目录

**一、C++基础1**

键入章标题(第 2 级)2

键入章标题(第 3 级)3

**二、键入章标题(第 1 级)4**

键入章标题(第 2 级)5

键入章标题(第 3 级)6

一、C++基础

1、sizeof();

占用的内存空间;

2、创建字符型变量的时候，要用单引号；单引号内只能有一个字符；

sizeof(char) = 1 个字节；a == 97 ; A == 65;

3、字符串型

1、C风格字符串：string **char[]** = “hello world”;

2、C++风格字符串：string str2 = “hello world”; 需要包含头文件：#include <string>

4、bool类型

1、sizeof(bool) = 1; 非 0 即为 真；

5、字符的输入：

cin；语法：cin>>数据;

6、递增/递减：

前置递增：

int a2 = 10;

int b2 = ++a2 \* 10; a2 = 11; b2 = 110;

后置递增：

int a3 = 10;

int b3 = a3++ \* 10; a3 = 11; b3 = 100;

7、赋值运算符

1、= ，+=，-=，\*=，/=，%= ;

8、比较运算符：

1、== , !=, < , >, <=, >=;

9、逻辑运算符

1、 ! (非); && (与); || (或);

10、程序流程结构

1、顺序结构

2、选择结构: if….else…..

3、循环结构

11、3目运算符

表达式1 ？ 表达式2 ： 表达式3

在C++中三目运算符返回的是变量，可以继续赋值

12、if 和 switch

switch缺点：判断时只能是整型或者字符型，不可以是一个区间

switch优点：结构清晰，执行效率高

13、while循环

在执行循环语句的时候，程序必须提供跳出循环的出口，避免出现死循环

14、break的使用时机

1、switch中使用；

2、for循环中使用；

3、嵌套循环中使用；

15、continue语句

可以筛选条件，执行到此就不再向下执行，执行下一次循环；

16、goto语句

如果标记的名称存在，执行到goto语句，会跳转到标记的位置

17、数组

1、数组中的每个数据元素都是相同的数据类型

2、数组是由连续的内存位置组成的

18、一维数组数组名

1、可以统计整个数组在内存中的长度

2、可以获取数组在内存中的首地址

3、数组名是常量，不可以进行赋值操作；

19、冒泡排序

排序的总轮数 = 元素个数 – 1；

每轮对比的次数 = 元素个数 – 排序轮数 – 1；

20、二维数组

21、函数：

返回值类型；

函数名；

参数列表；

函数体语句；

return 表达式；

返回值类型 函数名 (参数列表) {

函数体语句；

return 表达式；

}

函数的调用语法：函数名称(参数)

<https://github.com/microsoft/calculator>

https://github.com/microsoft/terminal

22、值传递

当我们做 值传递 的时候，函数的形参发生变化，并不会影响实参

23、函数常见的样式

1、无参无返

2、有参无返

3、无参有返

4、有参有返

24、函数的声明（返回值类型 函数名称 (参数列表)）

作用：告诉编译器 函数名称以及 如何调用函数，函数的实际主题可以单独定义；

函数的声明可以多次，但是函数的定义只能有一次；

25、函数的分文件编写

1、创建后缀名为.h的头文件

2、创建后缀名为.cpp的源文件

3、在头文件中写函数声明

4、在源文件中写函数定义

26、指针

1、指针的基本概念：

指针的作用：可以通过指针间接访问内存

内存编号是从 0 开始记录的，一般用16进制数字表示

可以利用指针变量保存地址；

2、指针变量的定义和使用

指针变量定义语法：数据类型 \* 变量名

27、指针所占内存空间

指针也是种数据类型，那么这种数据类型占用多少内存空间？

28、空指针和野指针

空指针：指针变量指向内存中编号为0 的空间

用 途：初始化指针变量

注 意：空指针指向的内存是不可以访问的；

野指针：指针变量指向非法的内存空间

int main()

{

// 野指针

int \*p = (int \*)0x1100;

// 访问野指针报错

cout<<”\*p”<<endl;

system(“pause”);

return 0;

}

在程序中，避免出现野指针；

29、const修饰指针

1、const修饰指针-------常量指针

2、const修饰常量-------指针常量

3、const即修饰指针，又修饰常量

P1 a 0x0011

10

0x0011

int a = 10;

int b =10; b 0x0022

10

int\*p = &a;

const int\* p = &a;

常量指针：

特点：指针的指向可以修改，但是指针指向的值不可以修改；

int\* const p = &a;

指针常量：

特点：指针的指向不可以改，但是指针指向的值可以修改；

\*p = 20; 正确，指向的值可以改；

P = &b; 错误，指针指向不可以改；

const int \* const p = &a;

即修饰指针又修饰常量：

特点：指针的指向和 指针指向的值都 不可以修改；

30、指针和数组

作用：利用指针访问数组中的元素；

31、指针和函数

作用：利用指针做函数参数，可以修改实参的值；

|  |
| --- |
| a 10 |
| B 20 |
| \*p1 0x0011 |
| \*p2 0x0022 |
| Temp 10 |
|  |
|  |
|  |
|  |
|  |

int a = 10, b = 20;

void swap02(int\* p1, int\* p2) {

int temp = \*p1;

\*p1 = \*p2; // 20赋值给\*p1,赋给a;

\*p2 = temp;

}

32、结构体

1、struct 结构体名 { 结构体成员列表 };

