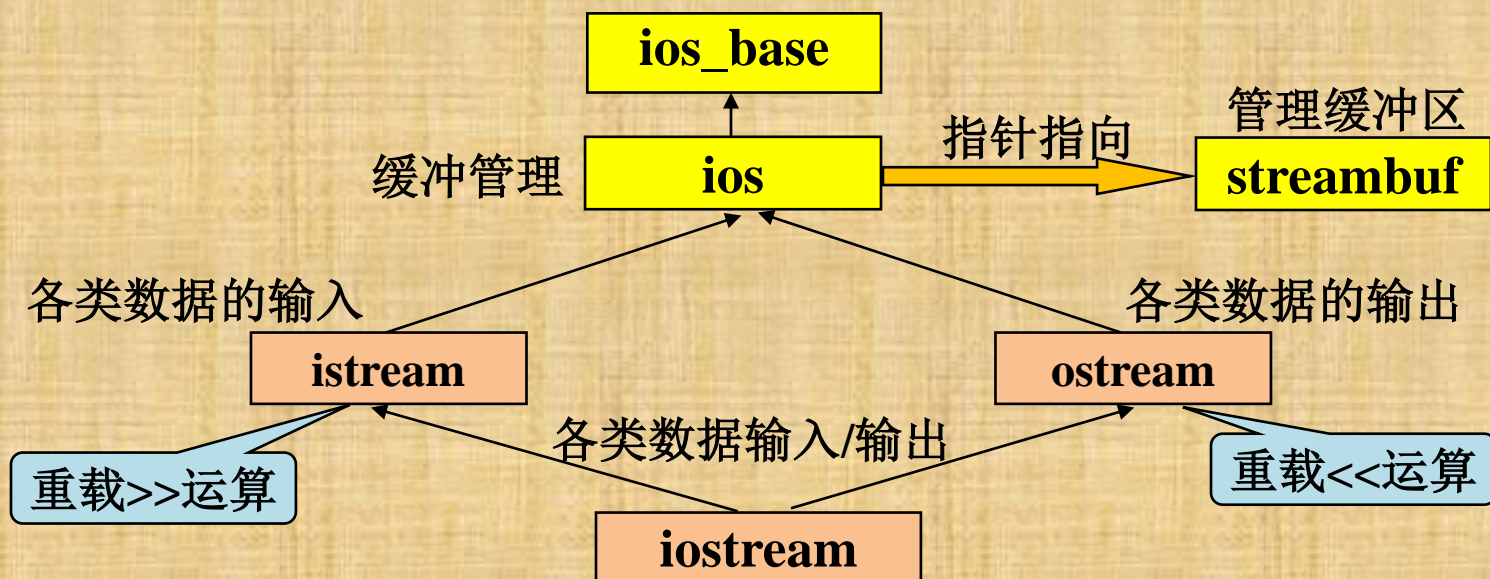


缓冲流图示

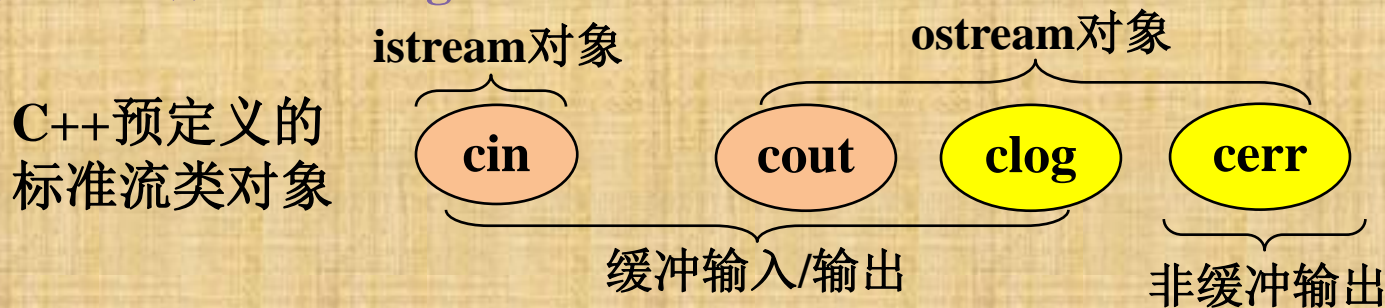


15.1 ○ C++ 输入/输出系统的结构

C++的基本流类结构： 处理格式、状态、错误检测



注1: C++中有一些预先定义的标准流类对象，最常见的就是cin和cout，其它还有cerr和clog。

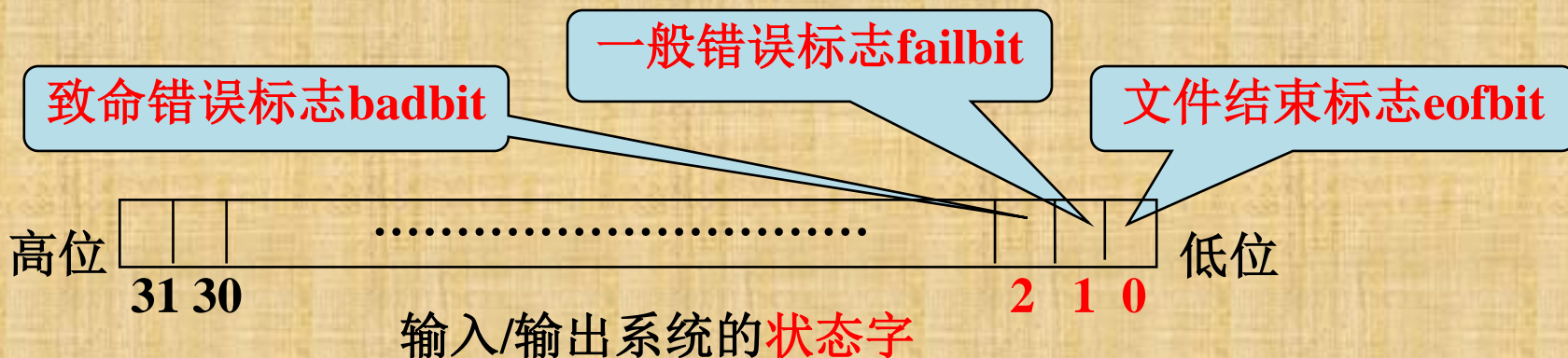


15.2 ○ C++流的状态

输入/输出过程中可能会发生错误，此时需要我们能够对输入/输出的状态进行检测和错误处理。

注1：在C++的输入/输出操作中，一旦出现错误将会自动设置一些错误标志，如果这些错误标志不消除，后续的输入/输出实际上不执行。

- 每个流对象中，都有自己独立的错误系统，在输入/输出操作时都会进行错误检测
