$$\mathbf{V}_1 \times \mathbf{V}_2 = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \frac{\partial x}{\partial u} & \frac{\partial y}{\partial u} & \theta \\ \frac{\partial x}{\partial v} & \frac{\partial y}{\partial v} & \theta \end{vmatrix}$$

| a | b | c |
| --- | --- | --- |
| | | |

---



# 类

# 本质：

类名本质：一种命名空间。

# 面向对象三大特性：

封装：从结构到类：封装是对成员的保护,比如结构list，需要封装头指针，不能对外开放，因此需要类。

继承：

多态：

# 访问权限：

类成员的默认访问权限：private

结构体默认的访问权限：public ，这是类与结构体的本质区别。

类有三种访问权限：private,public,project.

| 类\访问\权限 | private | public | projected |
| --- | --- | --- | --- |
| 类的内部 | yes | yes | yes |
| 子类的内部 | no | yes | yes |
| 类的外部 | no | yes | no |

# 继承方式：

三种继承方式：private，public,projected

作用: 继承是为了控制子类对父类的访问权限

public继承：不改变父类的成员访问权限，特别地，父类的private不可见。

protected继承：把父类中的public成员在本类中变成了protected的访问控制权限，其他不变。特别地，父类的private不可见。

private继承：把父类中的public和projected成员变成了private的访问控制权限，其他不变。特别地，父类的private不可见。

# 引用：

本质：指针

效果：扩展了作用域

## 构造函数：

系统自动提供一个空体的无参构造函数，但是只要你在类里面写了个构造函数（不管是有参还是无参的），系统提供的无参构造函数会被舍弃掉，不复存在。

区别对象定义和函数声明：

class_name cn： 这是一个对象定义，调用的是无参构造函数

class_name cn(): 这是一个函数声明，不能当做对象定义，并且调用了无参构造函数

初始化列表，初始化成员,效率较高。比如 Stack():size(0),spacesize(10){}

匿名对象：A()，直接构造出对象，在栈中。

## 析构函数：

系统提供一个默认空体的析构器，一经实现，默认析构器不复存在

对象存在的地方在栈上和堆上

栈对象：Stack s;

堆对象：Stack *s = new Stack(10);

对象在被销毁前会自动调用析构函数

析构函数用来处理清理工作，主要清理堆上的对象，比如释放对象申请的一些空间。如果对象没有申请空间时可以不用自己写析构函数。

析构函数里常放delete和delete[]

后构造的先析构

## 内存层次化：

申请内存从外到内，释放内存从里到外；也就是说，谁申请空间谁释放。

常在构造函数里申请(new)空间，在析构函数里释放空间，不需要在外部申请空间。

## 拷贝构造器（copy constructor):

一种构造函数，和constructor的地位是一致的

拷贝构造到底用来干啥呢?在哪些情况下需要用呢?

int a=0; 这是定义并初始化语句，相当于调用了构造函数初始化对象

a=5; 这是赋值语句，与初始化不同

int b=a; 由已有的对象创建新的对象，相当于调用 拷贝构造初始化对象

系统提供默认的拷贝构造函数,一经实现不复存在。默认的拷贝构造函数是非空体的，提供一个等位拷贝机制。系统提供的拷贝构造是一种浅拷贝（shallow copy),当然还有另一种拷贝即深拷贝（deep copy)。

拷贝构造的形式：

class_name(const class_name& anothor){

segvar1=anthor.segvar1;

segvar2=anthor.segvar2;

...

}

等位拷贝机制：成员与方法一一拷贝过去；

浅拷贝：如果对象不含有堆上的空间，此时浅拷贝就可以满足要求，使用系统提供的默认拷贝构造函数就可以了，不需要自己重实现。

深拷贝：对象包含堆上的空间，这时浅拷贝不能满足需求，浅拷贝会导致两个对象共享同一个堆空间，在对象析构时浅拷贝会导致重析构，即double free，需要自己实现。

浅拷贝系统默认提供，深拷贝自己实现。浅拷贝和深拷贝的关键点是否共享堆内存空间。

重析构：当对象析构时两个对象都对堆里相应的空间进行delete[]，因此同一个堆空间被重复释放了free。

深拷贝实现,另一个对象里需要堆空间时就重新申请，避免与之前的对象共享前一个对象的空间:

class_name(const class_name& anthor){

char *_str=new char[1024];

strcpy(_str,anthor.str);

}

在拷贝构造函数里，可以访问另一个对象的私有成员，不然也无法拷贝。

异类对象之间访问私处用友元。

class A{

public:

A(int x=0,int y=0):x(x),y(y){};

~A(){}

A(const A& a){x=a.x;y=a.y;}

private:

int x;

int y;

}

```
int main(){
A a;
A b(a); // 直接使用构造函数
return 0;
}
```

调用构造，会出现两次析构。

```
int main(){
A a;
A b=a; // 定义并初始化
return 0;
}
```

也会调用构造，也会出现两次析构。

```
void foo(A a){} //形参传对象时
int main(){
A a;
foo(a);
return 0;
}
```

也会调用构造，也会出现两次析构。

```
void foo(A& a){} //形参 传引用时
int main(){
A a;
foo(a);
return 0;
}
```

不会调用构造，只会出现一次析构。

# this指针：

指向当前对象的指针,在对象创建之后才出现。

this指针不会增加对象的空间大小，因为this指针作为成员函数参数隐式传进的，也即对象本身的地址隐式传入成员函数。

在所有的成员函数里可用，但是有一个地方不能用?

使用this指针的好处：1、使用this指针可以支持成员函数的形参名和成员相同的情况；2、可以实现链式表达(比如，a=b=c，a.f().f().f()..)。

class_name& f(){

this->age++;

class_name* const p=this; //注意这点

return *this;

}

这样的话，class_name A; 链式表达A.f().f().f()....;

# 赋值运算符重载：

系统提供默认的赋值运算重载，但是系统提供的赋值运算符重载也是浅赋值，当然还有深赋值。自己实现了赋值运算符重载后，系统提供的将不复存在。

class_name& operator= (const class_name& anthor){ return *this;}

string s1=s; // s已存在，并由已有的s创建新的s1,调用拷贝构造完成

s1=s; // s和s1这两个对象都存在，并且由s赋值给s1，调用该类重载的赋值运算符=完成，和string s1=s；完全不同。

浅赋值：对象中不包含堆空间。

深赋值：对象包含堆空间，需要自己重新实现，不然会出现重析构（double free)、内存泄漏（空间没使用但是也没释放，白白浪费空间）、自赋值问题。

浅赋值和深赋值的意义可以参考浅拷贝和深拷贝的意义。

自赋值：自己赋给自己

解决赋值时的内存泄漏和自赋值问题：先判断是否和this相等，如果相等就返回*this实现链式表达，如果不是就先把自己的堆空间释放，然后将另外对象的值赋给该对象，再返回*this实现链式表达。

大部分表达式是可以赋值的，比如（a=b)=c,但是 const class_name& operator= (const class_name& anthor){ return *this;}，加了const就不能赋值了。

# 对象的空间模型一：

数组被结构体或类打包后就可以进行等位拷贝。

对象在哪？对象里面成员或申请的空间在哪，大小是多小？成员属性分为堆属性和栈属性，如何区分呢，是否具有指针*,并且申请了新的空间，这时就具有堆属性。

堆对象：

A *pa = new A();

栈对象：

```
int f(){
int a=3;
return a;
}
int main(){
int i=4;
i=f();
return 0;
}
```

过程调用是通过栈实现的。

f()函数里面return 的 a 它去了CPU的寄存器里面，然后又把寄存器里的a值放到叫i的内存里。寄存器里面的变量叫做中间变量，是不能取地址的。

栈上的对象是可以返回的，但不可返回栈对象引用。

返回值优化，具名返回值优化RVO，不具名返回值优化NRVO，（RVO==return value optimization)。

```
A foo(){
return A(); // 返回一个匿名对象
}
int main(){
foo();
return 0;
}
```

一次构造和一次析构。

```
A foo(){
A a;
return a; // 返回一个具名对象
}
int main(){
foo();
return 0;
}
```

在VS里，一次构造，一次拷贝构造，和两次析构。

接收栈对象时；

A t; t=foo(); // 赋值

A t = foo(); // 拷贝构造

## 对象数组：

对象的数组

A a[100];sss

数组中有100个对象就有100次构造。

在构造函数里使用默认值，这样就能包含无参的和有惨的，而且初始化时不用自己去初始化100次了。这样的话，可以使用二段式初始化方式。

二段式初始化：有一个无参构造函数A()，和一个初始化函数init()，定义数组时自动调用无参构造，然后通过初始化函数init()进行100次初始化。

## 对象的空间模型二：

sizeof(A)=?