2、通过结构体创建变量的方式有三种：

struct 结构体名 变量名

struct 结构体名 变量名 = { 成员1值，成员2值}；

定义解结构体时顺便创建变量

3、定义结构体时的关键字是struct ,不可省略

创建结构体变量时，关键字struct可以省略

结构体变量利用操作符 “.”访问成员

33、结构体数组

将自定义的结构体放入到数组中，方便维护；

语法：struct 结构体名 数组名[元素个数] = { {}, {}, {},{}……};

34、结构体指针

作用：通过指针访问结构体中的成员；

利用操作符 -> 可以通过结构体指针访问结构体属性；

结构体嵌套结构体：

作用：结构体中的成员可以是另一个结构体；

结构体做函数参数：

作用：将结构体作为参数向函数中传递

传递方式又两种：值传递；地址传递；

总结：如果不想修改主函数中的数据，用值传递，反之用地址传递；

35、结构体中使用const的场景

作用：用const来防止误操作；

**C++核心编程：面向对象**

36、内存分区模型

C++程序在执行时，将内存大方向划分为4个分区

程序运行前

代码区：存放函数体的二进制代码，由操作系统进行管理的

全局区：存放全局变量和静态变量以及常量

程序运行后

栈 区：由编译器自动分配释放，存放函数的参数值，局部变量等；

堆 区：由程序员分配内存和释放，若程序员不释放，程序结束时由操作系统回收；

内存4区的意义：不同区域存放的数据，赋予不同的生命周期，给我们更大的灵活编程；

**程序运行前：**

代码区：

代码区是共享的：共享的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可；

代码区是只读的：使其只读的原因是防止程序意外的修改了他的指令；

全局区：

全局变量 和 静态变量存放于此；

全局区还包含了常量区，字符串常量和其他常量也存放于此；

该区域的数据在程序结束后由操作系统释放；

总结：C++在程序运行前分为 全局区 和 代码区

代码区的特点是 共享 和 只读

全局区中存放 全局变量，静态变量，常量

常量区中存放const修饰的全局常量和字符串常量

**程序运行后：**

栈区：由编译器自动分配释放，存放函数的参数值，局部变量等；

注意事项：不要返回局部变量的地址，栈区开辟的数据由编译器自动释放；

堆区：由程序员分配释放，若程序员不释放，程序结束时由操作系统回收；

在C++中主要利用new 在堆区开辟内存；new….delete new[]….delete[]

利用new创建数组：int\* arr = new int[10];

利用new创建的数据，会返回该数据对应的类型的指针；

37、引用

1、引用的基本使用：

作用：给变量起别名；

语法：数据类型 &别名 = 原名；

2、引用的注意事项

1、引用必须初始化

2、引用在初始化后，不可以改变；

3、引用做函数参数

作用：函数传参时，可以利用引用的技术让形参修饰实参

优点：可以简化指针修改实参

总结：通过引用参数产生的效果同按地址传递是一样的。引用的语法更清晰简单；

4、引用做函数的返回值

作用：引用是可以作为函数的返回值存在的

注意：不要返回局部变量引用；

用法：函数调用作为左值；

5、引用的本质

本质：引用的本质在C++内部实现是一个指针常量

6、常量引用

作用：常量引用主要用来修饰形参，防止误操作；

在函数形参列表中，可以加const 修饰形参，防止形参改变实参

38、函数提高

1、在C++中，函数的形参列表中的形参是可以有默认值的；

语法：返回值类型 函数名 （参数=默认值）{}

2、函数的占位参数

C++中函数的形参列表里可以有占位参数，用来做占位，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) {}

3、函数重载

作用：函数名可以相同，提高复用性

函数重载满足条件：① 同一个作用域下

② 函数名称相同

③ 函数的参数类型不同 或者 个数不同 或者 顺序不同

注意：函数的返回值不能作为函数重载的条件；

4、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；

39、类和对象

1、C++面向对象的3大特性：封装，继承，多态；C++认为万事万物都皆为对象，对象上有其属性和行为；

2、封装：

封装的意义一：将属性和行为作为一个整体，表现生活中的事务；

将属性和行为加以权限控制；

在设计类的时候，属性和行为写在一起，表现事务；

语法：class 类名 { 访问权限：属性/行为 };

封装的意义二：类在设计时，可以把属性和行为放在不同的权限下，加以控制

访问权限有3种：public 公共权限 类内可以访问，类外可以访问

protected保护权限 类内可以访问，类外不可以访问

private 私有权限 类内可以访问，类外不可以访问

struct 和 class 的区别：

struct 默认权限为公共；

class 默认权限为私有；

成员属性设置为私有：

优点1：将所有的成员属性设置为私有，可以自己控制读写权限

优点2：对于写权限，我们可以检测数据的有效性

**对象的初始化和清理：**

**构造函数和析构函数：**

对象的初始化和清理也是两个非常重要的安全问题；

一个对象或者变量没有初始状态，对其使用后果是未知；

同样的使用完一个对象或变量，没有及时清理，也会造成一定的安全问题；

C++利用了构造函数和析构函数解决上述问题，这两个函数将会被编译器自动调用，完成对象的初始化和清理工作；对象的初始化和清理工作是编译器强制要我们做的事情，因此如果我们不提供构造和析构，编译器会提供编译器提供的构造函数和析构函数是空实现。

