中间层（中介层）： 萃取机 iterator\_traits

Template <class I> I 是class iterator 进入这里

Struct iterator\_traits<T\*>

{typedef typename I::value\_type value\_type;}

Template <class T> I是pointer to T 进入这里

Struct iterator\_traits<T\*>

{typedef T value\_type;}

Template <class T> I是pointer to const T 进入这里

Struct iterator\_traits<const T\*>

{typedef T value\_type;}

88nkm1.字符串是以\0为结尾的，所以当传输一系列东西的中间有\0时无法传输所有的信息

\r回到本行最左边，会覆盖字符“abcd\rxy”==xycd（可以用来写时钟，一直更新在原处）

#include <algorithm>

sort（地址1，地址2）。例如：sort(\*a,\*a+3);（可以写二维数组的任意两行地址）

W25Q16 flash

2.NULL空指针==0==假。 ++v1先做自加后赋值，本行命令中v1=v1+1。v1++先赋值后自加，本行命令中

v1=v1,下一行中v1=v1+1。

强制类型转换转换的应该是变量的内容，变量本身的类型保持不变

连续判断中3<a<5实际格式为(3<a)<5

1？2:3 条件运算符（三目运算符）1为真选2，1为假选3,2、3为计算结果

枚举一般情况下内部是从0开始依次递增的整数序列

time(NULL)可以取得当前系统时间（秒数？）？#include<ctime>

long t=time(NULL); t+=5; while(time(NULL)==t);

rand()可以取随机整数

3.for(;&&;), continue忽略这轮循环剩下的循环内容，进入调整部分（下一次循环）

函数形参在调用时才创建，并且用实参初始化

相同函数名，不同函数表，函数重载，例如：f(int x)和f(float x,int y) 就是相同函数名，但实际功能与需要的参数却不相同

递归函数：自己调用自己，即f(x){x\*f(x-1)}

先声明后使用 只写函数头和分号,例如 void f1(); 里面参数可以不写

汉诺塔（递归）：只需要考虑如何把问题化简（即如何形成n-1）和最简时（n=1）问题如何处理，不需

要考虑中间步骤（只用考虑最简单的问题）。

#include <iostream>

using namespace std;

//a==from,b==target,c==temp

void hano(int n,char a,char b,char c);//n个盘子，A、B、C三个位置

int main()

{

hano(3,'A','B','C');

}

void hano(int n,char a,char b,char c)

{

if(n==1)

cout<<a<<"=>"<<c<<endl;

else

{

hano(n-1,a,c,b);

hano(1,a,b,c);

hano(n-1,b,a,c);

}

}

递归问题：当使用递归时，结束输出任务后，回到调用他的地方，例如

void paiwei(int a)

{

if(a<10)

cout<<a<<" ";

else

{

paiwei(a/10); ①

paiwei(a%10); ②

}

}

当输入123，先123->12①,再12->1①(输出1),再1->12①（回到调用的地方）,再12->12②（输出2）,再

12->123①（回到调用），再123->123②（输出3）

递归效率低

4.静态函数（static）只允许在本文件中使用，即参数或函数无法在其他文件中调用。

静态局部变量会在第一次遇到的时定义并初始化（保留，只执行一次，保持到程序结束），离开这个范

围它不会释放，之后再一次遇到这个变量它也不会再创建和初始化。

静态变量会在这段程序/函数（主函数main（））结束的时候进行释放，即变量生命周期延长

int counter()

{

static int a=0;

++a;

return a;

}

int main()

{

cout<<counter();//输出1

cout<<counter();//输出2

cout<<counter();//输出3

}

extern 类型 变量名//声明一个变量（使其能被外部文件调用）

5.int a[5];

sizeof(a); sizeof(int [5]);//int [5]表示a的类型

数组名表示第一个元素的地址（首地址）

输出字符变量地址（char类型，\*char）的时候，改成输出从这个内存单元开始存放的一系列字符，到向**后遇到的第一个‘\0’为止**（只有这个情况是这样处理，其余的类型均是输出地址）

cout<<"&ch="<**<hex<<(int)&ch<**<endl;//这样能取到ch字符的十六进制地址，若没有hex，则为十进制地址

赋值是赋值一个数据，不是一组数据，所以对于数组来说，初始化时可以a[]={};但赋值时a={}（不能用大括号）;是错误的

调用函数传递的是数值，s[],max(s);s（当做数值来看）传递的是地址。即，把数组名当做数值使用的时候，它仅仅是一个地址，大小与数值有关。

把数组传递给函数，除了传递数组名，还要将数组的元素个数n传递过去

a[2][2]={{1,20},{2,3}},二维数组传递形参只有第一维可以不写，后面的（列）需要写，例如int max(int s[][3])

name[20],"666" #include<cstring>

数组name[]

strcpy(字符数组名，字符串)=>strcpy(name,"666")//把字符串复制到字符数组中去

strlen(字符串或字符数组名)//取得其中字符串的长度（\0标志）

strcat(字符数组名，字符串)//把字符串追加到字符数组原来的字符串中

**new一个二维数组的方法**

**int \*\*a=new int\*[n];**

**for(int j=0;j<4;j++)**

**a[j]=new int[4]; 4行n列**4

6.struct xxx{};int main{xxx s1;} xxx为新的数据类型，int能够出现的地方xxx也可以。

**cin>>字符变量地址(char类型)；** 把输入的一串字符从这个地址表示的内存单元开始依次摆放，最后追加一个‘\0’，形成一个字符串

指针是用来保存相应类型的地址的变量，\*和&表示后面的变量是指针变量，代表一个地址，例如\*p。p为指针变量代表地址

\*指针表示指针所指向的变量，不是指针自己

int\* a[5]//5个指针

int (\*a)[10]表示a是个指针（一个指针），指向有10个元素的数组（数组指针）

p[5] \*a=p a=&p

**typedef int AA[10] 则AA表示：包含10个int类型元素的int类型数组，即AA==int[10] AA\*表示包含10个int类型元素的int类型指针，**

地址的运算统统以变量长度为单位：int a[5]={0,1,2,3,4}; &a[4]=003DFC78 &a[0]=003DFC68 &a[4]-&a[0]=4

int类型的变量 从a[0]到a[4]总共需要4步 因此&a[4]-&a[0]=4

Date \*p;(结构指针) 内部成员表示为:(\*p).year

堆空间的申请与释放：new/delete方式建立空间 int\* p1=new int[n];（数组首地址）

new数组的时候不能再命名一个新的数组名

new的结果是一个地址，在堆中申请一个变量的空间，返回这个变量的地址

type\* p1=new type;

**delete 地址;新建一个数组delete的话，需要写成： delete[] p1;**

指针数组：元素类型 数组名[个数] 所以int\* a[5]表示5个地址（int\*类型） 5个指针都是0==5个指针都为NULL

cstdlib

argv[0]是命令文件的名字，对其进行处理毫无疑义

**atof(字符串)把数字字符串转换成一个对应的浮点数字，相同的还有atoi（转换成整数）**

**引用（取别名） int& a=b;（要求：类型一致）** 说明了a和b是一个东西，这个东西的名字可以是a，也可以是b。

提倡用引用&a（取地址的时候用引用而非指针）

引用直接操作初始化它的那个变量，类型要一致

**尽量使用引用来传递参数，尽量用const来限制对引用的修改。例如 int max（const int &a）{}**

函数也是有地址的

**函数指针：像声明函数一样定义函数指针，不同在于，要把函数名改成“（\*指针名）”的形式**

**例如： void (\*fp)(int a[],int n)=NULL;**

**前面声明了： void output(int a[],int n); 所以 fp=output（这里也意味着fp可以当做output使用）**

;//因为**函数名就是函数地址**，所以output可以拿来用

函数指针可以作为其他函数的形参

**memset(a,0,sizeof(int)\*n);//作用是使数组a中的所有元素都是0**

7.**凡是指针一律是4个字节**

地址除了数值之外还有类型，没有类型的指针不能用来保持变量

结构体里面的东西想要弄出来，要使用引用或指针

**类型成员（函数）的调用使用“：：”（一般用于cpp中定义函数的时候：APP::Init(){xxx}），变量的成员就用“.”（一般用于主函数实现的时候）**

**在调用成员函数时，程序会自动把结构变量的地址传过来，可以通过this来使用这个地址。**例如：

**this.hour,this.min**（this->hour）等等

在成员函数中，把用来调用这个成员函数的变量称为当前变量，this总是指向当前变量

使用当前变量的成员，this可以（一般）不写。

封装（类？）：把数据和函数放在一起

类一般数据是私有的，函数是公开的

**类的变量，就是对象（?) 定义一个对象==创建一个变量，例如:CLOCK c；**

**调用成员函数==让对象做一件事情**

要求用户必须调用这个函数（强制）的办法：创建对象自动调用一个函数（希望这个对象调用这个函数）：

让函数名与类名一致即可(构造，自定义的话，需要创建时赋初值)