- 输入/输出操作一旦出现错误，就将错误的相关信息记录在称为**状态字**的一个int型输入输出流类成员变量的各个二进制位中

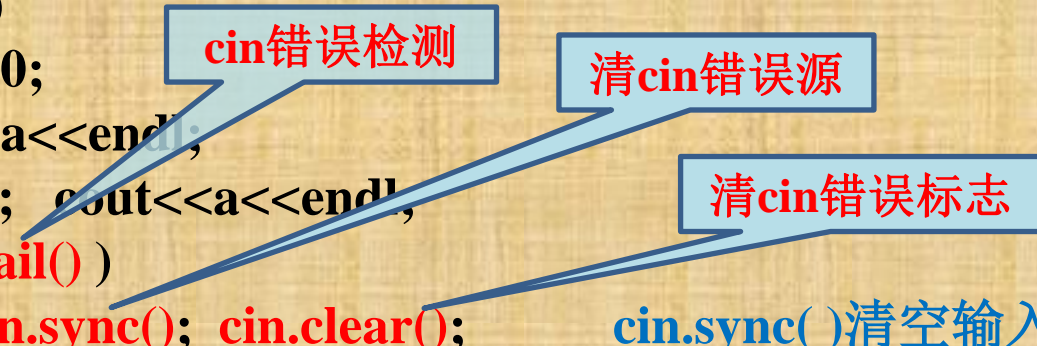


可以使用 `good()`、`fail()`、`bad()`检测错误，用`eof()`检测数据输入结束。

但是检测出错误后，需要进行错误处理，此时需要做两个工作：**1、清除错误源**，**2、清除错误标志**。

例A15.1 输入错误的检测与清理。

```
int main()  
{   int a=10;  
    cout<<a<<endl;  
    cin>>a; cout<<a<<endl;  
    if(cin.fail())  
    {   cin.sync(); cin.clear();      cin.sync()清空输入缓冲区  
        cin>>a; cout<<a<<endl;  
    }   return 0; }
```



