C++笔记

杂项

结构体对象赋初值：使用{}

```
Student g={1000, 23};
```

引用

定义

引用就是别名。

```
int a = 10;
int &b = a;
```

注意事项

- 引用必须初始化
- 引用初始化后就不能更改

引用做函数参数

```
int Swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

函数的形参是实参的别名。

可以直接更改实参的值。

引用做函数的返回值

不要返回局部变量的引用

```
// 返回局部变量的引用
int& fun() {
    int a = 10;
    return a;
}
int main(){
    int &ref = fun();
    cout<<"ref="<<ref<<endl;
}
```

这种情况在fun函数第一次使用后会使得a被释放，ref会得到随机值。

函数的调用可以作为左值

```
int& fun2() {
    static int a = 10; // a不会被释放
    return a;
}
int main(){
    int &ref = fun2();
    cout << "ref=" << ref << endl; // 输出10
    fun2()=1000; // 返回引用 引用赋值
    cout << "ref=" << ref << endl; // 输出1000
}
```

ref是返回引用的别名。

引用的本质

本质：**指向变量的指针**

```
int a = 10;
int& ref = a; // 等于int* const ref = &a;
```

常量引用

场景：用来修饰形参，防止误操作

```
const int& ref1 = 10;// 加入const后变为只读 不可修改
// 加入const后 编译器将代码修改为int temp = 10;const int& ref1 = temp;
```

实际情况

```
void showValue(int& val){
    val = 1;
    cout<<"val="<<val<<endl;
}
int main(){
    int c =100;
    showValue(c);
    cout<<"c="<<c<<endl;
}
```

在函数中修改val值会导致其指向的内容被修改。

```
void showValue(const int& val){
    // val = 1;加入const 修改后会报错
    cout<<"val="<<val<<endl;
}
```

函数提高

默认参数

- 如果函数有默认参数，那么它们必须要在没有默认参数的形参后面。
- 如果函数声明有默认参数，那么函数实现就**不能**有参数。
- 函数声明和实现只能有一个有默认参数

函数占位参数

C++中函数的形参列表里可以有占位参数，用来占位，调用函数时必须填补该位置。

语法：返回值类型 函数名 (数据类型) {}

```
void func(int a,int){
    cout<<a<<endl;
}

int main(){
    func(19,10);
}
```

必须要写两个整形数据。

函数重载

函数重载概述

作用：函数名可以相同，提高复用性。

函数重载**满足条件**：

- 同一个作用域下
- 函数名相同
- 函数参数**类型不同** or 个数不同 or **顺序不同**

注意：函数的**返回值不可以**作为函数重载的条件。

```
// 函数重载
void func(int a,int){
    cout<<a<<endl;
}

void func(){
    cout<<"func的调用";
}

int main(){
    func(19,10);
    func();
}
```

函数重载的注意事项

- 引用作为重载条件
- 函数重载碰到函数默认参数

示例：

```
// 引用作为重载条件
void fun(int &a){
    cout<<"(int &a)引用传递"<<endl;
}

void fun(const int &a){
    cout<<"(const int &a)引用传递"<<endl;
}

int main(){
    int a=10;
    fun(a); // 调用fun(int &a)
    fun(10); // 调用fun(const int &a) 相当于const int &a = 10;
}
```

```
// 函数重载遇到默认参数
void fun2(int a, int b = 10) {
    cout << "fun2(int a, int b = 10)调用" << endl;
}

void fun2(int a) {
    cout << "fun2(int a)调用" << endl;
}
int main() {
    fun2(10); // 会出现二义性 编译报错
}
```

数组、指针、字符串

数组

对象数组

存储对象的数组。注意初始化方式。

```
Point p[3]={Point(1,2),Point(2,3),Point(3,4)};
```

指针

函数指针

函数指针是指向函数的指针变量。

声明：

```
void (*functionPointer)(float); // 数据类型 (*函数指针名)(形参表)
```

调用：

```
functionPointer=printStuff;
// 这两种方式都可以
functionPointer(PI);
(*functionPointer)(PI);
```

对象指针

对象指针是用于存放对象地址的变量。

声明：

```
Point *pointPtr;  // 类名 *对象指针名
Point p1;
pointPtr=&p1;  // 取地址
```

指向类的非静态成员的指针

类的成员自身也是一些变量、函数或者对象等，因此也可以直接将它们的地址存放在一个指针变量中，这样就可以使用指针直接指向对象的成员。

声明：

```
类型说明符 类名::*指针名;
类型说明符 (类名::*指针名)(参数表);
```

赋值(初始化)：

```
指针名=&类名::数据成员(函数)名;
```

对于一个普通变量，可以直接使用"&"进行取地址。但是对于对象来说，在定义之初并没有确定相应的变量地址，而是记录相对位置(偏移量)。

由于类是通过对象而实例化的，在声明类的对象时才会为具体的对象分配内存空间。访问数据成员时，可以通过以下两种语法形式实现：

```
对象名.*类成员指针名
对象指针名->*类成员指针名
```

一个函数的函数名就表示它的起始地址。类的成员函数虽然并不在每个对象中复制一份副本，但是由于需要确定this指针，因而必须通过对象来调用非静态成员函数。访问函数成员时，可以通过以下两种方式实现：

```
(对象名.*类成员指针名)(参数表)
(对象指针名->*类成员指针名)(参数表)
```

如果对象为常成员，则要加const。

指向类的静态成员的指针

对类的静态成员的访问是不依赖于对象的，因此可以直接通过类名进行访问赋值。

```
int *ptr=&Point::count; // 成员数据
void (*funcPtr)()=Point::showCount;  // 成员函数
```

调用：

类似函数指针，直接写。

```
funcPtr();
```

类中的数组指针

直接看源代码

```
class ArrayOfPoints{
private:
    Point *points;
    int size;
public:
    ArrayOfPoints(int size):size(size){
        points = new Point[size]; // 初始化时这里最重要
    }
    ArrayOfPoints(const ArrayOfPoints &arr);

    ~ArrayOfPoints(){
        cout<<"Deleting..."<<endl;
        delete[] points;
    }

    Point& element(int index){
        return points[index];
    }

    int getSize(){return size;}
};

ArrayOfPoints::ArrayOfPoints(const ArrayOfPoints &arr){
    this->size=arr.size;
    this->points= new Point[size];
    for (int i = 0; i < size; i++)
    {
        this->points[i]=arr.points[i];
```

```
    }

}
```

类和对象

访问权限

- 公共权限 public　　成员 **类内可以访问 类外可以访问**
- 保护权限 protected　成员 **类内可以访问 类外不可以访问** 儿子可以访问父亲的**保护内容**
- 私有权限 private　　成员 **类内可以访问 类外不可以访问（类的默认权限）**

对象的初始化和清理

构造函数语法： 类名(){}

- 构造函数没有返回值
- 构造函数名称与类名相同
- 构造函数可以有参数，可以发生重载

析构函数语法： ~类名(){}

- 析构函数没有返回值
- 析构函数名称与类名相同，并且在前面加上~
- 析构函数不可以有参数

构造函数

```
class Person{
public:
    Person(int a=0):age(a){}
private:
    int age;
};