构造函数：主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用，无须手动调用；

析构函数：主要作用在于对象销毁前系统自动调用，执行一些清理工作；

构造函数语法：类名() {}

1. 构造函数，没有返回值也不写void
2. 函数名称与类名相同
3. 构造函数可以有参数，因此可以发生重载；
4. 程序在调用对象时候会自动调用构造，无须手动调用，而且只会调用一次；

析构函数语法：~类名() {}

1. 析构函数，没有返回值也不写void
2. 函数名称与类名相同，在名称前加上符号~；
3. 析构函数不可以有参数，因此不可以发生重载；
4. 程序在对象销毁前会自动调用析构，无须手动调用，而且只会调用一次；

**构造函数的分类及调用：**

两种分类方式：

按参数分类：有参构造和无参构造

按类型分类：普通构造和拷贝构造

三种调用方式：

括号法；

显示法；

隐式转换法；

拷贝构造函数的调用时机：

C++中拷贝构造函数调用时机通常有三种情况：

① 使用一个已经创建完毕的对象来初始化一个新对象

② 值传递的方式给函数参数传值

③ 以值方式返回局部对象

构造函数的调用规则：默认情况下，C++编译器 至少给一个类添加3个函数

① 默认构造函数（无参，函数体为空）

② 默认析构函数（无参，函数体为空）

③ 默认拷贝构造函数，对属性进行值拷贝

构造函数调用规则如下：

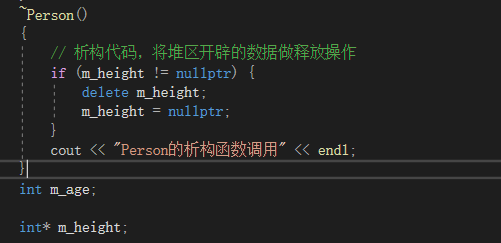
① 如果用户定义有参构造函数，C++不在提供默认无参构造函数，但是会提供默认拷贝构造；如果写了拷贝构造函数，编译器就不再提供其他普通的构造函数了；

② 如果用户定义拷贝构造函数，C++不会在提供其他构造函数

深拷与浅拷贝：

浅拷贝：简单的赋值拷贝操作

深拷贝：在堆区重新申请空间，进行拷贝操作；



P1 p2

|  |
| --- |
| Int m\_age  18 |
| Int\* m\_height  0x0011换成0x0022 |

|  |
| --- |
| Int m\_age  18 |
| Int \*m\_height  0x0011 |

Person p2(p1)

如果利用编译器提

供的拷贝构造函数，

会做浅拷贝操作

堆区（0x0011）

0x0022(160)

160

**浅拷贝带来的问题：堆区的内存重复释放**

P2释放的时候，\*m\_height 被释放，P1析构的时候再次释放m\_height,重复释放

浅拷贝的问题 要 利用 深拷贝进行解决；

**如果属性有在堆区开辟的，一定要自己提供拷贝构造函数，防止浅拷贝带来的问题；**

**初始化列表：**

作用：C++提供了初始化列表语法，用来初始化属性；

语法：构造函数():属性1(值1), 属性2(值2), …{}

**类对象作为类成员：**

C++类中的成员可以是另一个类的对象，我们称该成员为对象成员；

Class A {};

Class B { A b; };

当其他类对象作为本类成员，构造的时候先构造类对象，在构造自身；析构的顺序：和构造相反

**静态成员：**

静态成员就是在成员变量和成员函数前加上关键字static,称为静态成员；

静态成员分为：

静态成员变量：

※ 所有对象共享同一份数据；

※ 在编译阶段分配内存；

※ 类内声明，类外初始化；

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；

**C++对象模型和this指针：**

**Ⅰ、成员变量和成员函数分开存储**

在C++中，类内的成员变量和成员函数分开存储；

只有非静态成员变量才属于类的对象上；

**Ⅱ、this 指针**

C++中成员函数和成员变量是分开存储的；

每一个非静态成员函数只会诞生一份函数实例，也就是说多个同类型的对象会共用一块代码；那么问题是：这一块代码是如何区分那个对象调用自己的呢？

C++通过提供特殊的对象指针，this指针，解决上述问题，**this指针指向被调用的成员函数所属的对象；**

**This指针是隐含每一个非静态成员函数内的一种指针；**

**This指针不需要定义，直接使用即可；**

**This指针的用途：**

**当形参 和 成员变量同名时，可用this指针来区分；**

**在类的非静态成员函数中返回对象本身，可使用 return \*this;**

**Ⅲ、空指针访问成员函数**

C++中空指针也是可以调用成员函数的，但是也要注意有没有用到this指针；

如果用到this指针，需要加以判断保证代码的健壮性；

**Ⅳ、const 修饰成员函数**

**常函数：**

成员函数后加const后我们称这个函数为常函数；

常函数内不可以修改成员属性；

成员属性声明时加上关键字mutable后，在函数中依然可以修改；

**常对象：**

声明对象前加const称该对象为常对象；

常对象只能调用常函数；

**友元**：

在程序里，有些私有属性也想让类外特殊的一些函数或者类进行访问，就需要用到友元技术；

友元的目的就是让一个函数或者类访问另一个类中私有成员；

友元的关键字为friend;