自动调用的函数，取不到函数返回值，所以对自动调用的函数不用写返回类型（即连void也不需要）。

但是该函数里面的参数必须传递过去，否则无法编译通过。

**即在运用构造函数的时候，可以把它当成python中的def \_\_init\_\_()函数使用**

在创建对象的时候直接传递参数，例如 Clock c（10,45,30）//10小时45分30秒（这个地方不能用{}，与结构体不同）。这里传递的参数仅仅是传递给构造函数的形参。

函数名与类名同名的成员函数（方法）称为“构造函数”，这种情况下，该函数只能自动调用，无法主动调用

每创建一个对象，一定创建并只调用一次构造函数

函数重载：多个函数的函数名字一样，但参数不同

变量作用于：大范围变量和小范围变量同名的时候，优先使用小范围变量。例如：this->n = n; n为小范围变量n

**构造函数中的严格初始化：A():n(0),m(1),s(s){} 将变量n初始化为0，m初始化为1,s初始化为给定的s的值。**

:成员变量名（要初始化的数据）,多项用逗号隔开（形参可以有默认值）

成员变量是类里面定义了这些变量，例如上面中，类里面一定存在：int n,m,s;这种类似的定义

注意：**初始化列表不能用来初始化数组和结构变量**（即没有{}进行初始化）

因为数组初始化：a[]={},结构体struct：........？

**添加冒号是初始化列表（在构造函数的函数名和函数体之间添加冒号）**

类将头h文件和cpp文件分开，

头文件放置：函数声明与外类实例化。

cpp文件中：函数定义。

**外类实例化的变量在本类的cpp中可以直接使用。例如：在CRobot类中定义了 CAnalog类的对象（实例化）analog和int类对象a。则在cpp文件中可以直接： analog=xxx； a=xxx；。**

**类的嵌套：本类中的外类实例化，在外处（其他cpp文件）调用本类，则本类中的外类对象需要"."来修饰（子类调用也是如此...）。例如：在Quadruped.cc文件中使用CRobot的实例化对象robot，则需要robot.analog=xxx; robot.a=xxx;**

对象销毁（释放）的时候自动调用的函数：析构函数：~类名。

对象销毁是因为main()函数执行完毕，里面所有的变量都消除了

析构函数没有形参（即不存在重载）例如：~A（){}

析构函数里面一般也不需要什么内容

对对象来说，先执行成员的构造函数，再执行整个对象的成员函数。例如：

class B{

Int obj;//Int类

public:

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

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

};

最后输出的结果是：

Int();

B();

========

~B();

~Int();

全局变量会在所有变量之前创建，在程序完全结束后释放。

函数里面的变量会在所有程序运行结束后才按照顺序进行释放。

在一个函数中最后创建“局部变量”的先释放

构造函数和析构函数都有默认的（没任何卵用,里面没有任何内容）

一般在写一个类的时候总是会创建一个可以不传参数就能调用的构造函数。

方法：1.写一个没有参数的构造函数。2、在有参的构造函数中给参数赋初值。例如 A(int n=0):n(n){}

**对象也可以有别名： A &b=a;**

8.**java里面对成员函数必须在类中定义，c++中可以只在类中声明，外部进行定义**

初始化列表只能在构造函数的函数名和函数体之间定义，不能用在其他地方。

所以在定义其他函数的时候，活用this去对变量进行赋值

继承与多态：

**继承：1.单重继承（java唯一支持）。2.多重继承。3.虚继承。23很少使用**

**多态：1.虚函数。2.类型识别。3.纯虚函数（java里面是接口）。4.抽象类。**

继承：一个类自动获得另一个类的全部成员

基类->旧类->父类 派生类->新类->子类

写法： class 新类名 : public 旧类名{};

父类的所有成员都会继承到子类中。但是来自父类的私有private成员在子类中不能“直接”访问。（通过函数可以访问私有变量。如何操作？？）

访问成员除了私有（private，成员不能被访问）和公开（public，成员可以在任意地方被访问）之外，还有保护（protected）

子类中可以访问父类的保护成员，但是在其他地方（如main函数中），保护成员不能被访问

子类定义一个与父类同名的函数后，就会把父类的那个函数给覆盖。注意：只要同名就会覆盖，与函数重载无关

子类中定义了下面这个函数：

void show(bool bNewLine){

Animal::show(bNewLine);

};明确说明了子类中必须调用父类（Animal）的show函数

子类构造函数会首先调用父类构造函数（python中的super），然后再执行自己的语句 。子类析构函数

调用完之后会自动调用父类的析构函数

创建子类对象的时候，默认总是调用父类的无参构造函数。

子类接受参数传递到父类的方法：

**在初始化列表（子类的构造函数中）中将父类成员初始化（private中的变量可以用此方法）：**

**子类名称（int a）:子类变量名(a),父类的类名（父类变量给定值、这地方实际上是调用了父类的带参构造函数，赋予带参构造函数中形参的值){}//父类可以有多个。**

无名对象：毫无意义（个人理解）：A（3）；//这是一个无名对象，只在这一行有用，到下一行就释放

了

**A(5)与(A)5是一样的，可以把（A）5看成强制类型转换**

（可以自动进行类型转换）

例如：A a4; a4=40; 会自动类型转换（创建并释放一个无名对象40），把40给a4对象。

常量对象不能调用普通成员函数（能够修改当前对象的数据成员的函数）

成员函数声明自己不会修改当前对象的数据成员的方法：

void show() const{}//函数头与函数体之间写const（该函数不能修改数据成员）----》》这也是重载的一种方式

普通对象优先调用普通函数（可以调用const函数），const对象只能调用const函数

多重继承： class A:public B,public C{};//多个父类，全部继承

1.通过B::show();和C::show();解决同名函数的问题。

2.按照继承顺序依次调用父类的构造函数：先B后C。

虚继承：虚继承的父类称为虚基类

class A:virtual public B{};//虚继承，作标志，标志这部分来自于B？（什么意思）

class C:virtual public B{};

class D:public A,public C{ //来自同一个虚基类的成员会合 并成一份

D（...参数）:A(...参数),C(...参数),B(...参数){}//A、C、B三个的参数都得从D这传

递，B不能由A、C传递参数

};