对象所占空间的大小只取决于该对象中非static数据成员所占的空间，而与成员函数和static数据无关。

成员变量：每个对象的成员变量为对象自己所有。

成员函数的存储方式：成员函数是被所有对象公用的。虽然每个成员函数是公用的，但不要忘了，对象调用成员函数的时候隐式传了一个this指针，而this指针是指向该对象，因此不同对象调用公用成员函数能操作各自对象的数据了。All because of this。

## const：

const 语义是 不可更改。

必须初始化，初始化后不可变。

const 私有成员: 可以直接初始化，也可以通过自实现构造器初始化列表来实现初始化。

class A{

public: A(int xi=100):x(xi){}

private:

const int x;

int y；

}

const 成员函数：

const修饰函数可以构成重载。因此函数重载的条件是参数个数、类型、顺序、及有const修饰。

const可以修饰成员函数，但是不能修饰全局函数。

const修饰函数的作用：在该函数内不会改变数据成员。并且该函数内部只能调用const函数。非const对象优先调用非const函数。

class A{

public: A(int xi=100):x(xi){}

int foo(int i){return i;}

int foo(int i) const {

return i;

}

private:

const int x;

int y;

}

const对象：const对象只能调用const 函数，因此const修饰的对象里可以有非const函数，但是只能调用const函数，而且也不可更改数据成员。因此对象里的有些函数提供有const修饰的无const修饰的函数这两个版本的函数。可能还需自实现构造器，对数据成员进行初始化，不然数据成员没有初始化的机会了。

# static:

C语言中：static 全局变量或函数，不能被extern，函数或全局变量只能在本文件使用；static 局部变量，改变局部变量生存周期和存储位置；

对象中：全局性，对象共享数据，协调行为

static 数据成员：

类内声明，类外定义并初始化。

定义并初始化初始化后才能使用

不算入对象的大小之中，它存储在数据段的?(ro,rw)段

可以使用对象访问a.，也可以使用类名访问A::

private ： static int x; 如何访问x呢，保证对象和类都可以访问到x。

class A{

public:

static int foo(){return x;}

private:

static int x;

}

int A::x=1;

static 成员函数：

管理（访问）私有的静态数据成员。

static函数不能调用非static数据或函数，只能访问static的数据和static的函数。这样static的函数没有this指针，因为这样才能支持类访问static成员。

非static函数可以访问static的成员。

## static 与 const 搭配：

共享且不可更改

static const int a;

如何对a进行初始化？static和const的初始化要求不同。

static const int a=1;

## 指向类成员的指针：

类层面上使用指针

本质记录相对对象首地址的偏移量

回顾C里：

int *pa=&a; //指向数据的指针

cout<<*pa<<endl; //使用指向数据的指针

void (*pf)(int ) = foo; //定义函数指针

pf(1); //使用函数指针

void (*pf[n])(int ); //函数指针数组

如何定义一个指针，指向类的成员？注意：是类的成员，不是对象的成员。

指向数据成员（假设数据public）：

string A:: *pn = &A::name;

如何使用该指针：和具体对象相关

A a;

A *pa = &a;

cout<<a.*pn<<endl;

cout<*pn<<endl;

指向函数成员的指针：

void (A::*pf)(int ) = &A::f;

A a;

(a.*pf)(1);

指向类函数的指针数组：

void (A::*pf[n])(int );

指向static类数据和函数的指针：

int *pa = &A::data;

void (*pf)(int ) = &A::dis;

作用：1、更加统一的接口；2、更加隐蔽的接口；

更加统一接口:

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

struct point{
  int add(int x,int y)return x+y;
  int mius(int x,int y)return x-y;
  int multipy(int x,int y)return x*y;
  int div(int x,int y)return x/y;
}

int oper(point& p,int(point::*pf)(int, int),int x,int y){ return (p.*pf)(x,y);}

typedef int(point::*PF)(int, int);

int main(){
  point p;
  PF pf =&p::add;
  cout<<oper(p,pf,1,2);

  pf = &p::mius;
  cout<<oper(p,pf,2,1);

  return 0;
}
```

更加隐蔽的接口：

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

```
class game{

  public:
    game(){
      pf[0] = &game::f;
      pf[1] = &game::g;
      pf[2] = &game::h;
      pf[3] = &game::m;
    }

    int select(int i){
        return (i>=0 && i<=3)?((this->*pf)[i](i)):(-1);
    }

  private:
    void f(int d){cout<<"f "<<d<<endl;}
    void g(int d){cout<<"g "<<d<<endl;}
    void h(int d){cout<<"h "<<d<<endl;}
    void m(int d){cout<<"m "<<d<<endl;}

    enum{ nc = 4};

    void (game::*pf[nc])(int );
}

int main{
  game ga;
  ga.select(1);
  return 0;
}
```

# 友元 friend：

friend 方法，支持方法访问类的私有数据成员，避免频繁使用set和get方法，提高效率，但这样似乎破坏了封装。

同类之间无私处，异类之间有友元。同类表示同一个命名空间里。

声明为谁的友元，就可以通过谁的对象，访问谁的私有成员。

友元不是成员函数，不属于这个类的，因为不能隐式传递this指针，因此需要标记为该类的“朋友”，从而完成对私有成员的访问。

友元关系不能被继承

友元关系是单向的，不能传递的。

friend 函数（友元函数）：

全局函数作友元：

```
#include <iostream>
```

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

class point{
    friend float dist(const point& p1,const point& p2);
    public:
        point(int x=0,int y=0):x(x),y(y){}
    private:
        int x,y;
};

float dist(const point& p1,const point& p2){
    int dx=p1.x-p2.x;
    int dy=p1.y-p2.y;
    return sqrt((double)(dx*dx+dy*dy));
}

int main()
{
    point p1(1,1);
    point p2(4,1);
    point p3(4,5);

    cout<<"a: "<<dist(p1,p2)<<endl;
    cout<<"b: "<<dist(p2,p3)<<endl;
    cout<<"c: "<<dist(p1,p3)<<endl;

    return 0;
}
```

成员作友元，需前向声明：class B; friend void B::func(); 前向声明不能用于定义对象，但可以定义指针和引用，作为参数和返回值，但也仅用在函数声明中，而不是函数定义中。你中有我，我中有你，就会存在谁在前的问题，也就需要前向声明来化解。

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

//类成员作为友元，需要前向声明
//用于定义指针和引用
class point;

class calcus{
    public:
        float dist(const point& p1,const point& p2);
};

class point{
    friend float calcus::dist(const point& p1,const point& p2);
    public:
        point(int x=0,int y=0):x(x),y(y){}
    private:
```

```
        int x,y;
};

float calcus::dist(const point& p1,const point& p2){
    int dx=p1.x-p2.x;
    int dy=p1.y-p2.y;
    return sqrt((double)(dx*dx+dy*dy));
}


int main()
{
    calcus cal;

    point p1(1,1);
    point p2(4,1);
    point p3(4,5);

    cout<<"a: "<<cal.dist(p1,p2)<<endl;
    cout<<"b: "<<cal.dist(p2,p3)<<endl;
    cout<<"c: "<<cal.dist(p1,p3)<<endl;

    return 0;
}
```

friend 类（友元类）：

不需要前向声明

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

// 不需要前向声明

class point{
    friend class calcus;
    public:
        point(int x=0,int y=0):x(x),y(y){}
    private:
        int x,y;
};

class calcus{
    public:
        float dist(const point& p1,const point& p2){
            int dx=p1.x-p2.x;
            int dy=p1.y-p2.y;
            return sqrt((double)(dx*dx+dy*dy));
        }
};
```

```
int main()
{
    calcus cal;

    point p1(1,1);
    point p2(4,1);
    point p3(4,5);

    cout<<"a: "<<cal.dist(p1,p2)<<endl;
    cout<<"b: "<<cal.dist(p2,p3)<<endl;
    cout<<"c: "<<cal.dist(p1,p3)<<endl;

    return 0;
}
```

# 操作符重载 operator:

语义：对于自定义类型时，需要重载操作符，使其和基本类型表现一致。

本质调用过程：A ac = a.operator+(b)

注意点：不能有默认参数，const的位置（返回值-不能复制、参数值、函数-const对象调用），引用（返回值，参数），非引用-临时对象或返回匿名对象

struct和class里重载（)运算符后该类也叫仿函数

友元重载：即全局函数重载，操作符重载作为友元函数。

成员重载：操作符重载作为成员函数。

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

class point{
    friend class helper;
    public:
        point(int x=0,int y=0):x(x),y(y){}
        point& operator= (const point& p);
        const point operator+ (const point& p);
    private:
        int x,y;
};

point& point::operator= (const point& p){
    this->x=p.x;
    this->y=p.y;

    return *this;
```

```
}

const point point::operator+ (const point& p){
    point p1;
    p1.x=this->x+p.x;
    p1.y=this->y+p.y;
    return p1;
}

class helper{
    public:
        int getx(const point& p){return p.x;}
        int gety(const point& p){return p.y;}
};

int main()
{
    helper h;

    point p1(1,1);
    point p2(4,1);

    point p3=(p1+p2);

    cout<<"x:"<<h.getx(p3)<<"  y:"<<h.gety(p3)<<endl;

    return 0;
}
```

单双目操作符:

++a; //在本身对象上自增

a++; // 会创建对象，然后才自增

优先使用++a的形式;

流操作符重载:

```
//重载输入流
friend istream& operator>> (istream& ci, complex& c){
  ci>>c.real;
  ci>>c.image;
  return ci;
}

//重载输出流
friend ostream& operator<< (ostream& co, complex& c){
  co<<c.real<<endl;
  co<<c.image<<endl;
  return co;
}
```

# 自定义类型的转换：

自定义类型-转化构造函数：

explict 显示的；

implicit 暗示的；

```
class 目标类
{
  目标类（const 源类型& 源类型对象引用）
  {
    根据需要完成类型转换
  }
}

class pointD2
{
  friend class pointD3;
  public:
    pointD2(int x=0,int y=0):_x(x),_y(y){}
  private:
    int _x;
    int _y;
}

class pointD3
{
  public:
    pointD3(int x=0,int y=0, int z=0):_x(x),_y(y)_z(z){}

  explicit pointD3(const pointD2& d2){
    this->x=d2._x;
    this->y=d2._y;
    this->z=rand()%100;
  }

  private:
    int _x;
    int _y;
    int _z;
}

int main()
{
  srand(time(NULL));
  pointD2 d2(10,100);
  pointD3 d3 = (pointD3)d2; // explicit
  pointD3 d3 = d2; //implicit

  return 0;
}
```

自定义类型-操作符函数转化：

```
class 源类{
  operator 目标类 (void){
    return 目标类构造器（源类实参）
  }
}

class pointD3
{
  public:
    pointD3(int x=0,int y=0, int z=0):_x(x),_y(y)_z(z){}
  private:
    int _x;
    int _y;
    int _z;
}

class pointD2
{
  public:
    pointD2(int x=0,int y=0):_x(x),_y(y){}

    operator pointD3()
    {
      return pointD3(this->_x,this->_y,rand()%100);
    }

  private:
    int _x;
    int _y;
}

int main()
{
  srand(time(NULL));
  pointD2 d2(10,100);
  pointD3 d3 = (pointD3)d2; // explicit，显示转换
  pointD3 d3 = d2; //implicit，隐式转换

  return 0;
}
```

# 仿函数与智能指针：

仿函数：一个对象，使用起来像一个函数。

与一般函数的优势：可以像类那样携带更多信息：有参构造函数，私有成员等。

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

class power
{
  public:
  double operator()(double x, int y){
    double r=1;
    for(int i=0;i<y;i++)r*=x;
    return r;
  }
};

int main()
{
  double x=2.0;
  power mypower;
  cout<<mypower(x,3)<<endl;
}
```

智能指针：auto_ptr已被舍弃，后者share_ptr，weak_ptr的灵感均来自于auto_ptr。

RAII Theory:资源获取即初始化原理(初始化即构造），也是构造器和析构器的基本原理。

```
//非资源获取即初始化
A a;
A* p = new A;
delete p;
//资源获取即初始化
auto_ptr<A> p(new A);//通过对象p来获取对象A，并且会自动释放对象A。好处是不用担心忘记释放资源，避免内存泄漏。
```

# 重载new/delete

特点：不常用，格式较固定，主要用于定制对象内存。

重载new:

重载delete:

```
class A
{
  public:
    A(){
     x=100;
    }
    ~A();

    //重载new
```

```
    void* operator new(size_t _size){
        void *p=malloc(size);
        return p;
    }
    //重载delete
    void operator delete(void* p)
    {
        free(p);
    }
    //重载new[]
    void* operator new[](size_t _size){
        void *p=malloc(size);
        return p;
    }
    //重载delete
    void operator delete[](void* p)
    {
        free(p);
    }
  private:
    int x;
}
```

# 继承

作用：实现代码重用，也是面向对象编程的精华。

代码重用：

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

//老师 姓名 性别 年龄    教课 吃饭 睡觉
//学生 姓名 性别 年龄    学习 吃饭 睡觉
//抽象出老师和学生的共性(人)，把共性(人)作为重用的部分，也即继承共性(人)
//继承共性后，添加自己的个性 ，得到子类

class human{
    public:
        void eat(string food="noodle"){
            cout<<"Eating noodel"<<endl;
        }
};

class teacher:public human{
    public:
        void teach(string course="math"){
            cout<<"Teacher,I'm teaching "<<course<<endl;
        }
```

```
};

class student:public human{
    public:
        void study(string course="math"){
            cout<<"Student,I'm learning "<<course<<endl;
        }
};

int main()
{
  teacher t;
  t.teach();
  t.eat();

  student s;
  s.study();
  s.eat();

  return 0;
}
```

定性继承关系: is-a，不是has-a,比如dog is an animal。has-a表达的是一种包含关系，不是继承关系。

继承方式：private,project,public。

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

//总结六句话:
//继承不影响子类成员的访问属性，但是会改变子类访问父类成员的访问属性，public保持父类已有的访问属性，
protected将public变为protected，private将public和protected变为private
//类外只有public属性的成员才能被访问
//类内只有父类private属性的成员不能被访问
//protected属性的成员只能被自己或自己的子类内部访问
//先构造父类对象，在构造内嵌对象，最后构造子类对象；先析构子类对象，再析构内嵌对象，再析构父类对象；如果
父类或内嵌类的构造函数不是无参或默认参数的形式时，就需要子类显示调用父类或内嵌对象的构造函数
//子类全盘接收父类的东西 ,内存结构为:
/*
高地址:          b(子类的)(pub,pro,pri)
低地址: this->   a(父类)(pub,pro,pri)
this指向类中第一个成员pub.
子类和父类this值相同，但是，this的类型不同。
*/

//内嵌类c
class c{
    public:
//      c(int pub=12):pubc(pub){cout<<"construct:c()"<<endl;}

        c(int pub):pubc(pub),proc(22),pric(32){cout<<"construct:c()"<<endl;}
```

```
        ~c(){cout<<"free:~c()"<<endl;}
    public:
        int pubc;
    protected:
        int proc;
    private:
        int pric;
};

//基类a
class a{
    public:
        a(int pub=1,int pro=2,int pri=3):puba(pub),proa(pro),pria(pri){cout<<"construct:a()"
<<endl;cout<<"&a.pub:"<<&puba<<endl;cout<<"a.this:"<<this<<" type:"<<typeid(this).name()<<endl;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int puba;
    protected:
        int proa;
    private:
        int pria;
};

//子类b继承类a，并且内嵌类c
class b:private a{
    public:
        b(int pub=11,int pro=21,int pri=31,int pubc=12):cc(pubc),pubb(pub),prob(pro),prib(pri)
{cout<<"construct:b()"<<endl;cout<<"&b.pub:"<<&pubb<<endl;cout<<"b.this:"<<this<<" type:"
<<typeid(this).name()<<endl;}
        b(int pub=11,int pro=21,int pri=31,int pubc=12,int puba=1,int proa=2,int
pria=3):a(puba,proa,pria),cc(pubc),pubb(pub),prob(pro),prib(pri){cout<<"construct:b()"
<<endl;cout<<"&b.pub:"<<&pubb<<endl;cout<<"b.this:"<<this<<" type:"<<typeid(this).name()<<endl;}

        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int pubb;
    protected:
        int prob;
    private:
        int prib;
    //探究子类内访问成员
    public:
        void print(){
            //public继承
//          cout<< puba <<endl;
//          cout<< proa <<endl;
//          cout<< pubb <<endl;
//          cout<< prob <<endl;
//          cout<< prib <<endl;

            //protected继承
//          cout<< puba <<endl;

//          cout<< proa <<endl;
```

```
//            cout<< pubb <<endl;
//            cout<< prob <<endl;
//            cout<< prib <<endl;
//            cout<< cc.pubc <<endl;

              //private继承
              cout<< puba <<endl;
              cout<< proa <<endl;
              cout<< pubb <<endl;
              cout<< prob <<endl;
              cout<< prib <<endl;
              cout<< cc.pubc <<endl;
          }
      //探究内嵌对象的构造与析构问题
      public:
          c cc;
};

int main()
{
      //探究各继承方式的区别
      b ba(11,21,31,12,1);
//探究子类外部访问成员
  //public继承
//  cout<< ba.puba <<endl;
//  cout<< ba.pubb <<endl;
//  cout<< ba.cc.pubc <<endl;
  //protected继承
//  cout<< ba.pubb <<endl;
//  cout<< ba.cc.pubc <<endl;
  //private继承
//  cout<< ba.pubb <<endl;
//  cout<< ba.cc.pubc <<endl;

  //探究子类内部访问成员
  ba.print();

  //探究大小
  a aa;
  cout<< sizeof(a) <<endl; //size=12，父类和子类的数据成员都是12字节
  cout<< sizeof(ba) <<endl; //size=36=2*12+12=3*12，说明子类全盘接收父类的东西

  return 0;
}


/*运行结果：*/
/*
construct:a()
&a.pub:0x6dfe60
a.this:0x6dfe60 type:P1a
construct:c()

construct:b()
```

```
&b.pub:0x6dfe6c
b.this:0x6dfe60 type:P1b
1
2
11
21
31
12
construct:a()
&a.pub:0x6dfe84
a.this:0x6dfe84 type:P1a
12
36
free:~a()
free:~b()
free:~c()
free:~a()
*/
```