Person::Person(int a){
    age=a; // 构造函数的类外定义
}

int main(){
    // 显示法
    Person p1;
    Person p2 = Person(10);
    Person p3 = Person(p1);

    // 隐式转换法
    Person p4 = 10; // 相当于Person p4 = Person(10)
    Person p5 = p4;
```

```
    //括号法
    Person p5(10);
}
```

注意事项：

- 调用**默认**构造函数时不要加 () 。(编译器会认为这是一个函数声明)
- 不要利用复制构造函数初始化匿名对象 Person(p3) === Person p3;
- 如果写了有参构造函数，则编译器不会提供默认构造函数
- 如果带参数的构造函数含有默认值，则在定义对象时可以不带参数（并不是调用了默认构造函数）
- **当函数中定义了带有默认值的构造函数，参数默认值是在类里面给定的而不是在类外指定**

深拷贝与冷拷贝

同时也是深拷贝与冷拷贝的问题。

```
class Person {
    public:
        Person(int a) {
            height = new int(a);
        }

        // 深拷贝 防止释放同一块内存
        Person(const Person &p){
            height = new int(*p.height);
        }

        ~Person(){
            // 析构代码，将堆区开辟数据做释放操作
            if(height!=NULL){
                delete height;
                height = NULL;
            }
        }

        int *height;
};
```

初始化列表

```
class Person {
    public:
        Person(int a,int b):A(a),B(b){}
        int A;
        int B;
};
```

类中成员包含其他类

构造时先构造成员类，再构造本类。

析构时先析构本类，再析构成员类。

静态成员

在成员变量和成员函数前面加入关键字**static**

静态成员分为：

- 静态成员变量
  - 所有对象成员共享同一份数据
  - 在编译阶段分配内存
  - 类内声明，类外初始化 (类外 int Point::count=0;)
- 静态成员函数
  - 所有对象共享同一个函数
  - 静态成员函数只能访问静态成员变量

```
class Person{
public:
    static void fun(){
        cout<<"静态函数的调用"<<endl;
        A=100;
        // B=1; 不可访问，无法区分是哪个对象的B。
    }

    static int A;
    int B;

private:
    static void fun02(){
        cout<<"私有函数"<<endl; // 私有静态函数无法通过类名访问
    }
};

int Person::A=0; // 类外初始化

void test01(){

    // 通过对象访问
    Person p;
    p.fun();

    // 通过类名访问
    Person::fun();

}
```

C++对象模型和this指针

成员变量和成员函数分开存储

空对象占用内存空间为：**1**（为了区分空对象占内存的位置）

- 非静态成员变量 **属于**类的对象上
- 静态成员变量　不属于类的对象上
- 非静态成员函数 不属于类的对象上
- 静态成员函数　不属于类的对象上

可用**sizeof**验证

this指针

背景：每一个非静态成员函数只会生成一份函数实例，那么函数该如何区分哪个对象调用自己?

C++提供特殊的对象指针：this指针。

- this指针是隐含在每一个非静态成员函数内的一种指针
- this指针不需要定义

用途：

- 当形参和成员变量重名时，可以用this指针区分
- 在类的非静态成员函数中返回对象本身，可以使用return *this
- 在打印成员数据的时候就隐性使用了this指针

本质：this是指针常量 指针的指向是不可修改的

```
class Person{
public:
    Person(int age){
        //this指针指向 被调用的成员函数 所属的对象
        this->age=age;
    }

    // 返回类型为引用才能返回本体 不然会调用复制构造函数 创造很多新对象
    Person& PersonAddAge(Person &p){
        this->age +=p.age;

        return *this; // 返回p2
    }

    int age;
};

// 解决名称冲突
void test01(){
    Person p1(18);
    cout<<"p1的年龄:"<<p1.age<<endl;
}
```

```
void test02(){
    Person p1(10);
    Person p2(10);

    // 链式编程思想
    p2.PersonAddAge(p1).PersonAddAge(p1).PersonAddAge(p1);
    cout<<"P2的年龄:"<<p2.age<<endl;
}
```

空指针访问成员函数

```
class Person{
public:
    void showClassName(){
        cout<<"this is Person class"<<endl;
    }

    void showPersonAge(){
        if(this == nullptr) return; // 检测是否为空指针
        cout<<"age="<<age<<endl;
    }

    int age;

};


void test01(){
    Person *p = nullptr;
    p->showClassName();
    p->showPersonAge(); // 直接运行会报错 空指针没有实例化对象成员
}
```

const修饰成员函数

常函数:

- 在成员函数后加**const**
- 常函数内不可修改成员属性
- 成员属性声明时加关键字**mutable**后，在常函数中就可修改
- 在成员函数后面加const **修饰的是this指针** 让指针指向的值也不可修改

常对象:

- 声明对象前加**const**
- 常对象只能调用常函数（因为非常成员函数可以修改值）

```
class Person{
public:
    // const Perosn * const this;
```

```
    // 在成员函数后面加const 修饰的是this指针 让指针指向的值也不可修改
    void showPerson() const {
        // age=10; 不可修改
        this->name="Jerry";
    }

    int age;
    mutable string name; // 特殊变量

    Person(string name){ // 需要构造函数
        this->name=name;
    }

    void showName(){
        cout<<name<<endl;
    }
};

void test02(){
    const Person p2("name"); // 常对象
}
```

继承

继承是面向对象三大特性之一。

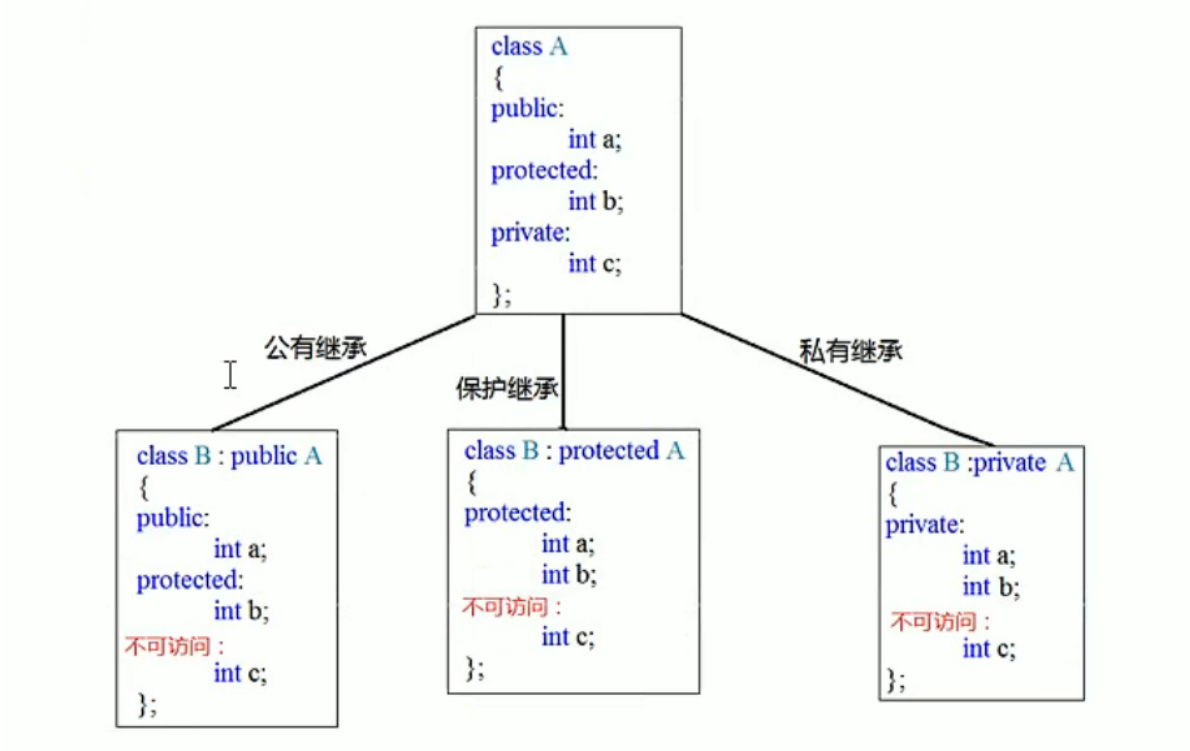
继承的基本语法

声明：