虚基类构造函数参数由下级子类直接传递

9. c语言中对字符串（字符数组）进行处理：必须考虑长度问题

c++中由于有string类的原因，输入自动处理长度

string类中可以：string name[下标] ; string obj;//obj是string类的对象

obj.size(),obj="Hello";obj[2]='\0',

注意：c++里面的字符串不是以‘\0’作为结束标志的，c里面是，所以即使obj[2]='\0',整个字符串的长度还是他本身的长度

getline(cin,obj);//会把输入的一行字符放到obj对象里面//cin.getline(地址，长度)

obj+="附加串"；//把两个字符串相连，无需考虑大小//strcat(旧串，附加串)-->c语言里面链接，要考虑旧串大小能否存放下附加串

obj1>obj2;//直接比较//strcmp(串一，串二);//c的方式

obj.c\_str();-->（理解为）const char \* //转换成c风格的字符串

**const int \*p;//p可变（p指向的地址），p指向的内容不可变（\*p），p=&A可以**

**int const \*p;//p可变（p指向的地址），p指向的内容不（\*p），p=&A可以**

**int \*const p;//p不可变（p指向的地址），p指向的内容可变（\*p）**

**const int \* const p;//p和p指向的内容均不可变。**

**口诀：左数右指**

**当const出现在\*号左边时，指针指向的数据为常量；当const出现在\*号右边时，指针本身为常量。**

**int const \*p=&i; int \* const p =&i;**

**\*p=5;(X) \*p=5;(√)**

**p=&j;(√) p=&j;(X)**

**cin和cout都是对象**

函数形参默认值在声明的时候定义//当声明和定义分开的时候

初始化列表在函数定义的时候写//当声明和定义分开的时候

多态：**本质：c++允许把派生类对象(子类实例化)的地址赋给基类的指针** 关键字：virtual

需要哪个方法（函数）用多态，就在其前面加virtual。

**加上virtual后，在调用该函数的时候，实际调用的是相应的类中的方法**（实例本身所指的类中的方法，而不是基类的方法）

例子：

virtual show（）；

根据对象的真实类型（最下层的类型，最详细的类型，子类中的同名函数），执行相应的类的函数（很重要）

不用该方法的话，F \* p=&f1; p->show();实际执行的是父类（父类是F）中的show函数，而不是我们要求的子类中的show函数（F \*p=&f1 等同于F \* p;p=&f1）

**P是指针，存放地址；&这里是取地址符,&f1是取f1变量的地址；f1变量存储的是数值**

**多态的优势是统一管理，统一操作接口**

有virtual的函数称之为虚函数，如果声明和定义分开，只在声明里写，定义不用写

**上面的指针程序也可以用引用表示：F &p=f1; p.show();**//与上面的程序结果一致

**sizeof(类名)//其大小是类里面变量大小的总和+是否使用虚函数（是+4，不是不加）**

有虚函数的类，编译器会申请一块内存空间，放上所有虚函数的地址（虚函数表），每个对象都会保存

虚函数表的地址，每个对象里面都会有一个指向虚函数表的指针（大小为个人字节）

**string内部封装的是一个指针，其大小4个字节**

**对虚函数来说，在子类里面覆盖虚函数时要注意以下几点：**

**1.子类中写同名函数的时候virtual可写可不写**

**2.子类的同名函数要和父类的函数完全一样（返回类型，参数表等等）**

纯虚函数：只有声明，没有定义

在声明的时候加上等于0.例如:

class Animal{

string name;

public:

virtual void eat()=0;

virtual void sleep()=0;

**virtual void shout()=0;//表示这个函数是一个一定不会被调用的真正函数，调用它时，会执行其他函数（同名）**

};

有纯虚函数的类称为抽象类。抽象类不允许直接创建对象，只能用来指向或引用子类对象

**纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”  
　virtual void funtion1()=0**

在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。

则编译器要求在派生类中必须予以重写以实现多态性

同时含有纯虚拟函数的类称为抽象类，它不能生成对象

纯虚函数最显著的特征是：它们必须在继承类中重新声明函数

**java和c#里面的抽象类进一步抽象成接口，因为java和c#里面不支持多重继承。**

类型转换：格式： **xxx\_cast<要转换的类型>（要转换的数据）** {[有地方不明白dynamic\_cast]}

static\_cast 静态类型转换 （可以进行转换的东西：两个数值类型之间进行相互转换、有类型的指针与void\*之间，无名对象支持的转换）

**static\_cast<A>(100)==A(100)**//无名对象

const\_cast 常量类型转换 用于把常量转换成变量用一次

**const\_cast<变量名&（这地方是引用）>（数据）=1**//常量依然是常量，这个转换的结果是变量。即数据依然是常量

register 寄存器 register int n；//把n存到寄存器里面，但是对n来说，个数有限，长度也有限

reinterpret\_cast 重新解释内存的转换 最危险的转换 用于不同类型指针之间转换

**reinterpret\_cast<z\*>(p)//把y\*类型的指针p转换成z\*类型**

dynamic\_cast 动态类型转换 向下转换时把关:父类对象或父类指针向子类转换。

dynamic\_cast<子类\*>(父类地址) 要求父类至少要有一个虚函数，有虚函数才能使用多态特性。

dynamic\_cast运用了多态特性。 转换成功返还一个地址，转换失败返回NULL

构造函数不能是虚函数。

**如果类中有任何一个成员函数是虚函数，那么析构函数应为虚函数**

10. 友员、静态成员

**友员：friend . 能够直接访问某个类中所有（公、私）成员的桥梁**

**友员：1.外面的函数. 2.另一个类**

**在类的内部用friend声明。friend是个关键字**

例子：

class A

{

int data;

public:

A(int d=0):data(d){}

void show()

{

cout<<"data="<<data<<endl;

}

**friend int add(A a1,A a2);//声明 允许这个函数访问类的私有成员**

**A add(A a1,A a2)**

**Friend class B;**

//**add不是成员函数（成员函数里有this）、友员中没有this、没有当前对象**

//**add函数（友员函数）在哪里写都可以（类内部的任一地方）。其定义也可以在里面（跟声明在一起**）

};

**无法重载仅按返回类型区分的函数**

int add(A a1,A a2)

{

int sum=a1.data+a2.data;

return sum;//A(sum)

}

**A A::add(A a1,A a2)**//改写成：A add(const A &a1,const A &a2) 能够保持a1 a2不变，同时也不是把值复制，而是直接使用a1 a2的本体

{

int sum=a1.data+a2.data;

return A(sum);

}

int main()

{

A a1(40),a2(50);

Int a;

a=add(a1,a2); //把a1和a2的值复制一份给add函数

a1.add(a1,a2).show();

Cout<<a;

**凡是再向函数中传递对象的时候，几乎都会用引用，只要不需要改变对象，几乎都会加const**

}

友员类：friend class B;//声明在A类内部

class B

{

public:

void twice(A& a)

{

a.data \*=2;

}

};

int main()

{

A a1(40),a2(50);

//add(a1,a2);

add(a1,a2).show();//把a1和a2的值复制一份给add函数

B ob;

ob.twice(a1);

a1.show();

}

**凡是所有对象共用一份的数据都要声明为静态数据成员：static**

**静态数据成员又称为类变量：它属于这个类，被该类的全体对象共享**

//只初始化一次

**静态数据成员像全局变量一样在所有函数之外初始化**

静态函数成员只能访问静态成员

类变量必须在class声明的外面定义。如果不初始化，则编译器自动初始化0

**int Counter::cnt;或int Counter::cnt=0;**

静态成员存在于内存

 int Myclass::n = 100;   // 静态数据成员初始化

类变量的初始化：class COUNTER（声明定义在一起）

{

static int cnt;//只是声明？？

}

例如：

**class a{**

**public:**

**static string teacher;**

**};**

**string a::teacher="Alex";**

允许但不提倡用对象.成员来访问静态成员。 提倡用：**类名::静态成员**

11.拷贝构造函数（一些构造函数的特殊名字而已。毫无特殊之处），旧对象初始化新对象

**自动产生的构造函数（拷贝构造函数0）：A()和A(const A&)**

**旧对象初始化新对象调用A(const A&)构造函数。例如：A newobject(oldobject);（？）**

把旧对象的内容逐个复制到新对象中

默认拷贝构造函数，再由指针成员（\*p）指向动态内存（new、delete）的时候有问题。这时需要自己写拷贝构造函数

例如：

name指针指向字符串的地址，通过A(const A&xx)(==A(const int& xx))传递给另一个对象的**name是这个字符串的地址，即s2.name指向该字符串，一旦s1.name改变(或者说是字符串内容改变)**，s2.name也会变化，不会保留原来的内容