派生类的拷贝构造：

构造函数和拷贝构造函数都不会被继承。

```
//总结(对于浅拷贝，默认就够了)：
//如果不实现拷贝构造，则使用系统默认的拷贝构造
//如果子类不实现拷贝构造时，则调用父类拷贝构造函数；如果在子类里实现拷贝构造，则会先调用父类的空体构造函数，然后在调用此拷贝构造
//如果子类和父类都没有拷贝构造函数，则不会发生拷贝函数或构造函数调用
//能正常实现拷贝构造的四种情况，第4种是正确的方式：
//1、子类和父类都不实现拷贝构造函数,使用系统默认的；2、子类实现包含所有成员拷贝的实体拷贝构造函数；3、父类实现自己成员拷贝的拷贝构造函数，子类不实现 ；
//4、父类和子类分别实现各自的类，不过子类的拷贝构造要调用父类的拷贝构造 （正确的方式，可用于深拷贝的自实现）

/* 子类b和父类a都没显示加拷贝构造，使用默认，此时拷贝构造正常*/
#include <iostream>
using namespace std;

class a{
    public:
        a():ai(10){cout<<"empty construct:a()"<<endl;}
        a(int ia):ai(ia){cout<<"params construct:a(ia)"<<endl;}
//      a(const a& another){cout<<"empty copy construct:a()"<<endl;}
//      a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};

class b:public a{

    public:
```

```
        b():bi(20){cout<<"empty construct:b()"<<endl;}
        b(int ia,int ib):a(ia),bi(ib){cout<<"params construct:b(ia,ib)"<<endl;}
//      b(const b& another){cout<<"empty copy construct:b()"<<endl;}
//      b(const b& another){ai=another.ai;bi=another.bi;cout<<"copy construct:b(another)"
<<endl;}
        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
};

int main(){
    //创建对象
    b b1(1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    //拷贝构造创建对象
    b b2(b1);
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;

    return 0;
}

/*运行结果:
construct:a()
construct:b()
b1.ai:1
b1.bi:2
b2.ai:1
b2.bi:2
free:~b()
free:~a()
free:~b()
free:~a()
*/


/* 子类显示空体拷贝构造，父类显示或不显示，拷贝非正常*/
#include <iostream>
using namespace std;

class a{
    public:
        a():ai(10){cout<<"empty construct:a()"<<endl;}
        a(int ia):ai(ia){cout<<"params construct:a(ia)"<<endl;}
        a(const a& another){cout<<"empty copy construct:a()"<<endl;}
//      a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};


class b:public a{
```

```
    public:
        b():bi(20){cout<<"empty construct:b()"<<endl;}
        b(int ia,int ib):a(ia),bi(ib){cout<<"params construct:b(ia,ib)"<<endl;}
        b(const b& another){cout<<"empty copy construct:b()"<<endl;}
//      b(const b& another){ai=another.ai;bi=another.bi;cout<<"copy construct:b(another)"
<<endl;}
        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
};

int main(){
    //创建对象
    b b1(1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    //拷贝构造创建对象
    b b2(b1);
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;

    return 0;
}

/*运行结果:
params construct:a(ia)
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
empty construct:a()
empty copy construct:b()
b2.ai:10
b2.bi:65535  乱七八糟，因为空体拷贝，值不确定
free:~b()
free:~a()
free:~b()
free:~a()
*/


/* 父类显示空体构造，子类不显示，此时拷贝非正常*/
#include <iostream>
using namespace std;

class a{
    public:
        a():ai(10){cout<<"empty construct:a()"<<endl;}
        a(int ia):ai(ia){cout<<"params construct:a(ia)"<<endl;}
        a(const a& another){cout<<"empty copy construct:a()"<<endl;}
//      a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:

        int ai;
```

```
};

class b:public a{
    public:
        b():bi(20){cout<<"empty construct:b()"<<endl;}
        b(int ia,int ib):a(ia),bi(ib){cout<<"params construct:b(ia,ib)"<<endl;}
//      b(const b& another){cout<<"empty copy construct:b()"<<endl;}
//      b(const b& another){ai=another.ai;bi=another.bi;cout<<"copy construct:b(another)"
<<endl;}
        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
};

int main(){
    //创建对象
    b b1(1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    //拷贝构造创建对象
    b b2(b1);
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;

    return 0;
}

/*运行结果:
params construct:a(ia)
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
empty copy construct:a()
b2.ai:4249168  乱七八糟，因为空体拷贝，值不确定
b2.bi:2
free:~b()
free:~a()
free:~b()
free:~a()
*/

/* 子类不实现拷贝构造，父类实现自己的实体拷贝构造，拷贝正常*/

#include <iostream>
using namespace std;

class a{
    public:
        a():ai(10){cout<<"empty construct:a()"<<endl;}
        a(int ia):ai(ia){cout<<"params construct:a(ia)"<<endl;}
//      a(const a& another){cout<<"empty copy construct:a()"<<endl;}
        a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}

        ~a(){cout<<"free:~a()"<<endl;}
```

```
    public:
        int ai;
};

class b:public a{
    public:
        b():bi(20){cout<<"empty construct:b()"<<endl;}
        b(int ia,int ib):a(ia),bi(ib){cout<<"params construct:b(ia,ib)"<<endl;}
//      b(const b& another){cout<<"empty copy construct:b()"<<endl;}
//      b(const b& another){ai=another.ai;bi=another.bi;cout<<"copy construct:b(another)"
<<endl;}
        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
};

int main(){
    //创建对象
    b b1(1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    //拷贝构造创建对象
    b b2(b1);
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;

    return 0;
}

/*运行结果:
params construct:a(ia)
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
copy construct:a(another)
b2.ai:1
b2.bi:2
free:~b()
free:~a()
free:~b()
free:~a()
*/

/*子类实现实体拷贝构造，父类实现或不实现，拷贝构造正常*/

#include <iostream>
using namespace std;

class a{
    public:
        a():ai(10){cout<<"empty construct:a()"<<endl;}
        a(int ia):ai(ia){cout<<"params construct:a(ia)"<<endl;}

//      a(const a& another){cout<<"empty copy construct:a()"<<endl;}
```

```
        a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};

class b:public a{
    public:
        b():bi(20){cout<<"empty construct:b()"<<endl;}
        b(int ia,int ib):a(ia),bi(ib){cout<<"params construct:b(ia,ib)"<<endl;}
//      b(const b& another){cout<<"empty copy construct:b()"<<endl;}
        b(const b& another){ai=another.ai;bi=another.bi;cout<<"copy construct:b(another)"
<<endl;}
        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
};

int main(){
    //创建对象
    b b1(1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    //拷贝构造创建对象
    b b2(b1);
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;

    return 0;
}

/*运行结果:
params construct:a(ia)
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
empty construct:a()
copy construct:b(another)
b2.ai:1
b2.bi:2
free:~b()
free:~a()
free:~b()
free:~a()
*/

/* 子类的拷贝构造的正确打开方式：在继承关系里的正确的拷贝构造 */
#include <iostream>
using namespace std;

class a{
    public:

        a(int ia=10):ai(ia){cout<<"params construct:a(ia)"<<endl;}
```

```
        a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};

class b:public a{
    public:
        b(int ia=10,int ib=20):a(ia),bi(ib){cout<<"params construct:b(ia,ib)"<<endl;}
        b(const b& another):a(another){bi=another.bi;cout<<"copy construct:b(another)"<<endl;}

        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
};

int main(){
    //创建对象
    b b1(1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    //拷贝构造创建对象
    b b2(b1);
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;

    return 0;
}

/*运行结果:
params construct:a(ia)
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
copy construct:a(another)
copy construct:b(another)
b2.ai:1
b2.bi:2
free:~b()
free:~a()
free:~b()
free:~a()
*/
```