```
class son:public father // 继承方式
```

继承方式

- 公有继承
- 保护继承
- 私有继承



私有成员都无法继承。

私有继承相当于断了继承。

保护成员可以认为是一种流通的数据成员。

继承中的对象模型

父类的所有非静态成员属性都会被子类继承下去。

父类的私有成员属性是被编译器**隐藏**了。

类型兼容规则

- 派生类对象可以隐含转换为基类对象
- 派生类对象可以初始化基类的引用
- 派生类指针可以隐含转换为基类的指针

继承中构造和析构

对象创建在栈上。

|**子类**|

|**父类**|

所以构造子类的时候父类先创建，析构子类的时候子类先析构。

子类**初始化**父类成员

```
class Derived:public Base1, public Base2, public Base3{
public:
    Derived(int a, int b, int c, int d): Base1(a), member2(b), member1(c),
Base2(d){}

private:
    Base1 member1;
    Base2 member2;
    Base3 member3;
};
```

注意：

每个类的构造函数负责自己的本类成员的初始化，基类不能初始化派生类定义的成员。默认情况下**基类的构造函数不被继承**。派生类要定义自己的构造函数，而这个构造函数就存在基类的构造函数（同名），调用时派生类的构造函数给基类的构造函数传递参数。如果在派生类的构造函数中没有调用基类的构造函数，则会调用基类的**默认**构造函数（但它会被有参数的构造函数覆盖，引发报错）。

当需要执行基类中带参数的构造函数来初始化基类成员时，派生类构造函数应在初始化列表中为基类构造函数提供参数。

using声明语句

问题：如果基类的构造函数重载了很多次，则派生类的构造函数也需要重新写很多个。

方法：使用using语句继承基类的构造函数进行基类成员的初始化。此时就不必再写派生类的构造函数，可使用就地初始化的方式进行派生类成员初始化。

```
class Derived:public Base{
public:
    using Base::Base; // 继承Base的构造函数
private:
    int i = 1; // 就地初始化
};
```

注意：带有默认值的形参不会被继承。

继承同名成员处理方式

问题：子类与父类出现同名成员，如何通过子类对象，访问到子类或父类中同名数据？

方法：使用作用域分辨父 "**::**"

- 访问子类同名成员，直接访问即可
- 访问父类同名成员，需要加**作用域**
- 访问静态成员，通过类名/对象来访问 (**Son::**)

```
class Base{
public:
    int a;
    Base(){
        a=100;
    }
};
```

```
class Son:public Base{
public:
    int a;
    Son(){
        a=200;
    }
};

void test01(){
    Son s;
    cout<<s.a<<endl;
    cout<<s.Base::a<<endl;
}
```

其中作用域就是 **Base::**

如果子类中出现了和父类同名的成员函数，子类的同名成员会**隐藏**掉父类中**所有的**同名成员（**重载无法实现**）

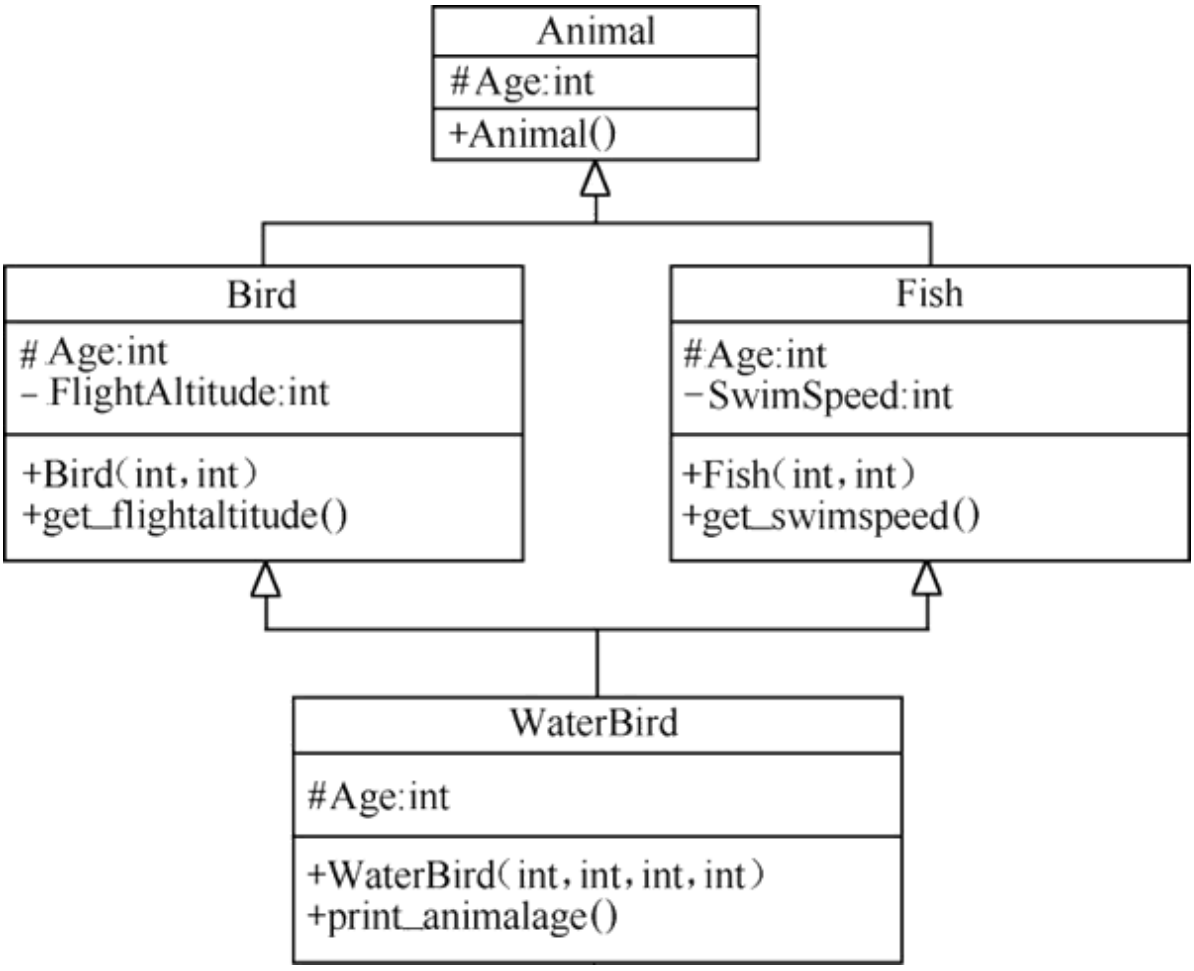
多继承时，出现同名成员，加**作用域区分**

虚基类

问题：当出现**菱形继承**状况的时候，对于间接基类成员会产生两份同名副本。

方法：将共同基类设置为虚基类。