A newobject(oldobject);传递数据的时候只是传递到构造函数的形参，并没有将数据传递到成员变量。

**自己写的拷贝构造函数**又称重载构造函数

**格式是怎样的？**

在一个类的成员函数中可以访问该类的任何一个对象的成员变量

要在自己定义的拷贝构造函数中处理所有需要处理的数据（因为没有默认的拷贝构造函数了）

双目运算符：

在c++中允许使用operator加上一个运算符（+、-、\*、/）作为函数名（类内？类外？还是都行？）

**这种形式的函数调用可以把函数直接写成运算符的形式。**

例如：

obj3+=operator+(obj1,obj2);

obj4=obj1+obj2;

obj3.show()==obj4.show()//两个的运算结果是一样的

operator+是运算符函数（自己写的）（让程序写起来更像数学式子），

从而实现自己规定运算符如何工作。没自己写就按照默认的方式工作。

**成员函数：obj1.operator+(obj2)//仅需要一个参数，因为已经有当前对象obj1，不需要把obj1再传一次**

**而友员：A add(const A&a1,const A&a2)则一定需要两个参数，因为友员不是成员函数，它没有当前对象。一定需要两个参数**

运算符重载实际是自己写运算符函数来规定这个运算符如何工作

**定义： 不能既写一个友元的，又写一个成员的**

**友元格式：返回类型 operator运算符（形参1，形参2）**

obj1运算符obj2==operator运算符（形参1，形参2）； 该运算符会自动调用之前定义的函数。

返回值作为计算结果。

**friend F operator/(const F& f1,const F& f2)**

{

**return F(f1.n\*f2.d,f1.d\*f2.n);**

}

成员：直接在类中定义

**返回类型 operator运算符（形参）== A operator + (a)**

**obj1运算符obj2==obj1.operator（obj2）**

返回值作为计算结果。

**F operator\*(const F& o)const//第一个const表示形参不变，第二个const表示当前对象不变（？）**

{

return F(n\*o.n,d\*o.d);//无名对象，分子相乘、分母相乘

F res(n\*o.n,d\*o.d);//res是F类的对象

return res;

}

**cout是ostream类型的，cin是istream类型的**

**ostream & operator<<(ostream& os,const F& f)**//只能写成友元类型，这是输出运算符标准的重载模式

例如：

**friend ostream& operator<<(ostream& os, const F& f){**

**os<<f.n<<'/'<<f.d;**

**return os;//返回了cout（一定要有return，不论是cin还是cout）**

**}**

**friend istream& operator>>(istream& is, F& f)//输入会使f变化，不能加const**

**cout传给第一个形参os，f1传给第二个形参f（cout<<f1），由于返回值是os（即cout类型），因此可以继续写<<,即<<’\*’**

这样在main函数中: cout<<f1<<'\*'<<f2<<f1\*f2<<endl;就能够直接输出分数的形式而不需要其他的函数

单目运算符：尽量用成员（++，--,前面的++--），++a，--b。

友元：返回类型 operator运算符（形参）

运算符obj==operator运算符（obj）

成员：返回类型 operator运算符（）//有当前对象，不用形参

运算符obj==obj.operator运算符（）

**A&（？）** operator--(){ //成员函数

data-=10;

**return \*this;//this是指针，\*this是当前对象，this分配的空间在离开函数后仍然存在**

}

**返回类型为INT，表示返回值  
返回类型INT&，表示返回引用，引用是一个变量的别名，也就是说引用必须用要赋初值以表示引用了一块内存，在返回的问题上表现为不能返回临时变量的引用，也不能返回常数**

friend **A&** operator++(**A&** a){

a.data+=10;

return a;

}

双目运算符：

**对后++、--作运算符重载的时候，把后++、--当做双目运算符，且第二个操作数是int型（人为添加，实际上不存在）**

所以

friend **A** operator++(A& a，int)//n++结果是10，n变为11了

{

**A old(a);//当前对象a初始化新对象old ，旧值**

a.data+=5;

**return old;}//仅仅是个局部变量，所以不能是引用（无意义）与上面形成对比**

**A operator--(int){ //成员函数**

**A old（\*this）；**

data-=5;

return old;

}

**至少有一个操作数是自定义类型的（自己定义的类），才可以对运算符进行重载操作。**

**强制类型转换重载： 只能写成成员函数**

返回类型和operator后面的类型相同，因此不必写返回类型

定义： operator 类型（） //operator和类型之间要有空格

例如： public：

operator int（）

{

return data；}

operator char（）

{

return （char）data；}

operator bool（）

{

return data！=0；}

对自定义类型的对象，使用运算符时总是调用响应的运算符函数

12.I/O：c：控制台（命令）。 f：文件 cout/fout 文件和控制台的I/O是完全统一的

键盘——》键盘缓冲——》输入缓冲区——》程序读取

cout输出有缓冲（？（会把数据先放到缓冲区，最后输出））

标准流：cerr和clog输出没有缓冲（立刻输出），也不能重定向（unix中和linux中的东西）。用法和cout完全相同

重定向: a.out>a.txt.把本来应该输出到屏幕（命令行）中的数据，输出到txt文档中

linux中的格式(可能是)：-bash-3.00$ g++ cerr.cc

-bash-3.00$ a.out

-bash-3.00$ a.out>a.txt

-bash-3.00$ cat a.txt

**#include<fstream>//文件流**

**文件对象： ofstream fout（“a.txt”）；//创建一个文件对象fout,文件名是a.txt，输出**

**fout<<**

**fout.close();//释放这个文件,把东西写入到文件中**

in、out都是针对电脑

**ifstream fin("a.txt");//输入 getline(fin,str);**

**fin>>变量名；**

**fin.close();//释放，把东西从文件中读取出来**

**cin是从键盘中读数据，fin是从文件中读数据**

一个I/O流对象在没有出错时可以当成真，在出错时可以当成假。

**ofstream fout("/abc");**

**cout<<fout<<endl; 这样输出的结果是0（NULL），fout错误**

**当输出结果正确，会输出文件所在地址。**

保存在string对象中的字符串是c++风格的，“adasfaf”的字符串是c风格的

从键盘输入文件名需要给fin（）中设置一个变量

**当name是c++风格的字符串时， ifstream fin(name.c\_str());**

**c风格字符串需要一个字符数组 char name[256];**

13.格式化输入输出:

istream流的操作：

get（）//从输入流中读取一个字符，返回字符的asc码（一个整数）

cin.get()

get(char&)//从输入流中读取一个字符，保存到形参中

char string1[256];cin.getline(string1,256);//这是c风格的用法，cin.getline(字符数组名,长度)

。长度要够用，比一行的实际长度大

getline(cin,str);//c++风格，str是string类的一个对象

有第三个默认值，为‘\n’：第三个参数制定结束字符，默认为'\n'

输入流一旦处于错误状态，就不再执行任何输入。

可以把输入流恢复到正常状态：

**cin.clear（）；//不是清除缓冲区**

**缓冲区里面的东西依然存在**

**while(cin.get()!='\n');//可以清空缓冲区的方式。**

**cin.ignore（字符个数（长度），‘\n’(结束条件，遇到就结束)）//清空缓冲区**

**char ch=cin.peek();/**/偷窥，偷看下一个字符是什么字符。返回下一字符的asc码（整数）

**get()是把一个字符读走，peek（）是看下一个字符，最后字符还在输入流中**

**putback（）；//把刚刚取走的一个字符退回到输入流中**

**读取的是哪个就退哪个**

**cin.width(10);//只读10个字符，限定宽度**

cout.width(10);