注2：当出现输入/输出错误时一定要用`clear()`清除错误标志才能继续后续的输入/输出。

15.3 输入/输出成员函数及插入符<<和提取符>>

缓冲输入和缓冲输出过程:

- **缓冲输入操作:** 从输入缓冲区读取数据, 若缓冲区空或者数据量不够, 则当缓冲区读空后暂停输入操作, 转而执行从输入设备或文件向输入缓冲区输入数据的过程, 其中键盘为输入一行回车结束, 而文件为输入缓冲区满或文件结束。缓冲区得到数据后继续执行未完成的输入操作。
- **缓冲输出操作:** 从程序向输出缓冲区输出数据, 在4种情况下会临时执行清除输出缓冲区(将输出缓冲区中的内容传送到显示器或文件)的操作, 即输出缓冲区已满、即将执行输入操作、遇到清除缓冲区操作(如`cout.flush()`)、程序正常结束。

最常用的输入/输出操作是：

`cin>>a;` 提取操作：位运算>>的重载

`cout<<a;` 插入操作：位运算<<的重载

其中变量a可以是int、short int、float、double、char等类型。

`cin>>a` 是istream类对>>的重载，即`cin.operator>>(a);`

`cout<<a`是ostream类对<<的重载，即`cout.operator<<(a)。`

这些重载都是成员重载形式，见如下函数说明：

`istream& istream::operator>> (基本数据类型 & _Val);`

`ostream& ostream::operator<< (基本数据类型 _Val);`

函数都返回流对象本身，即`return *this`

除了使用提取符和插入符进行输入/输出操作外,还有一些输入/输出成员函数:

- 与输入相关的成员函数有: **get(...)**、**getline(...)**、**ignore(...)**、**gcount()**
- 与输出相关的成员函数有: **put(...)**、**flush()**

输入/输出操作说明: **ch**为字符变量, **s**为字符数组

- **cin>>ch**用于输入一个字符到变量**ch**, 默认跳过空白间隔符, 而**ch=cin.get()**和**cin.get(ch)**也读取空白间隔符到**ch**。空格、跳格、回车
- **cin.get(s,80)**表示输入字符串到字符数组**s**中, **s**提供80个字节, 最多保存79个有效字符外加一个串结束标志。遇回车字符'\n'表示结束, 不读取回车字符, 字符串末尾加串结束标志, 若前80个字符都没有回车字符, 则读取79个字符, 串尾加串结束标志, 并产生输入错误。操作**cin.get(s,80,'A')**与双参数**get**基本相同, 只是遇字符'**A**'表示结束, 不读取该结束字符。
- **cin.getline(s,80)**与操作**cin.get(s,80)**基本相同, 只是也读取回车字符并丢弃。**cin.getline(s,80,'A')**与**cin.get(s,80,'A')**基本相同, 也读取并丢弃结束符'**A**'。
- **cin.ignore(80)**表示从输入缓冲区读取并丢弃80个字符, 遇文件结束符EOF(即-1)终止。**cin.ignore(80,'\n')**读取并丢弃80个字符遇'\n'结束, 丢弃'\n'。
- **cin.gcount()**用于返回最近一次读取字符串时从缓冲区实际读取的字节数
- **cout.put(ch)**用于输出字符变量**ch**的字符
- **cout.flush()**用于清除输出缓冲区, 即将缓冲区内的数据传送到显示器