```
class 派生类名:virtual 继承方式 基类名
```



```
class Base0{
public:
    int var0;
    void fun0(){cout<<"Member of Base0 "<<var0<<endl;}
};

class Base1:virtual public Base0{
public:
    int val1;
    void fun1(){
        Base0::fun0();
    }
};

class Base2:virtual public Base0{
public:
    int var2;
};

class Derived:public Base1, public Base2{
public:
    int var;
    void fun(){
        cout<<"Member of Derived"<<endl;
        Base1::Base0::var0; // 如果没有设置虚基类就会报错 原因未知
    }
};
```

对于上述代码，在派生时会遵循**隐藏规则**，且当出现多继承时会出现二义性导致报错。在使用虚基类后，如果在虚基类中的数据成员名是唯一的，直接使用"**对象名.成员名**"的方式就可以唯一标识和访问这些成员。

虚基类及其派生类的构造函数

如果虚基类声明中有非默认的（**即带形参的**）构造函数，并且没有什么默认形式的构造函数。此时，在整个继承关系中，直接或间接继承虚基类的所有派生类，**都必须在构造函数的成员初始化列表中列出对虚基类的初始化。**

```
class Base0 {
public:
    Base0(int a): age(a) {}
    int age = 0;
};

class Base1: virtual public Base0 {
public:
    Base1(int age): Base0(age) {}
    int var1 = 1;
};

class Base2: virtual public Base0 {
public:
    Base2(int age, int var): Base0(age), var2(var) {}
    int var2;
```

```
};

class Derived: public Base1, public Base2 {
public:
    Derived(int age, int var1, int age2, int var2)
    : Base0(age), Base1(var1), Base2(age2, var2) {}
};

int main() {
    Derived der(1, 2, 3, 4); // 定义了带参数的构造函数定义对象时一定要带参数(含初始值的不一定)
}
```

我们将建立对象的类称为最远派生类。建立一个对象时，如果这个对象中含有从虚基类继承来的成员，则虚基类的成员是由**最远派生类的构造函数通过调用虚基类的构造函数**进行初始化。

多态

多态意指相同的消息给予不同的对象会引发不同的动作（**一个接口，多种方法**）。其实更简单地来说，就是"在用**父类指针**调用函数时，实际调用的是指针指向的**子类的成员函数**"。

多态分为四种：重载多态、强制多态、包含多态和参数多态。

运算符重载

运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时导致不同的行为。

不可重载的运算符：

- 类属关系运算符“.”
- 成员指针运算符“.*”
- 作用域分辨符“::”
- 三目运算符“?:”

语法格式：

```
返回类型 类名::operator 运算符(形参表){
    函数体
}
```

如果是重载为非成员函数，则没有类名。

运算符重载为成员函数

运算符的重载实质上是成员函数。

对于**双目运算符B**，如果要重载为类的成员函数，使之能够实现表达式 **oprd1 B oprd2**，其中oprd1为A类的对象，则**应当把B重载为A类的成员函数**，该函数只有一个形参，形参的类型就是oprd2所属的类型。

对于**前置单目运算符U**，比如负号，则表达式 U oprd，其中oprd是A类的对象，则U应当重载为A类的成员函数，**函数没有形参。**

对于**后置运算符“++”和“--”**，用来实现oprd++或oprd--，那么运算符就应当重载为A类的成员函数，此时**函数要带一个整形(int)形参**。这里的int形参不起任何作用，单纯区分前置和后置“++”和“--”。

例子：对于复数运算符的重载

```
class Complex{
public:
    Complex(double r=0.0, double i=0.0)
    :real(r), imag(i){}

    Complex operator+(const Complex &c2) const {    // 重载运算符
        return Complex(real+c2.real, imag+c2.imag); // 创建一个临时无名对象作为返回值
    }
    Complex operator-(const Complex &c2) const {
        return Complex(real-c2.real, imag-c2.imag);
    }

    void display();
private:
    double real;
    double imag;
};
```

对于**前置和后置单目运算符**，有例子：

```
class Clock{
public:
    Clock& operator++();
    Clock operator++(int);
};

Clock& Clock::operator++(){ // 前置单目运算符
    second++;
    if(second>=60){
        second-=60;
        minute++;
        if(minute>=60){
            minute-=60;
            hour=(hour+1)%24;
        }
    }
    return *this; // 返回对象的引用
}

Clock Clock::operator++(int){ // 后置单目运算符
    Clock old = *this;
    ++(*this); // 调用前置“++”运算符
    return old; // 返回的是未++运算的值，而实际上已经改变了
}
```

虚函数

虚函数是动态绑定的基础。虚函数必须是非静态成员函数。

例子：P315 创建一个基类指针数组

```
Account *accounts[]={&sa1, &sa2, &cs};
for(int i=0;i<n;i++){
    accounts[i]->show();
}
```

问题：

show函数一直指向的是基类的show函数，但是我们希望指向的是派生类的show函数。

如果使用**基类类型的指针指向派生类对象**，就可以通过基类指针来访问该对象，问题是访问到的**只是从基类继承来的同名成员。**

解决方法：

如果需要通过**基类的指针**指向派生类的对象，并**访问某个与基类同名的成员**，那么首先在基类中将这个**同名函数说明成虚函数。**这样，通过基类类型指针，就可以使属于不同派生类的不同对象产生不同的行为。

一般虚函数成员

语法：

```
virtual 函数类型 函数名(形参表);
```

- **虚函数声明只能出现在类定义中的函数原型声明中，而不能在成员函数实现的时候。**
- **虚函数一般不声明为内联函数**

```
class Base1{
public:
    virtual void display() const;
};
void Base1::display() const {
    cout<<"Base1::display()"<<endl;
}

class Base2: public Base1{
public:
    void display() const;
};
void Base2::display() const {
    cout<<"Base2::display()"<<endl;
}

class Derived: public Base2 {
public:
    void display() const ;
};
void Derived::display() const {
    cout<<"Derived::display()"<<endl;
```