对输出指定宽度时，指定宽度小的话，不会丢失数据，会按照实际大小输出

**这是一次性的，只对下一个输出有效。**

**cin>>ws;//跳过输入缓冲区中的空白字符（空格、换行）**

cout.fill（‘\*’）；//会把没有到达设定长度的地方放置‘\*’，填充。

**这个不是一次性的，永久有效**

一般来说，没有卵用

非格式化输入输出：

**put(char);//输出一个字符**

**cout.put(cin.get())保证输出的一定是个字符**

**非格式化的输入输出只对字符进行操作**

内存中的内容写到文件中： fout.write(地址，字节数)；//内存的地址

文件中的内容读到内存中： fin.read(地址，字节数)；//内存的地址

**形参格式：（char \*addr，int bytes）必须是char\*的形式，所以需要强制类型转换**

**例如：fout.write((char\*)&n,sizeof(n));**

**string类封装的是指针，把他存到文件去实际上存的是个地址，所以往文件中存字符串用char addr[10];**

**fin.eof();//判断是否读到了文件末尾**

**添加输出控制标志：cout.setf(ios::left);//保持原有的输出标志，增加新的输出标志 。**

这个标志是输出左对齐

cout.unsetf(标志);//清除某个标志

**cout.setf(ios::left|ios::hex);//多个标志单竖线隔开**

**八进制带前缀输出其前缀是0，十六进制的前缀是0x。**

**ios::showbase//带前缀输出； ios::showpoint//带小数点输出小数**

**ios::scientific//科学记数法 ios::showpos//带符号输出**

**ios::uppercase//十六进制和科学记数法输出的时候，里面的字母变为大写**

**cout.precision（）//精度，表示有效数字多少位**

**cout.precision(2); cout.setf(ios::fixed);//表示定点。这两句加起来表示小数点后的精度**

设置进制的时候，由于有8/10/16三种进制，**所以设置一个的时候把其他两个清掉unsetf。**

输出格式控制符： **即可以写在<<后面的东西，需要#include<iomanip>**

**flush：清缓冲区。 endl：回车，并且清缓冲区 cout<<flush<<endl;**

**cout<<left;//不需要之前那么麻烦,之前是写ios：：xxx的样式。ios:: 后面的都可以这么写**

**setw（宽度）、setfill(填充字符)、setprecision(精度)**

**cout<<setw(20)**

**创建文件对象会默认把原文件中的内容删掉，如果不想把之前的内容删掉，需要追加app。 ios::app**

**ios::binary//二进制输出，即对文件中的内容原样读写**

**fout("app.txt",ios::app),顺序读写，顺序添加**

**fin.seekg(位置int)；//跳到某个地方读**

**char ch;**

**fin.get(ch);//再通过get得到当前位置的字符**

fout.seekp(位置int)；//写的位置(数据的第几个字符)

**fout.put('\a');//把该字符放到数据的当前位置，替换掉原来的字符**

fstream fio(文件名，ios::in|ios::out)；（既能读又能写） fstreamn即是头文件，也是类名

14. A::B//A里面的B类

**成员内部类（在另一个类里面的类），局部内部类（main函数中的类）。**

内部类处理外部类的数据：

**try{ //监视？？？**

**if（有异常情况）**

**throw 数据；**

**}//监视是否有数据抛出**

**catch（类型 变量名）{}//接已经被抛出的数据，只负责接一种类型的数据**

catch（A e）{}

一个try后面可以有若干个catch，用来接收不同类型的数据

catch就是用来集中处理错误信息

遇到throw，会跳出整个try块，找到适合的catch块（类型一致）

在异常没有处理前，一切的语句都不会正常执行，直到异常被处理

即，如果没有合适的catch，程序会终止

**抛出子类对象，catch父类对象，是没有问题的，因为子类对象一定是父类对象**

先catch子类，后catch父类

catch（int e）{ throw ； }//表示重新抛出这个异常

能够接收所有的类型的数据： catch（...）{}//...表示不管什么类型，这个catch最后执行

**what();//返回一个用来描述异常的字符串**

15：数据结构、算法、链表

链表：myHead--->（头结点myNext）--->尾节点 指针指向

头指针：用来指向头节点的指针

**head指针指向头节点，它并不是头节点。：其他指针同理**

**指针指向下个节点： struct Node{ T data； Node \*next；//为了指向下一个节点（下一个数组），必须是Node类型的（内部类型）}**

结构里面也可以写构造函数

**int()//表示0，即，基本类型()表示为0。]**

**struct Node{**

**T data;//typedef int T**

**Node\* next;**

**Node(const T& d=T()):data(d) //,next(NULL)**

**{**

**next=NULL;**

**}//结构体里面的构造函数**

**~Node(){cout<<'X'<<endl;}//析构函数不是必须的**

**}；**

**Node\* head=NULL;//头指针指向空**

T item;

do

{

**cin>>item;**

**Node\* p=new Node(item);//item最终传递给Node类中的data，每个节点的next都是NULL**

**p->next=head;//指针“指向”了原head指针指向的地址。head指针所在位置与其指向的地址所在位置是不同的（两个方框）。每个struct里面有变量以及，指向下一个节点的指针.**

**//创建新的Node类型的结构体，即创建一个新的节点。p指针是一个新的指针**

**head=p;//新节点插入到原来链表的头上**

}

while(cin.get()!='\n');

Node \*p=head;

while(p!=NULL)

{

cout<<p->data<<' '

**p=p->next;//p指向了p->next指向的节点，即p指向了原先指向的节点的下一个节点**

//**Node\* q=p->next;**

**//delete p;//确保释放**

**//p=q;**

}//遍历作用

**//释放也只能是一个循环释放**

**p=head;**

**while(p!=NULL)**

**{**

**Node\* q=p->next;**

**delete p;**

**p=q;**

**}**

**//指针不代表当前位置，只是存储位置（位置数据）**

链表类： **linkedlist**

**class List{**

**struct Node{**

**T data;//T==int**

**Node\* next;**

**Node(const T& t=T()):data(t)**

**{next=NULL;}**

**};**

**Node\* head;**

**public:**

**List():head(NULL){}**

**void clear(){**

**while(head!=NULL)**

**{Node\* q=head->next;**

**delete head;**

**head =q;}**

**}**

**~List(){clear();}**

void insert\_front(**const T& t)**{ //插入

**Node\* p=new Node(t);**

**p->next=head;**

**head=p;**

}

void insert\_back(const T& t){//从最后面插入

**Node\* p=new Node(t);**

**if(head==NULL)//如果没有尾节点（空链表）**

**head=p;**

**else{**

**getPointer(size()-1)->next=p;//getPointer(size()-1)是尾节点**

**}**

}

void travel(){ //遍历

**Node\* p=head;**

**while(p!=NULL){**

**cout<<p->data<<' ';**

**p=p->next;//经典用法，表示让这个指针指向下一个节点**

**}**

cout<<endl;

}

int size(){

**Node\* p=head;**

**int cnt=0;**

**while(p!=NULL){**

**cnt++;**

**p=p->next;//经典用法，表示让这个指针指向下一个节点**

**}**

**return cnt;**

}

T getHead(){

**if(head==NULL)**

**throw "no head";**

**return head->data;//head为空，则没有data，函数会出问题，所以需要上面的语句**

}

T getTail(){

**if(head==NULL)**

**throw "no tail";**

**Node\* p=head;**

**while(p->next!=NULL)**

**p=p->next;**

**return p->data;//p->next为空说明p是尾节点**

}

bool empty(){

**return head==NULL;**

}

int find(const T& t){ //用于寻找位置

**int pos=0;**

**Node\* p=head;**

**while(p!=NULL){**

**if(p->data==t)**

**return pos;**

**p=p->next;**

**pos++;**

**}**

**return -1;//如果都没有，返回-1**

}

bool **update(const T& o, const T&n)**{

**int pos=find(o);**

**if(pos==-1)**

**return flase;**

**Node\* p=getPointer(pos);**

**p->data=n;**

**return true;**

}

bool erase(const T& t){

**int pos=find(t);**

**if(pos==-1)**

**return flase;**

**if(pos==0){**

**Node\* q=head->next;**

**delete head;**

**head=q;**

**}**

**else{**

**Node\* pre=getPointer(pos-1);**

**Node\* cur=pre->next;//cur指向要删的那个节点**

**pre->next=cur->next;**

**delete cur;//为了删除这个节点，并不仅仅是把他放空**

**}**

}

private:

// **Node\* getHead(){//取头节点,Node和Node\*都是内部类型，不对外公开,所以要写成私有的**

// //想要公开的函数，返回类型只能是data？？（data不也是内部成员？）

// }

**Node\* getPointer(int pos){**

**Node\* p=head;**

**for(int i=0;i<pos;i++)**

**p=p->next;**

**return p;**

}//取得指向指定位置的指针

}

双向链表：

**struct Node{**

**int v;**

**Node \*prev;**

**Node \*next;**

**};**

**头指针head。尾指针tail。**

栈和队列。

**栈：push pop 只允许在同一端进行插入和删除**

pop和push需要自己写吗???