友元的三种实现：

全局函数做友元；

类做友元；

成员函数做友元；

**运算符重载**：

运算符重载概念：对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型；

**Ⅰ、加号运算符重载**

**作用：实现两个自定义数据类型相加的运算；**

**总结1：对于内置的数据类型的表达式的运算符是不可能改变的；**

**总结2：不要滥用运算符重载**

**Ⅱ、左移运算符的重载**

**作用：可以输出自定义数据类型；**

**总结：重载左移运算符配合友元可以实现输出自定义数据类型；**

**Ⅲ、递增运算符的重载**

**作用：通过重载递增运算符，实现自己的整型数据；**

**Ⅳ、赋值运算符的重载**

C++编译器至少给一个类添加4个函数：

1. 默认构造函数（无参，函数体为空）
2. 默认析构函数（无参，函数体为空）
3. 默认拷贝构造函数，对属性进行值拷贝
4. 赋值运算符operator=,对属性进行值拷贝

如果类中有属性指向堆区，做赋值操作时也会出现深浅拷贝问题；

**Ⅴ、关系运算符的重载**

作用：重载关系运算符，可以让两个自定义类型对象进行对比操作；

**Ⅵ、函数调用运算符的重载**

函数调用运算符() 也可以重载；

由于重载后使用的方式非常像函数的调用，因此称为仿函数；

仿函数没有固定写法，非常灵活；

**3、继承：**

**好处：减少重复的代码；**

**语法：class 子类 ： 继承方式 父类**

Class A:public B

A类称为子类 或 派生类

B类称为父类 或 基类

派生类中的成员，包含两大部分：

一类是从基类继承过来的，一类是自己增加的成员；

从基类继承过来的表现其共性，而新增的成员体现了其个性；

**继承的方式：**

**公共继承**

**保护继承**

**私有继承**

|  |
| --- |
| **Class A{**  **Public:**  **Int a;**  **Protected:**  **Int b;**  **Private:**  **Int c;**  **};** |

**公共继承 私有继承**

**保护继承**

|  |
| --- |
| **Class B:public A**  **{**  **Public:**  **Int a;**  **Protected:**  **Int b;**  **不可访问**  **Private:**  **Int c;**  **};** |

|  |
| --- |
| **Class B:protected A{**  **protected:**  **Int a;**  **Int b;**  **不可访问**  **Private:**  **Int c;**  **};** |

|  |
| --- |
| **Class B：private A{**  **private:**  **Int a;**  **Int b;**  **不可访问**  **Private:**  **Int c;**  **};** |

**继承中的对象模型：**

利用开发人员命令提示工具查看对象模型

cl /d1 reportSingleClass类名 文件名

结论：父类中的私有成员也是被子类继承下去的，只是由编译器给隐藏后访问不到；



**继承中的构造和析构顺序：**

**子类继承父类后，当创建子类的对象，也会调用父类的构造函数；**

**总结：**继承中 先调用父类的构造函数， 在调用子类的构造函数，析构顺序与构造相反；

**继承中同名成员的处理方式：**

**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；**

**总结：1、子类对象可以直接访问到子类中同名的成员；**

**2、子类对象加作用域可以访问到父类同名成员；**

**3、当子类与父类拥有同名的成员函数，子类会隐藏父类中同名成员函数，加作用域可以访问到父类中同名函数；**

**继承中同名静态成员处理方式：**

**静态成员和非静态成员出现同名，处理方式一掷；**

**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；**

**总结：同名静态成员处理方式和非静态处理方式一样，只不过有两种访问方式（通过 对象 和 类名）；**

**多继承语法：**

**C++允许一个类继承多个类**

**语法：class 子类： 继承方式 父类1，继承方式 父类2…**

**多继承可能会引发父类中有同名成员出现，需要加作用域区分；**

**总结：多继承中如果出现了同名情况，子类使用的时候要加作用域；**

**菱形继承：**

**两个派生类继承同一个基类；**

**又有某个类同时继承两个派生类；**

**这种继承方式被称为菱形继承或者钻石继承；**

**总结：菱形继承带来的主要问题是子类继承两份相同的数据，导致资源浪费以及毫无意义；**

**利用 虚继承 可以解决 菱形问题；**

**4、多态：**

**多态分为两类：**

**静态多态：函数重载和运算符重载属于静态多态，复用函数名**

**动态多态：派生类和虚函数实现运行时多态**

**静态多态和动态多态的区别：**

**静态多态的函数地址早绑定----编译阶段确定函数地址；**

**动态多态的函数地址晚绑定----运行阶段确定函数地址；**

**总结：** **动态多态满足条件：**

**1、有继承关系**

**2、子类重写父类的虚函数；**

**动态多态的使用：**

1. **父类的指针或引用 指向子类对象**

**重写：函数返回值类型，函数名，参数列表，完全一致称为重写；**

**当父类的指针或者引用指向子类对象的时候，发生多态；**

**多态的优点：**

**1、代码组织结构清晰；**

**2、可读性强；**

**3、利于前期和后期的扩展以及维护；**

**总结：C++开发提倡利用多态涉及程序架构，因为多态优点多；**

**纯虚函数和抽象类：**

在多态中，通常父类中虚函数的实现是毫无意义的，主要都是调用子类重写的内容；

因此可以将虚函数改为纯虚函数；

**纯虚函数语法：virtual 返回值类型 函数名 （参数列表）= 0；**

**当类中有了纯虚函数，这个类也称为抽象类；**

**抽象类的特点：**

* **无法实例化对象**
* **子类必须重写抽象类中的纯虚函数，否则也属于抽象类；**

**虚析构和纯虚析构：**

**多态使用时，如果子类中有属性开辟到堆区，那么父类指针在释放时无法调用到子类的析构代码**

**解决方法：将父类中的析构函数改为虚析构或者纯虚析构；**

**虚析构和纯虚析构共性：**

**· 可以解决父类指针释放子类对象；**

**· 都需要有具体的函数实现；**

**虚析构和纯虚析构的区别：**

**如果是纯虚析构，该类属于抽象类，无法实例化对象；**

虚析构语法：

**Virtual ~类名() {};**

纯虚析构语法：

**Virtual ~类名() = 0;**

**类名::~类名() {}**

**总结：**

1. **虚析构和纯虚析构就是用来解决通过父类指针释放子类对象；**
2. **如果子类中没有堆区数据，可以不写为虚析构或纯虚析构；**
3. **拥有纯虚析构函数的类也属于抽象类；**