```
}

void fun(Base1 *ptr){ // 参数为指向基类对象的指针
    ptr->display();
}
void fun2(Base2 *ptr){ // 修改成派生类的指针时仍可通过强转完成操作
    ptr->display();
}

int main(){
    Base1 base1;
    Base2 base2;
    Derived derived;
    fun2(static_cast<Base2 *>(&base1));
    fun2(&base2);
    fun2(&derived);

    return 0;
}
```

程序中使用对象指针来访问函数成员，这样绑定的过程就是在运行中完成的。

在本程序中，派生类并没有显式给出虚函数声明，此时系统会遵循以下规则判断派生类的一个成员函数是不是虚函数。

- 该函数是否与基类的虚函数有相同的名称
- 该函数是否与基类函数的虚函数有相同的参数个数以及相同的对应参数类型
- 该函数是否与基类的虚函数有相同的返回值或满足赋值兼容规则的指针、引用型的返回值

派生类的虚函数会**隐藏**基类中的同名函数的所有其他重载形式。

注意：

- 只有虚函数是动态绑定的，如果派生类需要修改基类的行为，就应当在基类中将对应的函数声明成虚函数。
- 在重写继承来的虚函数时，如果函数有缺省形参值(int a=10)，不要重新定义不同的值
- 只有通过基类的指针或引用调用虚函数时，才会发生动态绑定。

final和override说明符

问题：派生类定义了一个函数与基类中虚函数的名字相同但是形参列表不同，派生类的函数并不会覆盖掉基类的版本。这时常会导致错误的发生，但是这种错误又难以找到。

方法：在C++11标准下，使用**override**(重写)关键字来标记派生类中的虚函数。如果使用override标记了某个函数，但该函数并没有覆盖已存在的虚函数，编译器就会报错。

```
public:
    virtual void f1(int) const;
    virtual void f2();
    void f3();
};

class Derived: public Base{
public:
    void f1(int) const override;
    void f1(int) override; // 报错 没有const
    void f2(int) override; // 报错 没有形参是这样的函数
    void f3() override; // 报错 f3不是虚函数
};
```

相应的还可以把某个函数指定为**final**，意味着该函数不能被覆盖，任何试图覆盖该函数的操作都会引发错误。

虚析构函数

在C++中，不能声明虚构造函数，但是可以声明虚析构函数。虚析构函数的作用就是当基类指针指向派生类对象时，可以调用派生类的析构函数进行内存释放（如果不是虚析构函数只会释放基类部分的数据）。

```
class Base{
public:
    virtual ~Base();
};

Base::~Base(){
    cout<<"Base destructor"<<endl;
}

class Derived: public Base{
public:
    Derived();
    ~Derived();
private:
    int *p;
};
Derived::Derived(){
    p = new int(0);
}
Derived::~Derived(){
    cout<<"Derived destructor"<<endl;
    delete p;
}

void fun(Base* b){
    delete b;
}

int main(){
    Base* b = new Derived();
    fun(b);
    return 0;
```

```
}
```

抽象类

抽象类是一种特殊的类，它为类族提供统一的操作界面。建立抽象类，就是为了通过它多态地使用其中的成员函数。抽象类处于类层次地上层，一个抽象类自身无法实例化。

纯虚函数

问题：基类中有虚函数但是并不能执行具体操作，必须在派生类中实例化操作。

方法：将基类的虚函数定义为纯虚函数，只提供整个类族的统一接口形式。

格式：

```
virtual 函数类型 函数名(形参表) = 0;
```

声明为纯虚函数后，基类就**可以**不再给出函数的实现部分。基类中仍然允许对纯虚函数给出实现，但即使给出实现，也必须由派生类覆盖，否则无法实例化。

如果将虚构函数声明为纯虚函数，就必须给出它的实现，因为派生类的析构函数执行完后需要调用基类的纯虚函数。

抽象类

带有纯虚函数的类就是抽象类。抽象类的主要作用就是通过它为一个类族建立一个公共的接口，使它们更好发挥多态性。

抽象类不能实例化。即不能定义一个抽象类的对象，但是，我们可以定义一个抽象类的指针和引用。

模板和群体数据

模板就是对参数化多态性，就是将程序所处理的对象的类型参数化，使得一段程序可以用于处理多种不同类型的对象。

函数模板

定义：

```
template<模板参数表>
类型名 函数名(参数表){
    函数表的定义
}
```

例子：

```
template<class T>
void outputArray(const T * array, int count){
    for(int i=0;i<count;i++){
        cout<<array[i]<<" ";
    }
    cout<<endl;
}

int main(){
    const int A_COUNT=8, B_COUNT=8, C_COUNT=20;
    int a[A_COUNT]={1,2,3,4,5,6,7,8};
    outputArray(a, A_COUNT);
}
```

通过T自动推断类型。

类模板

定义：

```
template<模板参数表>
class 类名{
    函数表的定义
}
```

如果需要在类模板以外定义其成员函数，则需要采用以下形式：

```
template<模板参数表>
类型名 类名<模板参数标识符列表>::函数名(参数表)
```

一个类模板声明自身并不是一个类，只有当被其他代码引用时，模板才根据引用需要生成相应具体的类。

使用一个类模板建立对象时，应使用如下形式声明：

```
模板名<模板参数表> 对象1,...,对象n;
vector<int> v; // vector就是一个类模板
```

例子：

```
struct Student{
    int id;
    float gpa;
};

template<class T>
class Store{
private:
    T item;
    bool haveValue;
public:
```

```
    Store();
    T &getElem();
    void putElem(const T &x);
};

template<class T>
Store<T>::Store():haveValue(false){}

template<class T>
T &Store<T>::getElem(){
    if(!haveValue){
        cout<<"No Item!"<<endl;
        exit(1);
    }
    return item;
}

template<class T>
void Store<T>::putElem(const T &x){
    haveValue=true;
    item = x;
}

int main(){
    Store<int> s1,s2; // 定义两个对象 其中item是int型
    s1.putElem(3);
    s2.putElem(-7);

    cout<<s1.getElem()<<" "<<s2.getElem()<<endl;

    Student g={1000, 23};
    Store<Student>s3;
    s3.putElem(g);
    cout<<"id="<<s3.getElem().id<<endl;

    Store<double> d;
    cout<<"Retrieving object d...";
    cout<<d.getElem()<<endl;
}
```

线性群体

线性群体中，按照访问元素的方式不同分为直接访问、顺序访问和索引访问。

数组类模板

一个动态数组模板：

```
#include<iostream>
#include<cstdlib>
#include<assert.h>
using namespace std;

// 数组类模板定义
template<class T>
```

```
class Array{
private:
    T* list; // T类型指针，用于存放动态分配的数组内存首地址
    int size; // 元素个数
public:
    Array(int size=50);
    Array(const Array<T>&a);
    ~Array();
    Array<T>& operator=(const Array<T>&rhs); // 重载"="使数组对象可以整体赋值
    T& operator [] (int i); // 重载[]，使得Array对象可以起到C++普通数组的作用
    const T& operator[](int i) const; // []运算符的const版本
    // 自定义类型转换
    operator T* (); // 重载到T*类型的转换，使Array对象起到C++普通数组的作用
    operator const T* () const;
    int getSize() const;
    void resize(int size);
};

// 构造函数
template<class T>
Array<T>::Array(int size){
//  assert(size>=0);
    this->size = size;
    list = new T[size];
}

// 析构函数
template<class T>
Array<T>::~Array(){
    delete[] list;
}

// 复制构造函数
template<class T>
Array<T>::Array(const Array<T>&a){
    this->size = a.size;
    this->list = new T[size];
    for(int i=0;i<size;i++){
        this->list[i]=a.list[i];
    }
}