继承是重用的方式之一

class Stack:private List{};//通过private继承，避免因public继承而可能产生的问题

？ 什么问题

通过链表实现栈

**class Stack:private{ //组合重用**

**List l;**

**public:**

**void push(const T& t){l.insert\_front(t);}//数据入栈**

**void pop(){l.erase(l.getHead());};//删除栈顶元素**

**T top(){return l.getHead();}//取得栈顶上的数据**

**bool empty(){return l.empty();}//判断栈是否为空**

**int size(){return l.size();}//取得栈中元素个数**

**void clear(){l.clear();}//清空整个栈**

};

队列：queu FIFO 必须在不同端进行插入和删除

链表实现队列：

class Queue{ //队列有首尾

List l;

public:

void push(const T& t){l.insert\_back(t);}//数据入队

void pop(){l.erase(l.getHead());}//删除队首元素

T front(){return l.getHead();}//取得队首元素

T back(){return l.getTail();}//取得队尾元素

bool empty(){return l.empty();}//判断队列是否为空

int size(){return l.size();}//取得元素个数

void clear(){l.clear();}//清空整个队列

};

16.二叉树：树：1.树可以是空的。2.根节点和若干个互不相关的子树组成

**因此，对树进行操作绝大部分都是运用递归**

二叉（查找\排序）树：BST： **左子树小于父节点的数据，右子树大于等于父节点的数据**

根指针指向根节点

struct Node{ //仅仅是构建了一个节点，并不是树

T data;

**Node\* left;//指向左子树的根节点**

**Node\* right;**

**Node(const T& t):data(t),left(NULL),right(NULL){}//构造函数初始化**

**};**

**Node\* root=NULL;**

void travel(Node\* tree){ 1

if(tree==NULL) return ; 2

cout<<tree->data<<' '; 3

**travel(tree->left); 4**

**travel(tree->right);//把整棵树遍历完 5**

}

**根->左->右 前序遍历**

**左->根->右 中序遍历 (用的最多):交换上面第三行和第四行的顺序。遍历后的数据从小到大排好顺序**

**左->右->根 后序遍历**

**void clear(Node\*（Node\*类型指针） &（引用符） tree（指针形参）){**

**//把指针本身传过去而不要复制一份**

if(tree==NULL) return ;

**clear(tree->left);**

**clear(tree->right);**

**delete tree;**

}

**&变量名（&指取地址符）**

int size(Node\* tree){

**if(tree==NULL) return 0;**

**return size(tree->left)+size(tree->right)+1;//最底层只有1个**

}

void insert(Node\* & tree,**Node\* p){ //Node\* p是树里面的一个新的节点**

**//不停迭代，直到最后找到可以添加新节点的位置。**

**if(tree==NULL) tree=p;**

**else if(p==NULL) return;**

**else if(p->data<tree->data)**

**insert(tree->left,p);**

**else**

**insert(tree->right,p);**

}

void\* & find(Node\* & tree,const T& t){

**if(tree==NULL) return tree;//不能用return NULL。因为void\*的引用是一个变量，NULL不是变量**

**if(tree->data==t) return tree;**

**if(t<tree->data)**

**return find(tree->left,t);**

**else**

**return find(tree->right,t);**

}

**二叉树中删除某一子节点（小根节点）：//将原右半部分最左边的子节点，作为原有的根节点**

**1.合并左右分支（右半边是树，左半边根节点当做节点） 2.让指针指向合并后的分支 3.释放要删的节点**

void **erase(Node\* & tree,const T& t){**

**Node\* & p=find(tree,t);//p指向要被删除的节点的“指针”**

**if(p==NULL) return;**

**insert(p->right,p->left);//合并 p->left表示一个新节点，p->right表示树**

**Node\* q=p;//将要删的子节点保存一份**

**p=p->right;**

**delete q;**

}

void update(const T& o,const T& n){//数据替换，

**Node\* p=find(root,o);**

**if(p==NULL) return**

**erase(root,o);//没有递归**

**p=new Node(n);**

**insert(root,p);//删除旧节点，添加新节点，不能直接更改数据（可能会出现23替换为100的错误情况）**

}

如何建立二叉树？？？ 未知

17.排序算法：时间复杂度、空间复杂度

时间复杂度：

**大O表示法：渐进上界法**

**O表示：最多怎样，不超过...。 表示计算机算法的效率：algorithm**

**大Ω表示法：渐进下界法**

例如：

**1.线性查找：O(N)**

**2.二分查找（折半查找）：O(logN)**

**logN<N<N^2 一个for循环是一个N，两个for循环就是N^2**

**选择排序法:O(N^2)**

**冒泡排序法:O(N^2)**

**插入排序法:O(N^2)**

**快速排序法:O(N\*logN)**

**ctime看时间**

time\_t t1=time(NULL);

**sort(a,N);//N=10240 选择排序，a是数组，N是数组元素个数**

time\_t t2=time(NULL);

cout<<t2-t1<<endl;

**选择排序：遍历多遍，每次都选择合适的（最大\最小）放到开始位置，直至序列排好顺序**

void **sort(T a[],int n)//(T\* a,int n)**

{

for(int i=0;i<n-1;i++)

{

int pos=i;

for(int j=i+1;j<n;j++)

if(a[j]<a[pos])

pos=j;

swap(a[pos],a[i]);

}

}

**冒泡排序：每次只能比相邻的两个元素，如果顺序不合适就交换**

void bubble\_sort(int a[], int n)

{

int i, j, temp;

**for (j = 0; j < n - 1; j++)**

**for (i = 0; i < n - 1 - j; i++)**

**{**

**if(a[i] > a[i + 1])**

**{**

**temp = a[i];**

**a[i] = a[i + 1];**

**a[i + 1] = temp;**

**}**

**}**

}

**插入排序：**

**对每个新元素（除第一个）：**

**1.找合适的插入位置。：先保存一份**

**2.把从那个位置开始的所有数据右移**

**3.把新元素放到要插入的位置**

void sort(T a[],int n)

{

**T temp;**

**int pos;**

**for(int i=1;i<n;i++)**

**{**

**temp=a[i];//从右向左找**

**for(pos=i;pos>0&&a[pos-1]>temp;pos--)**

**a[pos]=a[pos-1];**

**a[pos]=temp;**

**}**

}

**快速排序：分组， 第一个元素，中间的元素，最后的元素。这三个谁居中就拿谁做分界值**

**void sort(T a[],int n)**

**{**

**if(n<=1) return;//递归的结束条件，没有这个，程序会出现段错误**

**swap(\*a,a[n>>1]);//a[0]和a[n/2]进行交换，分界值a[n/2]到达最左边**

**T\* L=a+1;//左边的第一个元素（从左向右进行查找，找大于分界值的数据）**

**T\* R=a+n-1;//右边第一个元素（从右向左进行查找，找小于分界值的数据）**

**T v=\*a;//v是分界值**

**while(L<R)**

**{**

**while(\*L<v&&L<R) L++;**

**while(\*R>=v&&R>a) R--;**

**if(L<R)**

**swap(\*L,\*R);//数据交换**

**}**

**if(v>\*R)**

**swap(\*a,\*R);//分组完毕，中间值左边小于中间值，右边大于中间值**

**sort(a,R-a);//快速排序的递归**

**sort(R+1,n-(R-a)-1);**

**}**

**排序速度：快速>选择>插入>冒泡**

18.模板

**template <typename T>//表示T是类型名，这样T就不用再受约束,可以表示任何类型，不需要typedefine int T;**

void sort(T a[],int n)

{

T temp;

int pos;

for(int i=1;i<n;i++)