输入/输出时有时也需要检测输入/输出错误，有两个方便的检测方式：**(void*)cin**和**!cin**，对应于转换函数**(void*)**和运算符**!**重载，原始定义如下：

```
ios_base::operator void *( ) const { return (fail( ) ? 0 : (void *)this); }
```

```
bool ios_base::operator!( ) const { return (fail( )); }
```

例A15.2 输入错误的检测。

```
int main( )
{   int x; double d,sum;
    cin>>x;
    if(!cin) { cout<< "输入错误，终止程序"<<endl; exit(1); }
    cout<< "x="<<x<<endl;
    for(sum=0;cin>>d;sum+=d); 等价于 for(sum=0;(void*)(cin>>d);sum+=d);
    cout<< "总和sum="<<sum<<endl;
    return 0;
}
```

例B15.3 演示getline与get的使用。

```
int main( )
```

```
{  char ch,str[80];  
    ch=cin.get(); cout<<" "<<ch<<"\t";  
    cin>>ch;   cout<<" "<<ch<<"\t";   跳过空白间隔符  
    cin.get(ch); cout<<" "<<ch<<"\n";   也读取空白间隔符  
    while((ch=cin.get())!= '\n') cout<<ch; cout<<endl<<endl;  
    cin.getline(str,80); cout<<str<<endl;  
    cin.getline(str,5); cout<<str<<endl;  
    if(!cin) cin.clear();  若输入出错，清除错误标志  
    cin.getline(str,80); cout<<str<<endl;  
    cin.getline(str,80, 'e'); cout<<str<<endl;  结束字符'e'读出并丢弃  
    cin.getline(str,80);   cout<<str<<endl;  
    cin.get(str,80, 'e'); cout<<str<<endl;  结束字符'e'留在缓冲区  
    cin.get(str,80);   cout<<str<<endl;  
    ch=cin.get(); ..... }
```

注1: cin>>、getline、get都可以输入字符串，cin>>输入的是一个单词，空白间隔符(空格、跳格、回车)为结束符，而getline和get输入的是一行字符串。

注2: 两个参数的getline和get输入一行字符串时，字符串都不含回车字符'\n'。但是getline会读取回车字符并丢弃，而get并不从输入缓冲区中读出回车字符，回车字符仍然保留在输入缓冲区中。

例A15.4 ignore与gcount的使用。

```
int main( )
{   char str[80]; int len;
    cin.getline(str,80); cout<<str<<endl;
    len=cin.gcount(); cout<<len<< "\\t"<<strlen(str)<<endl;
    cin.getline(str,5); cout<<str<<endl;
    len=cin.gcount(); cout<<len<< "\\t"<<strlen(str)<<endl;
    if(!cin)
    {   cin.clear(); cin.ignore(80, '\\n'); len=cin.gcount();
        cout<< "输入缓冲区残留的"<<len<<"个字符已被清除"<<endl;
    }   return 0; }
```

注3：用提取符>>输入数据(尤其是数值型数据)时会跳过数据前面的空白间隔符，然后读取数据，但数据后面的间隔符不读出来，仍留在输入缓冲区内，故提取符输入数据至少会残留一个回车字符。

例A15.5 put的使用。

```
int main( )
{   char str[80]= "This is a book!";
    for(int i=0;str[i];i++) cout.put(str[i]); cout.put( '\\n');
    return 0; }
```