不含嵌入对象的派生类拷贝构造函数的正确打开方式:

```
/* 子类的拷贝构造的正确打开方式: 在继承关系里的正确的拷贝构造 */
#include <iostream>
using namespace std;

class a{
```

```
    public:
        a(int ia=10):ai(ia){cout<<"params construct:a(ia)"<<endl;}
        a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};

class b:public a{
    public:
        b(int ia=10,int ib=20):a(ia),bi(ib){cout<<"params construct:b(ia,ib)"<<endl;}
        b(const b& another):a(another){bi=another.bi;cout<<"copy construct:b(another)"<<endl;}

        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
};

int main(){
    //创建对象
    b b1(1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    //拷贝构造创建对象
    b b2(b1);
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;

    return 0;
}

/*运行结果:
params construct:a(ia)
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
copy construct:a(another)
copy construct:b(another)
b2.ai:1
b2.bi:2
free:~b()
free:~a()
free:~b()
free:~a()
*/
```

包含嵌入对象的派生类拷贝构造函数:

```
//总结:
//子类不实现拷贝构造函数时，默认调用嵌入类的拷贝构造函数
//子类实现拷贝构造函数时，先调用嵌入类的构造函数(不是嵌入类的拷贝构造)，然后再调用子类的拷贝构造函数
```

```
#include <iostream>
using namespace std;
//嵌入类
class c{
    public:
        c(int ic=0):ci(ic){cout<<"params construct:c(ic)"<<endl;}
        c(const c& another){ci=another.ci;cout<<"copy construct:c(another)"<<endl;}
        ~c(){cout<<"free:~c()"<<endl;}
    public:
        int ci;
};
//基类
class a{
    public:
        a(int ia=10):ai(ia){cout<<"params construct:a(ia)"<<endl;}
        a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};
//子类
class b:public a{
    public:
        b(int ia=10,int ic=0,int ib=20):a(ia),cc(ic),bi(ib){cout<<"params construct:b(ia,ib)"
<<endl;}
//      b(const b& another):a(another){bi=another.bi;cout<<"copy construct:b(another)"<<endl;}

        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
    //嵌入的对象
    public:
        c cc;
};

int main(){
    //创建对象
    b b1(1,-1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    cout<<"b1.cc.ci:"<<b1.cc.ci<<endl;
    //拷贝构造创建对象
    b b2(b1);
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;
    cout<<"b2.cc.ci:"<<b1.cc.ci<<endl;

    return 0;
}

/*运行结果:
params construct:a(ia)

params construct:c(ic)
```

```
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
b1.cc.ci:-1
copy construct:a(another)
copy construct:c(another)
b2.ai:1
b2.bi:2
b2.cc.ci:-1
free:~b()
free:~c()
free:~a()
free:~b()
free:~c()
free:~a()
*/
```

包含嵌入对象的派生类拷贝构造的正确打开方式：

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

//嵌入类
class c{
    public:
        c(int ic=0):ci(ic){cout<<"params construct:c(ic)"<<endl;}
        c(const c& another){ci=another.ci;cout<<"copy construct:c(another)"<<endl;}
        ~c(){cout<<"free:~c()"<<endl;}
    public:
        int ci;
};
//基类
class a{
    public:
        a(int ia=10):ai(ia){cout<<"params construct:a(ia)"<<endl;}
        a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};
//子类
class b:public a{
    public:
        b(int ia=10,int ic=0,int ib=20):a(ia),cc(ic),bi(ib){cout<<"params construct:b(ia,ib)"
<<endl;}
        b(const b& another):a(another),cc(another.cc){bi=another.bi;cout<<"copy
construct:b(another)"<<endl;}
        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
    //嵌入的对象
```

```
    public:
        c cc;
};

int main(){
    //创建对象
    b b1(1,-1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    cout<<"b1.cc.ci:"<<b1.cc.ci<<endl;
    //拷贝构造创建对象
    b b2(b1);
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;
    cout<<"b2.cc.ci:"<<b1.cc.ci<<endl;

    return 0;
}

/*运行结果:
params construct:a(ia)
params construct:c(ic)
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
b1.cc.ci:-1
copy construct:a(another)
copy construct:c(another)
copy construct:b(another)
b2.ai:1
b2.bi:2
b2.cc.ci:-1
free:~b()
free:~c()
free:~a()
free:~b()
free:~c()
free:~a()
*/
```

派生类的赋值重载:

```
//总结:
//
/* 不含内嵌子对象时 */
#include <iostream>
using namespace std;

//基类
class a{
    public:
        a(int ia=10):ai(ia){cout<<"params construct:a(ia)"<<endl;}
```

```
        a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        a& operator= (const a& another){ai=another.ai;cout<<"operator=:a&(another)"
<<endl;return *this;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};
//子类
class b:public a{
    public:
        b(int ia=10,int ib=20):a(ia),bi(ib){cout<<"params construct:b(ia,ib)"<<endl;}
        b(const b& another):a(another){bi=another.bi;cout<<"copy construct:b(another)"<<endl;}

        b& operator= (const b& another){a::operator=(another);bi=another.bi;cout<<"operator=:b&
(another)"<<endl;return *this;}
        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
};

int main(){
    //创建对象
    b b1(1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    //赋值
    b b2(3,4);
    b2=b1;
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;
    return 0;
}

/*运行结果:
params construct:a(ia)
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
params construct:a(ia)
params construct:b(ia,ib)
operator=:a&(another)
operator=:b&(another)
b2.ai:1
b2.bi:2
free:~b()
free:~a()
free:~b()
free:~a()
*/

/* 包含内嵌子对象时 */
#include <iostream>

using namespace std;
```

```
//嵌入类
class c{
    public:
        c(int ic=0):ci(ic){cout<<"params construct:c(ic)"<<endl;}
        c(const c& another){ci=another.ci;cout<<"copy construct:c(another)"<<endl;}
        c& operator= (const c& another){ci=another.ci;cout<<"operator=:c&(another)"
<<endl;return *this;}
        ~c(){cout<<"free:~c()"<<endl;}
    public:
        int ci;
};
//基类
class a{
    public:
        a(int ia=10):ai(ia){cout<<"params construct:a(ia)"<<endl;}
        a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        a& operator= (const a& another){ai=another.ai;cout<<"operator=:a&(another)"
<<endl;return *this;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};
//子类
class b:public a{
    public:
        b(int ia=10,int ic=0,int ib=20):a(ia),cc(ic),bi(ib){cout<<"params construct:b(ia,ib)"
<<endl;}
        b(const b& another):a(another),cc(another.cc){bi=another.bi;cout<<"copy
construct:b(another)"<<endl;}
        b& operator= (const b& another){a::operator=
(another);cc=another.cc;bi=another.bi;cout<<"operator=:b&(another)"<<endl;return *this;}
        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
    public:
        c cc;
};

int main(){
    //创建对象
    b b1(1,-1,2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    cout<<"b1.cc.ci:"<<b1.cc.ci<<endl;
    //赋值
    b b2(3,-2,4);
    b2=b1;
    cout<<"b2.ai:"<<b2.ai<<endl;
    cout<<"b2.bi:"<<b2.bi<<endl;
    cout<<"b2.cc.ci:"<<b2.cc.ci<<endl;
    return 0;

}
```

```
/*运行结果:
params construct:a(ia)
params construct:c(ic)
params construct:b(ia,ib)
b1.ai:1
b1.bi:2
b1.cc.ci:-1
params construct:a(ia)
params construct:c(ic)
params construct:b(ia,ib)
operator=:a&(another)
operator=:c&(another)
operator=:b&(another)
b2.ai:1
b2.bi:2
b2.cc.ci:-1
free:~b()
free:~c()
free:~a()
free:~b()
free:~c()
free:~a()
*/
```

派生类的友元函：友元函数不能被继承，但是有时又需要父类的友元函数,达到代码重用的效果。

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

//基类
class a{
    friend ostream& operator<< (ostream& out, a& aa);
    public:
        a(int ia=10):ai(ia){cout<<"params construct:a(ia)"<<endl;}
        a(const a& another){ai=another.ai;cout<<"copy construct:a(another)"<<endl;}
        a& operator= (const a& another){ai=another.ai;cout<<"operator=:a&(another)"
<<endl;return *this;}
        ~a(){cout<<"free:~a()"<<endl;}
    public:
        int ai;
};
//友元函数
ostream& operator<< (ostream& out, a& aa){
    cout<<"ai:"<<aa.ai<<endl;
    return out;
}

//子类
class b:public a{
    friend ostream& operator<< (ostream& out, b& bb){
        cout<<(a&)bb;
```

```
        cout<<"bi:"<<bb.bi<<endl;
        return out;
    }
    public:
        b(int ia=10,int ib=20):a(ia),bi(ib){cout<<"params construct:b(ia,ib)"<<endl;}
        b(const b& another):a(another){bi=another.bi;cout<<"copy construct:b(another)"<<endl;}

        b& operator= (const b& another){a::operator=(another);bi=another.bi;cout<<"operator=:b&
(another)"<<endl;return *this;}
        ~b(){cout<<"free:~b()"<<endl;}
    public:
        int bi;
};


int main(){
    //创建对象
    b b1(1,-2);
    cout<<"b1.ai:"<<b1.ai<<endl;
    cout<<"b1.bi:"<<b1.bi<<endl;
    cout<<b1<<endl;
    return 0;
}

/*运行结果:
params construct:a(ia)
params construct:b(ia,ib)
b1.ai:1
b1.bi:-2
ai:1
bi:-2

free:~b()
free:~a()
*/
```