40、文件操作

程序运行时产生的数据都属于临时数据，程序一旦运行结束都会被释放

**通过文件可以将数据持久化**

C++中对文件操作需要包含头文件<fstream>

文件类型分为两种：

1. **文本文件** 文件以文本的ASCII码形式存储在计算机中
2. **二进制文件** 文件以文本的二进制形式存储在计算机中，用户一般不能直接读懂他们；

操作文件的三大类：

1. ofstream: 写操作
2. ifstream: 读操作
3. fstream : 读写操作

文本文件：

写文件：

写文件步骤如下：

1. 包含头文件 #include<fstream>
2. 创建流对象 ofstream ofs;
3. 打开文件 osf.open(“文件路径”, 打开方式);
4. 写数据 ofs<<”写入的数据”
5. 关闭文件 ofs.close();

文件打开方式：

ios::in 为读文件而打开文件

ios::out 为写文件而打开文件

ios::ate 初始位置：文件尾

ios::app 追加方式写文件

ios::trunc 如果文件存在先删除，在创建

ios::binary 二进制方式

注意：文件打开方式可以配合使用，利用 | 操作符；

例如：用二进制方式写文件 ios::binary | ios::out;

总结：

· 文件操作必须包含头文件<fstream>

· 读文件可以利用ofstream,或者fstream

· 打开文件时候需要指定操作文件的路径，以及打开方式

· 利用 << 可以向文件中写数据

· 操作完毕，要关闭文件

读文件：

读文件与写文件步骤相似，但是读取方式相对于比较多；

读文件步骤如下：

1. 包含头文件 #include<fstream>
2. 创建流对象 ifstream ifs;
3. 打开文件并判断文件是否打开成功

isf.open(“文件路径”, 打开方式);

1. 读数据 四种方式读取
2. 关闭文件 ifs.close();

// 读数据

// 第一种方式

/\*char buff[1024] = { 0 };

while (ifs >> buff) {

cout << buff << endl;

}\*/

// 第二种方式

/\*char buff[1024] = { 0 };

while (ifs.getline(buff, sizeof(buff))) {

cout << buff << endl;

}\*/

// 第三种方式

string buf;

/\*while (getline(ifs, buf)) {

cout << buf << endl;

}\*/

// 第四种方式

char c;

while ((c = ifs.get()) != EOF) { // EOF == End Of File

cout << c;

}

总结：

1. 读文件可以利用ifstream,或者fstream
2. 利用is\_open()函数可以判断文件是否打开成功
3. close 关闭文件

二进制文件：

以二进制的方式对文件进行读写操作；

打开方式要指定为 ios::binary;

写文件：

二进制方式写文件主要利用流对象调用成员函数write;

函数原型：ofstream& write (const char\* buffer, int len)

参数解释：字符指针buffer指向内存中一段存储空间，len是读写的字节数

读文件：

二进制方式读文件主要利用流对象调用成员函数read;

函数原型：ofstream& read (char\* buffer, int len)

参数解释：字符指针buffer指向内存中一段存储空间，len是读写的字节数

总结：

文件输入流对象 可以通过read函数，以二进制方式读数据；

**C++提高编程**

41、模板

模板的概念：模板就是建立通用的模具，大大提高复用性

函数模板：

C++另一种编程思想是泛型编程，主要利用的技术就是模板

C++提供两种模板机制：**函数模板**和**类模板**

**函数模板语法：**

函数模板作用：建立一个通用函数，其函数返回值类型和形参类型可以不具体确定，用一个**虚拟的类型**来代表。

语法：**template <typename T>**

函数声明或定义

解释：template --- 声明创建模板

Typename --- 表面其后面的符号是一种数据类型，可以用class 代替；

T --- 通用的数据类型，名称可以替换，通常为大写字母；

总结：

\* 函数模板利用关键字template

\* 使用函数模板有两种方式：自动类型推导，显示指定类型

\* 模板的目的是为了提高复用性，将类型参数化

函数模板注意事项：

自动类型推导，必须推导出一致的数据类型T，才可以使用

模板必须要确定出T的数据类型，才可以使用；

总结：使用模板时必须确定出通用数据类型T，并且能够推导出一致的类型；

普通函数与函数模板的区别：

\* 普通函数调用时可以发生自动类型转换（隐式类型转换）

\* 函数模板调用时，如果利用自动类型推导，不会发生隐式类型转换

\* 如果利用显示指定类型的方式，可以发生隐式类型转换；

总结：建议使用显示指定类型的方式，调用函数模板，因为可以自己确定通用类型T;