15.5 文件流

很容易将输出到屏幕的C++程序改成输出到文件。

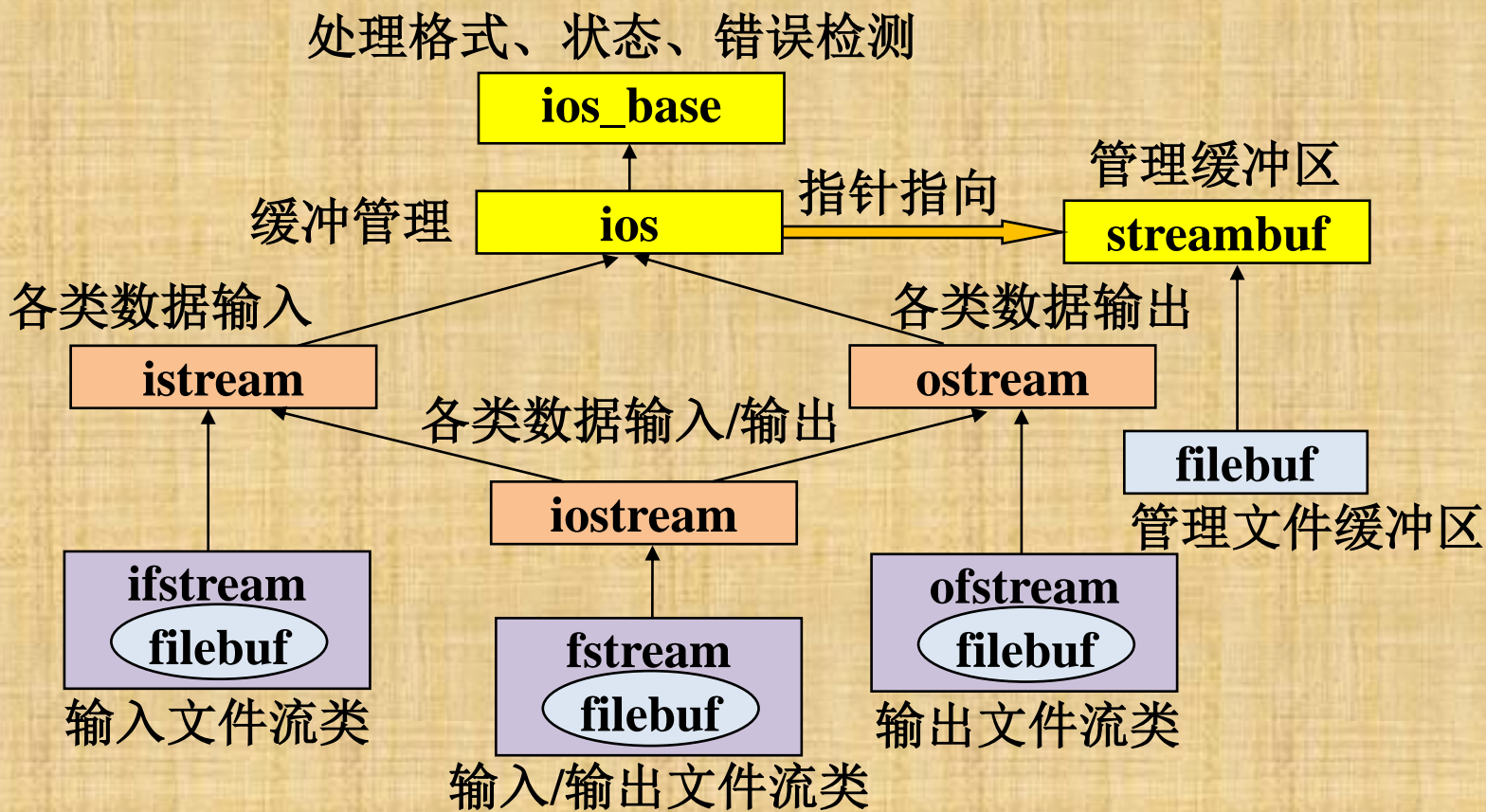
一个普通程序：

```
# include <iostream>
using namespace std;
int main( )
{   int  a=3,b=5;
    cout<<a<< "\\t"<<b<<endl;
    return 0;
}
```

改成输出到文件fileout.txt:

```
# include <iostream>
# include <fstream>
using namespace std;
int main( )
{   int  a=3,b=5;
    ofstream out("fileout.txt");
    out<<a<< "\\t"<<b<<endl;
    out.close();
    return 0;
}
```

○C++文件流类组成结构:



注1: 文件流类的使用要包含头文件 `fstream`。

15.5.1 文件的打开与关闭

使用文件之前需要对文件进行开启和停用的操作，称为**打开文件**和**关闭文件**。

文件的主要操作分类如下：

- **打开文件**：让文件流对象关联到某个文件上
- **输入文件数据**：从文件流对象关联的文件中读取数据
- **输出数据到文件**：将数据传送到文件流对象关联的文件中
- **关闭文件**：让文件流对象脱开某个文件，不再关联该文件