派生类的析构：1、析构顺序和构造顺序相反，见重载的类型那里。2、无需指明析构关系，因为析构函数只有一种，无重载，无默认参数。

隐藏：父子类或多继承中的多个类出现了同名的标识符（函数，变量），这时需要加父类的作用域A::。

多重继承:继承多个父类，每个父类都可以有不同的继承方式，好处是可使用继承自不同类的方法，不好的地方是，如果有重名会出现二义性。使用作用域可以避免二义性，其实继承自不同类中同名的成员是有作用域分开的，作用域就是类名。使用虚继承可以解决成员重名问题。

```
/* 不使用虚继承 */
#include <iostream>
using namespace std;

class base{
    public:
        base(int var=0):var_base(var),name("base"){cout<<"base(var=0)"<<endl;}

        ~base(){cout<<"~base()"<<endl;}
```

```
    public:
        void print(){
            cout<<"var_base:"<<var_base<<endl;
        }
    private:
        string name;
        int var_base;
};

class base1{
    public:
        base1(int var=1):var_base1(var),name("base1"){cout<<"base1(var=1)"<<endl;}
        ~base1(){cout<<"~base1()"<<endl;}
    public:
        void print(){
            cout<<"var_base1:"<<var_base1<<endl;
        }
    private:
        string name;
        int var_base1;
};

class a:public base,public base1{
    public:
        a(int var=2,int var0=0,int var1=1):base(var0),base1(var1),var_a(var),name("a")
{cout<<"a(var=2)"<<endl;}
        ~a(){cout<<"~a()"<<endl;}
    public:
        void print(){
            cout<<"var_a:"<<var_a<<endl;
        }
    private:
        string name;
        int var_a;
};

int main(){
    a aa(3,1,2);
    aa.print(); //调用a自身的print()
    aa.base::print(); //调用继承自base类的print()
    aa.base1::print(); //调用继承自base1类的print()
}

/*运行结果:
base(var=0)
base1(var=1)
a(var=2)
var_a:3
var_base:1
var_base1:2
~a()
~base1()

~base()
```

```
*/

/* 使用虚继承 */
#include <iostream>
using namespace std;

class base_base{
    public:
        base_base(){cout<<"base_base()"<<endl;}
        ~base_base(){cout<<"~base_base()"<<endl;}
    public:
        void print(){cout<<"var_base_base:"<<var_base_base<<endl;}
    protected:
        int var_base_base;
};
class base:virtual public base_base{
    public:
        base(int var=0):name("base"){var_base_base=var;cout<<"base(var=0)"<<endl;}
        ~base(){cout<<"~base()"<<endl;}
    public:
        void set_var(int var){
            var_base_base=var;
        }
    private:
        string name;
};

class base1:virtual public base_base{
    public:
        base1(int var=1):name("base1"){var_base_base=var;cout<<"base1(var=1)"<<endl;}
        ~base1(){cout<<"~base1()"<<endl;}
    public:
        int get_var(){
            return var_base_base;
        }
    private:
        string name;
};

class a:public base,public base1{
    public:
        a(int var=1):name("a"){var_base_base=var;cout<<"a(var=1)"<<endl;}
        ~a(){cout<<"~a()"<<endl;}
    private:
        string name;
};

int main(){
    //创建对象
    a aa(-1);
    aa.print();
    //修改值

    aa.set_var(-100);
```

```
    cout<<aa.get_var()<<endl;
    aa.print();
}

/*运行结果:
base_base()
base(var=0)
base1(var=1)
a(var=1)
var_base_base:-1
-100
var_base_base:-100
~a()
~base1()
~base()
~base_base()
*/
```

# 多态

基本定义：具有继承关系的类，这些对象对同一消息会作出不同的响应。

赋值兼容：在需要基类对象的任何地方，都可以使用该基类的公有派生类的对象来代替。赋值兼容是多态的前提，是一种默认的行为，不需要任何的显示的转化步骤，并且是安全的。赋值兼容的三个特点：

```
//总结:
//1、公有派生类对象可以赋值给基类对象
//2、公有派生类对象可以初始化基类对象的引用
//3、公有派生类对象的地址可以赋给指向基类的指针
//第3点与多态关系更紧密，因为父类指针只会访问父类成员的空间
//从结果看，子类对象(引用、指针）赋给父类对象后，父类对象(引用、指针）看不见子类的成员，无法调用子类的成员,因此是安全的
//相反，父类对象(指针）赋给子类对象，扩大了指针的作用域，是不安全的。
#include <iostream>
using namespace std;

class a{
    public:
        a(int ia=1):ai(ia){cout<<"a()"<<endl;}
    public:
        int ai;
};

class b:public a{
    public:
        b(int ib=2,int ia=3):a(ia),bi(ib){cout<<"b()"<<endl;}
    public:
        int bi;
};
int main(){
    a aa(-1);
```

```
    b bb(-2,-3);
    //1、公有派生类对象可以赋值给基类对象
    aa=bb;
    cout<<aa.ai<<endl;
    //2、公有派生类对象可以初始化基类对象的引用
    a& a2 = bb;
    cout<<a2.ai<<endl;
    //3、公有派生类对象的地址可以赋给指向基类的指针
    a* pa = &bb;
    cout<<"&bb:"<<&bb<<" "<<bb.ai<<endl;
    cout<<"pa:"<<pa<<" "<<pa->ai<<endl;
}

/*运行结果:
baa()
a()
b()
-3
-3
&bb:0x6ffe10 -3
pa:0x6ffe10 -3
*/
```

多态的类别：静态多态和动态多态。静多态就是函数重载，表面上是由重载规则来限定，实质是命名倾轧，发生在编译期，故称为静多态。动态多态不是发生在编译时期，而是在运行阶段决定，故称为动态多态。

动态多态形成条件：1、父类中要有虚函数，即公共接口；2、子类覆写父类中的虚函数；3、通过已被子类对象赋值的父类指针，调用共用接口。

```
//总结:
//动态多态形成条件:
//1、父类中要有虚函数(virtual)，即公共接口;
//2、子类覆写父类中的虚函数;
//3、通过已被子类对象赋值的父类指针，调用共用接口。
//动多态本质原理:
//在编译时不知道调用那个函数，而是在运行时才能确定，virtual关键字来达到此目的。
//多态的语法:
//多态中的virtual与虚继承中的virtual没有关系，前面的virtual是为了实现多态的目的，后面的virtual是为了实现虚继承。
//子类覆写是指：同名，同参，同返回值，override。
//虚函数在子类中的访问属性并不影响多态。具体需求要看子类。

#include <iostream>
using namespace std;
//父类
class father{
    public:
        father(int f=1):fer(f){}
        ~father(){}
    public:
        //条件1: virtual
        virtual void print();

    private:
```

```
        int fer;
};
void father::print(){cout<<"fer:"<<fer<<endl;}

//子类
class child:public father{
    public:
        child(int c=2,int f=1):father(f),cer(c){}
        ~child(){}
    public:
        //条件2: 子类覆写
        virtual void print();
    private:
        int cer;
};
void child::print(){cout<<"cer:"<<cer<<endl;}

//子子类
class grandson:public child{
    public:
        grandson(int g=3,int c=2,int f=1):child(c,f),ger(g){}
        ~grandson(){}
    public:
        //条件2: 子类覆写
        virtual void print();
    private:
        int ger;
};
void grandson::print(){cout<<"ger:"<<ger<<endl;}

int main(){
    father f(-1), *pf;
    child c(-2,-3), *pc;
    grandson g(-3,-4,-5);

    f.print();//调用的是父类本身的print()
    c.print();//调用的是子类本身的print()
    g.print();

    //条件3:父类指针被子类对象赋值
    pf = &c;
    pf->print(); //调用子类的print()

    pf = &g;
    pf->print();

    pc = &g;
    pc->print();

    return 0;
}


/*运行结果:
```

```
fer:-1
cer:-2
ger:-3
cer:-2
ger:-3
ger:-3
*/
```