{

temp=a[i];

for(pos=i;pos>0&&a[pos-1]>temp;pos--)

a[pos]=a[pos-1];

a[pos]=temp;

}

}

**//这样组成一个完整的模板**

**template <typename T> 模板头，每个模板（函数）都有自己的模板头。各个模板头不能共用**

**sort<char>(ac,4);//强行指定模板中T类型为char**

**void sort(char\* a[],int n)**

**{**

**for(int i=0;i<n;i++)**

**for(int j=i+1;j<n;j++)**

**if(strcmp(a[j],a[i])<0)//比较两个字符串，右边比左边的小**

**swap(a[i],a[j]);//交换两个字符串内容**

**}**

**char\* ac[4]={"x","a","v","r"};//4个字符串，字符串是char\*类型的，ac是char\*类型的数组。所以在T\* a中带入的实际是ac[i]，即T==char\*，char\*\*才表示内容**

**写函数模板时：声明和定义不要分开**

**写好的模板如果要单独保存，模板要保存在头文件中（声明和定义不分开**）

类模板： 和函数模板差不多，都是在最前面添加template<typename T>。这被称为模板形参

**因为创建对象的时候未必传参数，因此类模板在使用的时候不一定能确定T是什么类型。因此在使用类模板时必须明确指定模板中的类型。 确定模板形参的过程称为实例化。**

**类模板实例化后才能称为类，才能够创建对象**

例如：

int main(){

**Stack<int> si;**

Stack<double> sd;

Stack<char> sc;

}

Stack<int>这是类名：模板名<类型实参>

**template<typename T,typrname U>//T和U各自是一个类型**

**struct Pair{**

**T first;**

**U second;**

**Pair():first(T()),second(U()){}// 类型名()表示0，所以T()==0** 区间？？？？？

**Pair(const T& f,const U& s):first(f),second(s){}**

};

int main()

{

**Pair<int,double> p1;**

Pair<int,string> p2;//p1是p2来自同一模板的两个类型（两个对象）

}

**函数模板（算法）**

**所有的类模板(容器)都有内部类：iterator(迭代器)。 几乎所有的函数模板形参都是iterator类型的迭代器**

**class iterator{**

**T\* o;**

**public:**

**iterator();**

**iterator(const iterator&i);//拷贝构造函数**

**iterator(T\* q);**

**operator\*,operator->,operator++,operator==,operator!=//支持运算符重载**

**}//模拟指针的能力**

**指针的操作符(在指针中可用)：operator\*,operator->,operator++,operator==,operator!=**

两个地址相减是他们间隔的变量的个数:22(地址)-12(地址)=10(变量个数)。 也就是区间

**区间只包含第一个指针指向的元素，不包含最后一个指针指向的元素：**

**(头指针)1(1表示头指针指向的元素)234(尾指针)5(5是尾指针指向的元素) 1234是区间内部元素，5不是**

19.**序列式容器：1.vector 动态数组 2.deque 双端队列 3.list 双向链表**

**关联式容器：1.map 映射 2.multimap 多映射 3.set 数据集 4.multiset 多数剧集**

**所有关联式容器都是二叉查找树模板(底层BST)**

**区间是由两个迭代器界定的一个范围**

**vector<int> v1;**

**vector<int> v2(v1);//拷贝构造函数，把v1里面的内容 复制 一份**

**vector<int> v3(begin,end)；//两个迭代器begin、end，这两个迭代器区间里的内容都会被复制一份到新容器中。这是区间构造函数**

**链表的循环中，不要用<=这类的循环判断点，用!=做判断**

例如：

**for(it=l2.begin();it!=l2.end();it++)**

**begin和end都是迭代器里的，只有用迭代器的时候才能这么写**

**vector和deque支持“[]”运算符，支持at（）运算符。 vi.at(1000)**

at就是找到第1000个数的位置

**iterator总是容器内部类型，所以用到迭代器的时候总是：**

**容器名<类型>::iterator**

**reverse\_iterator 反向迭代器**

const\_iterator

const\_reverse\_iterator

容器适配器：1.stack 栈 2.queue 队列 3.priority-queue 优先队列 操作与栈相同

插入指定位置，插入的是迭代器，**insert(pos,n,val),pos指迭代器的位置**

支持的函数：

序列式容器： 构造：Constructor(int n) Constructor(int n, T val)

调整: resize(int n) resize(int n,T val)

赋值：assign(n,val) assign(beg,end) 区间

**插入：insert(pos,n,val) insert(pos,区间)**

末尾插入新数据：push\_back(val)

首尾元素: front() back()

尾删: pop\_back()

void show(vector<int> vi)

{

vector<int>::iterator it;

it=vi.begin();

while(it!=vi.end())

**cout<<\*it++<<' ';//先取it里面的数据，即先执行\*it，再对it++**

cout<<endl;

}

**使用时加头文件： #include <vector>**

int main()

{

**vector<int> vi(3,90);// 放置3个“90”**

show(vi);

int a[50]={3,4,5,6,7};

vi.insert(vi.begin(),a,a+5);

show(vi);

vi.push\_back(100)

show(vi);

cout<<"size:"<<vi.size()<<endl;//大小为9

**vi.assign(5,99);//新数据放进去，旧的数据全部丢失**

**show(vi);**

**cout<<"size:"<<vi.size()<<endl;//大小为5 vi中变为5个“99”**

}

所有的容器在插入数据的时候会自动增大

**vector<int> v1(10);**

**vector<int> v2(5);**

**sizeof(v1)==sizeof(v2)//因为sizeof只关心类型，**

**vector只适合在末尾插入和删除数据**

凡是标准库抛出的异常都可以用 catch(exception& e),进行处理

catch(exception& e)

{

cout<<e.what()<<endl;

}

**链表中适用的函数：**

**splice 转移**

**c1.splice(pos,c2);//把c2里面的元素转移到c1中的制指定位置，c2中就没有这个元素了**

**c1.splice(pos,c2,it);//把c2里面it位置的元素转移到c1的pos位置中**

merge归并

c1.merge(c2);//把c2里面的元素合并到c1中合适的位置

remove删除

remove(val)//删除指定值的元素

unique：==

sort:排序 < 小于比较 ，排好序是从小到大

**关联式容器： 插入：insert(val) 把数直接插入，会自动到达合适的位置 不允许改变容器内的元素**

查找：find(val) 返回指向的找到的元素的迭代器（找到），没找到返回end()(无效位置迭代器)

map（数据不允许重复）、multimap（数据允许重复）、set、multiset

重复只是key不允许重复，value没有要求 (类似python中的键值对？)

**map中的元素：Pair<key,value> ，支持[key],用作索引**

**multimap:Pair<key,value>**

set:key

multiset:key

**erase(key)删除指定key的所有元素**

**寻找容器中某一个值最开始的地方**

**it=lower\_bound(val)**

**ie=upper\_bound(val)//等于指定值的最后一个元素的后面的位置**

**这两个函数组成的区间为等于指定值的所有的元素**

**（因为关联式容器会将数据自动排序）**

while(ib!=ie)

cout<<\*ib++;

equal\_range(val)//返回值是一个pair，一对数据，pair里面的两个元素都是迭代器，

一个first，一个second。

first==\*it，second==\*ie

#include <iostream>

using namespace std;

**#include <map>//multimap与map同在这个头文件里**

#include <sting>

int main()

{

**map<int,string> mis;**

**mis.insert(make\_pair(62,"李明"));//insert插入，make\_pair 制造pair，62对应int，“李明”对应string，是一个pair。（定义在上面）**

**mis.insert(make\_pair(32,"王岩"));**

**mis.insert(make\_pair(36,"李鹏"));**

**mis[20]="契丹"//方括号里面的类型与上面的int有关，若不是int，则里面不是整型，而是其它**

**mis.insert(make\_pair(32,"lala"));//输出结果中消失了**

**map<int,string>::iterator it;//迭代器**

**it=mis.begin(); //\*it是个pair，pair是个结构，不能用cout<<\*it输出**

**while(it!=mis.end())**

**{**

**cout<<it->first<<":"<<it->second<<endl;**

**++it;**

**}**

}

**输出结果：重复插入被忽略：20：契丹 32：王岩 36：李鹏 62：李明 会自动排好顺序**

int main()