普通函数和函数模板的调用规则

调用规则如下：1、如果函数模板和普通函数都可以实现，优先调用普通函数

2、可以通过空模板参数列表来强制调用函数模板

3、函数模板也可以发生重载

4、如果函数模板可以产生更好的匹配，优先调用函数模板

总结：既然提供了函数模板，最好就不要提供普通函数，否则容易出现二义性；

模板的局限性：

模板的通用性并不是万能的；

例如：

Template<typename T>

Void f(T a,T b)

{

A=b;

}

在上述代码中提供的赋值操作，如果传入的a和b是一个数组，就无法实现了；

在例如：

Template <typename T>

Void f(T a,T b)

{

If(a>b) {……}

}

在上述代码中，如果T的数据类型传入的是像Person这样自定义的数据类型，也无法正常运行；

因此C++为了解决这种问题，提供模板的重载，可以为这些特定的类型提供具体化的模板；

总结：利用具体化的模板，可以解决自定义类型的通用化

学习模板并不是为了写模板，而是在STL能够运用系统提供的模板；

42、类模板

类模板语法：

类模板作用：建立一个通用类，类中的成员，数据类型可以不具体制定，用一个虚拟的类型来代表。

语法：template <typename T>

类

解释：template --- 声明创建模板

typename --- 表明其后面的符号是一种数据类型，可以用class 代替

T --- 通用的数据类型，名称可以替换，通常为大写字母；

总结：类模板和函数模板语法相似，在声明模板template后面加类，此类称为类模板；

类模板和函数模板的区别：1、类模板没有自动类型推导的使用方式；

2、板在模板参数列表中可以有默认参数；

总结：类模板使用只能用显示指定类型；

类模板中的模板参数列表可以有默认参数；

类模板中成员函数的创建时机：

类模板中成员函数和普通类中成员函数创建时机是有区别的：

普通类中的成员函数一开始就可以创建；

类模板中的成员函数在调用时才创建；

总结：类模板中的成员函数并不是一开始就创建的，在调用时才去创建；

类模板对象做函数参数：

学习目标：类模板实例化出的对象，向函数传参的方式；

一共有三种传入方式：

① 指定传入的类型 --- 直接显示对象的数据类型

② 参数模板化 --- 将对象中的参数变为模板进行传递

③ 整个类模板化 --- 将这个对象类型模板化进行传递

总结：通过类模板创建的对象，可以有三种方式向函数中进行传参

使用比较广泛是第一种：指定传入类型；

类模板与继承：

当类模板碰到继承时，需要注意以下几点：

当子类继承的父类是一个模板时，子类在声明的时候，要指定出父类中T的类型；

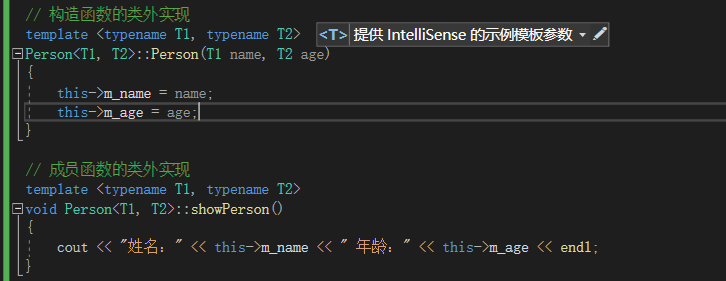
如果不指定，编译器无法给子类分配内存；

如果想灵活指定出父类中T的类型，子类也需变为类模板；

总结：如果父类是类模板，子类需要指定出父类中T的数据类型；

类模板成员函数类外实现

学习目标：能够掌握类模板中的成员函数类外实现



类模板分文件编写

学习目标：掌握类模板成员函数分文件编写产生的问题以及解决方式

问题：类模板中成员函数创建时机是在调用阶段，导致分文件编写时链接不到；

解决：解决方式1：直接包含.cpp源文件

解决方式2：将声明和实现写到同一个文件中，并更改后缀名为.hpp，hpp是约定的名称，并不是强制；

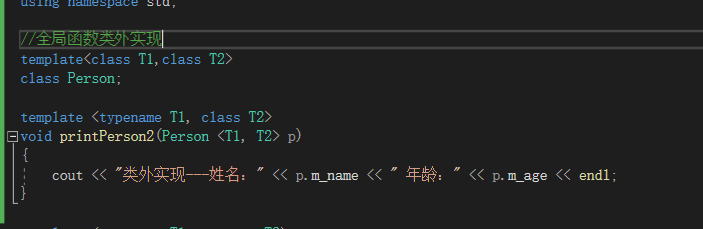
总结：主流的解决方式是第二种，将类模板成员函数写到一起，并将后缀名改为.hpp；

类模板与友元

学习目标：掌握类模板配合友元函数的类内和类外实现

全局函数类内实现 – 直接在类内声明友元即可

全局函数类外实现 – 需要提前让编译器知道全局函数的存在



总结：建议全局函数做类内实现，用法简单，而且编译器可以直接识别；

43、STL初识

1、STL的诞生

长久以来，软件界一直希望建立一种可重复利用的东西；

C++的**面向对象**和**泛型编程**思想，目的就是**复用性的提升**；

大多数情况下，数据结构和算法都未能有一套标准，导致被迫从事大量重复工作；

为了建立数据结构和算法的一套标准，诞生了**STL**;

2、STL基本概念：

. STL(标准模板库);