文件打开的两种方式:

(1) 先定义文件流对象, 再用open打开文件

```
ifstream infile;  
ofstream outfile;  
fstream iofile;  
infile.open("data1.txt");  
outfile.open("result.txt");  
iofile.open("file1.txt");
```

(2) 定义文件流对象的同时打开文件

```
ifstream infile("data1.txt");  
ofstream outfile("result.txt");  
fstream iofile("file1.txt");
```

文件打开成功与否的检测:

```
if(!infile) { cerr<<"文件打开失败!\n"; exit(1); }
```

文件打开后进行输入输出：

```
int x;
```

```
infile>>x; 从infile关联的文件输入整数给x
```

```
outfile<<"x="<<x<<endl; 将数据输出到outfile关联的文件
```

```
outfile<<hex<<setiosflags(ios_base::uppercase)<<x<<endl;  
    格式化(大写字母十六进制)输出到outfile关联的文件
```

关闭文件：

```
infile.close(); outfile.close(); iosfile.close();
```

注2：文件的输入/输出在使用时与其它输入/输出相比使用前多出了一步关联文件(需要检测成功与否)，使用后多出了一步关闭文件。

○用于打开文件的open函数的函数说明:

```
void ifstream::open(const char *_Filename, int _Mode = ios_base::in);  
void ofstream::open(const char *_Filename, int _Mode = ios_base::out);  
void fstream::open(const char *_Filename,  
                   int _Mode = ios_base::in|ios_base::out);
```

○用于打开文件的流类构造函数的函数说明:

```
ifstream::ifstream (const char *_Filename, int _Mode = ios_base::in);  
ofstream::ofstream (const char *_Filename, int _Mode = ios_base::out);  
fstream::fstream(const char *_Filename,  
                 int _Mode = ios_base::in|ios_base::out);
```

○文件打开的第二参数通常省略，表示文件打开方式

```
static const int in= 0x01;  打开文件用于读(输入数据)    I 即用于输入  
static const int out= 0x02; 打开文件用于写(输出数据)    O 即用于输出  
static const int ate= 0x04; 打开文件并将文件指针移到文件末尾 IO  
static const int app= 0x08; 打开文件用于在末尾添加数据 O  
static const int trunc= 0x10; 打开文件，若文件存在则清空文件内容 O  
static const int _Nocreate= 0x40; 打开已有文件，且不清空文件内容 O  
static const int binary= 0x20; 以二进制方式打开文件 IO
```


15.5.2 文本文件的使用

文本文件就是可用Windows操作系统中记事本打开阅读的文件。

文本文件输入/输出与键盘和屏幕的输入/输出(标准输入/输出)用法差不多，只是输入/输出流对象不再是cin和cout，而是与文件关联的文件流对象，并且之前要打开文件，之后要关闭文件。

例A15.9 将数据输出到文本文件。

include <fstream> 使用文件

include <iostream>

include <iomanip> 使用输入/输出格式

include <string> 使用字符串

using namespace std;

int main()

{ string filename; int k,i,cnt; cin>>filename;

ofstream out(filename.c_str()); 定义文件流对象同时打开文件

if(!out) { cerr<< "打开文件: "<<filename<< "失败\n"; exit(1); }

for(k=2;k<=40;k+=2) { out<<setw(4)<<k; if(k%10==0) out<<endl; }

out.close(); 关闭文件

格式输出：每个输出整数占4个字符

cin>>filename; out.open(filename.c_str()); 打开第二个文件

if(!out) { cerr<< "打开文件: "<<filename<< "失败\n"; exit(2); }

out<<setw(8)<<2; cnt=1;

for(k=3;k<=50;k+=2)

{ for(i=3;i<k;i+=2) if(k%i==0) break; 判断素数

if(i>=k) { out<<setw(8)<<k; if(++cnt%5==0) out<<endl; }

} out.close();