纯虚函数与需析构：

```
//总结:
//纯虚函数:
//1、无实现体，初始化为0;
//2、拥有纯虚函数的类成为 抽象基类，并且抽象基类不能被实例化;
//3、用作接口类使用;
//4、如果派生类没有实现纯虚函数，那么该虚函数在派生类中仍然为纯虚函数，派生类仍然为纯虚基类。
//虚析构:
//1、含有虚函数的类，析构函数也应该为虚函数，也即虚析构函数
//2、因为父类指针指向的是堆上的子对象，当delete该指针时会调用子类的析构函数，实现完整的析构。

#include <iostream>
using namespace std;
//虚基类
class father{
    public:
        father(int f=1):fer(f){cout<<"father()"<<endl;}
        virtual ~father(){cout<<"~father()"<<endl;}
    public:
        //条件1: 纯虚函数: virtual ... = 0;
        virtual void print()=0;
    private:
        int fer;
};

//子类
class child:public father{
    public:
        child(int c=2,int f=1):father(f),cer(c){cout<<"child()"<<endl;}
        ~child(){cout<<"~child()"<<endl;}
    public:
        //条件2: 子类覆写
        virtual void print();
    private:
        int cer;
};
void child::print(){cout<<"cer:"<<cer<<endl;}

//子子类
class grandson:public child{
    public:
        grandson(int g=3,int c=2,int f=1):child(c,f),ger(g){cout<<"grandson()"<<endl;}
        ~grandson(){cout<<"~grandson()"<<endl;}
    public:
```

```
        //条件2: 子类覆写
        virtual void print();
    private:
        int ger;
};
void grandson::print(){cout<<"ger:"<<ger<<endl;}

int main(){
//  father f(-1); //抽象基类不能被实例化
    father *pf;
    child *pc;
    child* c = new child(-2,-3); //堆对象
    grandson* g = new grandson(-3,-4,-5); //堆对象

    c->print();//调用的是子类本身的print()
    g->print();

    //条件3:父类指针被子类对象赋值
    pf = c;
    pf->print(); //调用子类的print()

    pc = g;
    pc->print();

    //析构
    delete pc;
    delete pf;

    return 0;
}

/*运行结果:
//虚基类中的析构不是虚析构时(无virtual),这时无法调用子类对象自身的析构函数
father()
child()
father()
child()
grandson()
cer:-2
ger:-3
cer:-2
ger:-3
~child()
~father()
~father()
//虚基类里有虚析构时(virtual),这时能调用子类对象自身的析构函数了
father()
child()
father()
child()
grandson()
cer:-2

ger:-3
```

```
cer:-2
ger:-3
~grandson()
~child()
~father()
~child()
~father()
*/
```

倒置依赖原则（DIP）实现多态：设计模式中有个很重要的原则，就叫倒置依赖原则，是基于多态的。DIP的设计原则：将中间层抽象为抽象层，让高层模块和低层模块依赖于中间抽象层。核心思想是面向接口编程。减少强耦合。具体是高层依赖接口，低层实现接口。

```
//总结:
//倒置依赖原则：面向接口编程，减少强耦合，尽量减少高层的修改;
//具体是：高层依赖接口，低层实现接口。

#include <iostream>
using namespace std;

//中间接口
class story{
    public:
        virtual string getstory()=0;
};
//低层的东西
class book:public story{
    public:
        string getstory(){
            return "在很久很久以前，整个世界混沌一片，盘古挥着斧头，将天地分开...";
        }
};
//上层调用中间接口，低层变化，上层不用改变
class newspaper:public story{
    public:
        string getstory(){
            return "2018年的钟声即将敲响，在此祝贺大家，在即将到来的戊戌狗年里旺旺旺...";
        }
};

class mother{
    public:
        //函数重载
        void tellstroy(story& s){
            cout<< s.getstory() <<endl;
        }
        void tellstroy(story* ps){
            cout<< ps->getstory() <<endl;
        }
};

int main(){
```

```
    story *s;
    book b;
    newspaper np;

    mother m;

    s=&b;
    m.tellstroy(s);
    s=&np;
    m.tellstroy(s);

    m.tellstroy(b);
    m.tellstroy(np);

    return 0;
}

/*运行结果:
在很久很久以前，整个世界混沌一片，盘古挥着斧头，将天地分开...
2018年的钟声即将敲响，在此祝贺大家，在即将到来的戊戌狗年里旺旺旺...
在很久很久以前，整个世界混沌一片，盘古挥着斧头，将天地分开...
2018年的钟声即将敲响，在此祝贺大家，在即将到来的戊戌狗年里旺旺旺...
*/
```

# 多态实现原理--虚函数表：

虚函数表（vt）：表中主要是一个类的虚函数的地址表，保证了继承和覆写的问题。

虚函数表的内存模型：类的实例里只保存虚函数表的首地址，通过表首地址可以遍历所有的虚函数地址。并且表首地址放在类实例内存的最前面，这样能保证在多层继承和多重继承时有最高的性能。

构造结束：多态发生在构造结束之后，因为这时虚函数表才能生成。并且多态发生点不适宜放在析构函数里。

```
//总结:
//多态的实现原理--虚函数表:
//1、虚函数表：保存类中虚函数的地址表 。
//2、虚函数表的内存模型：类实例中只保存虚函数表的首地址，且在最前面。
//3、继承性

/* 不含子类继承时 */
#include <iostream>
#include <typeinfo>
using namespace std;

class a{
    public:
        a(int ia=1):ai(ia){}
        ~a(){}
    public:
        virtual void print(){cout<<"a::print()"<<endl;}
```

```
        virtual void foo(){cout<<"a::foo()"<<endl;}
        virtual void func(){cout<<"a::func()"<<endl;}
    public:
        int ai;
};

typedef void (*Func)();

int main(){
    a aa;
    //大小
    cout<<"sizeof(a):"<<sizeof(a)<<endl;  // 8, //从结果表明，类实例只保存虚函数表的首地址。

    //对象地址
    cout<<"&aa:"<<(&aa)<<endl;
    //虚函数表的地址
    cout<<"虚函数表的首地址:"<<(int**)*(int*)(&aa)<<endl;
    //虚函数表的地址
    cout<<"虚函数表的第一个函数的地址:"<<*((int**)*(int*)(&aa))<<endl;

    //调用第一个虚函数
    Func pf = (Func)*((int**)*(int*)(&aa));
    pf();
    //调用第二个虚函数
    Func pf1 = (Func)*((int**)*(int*)(&aa)+1);
    pf1();
    //调用第三个虚函数
    Func pf2 = (Func)*((int**)*(int*)(&aa)+2);
    pf2();

    return 0;
}

/*运行结果:
sizeof(a):8
&aa:0x6dfe7c
虚函数表的首地址:0x48f378
虚函数表的第一个函数的地址:0x421da0
a::print()
a::foo()
a::func()
*/

/* 包含子类继承时 */
//总结:
//多态的实现原理--虚函数表:
//1、虚函数表：保存类中虚函数的地址表 。
//2、虚函数表的内存模型：类实例中只保存虚函数表的首地址，且在最前面。
//3、继承性 :1、不仅有父类的虚函数，而且还有子类的虚函数；2、一旦子类覆写了父类的虚函数，父类的虚函数就被子类取代了；
//4、多态能发生的原因：遍历虚函数表中对应名称的函数，该函数可能被子类覆盖了，这时多态就发生了。


#include <iostream>
```

```cpp
#include <typeinfo>
using namespace std;

//函数指针
typedef void (*Func)();
//虚基类
class a{
    public:
        a(int ia=1):ai(ia){}
        ~a(){}
    public:
        virtual void print(){cout<<"a::print()"<<endl;}
        virtual void foo(){cout<<"a::foo()"<<endl;}
        virtual void func(){cout<<"a::func()"<<endl;}
    public:
        int ai;
};

//子类
class b:public a{
    public:
        b(int ib=2):bi(ib){}
        ~b(){}
    public:
        void print(){cout<<"b::print()"<<endl;}  //覆写父类a中的print()函数
        virtual void foo1(){cout<<"b::foo()"<<endl;} //自己的虚函数
        virtual void func1(){cout<<"b::func()"<<endl;} //自己的虚函数
    public:
        int bi;
};

int main(){
    a aa;
    //大小
    cout<<"sizeof(a):"<<sizeof(a)<<endl;  // 8, //从结果表明，类实例只保存虚函数表的首地址。

    //对象地址
    cout<<"&aa:"<<(&aa)<<endl;
    //虚函数表的地址
    cout<<"虚函数表的首地址:"<<(int**)*(int*)(&aa)<<endl;
    //虚函数表的地址
    cout<<"虚函数表的第一个函数的地址:"<<*((int**)*(int*)(&aa))<<endl;

    //调用第一个虚函数
    Func pf = (Func)*((int**)*(int*)(&aa));
    pf();
    //调用第二个虚函数
    Func pf1 = (Func)*((int**)*(int*)(&aa)+1);
    pf1();
    //调用第三个虚函数
    Func pf2 = (Func)*((int**)*(int*)(&aa)+2);
    pf2();
```

```
    cout<<"子类b:"<<endl;

    b bb;
    //大小
    cout<<"sizeof(b):"<<sizeof(b)<<endl;  // 8, //从结果表明，类实例只保存虚函数表的首地址。

    //对象地址
    cout<<"&bb:"<<(&bb)<<endl;
    //虚函数表的地址
    cout<<"虚函数表的首地址:"<<(int**)*(int*)(&bb)<<endl;
    //虚函数表的地址
    cout<<"虚函数表的第一个函数的地址:"<<*((int**)*(int*)(&bb))<<endl;

    //调用第一个虚函数
    Func pff = (Func)*((int**)*(int*)(&bb));
    pff();
    //调用第二个虚函数
    Func pff1 = (Func)*((int**)*(int*)(&bb)+1);
    pff1();
    //调用第三个虚函数
    Func pff2 = (Func)*((int**)*(int*)(&bb)+2);
    pff2();

    return 0;
}

/*运行结果:
sizeof(a):8
&aa:0x6dfe70
虚函数表的首地址:0x48f3f8
虚函数表的第一个函数的地址:0x421f30
a::print()
a::foo()
a::func()
子类b:
sizeof(b):12
&bb:0x6dfe64
虚函数表的首地址:0x48f410
虚函数表的第一个函数的地址:0x422028
b::print()
a::foo()
a::func()
*/
```