// 重载"="运算符，将对象rhs赋值给本对象。实现对象之间的整体赋值
template<class T>
Array<T>& Array<T>::operator=(const Array<T>& rhs){
    if(&rhs!=this){
        // 如果本对象数组大小与rhs不同，则删除数组原有内存，然后重新分配
        if(this->size!=rhs.size){
            delete[] this->list;
            this->size = rhs.size;
            this->list = new T[size];
        }
        for(int i=0;i<size;i++){
            this->list[i]=rhs.list[i];
        }
    }
    return *this; // 返回当前对象的引用
}
```

```
// 重载下标运算符，实现与普通数组一样通过下标访问元素
// 并且具有越界检查功能
template<class T>
T& Array<T>::operator[](int n){
//  assert(n>=0&&n<size);
    return list[n];
}

// 重载指针转换运算符，将Array类的对象名转换为T类型的指针
// 因而可以像使用普通数组首地址一样使用Array类的对象名
// 转化操作符的重载函数不用指定返回值类型
template<class T>
Array<T>::operator T*(){
    return this->list;
}

template<class T>
Array<T>::operator const T* () const{
    return this->list;
}

// 取当前数组的大小
template<class T>
int Array<T>::getSize() const{
    return size;
}

// 将数组大小修改为size
template<class T>
void Array<T>::resize(int size){
//  assert(size>=0);
    if(size==this->size)
        return;
    T* newList=new T[size];
    int n = (size<this->size) ? size : this->size; // 将size和this size中较小的赋值
给n
    for(int i=0;i<n;i++){
        newList[i] = list[i];
    }
    delete[] list;
    list = newList;
    this->size = size;
    return ;
}

int main(){
    Array<int>(10);
}
```

泛型程序设计与C++ STL

迭代器

istream_iterator (输入迭代器)

注意:

- ++p1　对迭代器实例可以使用前置"++"使迭代器指向下一个元素，且该表达式的返回值是p1自身的引用
- p1++　对迭代器实例可以使用前置"++"使迭代器指向下一个元素，且该表达式的返回类型是不确定的

```
#include <iterator>
using namespace std;
// 调用 istream_iterator 模板类的默认构造函数，可以创建出一个具有结束标志的输入流迭代器。要知道，当我们从输入流中不断提取数据时，总有将流中数据全部提取
istream_iterator<double> eos();
// 还可以创建一个可用来读取数据的输入流迭代器
istream_iterator<double> iit(cin);
// istream_iterator 模板类还支持用已创建好的 istream_iterator 迭代器为新建
istream_iterator 迭代器初始化，例如，在上面 iit 的基础上，再创建一个相同的 iit2 迭代
istream_iterator<double> iit2(iit);
```

ostream_iterator (输出迭代器)

```
#include <iterator>
using namespace std;
// 创建输出流迭代器
ostream_iterator<int> out_it(cout);
// 创建输出流迭代器并初始化,以逗号隔开
ostream_iterator<int> out_it1(cout,",");
// 拷贝out_it1到out_it2
ostream_iterator<int> out_it2(out_it1)
```

前向迭代器

既是输入迭代器又是输出迭代器，并且可以对序列进行单向的遍历

双向迭代器

与前向迭代器相似，但是在两个方向上都可以对数据遍历

增加--p1，p1--

随机访问迭代器

也是双向迭代器，但能够在序列中的任意两个位置之间进行跳转，如指针、使用vector的begin()、end()函数得到的迭代器

类似指针

迭代器的区间

假设p1和p2是两个输入迭代器，则它们构成的区间是 **[p1, p2)**

当p1==p2时，[p1, p2)是一个没有任何元素的空区间

以下是综合运用几种迭代器的案例：

```
// 将来自输入迭代器p的n个T类型的数值排序，将结果通过输出迭代器result输出
template<class T, class InputIt, class OutputIt>
void mySort(InputIt first, InputIt last, OutputIt result){
    vector<T>s;
    for(;first!=last;++first)
        s.push_back(*first);
    sort(s.begin(),s.end()); // 对s进行排序，sort函数的参数必须是随机访问迭代器
    copy(s.begin(),s.end(),result); // 将s序列通过输出迭代器输出 将某容器中的指定元素拷贝到另一个容器中
}

int main(){
    // 将a数组内容排序后输出
    double a[]={1.2,3.2,0.8,3.3,3.2};
    mySort<double>(a,a+5,ostream_iterator<double>(cout," "));

    // 从标准输入读取若干整数，将排序后的结果输出
    mySort<int>(istream_iterator<int>(cin),istream_iterator<int>(),ostream_iterator<int>(cout," "));
    cout<<endl;

    return 0;
}
```

迭代器的辅助函数

- advance(p, n)
    - 对p执行n次自增操作
- distance(first, last)
    - 计算两个迭代器first和last的距离，即对first执行多少次“++”操作后能够使得first == last

容器

C++中的STL容器有

- 顺序容器
    - array（数组）、vector（向量）、deque（双端队列）、forward_list（单链表）、list（列表）
- (有序)关联容器
    - set（集合）、multiset（多重集合）、map（映射）、multimap（多重映射）
- 无序关联容器
    - unordered_set （无序集合）、unordered_multiset（无序多重集合）
    - unordered_map（无序映射）、unorder_multimap（无序多重映射）

容器的基本功能

功能	例子	说明
赋值	a.push_back(100);	在尾部添加元素
尾部插入	a.insert(a.end() ,10, 5);	在尾部插入10个值为5的元素
删除尾部	a.pop_back();	删除末尾元素
删除具体元素	a.erase(a.begin()+2);	删除第三个元素
排序	sort(a.begin(), a.end());	从小到大排序
交换	s1.swap(s2)	将s1和s2两容器内容交换
迭代器	S::iterator	指向容器元素的迭代器类型
常迭代器	S::const_iterator	常迭代器类型
逆向迭代器	rbegin()	指向容器尾的逆向迭代器
逆向迭代器	rend()	指向容器首的逆向迭代器

顺序容器

构造函数

```
S s(n,t); // 构造一个由n个t元素构成的容器实例s
S s(p1,p2); // 使用将区间[p1,p2)区间的数据作为s的元素构造s
```

赋值函数

类似于构造函数，函数名assign

```
s.assign(n,t);
```

插入函数

插入时需要通过一个指向当前容器元素的迭代器来指示插入位置，函数名insert

```
s.insert(p1,n,t); // 在容器s中p1所指的位置插入n个新元素t
s.insert(p1,q1,q2); // 使用将区间[q1,q2)区间的数据顺序插入到p1处
```

删除函数

函数名erase

```
s.erase(p1); // 删除s容器中p1所指的元素，返回被删除的下一个元素的迭代器
```

改变容器大小

函数名resize

```
s.resize(n); // 将容器的大小变为n 如果原有元素个数小于n，则会在容器末尾填充T()
```

首尾元素的直接访问

```
s.front(); // 获取容器首元素的引用
s.back(); // 获取容器尾元素的引用(不包括forward_list)
```

向容器收尾插入、删除元素

尾部