{

**typedef multimap<string,string> M;**

**M mss;**

**M::iterator ib,ie;**

**mss.insert(make\_pair("aa","bb")); //键值对、 first，second**

**mss.insert(make\_pair("cc","dd"));**

**mss.insert(make\_pair("ee","ff"));**

**mss.insert(make\_pair("cc","pp"));**

**mss.insert(make\_pair("aa","mm"));**

**mss.insert(make\_pair("cc","dd"));**

**mss.insert(make\_pair("aa","kk"));**

**ib=mss.begin();**

**ie=mss.end();**

**while(ib!=ie)**

**{//是个pair，只能通过first和second来找到ib**

**cout<<ib->first<<"的好友"<<ib\_second<<endl;**

**++ib;**

**}**

**cout<<"aa的好友数量："<<mss.count("aa")<<endl;**

**cout<<"cc的所有好友:"<<endl;**

**ib=mss.lower\_bound("cc");**

**ie=mss.upper\_bound("cc");**

**while(ib!=ie)**

**{**

**cout<<ib->second<<' ';**

**++ib;**

**}**

**cout<<endl;**

}

输出结果排好顺序，按照key从小到大排好顺序，并且全部都保存下来，即有重复.不会按照value排序，key必须支持小于运算符

#include <set>

int main()

{

set<int> si;

**int userid[5]={3,8,6,1,3};**

for(int i=0;i<5;i++)

si.insert(userid[i]);

set<int>::iterator it;

it=si.begin();

while(it!=si.end())

**cout<<\*it++<<' ';**

cout<<endl;

cout<<"user 3:"<<(si.find(3)!=si.end())<<endl;

}\

输出结果：1,3,6,8 重复结果被忽略

#include<set>

template<typename Iter>

void show(Iter ib,Iter ie)

{

while(ib!=ie)

cout<<\*ib++<<' ';

cout<<endl;

}

int main()

{

int a[5]={5,1,7,5,1};

multiset<int> pids(a,a+5);

show(pids.begin(),pids.end());

pids.insert(7);

pids.insert(7);

pids.insert(7);

**pids.erase(pids.find(5)这里find是迭代器？？？？);//删除容器中元素为5的一个数。而不是把元素为5的数全都删掉**

show(pids.begin(),pids.end());

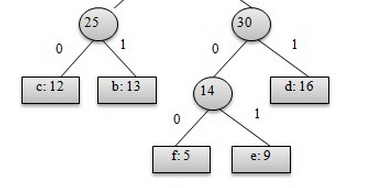
}

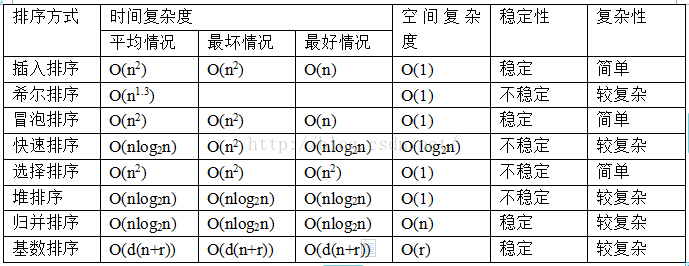
结果为：1,1,5,5,7

结果2位：1,1,5,7,7,7,7

**数值函数需要用到的头文件：<numeric> 例如：min(),max(),count(),swap(),accumulate()等等**

**其他的用到<algorithm>头文件。例如：sort()排序，reverse()排序等等**



****

char str[10];  
int a=1234321;  
(str,"%d",a);s

string->double  
  double d=atof(s.c\_str());

strrev函数实现字符串反转

 char s[]="hello";



     strrev(s);



使用algorithm中的reverse函数

 reverse(s.begin(),s.end());

前缀递增递减和\*优先级相同，从右到左；

后缀递增递减比前缀优先级高，从左到右。

\*++p：p先自＋，然后\*p，最终为3

++\*p：先\*p，即arr[0]=1，然后再++，最终为2

**\*p++：值为arr[0],即1，该语句执行完毕后，p指向arr[1]**

**(\*p)++：先\*p，即arr[0]=1，然后1++，该语句执行完毕后arr[0] =2**

**\*（p++）：效果等同于\*p++**

inline函数 在调用点展开

**编译器必须随处可见内联函数的定义**

每个调用了内联函数的文件都出现了该**内联函数的定义**。

因此，将**内联函数的定义**放在**头文件**里实现是合适的，省却你为每个文件实现一次的麻烦

**定义**在类中的**成员函数**缺省都是**内联的**

如果在类中未给出成员函数定义，而又想内联该函数的话，那在类外要加上inline，否则就认为不是内联的

关键字inline 必须与**函数定义体**放在一起才能使函数成为内联

inline 是一种“**用于实现的关键字**”

next数组值的程序设计求解方法：首先可以肯定的是第一位的next值为0，第二位的next值为1，后面求解每一位的next值时，根据前一位 进行比较。首先将前一位与其next值对应的内容进行比较，如果相等，则该位的next值就是前一位的next值加上1；如果不等，向前继续寻找next值对应的内容来与前一位进行比较，直到找到某个位上内容的next值对应的内容与前一位相等为止，则这个位对应的值加上1即为需求的next值；如果找到 第一位都没有找到与前一位相等的内容，那么需求的位上的next值即为1。

**举例：**  
模式串  a  b  a  a  b  c  a  c  
next值  0  1  1  2  2  3  1  2  
1.前两位必为0，1。  
2.计算第三位的时候，看第二位b的next值，为1，则把b和1对应的a进行比较，不同，则第三位a的next的值为1，因为一直比到最前一位，都没有发生比较相同的现象。  
3.计算第四位的时候，看第三位a的next值，为1，则把a和1对应的a进行比较，相同，则第四位a的next的值为第三位a的next值加上1，为2。因为是在第三位实现了其next值对应     
的值与第三位的值相同。  
4.计算第五位的时候，看第四位a的next值，为2，则把a和2对应的b进行比较，不同，则再将b对应的next值1对应的a与第四位的a进行比较，相同，则第五位的next值为第二位b的     
next值加上1，为2。因为是在第二位实现了其next值对应的值与第四位的值相同。  
5.计算第六位的时候，看第五位b的next值，为2，则把b和2对应的b进行比较，相同，则第六位c的next值为第五位b的next值加上1，为3，因为是在第五位实现了其next值对应的     
值与第五位相  
6.计算第七位的时候，看第六位c的next值，为3，则把c和3对应的a进行比较，不同，则再把第3位a的next值1对应的a与第六位c比较，仍然不同，则第七位的next值为1。  
7.计算第八位的时候，看第七位a的next值，为1，则把a和1对应的a进行比较，相同，则第八位c的next值为第七位a的next值加上1，为2，因为是在第七位和实现了其next值对应的值与第七位相同。

模式串      a b a a b c a c

　　next值      0 1 1 2 2 3 1 2

　　nextval值 0 1 0 2 1 3 0 2

　　1.第一位的nextval值必定为0，第二位如果于第一位相同则为0，如果不同则为1。

　　2.第三位的next值为1，那么将第三位和第一位进行比较，均为a，相同，则，第三位的nextval值为0。

　　3.第四位的next值为2，那么将第四位和第二位进行比较，不同，则第四位的nextval值为其next值，为2。

　　4.第五位的next值为2，那么将第五位和第二位进行比较，相同，第二位的next值为1，则继续将第二位与第一位进行比较，不同，则第五位的nextval值为第二位的next值，为1。

　　5.第六位的next值为3，那么将第六位和第三位进行比较，不同，则第六位的nextval值为其next值，为3。

　　6.第七位的next值为1，那么将第七位和第一位进行比较，相同，则第七位的nextval值为0。

　　7.第八位的next值为2，那么将第八位和第二位进行比较，不同，则第八位的nextval值为其next值，为2。