}

例A15.10 从文本文件读取数据。

```
# include <fstream>
# include <iostream>
using namespace std;
int main( )
{   double score,average,sum,max,min;   int num;
    ifstream infile( "score1.txt" );
    if(!infile) { cerr<< "打开文件: score1.txt失败\n" ; exit(1) ; }
    sum=0; num=0; max=-1; min=200;
    while(infile>>score) 等价于(void*)(infile.operator>>(score) )
    {   sum+=score; num++;
        if(score>max) max=score;
        if(score<min) min=score; //书上程序略有问题: 第一个分数最小
    }
    infile.close();
    average=sum/num ;
    cout<< "文件: score1.txt 中共有"<<num<< "个成绩\n";
    cout<< "其中最高分: "<<max<< ",最低分: "<<min;
    cout<< ",平均分: "<<average<<endl; return 0;
}
```


例A15.11 用凯撒密码将一个文本文件中的文字加密到另一个文件。 读文件和写文件

```
# include <fstream>
# include <iostream>
# include <string>
using namespace std;
char encrypt(char ch) ;
int main( )
{   string filename1, filename2;   char ch ;
    cout<< "请输入要加密的源文件名和目的文件名: ";
    cin>>filename1>>filename2;
    ifstream in(filename1.c_str()); ofstream out(filename2.c_str());
    if( !in| !out ) { cerr<< "文件打开失败\n"; exit(1); }
    while( in.get(ch) ) out.put( encrypt(ch) ); 等价于 (void*)in.get(ch)
    in.close(); out.close(); return 0;
}
char encrypt(char ch) 对单个字符进行加密 ( 凯撒密码 )
{   if(ch>= 'A'&&ch<= 'Z')  ch= (ch- 'A'+3)%26+ 'A';
    else if(ch>= 'a'&&ch<= 'z') ch=(ch- 'a'+3)%26+ 'a';
    return ch; }
```

15.5.3 ○二进制文件的使用

打开二进制文件:

```
ifstream in("file1.dat", ios_base::in | ios_base::binary);  
ofstream out;  
out.open("file2.dat", ios_base::out | ios_base::binary);
```

注3: 二进制文件的输入和输出不用>>和<<, 而是用成员函数read和write, 这两个函数的形式如下:

```
istream& istream::read(char *_Str, int _Count);  
ostream& ostream::write(const char *_Str, int _Count);
```

注4: 当反复进行输入时二进制文件用以下形式判断数据是否结束:

```
bool ios_base::eof() const
```

istream& istream::read(char *_Str, int _Count); 读_Count个字节到 _Str

ostream&ostream::write(const char *_Str, int _Count);输出 _Str中_Count字节

注5: 用read和write进行输入/输出时, 第一个参数若不是char*类型, 则都要显式转换成char*类型, 输出时也可以转换成const char*类型。

二进制文件输入结束的判别可使用 eof

```
bool ios_base::eof() const
```

例A15.12 用二进制文件存取数据。

```
# include <fstream>
```

```
# include <iostream>
```

```
# include <string>
```

```
using namespace std;
```

```
int main( )
```

```
{ string filename,ioselection; int k,i; cin>>filename>>ioselection;
```

```
if(ioselection=="输入")
```

定义二进制输入文件对象

```
{ ifstream in(filename.c_str(),ios_base::in|ios_base::binary);
```

```
if(!in) { cerr<< "open:"<<filename<< " failure\n";exit(1);}
```

```
while(true) { in.read((char*)&k,sizeof(int)); 输入数据到k
```

```
if(in.eof()) break; cout<<k<< "\t"; } 判断若文件未结束则输出k
```

```
in.close(); cout<<endl<<endl; }
```

地址要转换成char*

```
else
```

```
{ ofstream out(filename.c_str(),ios_base::out|ios_base::binary);
```

```
if(!out) { cerr<< "open:"<<filename<< " failure\n";exit(2);}
```

```
k=2; out.write((char*)&k,sizeof(int));
```

```
for(k=3;k<=60;k+=2) { for(i=3;i<k;i+=2) if(k%i==0) break;
```

```
if(i>=k) out.write((char*)&k,sizeof(int));
```

```
} out.close(); } return 0; }
```