```
s.push_back(t);
s.emplace_back(args); // 将参数args传递给T的构造函数构造新元素t，向容器尾部插入该元素
s.pop_back();
```

头部

列表list和双端队列deque支持高效地从容器头部插入或删除新的元素，但vector不支持（但可以用insert在头部插入，效率低）。这种支持这一操作的概念构成了“前插顺序容器”

```
s.push_front(t);
s.emplace_front(args);
s.pop_front();
```

列表容器splice操作

```
#include<list>
#include<iterator>
#include<string>
#include<iostream>
using namespace std;

int main(){
    string names1[]={"Alice","Helen","Lucy","Susan"};
    string names2[]={"Bob","David","Levin","Mike"};
    list<string> s1(names1,names1+4);
    list<string> s2(names2,names2+4);
```

```
    s2.splice(s2.end(),s1,s1.begin()); // s1第一个元素放在s2后面
    list<string>::iterator iter1 = s1.begin();
    advance(iter1,2); // ITER1前进2个元素，指向第三个元素
    list<string>::iterator iter2 = s2.begin();
    ++iter2;

    list<string>::iterator iter3 = iter2;
    advance(iter3,2); // iter3前进2个元素，指向s2的第四个元素

    // 将iter2，iter3范围内的结点接到s1中iter1指向的结点前
    s1.splice(iter1,s2,iter2,iter3);

    copy(s1.begin(),s1.end(),ostream_iterator<string>(cout," "));
    cout<<endl;
    copy(s2.begin(),s2.end(),ostream_iterator<string>(cout," "));
    cout<<endl;

    return 0;
}
```

关联容器

set集合

```
#include<set>
#include<utility>
#include<iterator>
#include<iostream>

using namespace std;

int main() {
    set<double> s;
    double temp;
    while(cin>>temp){
        auto r = s.insert(temp);
        if(!r.second) // 如果r存在则提示输出信息
            cout<<temp<<" is duplicated"<<endl;
    }
    auto iter1=s.begin();
    auto iter2=s.end();
    auto medium = (*iter1+*(--iter2))/2;

    cout<<"medium="<<medium<<endl;

    cout<<"<=medium:";
    copy(s.begin(),s.upper_bound(medium),ostream_iterator<double>(cout," "));
    cout<<endl;

    cout<<">=medium:";
    copy(s.lower_bound(medium),s.end(),ostream_iterator<double>(cout," "));
    cout<<endl;
    return 0;
}
```

map映射

```
#include<iostream>
#include<map>
#include<string>
#include<utility>

using namespace std;

int main(){
    map<string,int> courses;
    courses.insert(make_pair("CSAPP", 3)); // 构造二元组
    courses.insert(pair<string, int>("C++", 2)); // 比较麻烦
    courses.insert(make_pair("CSARCH", 4));
    courses.insert(make_pair("OS",5));

    int n = 3;
    int sum = 0;
    while(n>0) {
        string name;
        cin>>name;
        auto iter = courses.find(name);
        if(iter==courses.end()) // 判断是否找到
            cout<<name<<" is not available"<<endl;
        else {
            sum += iter->second;
            courses.erase(iter); // 将刚选的课程从映射中删除
            n--;
        }
    }

    cout<<"Total credit: "<<sum<<endl;
    return 0;

}
```

multimap多重映射

多重映射是允许有重复元素的集合，多重映射是允许一个键对应多个附加数据的映射。

```
#include<iostream>
#include<map>
#include<utility>
#include<string>

using namespace std;

int main() {
    multimap<string, string> courses;
    typedef multimap<string, string>::iterator CourseIter;
    courses.insert(make_pair("C++","2-1"));
    courses.insert(make_pair("C++","2-2"));
    courses.insert(make_pair("C++","2-3"));
    courses.insert(make_pair("C++","2-4"));
    courses.insert(make_pair("C++","2-5"));
```

```
    string name;
    int count;

    do {
        cin>>name;
        count = courses.count(name);
        if(count==0)
            cout<<"Cannot find this course!"<<endl;
    } while (count==0);

    cout<<count<<" lesson(s) per week: ";
    auto range = courses.equal_range(name);
    for(CourseIter iter = range.first; iter!=range.second;iter++)
        cout<<iter->second<<" ";

    return 0;
}
```

equal_range是C++ STL中的一种二分查找的算法，试图在已排序的[first,last)中寻找value，它返回一对迭代器i和j，其中i是在不破坏次序的前提下，value可插入的第一个位置（亦即lower_bound），j则是在不破坏次序的前提下，value可插入的最后一个位置（亦即upper_bound），因此，[i,j)内的每个元素都等同于value，而且[i,j)是[first,last)之中符合此一性质的最大子区间
count是计算映射中键值出现的次数

函数对象

所谓函数对象就是一个行为类似函数的对象，它可以不需要参数，也可以带若干参数，其功能是获取一个值，或者改变操作的状态。在C++中，任何**普通的函数、函数指针、lambda表达式和任何重载了调用运算符operator()的类对象**都可以是函数对象。

标准函数对象

- 产生器：函数参数0个
- 一元函数对象：函数参数1个
- 二元函数对象：函数参数2个
- 一元谓词 函数参数1个，函数返回值是bool类型，可以作为一个判断式，谓词可以是一个仿函数，也可以是一个回调函数。
- 二元谓词 函数参数2个，函数返回值是bool类型

lambda表达式

lambda表达式又称匿名函数，其构造了一个可以在其作用范围内捕获变量的函数对象。

•定义：[捕获列表] (参数列表) -> 返回类型 {函数体}

▫捕获列表可捕获lambda所在函数的局部变量

▫参数列表、返回类型和函数题与普通函数一致

▫可定义在函数内部，理解为未命名的内联函数

▫auto lmda = [] { return "Hello World!"; };

▫cout<< lmda() <<std::endl; //执行与函数对象一致

•捕获列表有值捕获、引用捕获和隐式捕获方式

▫int size = 10, base = 0; //局部变量

▫auto longer = size{return s.size()>size;} //值捕获

▫auto longer = &size{return s.size()>size;}//引用捕获

▫auto longer = ={return s.size()>base;}//隐式值捕获

▫auto longer = &{return s.size()>size;}//隐式引用捕获

函数对象参数绑定

•算法模板对作为参数的函数对象有严格要求

•通过bind参数绑定模板，实现将一种函数对象转化为另一种符合要求的函数对象。

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<functional>
#include<iterator>
using namespace std;
using namespace placeholders; // 占位符_n的命名空间
int main() {
    vector<int> v = {10,20,30,40};
    auto p = find_if(v.begin(),v.end(),bind(greater<>(),_1,20));
    if(p==v.end())
        cout<<"NO"<<endl;
    else
        cout<<*p<<endl;
}
```