. STL从广义上分为：容器 算法 迭代器

. **容器**和**算法**之间通过**迭代器**进行无缝链接

. STL几乎所有的代码都采用了模板类或者模板函数

3、STL 6大组件  
 STL大体分为六大组件，分别是：**容器，算法，迭代器，仿函数，适配器(配接器)、空间配置器；**

容器：各种数据结构：如vector、list、deque、set、map等，用来存放数据

算法：各种常用的算法：如sort,find,copy,for\_each等

迭代器：扮演了容器与算法之间的胶合剂

仿函数：行为类似函数，可作为算法的某种策略

适配器：一种用来修饰容器或者仿函数或迭代器接口的东西

空间配置器：负责空间的配置与管理

4、STL中容器、算法、迭代器

容器：置物之所也

STL容器就是将运用最广泛的一些数据结构实现出来

常用的数据结构有：数组、链表、树、栈、队列、集合、映射表等

这些容器分为序列式容器和关联式容器两种：

序列式容器：强调值的排列，序列式容器中的每个元素均有固定的位置

关联式容器：二叉树结构，各元素之间没有严格的物理上的顺序关系

算法：问题之解法也

有限的步骤，解决逻辑上或者数学上的问题，这一门学科我们叫做算法；

算法分为：质变算法和非质变算法。

质变算法：是指运算过程中会更改区间内的元素内容，例如：拷贝，替换，删除等等；

非质变算法：是指运算过程中不会更改区间内的元素内容，例如：查找， 计数，遍历，寻找极值等等；

迭代器：容器和算法之间的粘合剂

提供一种方法，使之能够依序寻访某个容器所含的各个元素，而又无需暴露该容器的内部表示方式；

每个容器都有自己专属的迭代器；

迭代器使用非常类似于指针，初学阶段我们可以先理解迭代器为指针；

迭代器种类：

|  |  |  |
| --- | --- | --- |
| 种类 | 功能 | 支持运算 |
| 输入迭代器 | 对数据的只读访问 | 只读，支持++，==，！= |
| 输出迭代器 | 对数据的只写访问 | 只写，支持++ |
| 前向迭代器 | 读写操作，并能向前推进迭代器 | 读写，支持++，==，！= |
| 双向迭代器 | 读写操作，并能向前和向后操作 | 读写，支持++，-- |
| 随机访问迭代器 | 读写操作，可以以跳跃的方式访问任意数据，功能最强的迭代器 | 读写，支持++，--，[n]，-n，<，<=，>，>= |

常用的容器中迭代器种类为双向迭代器，和随机访问迭代器；

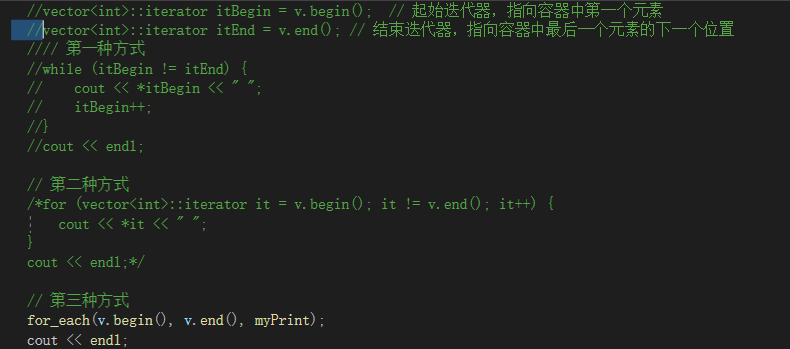
5、容器算法迭代器初识

Vector存放内置数据类型

容器：vector

算法：for\_each();

迭代器：vector<int>::iterator



Vector中存放自定义数据类型

学习目标：Vector中存放自定义数据类型，并打印输出

Vector容器嵌套容器

学习目标：容器中嵌套容器，我们将所有数据进行遍历输出

44、**string基本概念**

本质：string 是c++风格的字符串，而string本质上是一个类；

String 和 char\* 区别：

Char\* 是一个指针

String 是一个类，类内部封装了char\*，管理这个字符串，是一个char\*型容器；

特点：string类内部封装了很多成员方法；

例如：查找find，拷贝copy，删除delete，替换replace，插入insert

String管理char\*所分配的内存，不用担心复制越界和取值越界等，由类内部进行负责；

**String 构造函数**：

构造函数原型：

String(); // 创建一个空的字符串，例如：string str;

String(const char\* s); // 使用字符串s初始化

String(const string& str); // 使用一个string对象初始化另一个string对象

String(int n, char c); // 使用n个字符c初始化；

总结：string的多种构造方式没有可比性，灵活使用即可；

**String赋值操作**

功能描述：给string字符串进行赋值

赋值的函数原型：

String& operator=(const char\* s) // char\*类型字符串 赋值给当前的字符串

String& operator=(const string& s) //把字符串s赋给当前的字符串

String& operator=(char c); //字符赋值给当前的字符串

String& assign(const char\* s) //把字符串s赋给当前的字符串

String& assign(const char\* s,int n) // 把字符串s的前n个字符赋给当前的字符串

String& assign(const string& s) //把字符串s赋给当前字符串

String& assign(int n,char c) //用n个字符c赋给当前字符串

总结：string的赋值方式很多，operator= 这种方式是比较实用的；

**String字符串拼接**

功能描述：实现在字符串末尾拼接字符串

函数原型：

String& operator+=(const char\* str) // 重载+=操作符

String& operator+=(const char c) //载+=操作符

String& operator+=(const string& str); //载+=操作符

String& append(const char\* s) //把字符串s连接到当前字符串结尾

String& append(const char\* s,int n) // 把字符串s的前n个字符连接到当前字符串结尾

String& append(const string& s) //同operator+=(const string& str)