# 运行时类型信息RTTI:

运行时信息识别：运行时类型信息：运行程序在运行时对对象信息进行识别

typeid:是运算符,返回包含数据类型信息的type_info对象的一个引用。常用于检查调试用。

```
//总结:
//运行时类型信息：运行程序在运行时对对象信息进行识别

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

int func(int a,int b){
    return a+b;
}

class a{
    public:
        a(int ia=1):ai(ia){}
    public:
        int ai;
};

class b:public a{
    public:
        b(int ib=2,int ia=1):a(ia),bi(ib){}
    public:
        int bi;
};

int main(){
    //基本数据类型
    cout<<typeid(char).name()<<endl;
    cout<<typeid(short).name()<<endl;
    cout<<typeid(int).name()<<endl;
    cout<<typeid(long).name()<<endl;
    cout<<typeid(long long).name()<<endl;
    cout<<typeid(float).name()<<endl;
    cout<<typeid(double).name()<<endl;
    cout<<typeid(string).name()<<endl;

    //指针类型
    cout<<typeid(int*).name()<<endl;
    cout<<typeid(const int*).name()<<endl;
    cout<<typeid(int* const).name()<<endl;

    //函数类型
    void (*pf)(int);
    cout<<typeid(pf).name()<<endl;
    cout<<typeid(func).name()<<endl;

    //对象类型
    a aa, *pa;
    b bb;
    pa=&bb;
    cout<<typeid(aa).name()<<endl;

    cout<<typeid(bb).name()<<endl;
```

```
    cout<<typeid(pa).name()<<endl;

    return 0;
}

/*运行结果:
c
s
i
l
x
f
d
Ss
Pi
PKi
Pi
PFviE
FiiiE
1a
1b
P1a
*/
```

dynamic_cast：由于上转不需要显示的转换，因而dynamic_cast主要用在下转（downcast)中，注意：在多态体系中使用RTTI时，必须使用虚析构。

# 模板(Template):

作用：支持泛型编程，类型参数化，减少重载。

函数模板：

```
//总结:
//函数模板:
//1、过程：函数模板myswap ->实例化-> 模板函数myswap<int> ->调用模板函数-> myswap<int>(1,2); 也即类型检查和代码生成两部分;
//2、类型参数化：可以设置默认类型 ,但是一般不写默认，因为类型推导很强大。这也显示出了类型的参数化。
//3、缺点:不适宜处理参数类型不同的情况；适宜使用统一类型的地方。
//4、泛型编程是算法抽象的基础。

#include <iostream>
using namespace std;

template<typename T = int>
void myswap(T& a, T& b){
    T t=a;
    a=b;
    b=t;
}
```

```
int main(){

    int a=1,b=2;
    myswap(a,b);
    cout<<a<<" "<<b<<endl;
    myswap<int>(a,b);
    cout<<a<<" "<<b<<endl;

    //探究函数模板的缺点:
//  int i=1;
//  double j=1.2;
//  myswap(i,j);

    return 0;
}

/*结果:
2 1
1 2
*/
```

类模板:

```
//总结:
//函数模板:
//1、过程:函数模板myswap ->实例化-> 模板函数myswap<int>(可省略) ->调用模板函数-> myswap<int>(1,2);
也即类型检查和代码生成两部分;
//2、类型参数化:可以设置默认类型 ,但是一般不写默认,因为类型推导很强大。这也显示出了类型的参数化。
//3、缺点:不适宜处理参数类型不同的情况;适宜使用统一类型的地方。
//4、泛型编程是算法抽象的基础。
//类模板:
//1、过程:类模板a ->实例化-> 模板类a<int>(不可省略) -> 类对象a<int> aa;
//2、模板类的友元函数,常规使用方法是放在模板类的声明里;也可以砸头文件里声明,在定义里实现,只是要注意一
下下:前向类声明等,为了保证编译阶段看到模板的全部。
#include <iostream>
using namespace std;

template<typename T = int>
void myswap(T& a, T& b){
    T t=a;
    a=b;
    b=t;
}

template<typename T,class Tool>
class A;

template<typename T,class Tool>
ostream& operator<< (ostream& out, A<T,Tool>& a);

template<typename T>
```

```
class B{
    public:
        T add(T ia){
            return ia+1;
        }
};

template<typename T,class Tool>
class A{
    public:
        void setai(T ia){
            ai=ia;
        }
        T getai(){
            return ai;
        }
        //友元函数
        friend ostream& operator<< <>(ostream& out, A<T,Tool>& a);//<>表示空的意思
//      friend ostream& operator<< (ostream& out, A<T,Tool>& a){
//          out<<"A.ai:"<<a.ai<<endl;
//          return out;
//      }
    public:
        T ai;
        Tool b;
};

template<typename T,class Tool>
ostream& operator<< (ostream& out, A<T,Tool>& a){
    out<<"A.ai:"<<a.ai<<endl;
    return out;
}

int main(){

    int a=1,b=2;
    myswap(a,b);
    cout<<a<<" "<<b<<endl;
    myswap<int>(a,b);
    cout<<a<<" "<<b<<endl;

    //探究函数模板的缺点:
//  int i=1;
//  double j=1.2;
//  myswap(i,j);

    //模板类
    A<int,B<int>> aa;
    aa.setai(10);
    cout<<aa.getai()<<endl;
    cout<<aa.b.add(aa.getai())<<endl;
    //使用友元函数

    cout<<aa<<endl;
```

```
    return 0;
}

/*结果:
2 1
1 2
10
11
A.ai:10
*/
```

# 异常处理:

作用：实现返回与错误的分离 。

```
//总结:
//作用：实现返回与错误的分离
//流程：1、抛出异常码throw；2、发生了异常后，在异常语句后的语句不在被执行，捕获异常后会进入异常处理里，注意：捕获的异常码和抛出的异常码的类型需一致，不然捕获不到，会被系统杀掉。
//捕获异常的方式：自里向外层层抛出。
//显示指明函数不会抛异常,在函数f()后加throw()，即void f() thrwo() {}
//显示指明函数抛何种类型的异常，void fun() throw(char) {}
#include <iostream>
using namespace std;

int add(int a, int b){
    if(a<0||b<0)throw -1; //抛出异常码
    return a+b;
}

//显示指出该函数不抛异常
void fun() throw(){
    cout<<"我不会抛异常的！"<<endl;
}
//显示指出该函数抛char类型的异常
void func() throw(char) {
    cout<<"我会抛出char类型的异常！"<<endl;
}

int main(){

    int a,b,c;
    while(1){ //可能会出现异常的语句
        try{
            cin>>a>>b;
            c=add(a,b);
        }catch(int e){ //捕获异常

            cout<<"输入应该大于0."<<endl;
```

```
            continue;
        }
        cout<<c<<endl;
    }

    return 0;
}
```

# I/O与文件I/O:

I/O:

```
//总结:
//I/O:
//文件打开方式：ios::in读/ios::out打开/ios::trunc不存在创建，存在则清空;
//流家族:
//                              ios
//                          /     \
//                  istream        ostream
//               /     |   \     /     |    \
//    istrstream ifstream iostream ofstream ostrstream
//                            /  \
//                 strstream   fstream

//不可赋值和复制:

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

int main(){
    fstream fs("input.txt",ios::in|ios::out|ios::trunc);
    if(!fs)cout<<"error"<<endl;
    fs.put('L');
    fs.seekg(0,ios::beg);
    fs<<"lixiaobiing,2018/2/26.";

    fs.close();

}
```

文件I/O:

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

int main(){
    fstream fs("input.txt",ios::in|ios::out|ios::trunc);
    if(!fs)cout<<"error"<<endl;
    fs.put('L');
    fs.close();
}
```