算法

transform算法

- transform算法顺序遍历first和last两个迭代器所指向的元素；
- 将每个元素的值作为函数对象op的参数；
- 将op的返回值通过迭代器result顺序输出；
- 遍历完成后result迭代器指向的是输出的最后一个元素的下一个位置，transform会将该迭代器返回

```
template <class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last, OutputIterator
result, UnaryFunction op) {
    for (;first != last; ++first, ++result)
        *result = op(*first);
    return result;
}

transform(s.begin(), s.end(), ostream_iterator<int>(cout," "), negate<int>());
// 输出相反数
```

accumulate算法

该算法函数在numeric头文件中定义：`#include<numeric>`

```
#include<iostream>
#include<numeric>//accumulate函数在这个库中定义
#include<string>//包含这个库，可以直接输出字符串
#include<vector>//vector是向量类型，可容纳许多类型的数据，因此也被称为容器
using namespace std;
int main() {
//功能一：求和
 int list[10] = { 1,2,3,4,5,6,7,8,9,10 };
 int sum= accumulate(list, list+10, 0) ;
 cout <<"和："<<sum<<endl;
 //功能二：求连乘积
 int  con_product  = accumulate(list, list + 3, 1, multiplies<int>());
 cout<<"连乘积："<<con_product<<endl;
 //功能三：string合并
 vector<string>a{"1","-2345","+6"};
 string a_sum=accumulate(a.begin(), a.end(),string("out: "));
cout<<"string合并后输出："<<a_sum<<endl;
 return 0;
}
```

流类库与输入输出

I/O流的概念及流类库结构

当程序与外界环境进行信息交流时，存在着两个对象，一个是程序中的对象，另一个是文件对象。流是一种抽象，它负责在数据的生产者和数据的消费者之间建立联系，并管理数据流动。

输出流

最重要的三个输出流**ostream,ofstream,ostringstream。**

输出流简介

ostream

预先定义的ostream类对象用来完成向标准设备的输出：

- cout是标准输出流
- cerr是标准错误输出流，没有缓冲，发送给它的内容立即被输出
- clog类似于cerr，但有缓冲，缓冲区满时被输出

ofstream

ofstream类支持磁盘文件输出。在打开文件之前可以指定ofstream对象接受**文本模式数据**和**二进制数据。**

构造输出流对象

```
ofstream myFile;
myFile.open("filename"); //  使用open函数打开文件

ofstream myFile("filename"); // 调用构造函数时打开

ofstream file;
file.open("FILE1")
    ...
file.close(); // 关闭FILE1

file.open("FILE2");
    ...
file.close();
```

插入运算符

插入运算发("<<")用于传送字节到一个输出流对象。插入运算符与预先定义的操作符一起工作可以控制输出格式。

插入文本数据

使用插入运算符可以为一个流插入文本数据。

```
ofstream os("filename");
os<<"Hello World! ";
```

输出宽度

使用width或者setw为每个项指定输出宽度。

```
cout.fill("*"); // 可以将填充的空格改成*
for(int i=0;i<4;i++) {
    cout.width(10);
    cout<<value[i]<<endl;
}
```

```
for(int i=0;i<4;i++) {
    cout<<setw(6)<<name[i]<<setw(10)<<values[i]<<endl;
}
```

对齐方式

使用setiosflags操纵符来进行左对齐等操作，其定义在iomanip头文件中。

参数值	作用
ios_base::left	设置左对齐
ios_base::scientific	以科学格式显示浮点数值
ios_base::fix	以顶点格式显示

使用resetiosflags关闭操作符。

文件输出流成员函数

open函数

打开一个与输出流关联的文件时，可以指定一个open_mode标志。可以通过按位或（“|”）运算符组合这些标志。

```
ofstream file;
file.open("filename",iosbase::out|ios::binary); // 以二进制打开
```

标志	功能
ios_base::app	打开一个输出文件用于在文件尾添加数据
ios_base::ate	打开一个现存文件（用于输入输出）并查找到结尾
ios_base::in	打开一个文件，用以向计算机输入
ios_base::out	打开一个文件，用以输出
ios_base::binary	以二进制模式打开一个文件（默认是文本）

close函数

close函数用来关闭一个与文件输出流关联的磁盘文件。如果写入后不关闭，在读取时就不会读取到写入的内容（大概因为没保存）。

put函数

```
cout.put("A"); // 精确地输出一个字符
cout<<"A"; // 输出一个字符且受到之前设置的宽度等参数影响
```

write函数

write函数用来把内存中的一块内容写到一个文件输出流中。

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

struct Date {
    int money, day, year;
};

int main() {
    Date dt={6,10,92};
    ofstream file("date.dat", ios_base::binary);
    file.write(reinterpret_cast<char*>(&dt),sizeof(dt));  // 把二进制数据强转字符串才能写文件，然后读也是要强转
    file.close();
    return 0;
}
```

seekp和tellp函数

一个文件输出流保存一个内部指针指出下一次写数据的位置。seekp函数设置这个指针。tellp函数返回该文件位置指针值。

输入流

提取运算符

提取运算符(">>")用于格式化文本输入，以空白符为分隔。

如果想不已空白符分隔，可以使用getline.

getline函数

```
string line;
getline(cin,line,"t"); // 从输入流读取数据到line，以t作为分隔符
```

read函数

read函数从一个文件读字节到一个指定的储存器区域，由长度参数确定要读取的字节数。

```
struct SalaryInfo{
    unsigned id;
    double salary;
};

int main() {
    SalaryInfo employee1 = {600001, 8000};
    ofstream os("payroll", ios_base::out | ios_base::binary);
    os.write(reinterpret_cast<char*>(&employee1), sizeof(employee1));
    os.close();

    ifstream is("payroll",ios_base::in | ios_base::binary);
    if(is) {
        SalaryInfo employee2;
        is.read(reinterpret_cast<char*>(&employee2),sizeof(employee2));
        cout<<employee2.id<<" "<<employee2.salary<<endl;
    } else {
        cout<<"Error"<<endl;
    }

    return 0;
}
```

类似于将文件内容写入到内存中。

异常处理

C++异常处理的实现

```
throw 表达式

try
    复合语句
catch (异常声明)
    复合语句
catch (异常声明)
    复合语句
```

throw

如果某段程序发现了自己无法处理的异常，就可以使用throw来抛出这个异常。throw的操作数表示异常类型语法上与return类似。如果程序要抛出多种异常，应该使用不同的操作数类型来互相区别。

try

try后的代码是代码的保护段。如果一段代码预期会产生异常，则就应把代码放在try后。如果真的发生了异常，其中的throw会抛出这个异常。

catch

catch后的代码时异常处理程序，捕获由throw抛出的异常。如果异常声明是一段省略号(...)，catch子句便会处理任何异常，所以这种应该放在最后。一旦执行了一段catch，后面的catch就不会再执行。

异常接口声明

对于一段函数可以规定它能抛出的异常类型。

```
void func() throw(A,B,C,D); // 这表明func函数只能抛出类型ABCD及其子类的异常
```

如果没写throw，则可抛出所有异常

```
void func();
```

一个不抛出任何异常的程序可以声明如下

```
void func() throw();
```

异常处理中的构造与析构

如果在一段catch后面的代码中出现了一个值参数，则其初始化的方式是复制被抛出的异常对象。如果catch子句的异常声明是一个引用，则其初始化的方式是使该引用指向异常对象。

标准程序库异常处理

C++标准提供了一组标准异常类，这些类以基类Exception开始。该基类提供一个成员函数what()，用于返回错误信息。

——Jerry