String& append(const string& s,int pos,int n) //字符串s中pos位置开始的n个字符连接到字符串结尾；

总结：字符串拼接的重载版本很多，初学阶段记住几种即可；

**String查找和替换**

**功能描述：**

**查找：查找指定字符串是否存在**

**替换：在指定的位置替换字符串**

**函数原型：**

**Int find(const string& str,int pos =0) const;** //查找str第一次出现位置，从pos开始查找

**Int find(const char\* s,int pos =0) const;** //查找s第一次出现位置，从pos开始查找

**Int find(const char\* s,int pos,int n) const;** //从pos位置查找s的前n个字符第一次位置

**Int find(const char c,int pos=0) const;** //查找字符c第一次出现位置

**Int rfind(const string& str,int pos =npos) const;** //查找str最后一次位置，从pos开始查找

**Int rfind(const char\* s,int pos =npos) const;** // 查找s最后一次出现位置，从pos开始查找

**Int rfind(const char\* s,int pos,int n) const;** //从pos位置查找s的前n个字符最后一次位置

**Int rfind(const char c,int pos=0) const;** // 查找字符c最后一次出现位置

**String& replace(int pos,int n,const string& str);** //替换从pos开始n个字符为字符串str

**String& replace(int pos,int n,const char\* s);** //替换从pos开始的n个字符为字符串s

**总结：**find查找是从左往右，rfind从右往左

Find找到字符串后返回查找的第一个字符位置，找不到返回-1；

Replace在替换时，要指定从哪个位置起，多少个字符，替换成什么样的字符串

**String字符串比较**

**功能描述：**字符串之间的比较；

**比较方式：**字符串比较是按字符的ASCII码进行对比

**=** 返回 0

>返回 1

<返回 -1

函数原型：

Int compare(const string& s) const; // 与字符串s比较

Int compare(const char\* s) const; // 与字符串s比较

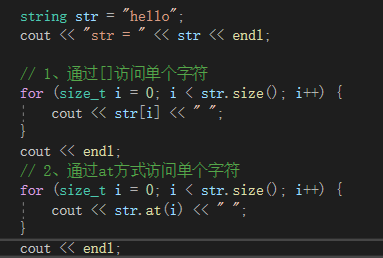
总结：字符串对比主要是用于比较两个字符串是否相等，判断谁大谁小的意义并不是很大；

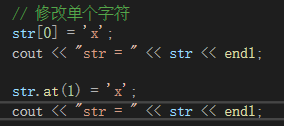
**String字符存取**

**String中单个字符存取方式有两种：**

**Char& operator[](int n);** // 通过[]方式取字符

**Char& at(int n);** // 通过at方法获取字符





**总结：string字符串中单个字符存取有两种方式：利用[] 或者 at;**

**String插入和删除**

**功能描述：对string字符串进行插入和删除字符操作**

**函数原型：**

**String& insert(int pos, const char\* s); // 插入字符串**

**String& insert(int pos,const string& str); // 插入字符串**

**String& insert(int pos,int n,char c); // 在指定位置插入n个字符c**

**String& erase(int pos,int n =npos); // 删除从pos开始的n个字符**

**总结：插入和删除的起始下标都是从0开始；**

**String子串**

**功能描述：从字符串中获取想要的子串**

**函数原型:**

**String substr(int pos =0,int n =npos) const; // 返回由pos开始的n个字符组成的字符串；**

总结：灵活的运用求子串的功能，可以在实际开发中获取有效的信息；

45、vector容器

1、vector基本概念

功能：vector数据结构和数组非常相似，也称为单端数组；

Vector与普通数组的区别：

不同之处在于数组是静态空间，而vector可以动态扩展

动态扩展：

并不是在原空间之后续接新空间，而是找更大的内存空间，然后将原数据拷贝新空间，释放原空间；

Vector容器的迭代器是支持随机访问的迭代器；

2、vector构造函数

功能描述：创建vector容器

函数原型：

Vector<T> v; // 采用模板实现类实现，默认构造函数

Vector(v.begin(),v.end()) // 将[v.begin(),v.end())区间中的元素拷贝给本身

Vector(n,elem); // 构造函数将n个elem拷贝给本身

Vector(const vector &vec) // 拷贝构造函数

总结：vector的多种构造方式没有可比性，灵活使用即可；

3、vector赋值操作

4、vector容量和大小

5、vector插入和删除

6、vector数据获取

7、vector互换容器

8、vector预留空间

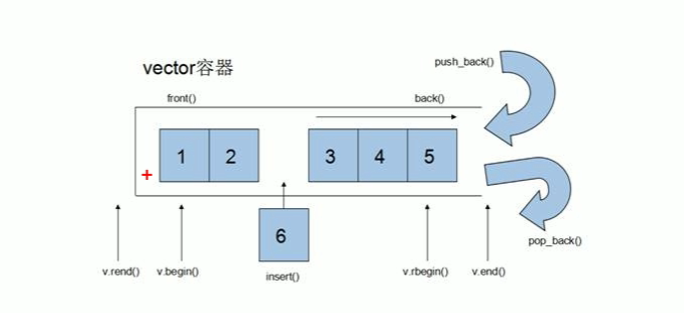
Vector容器

1. Vector基本概念

功能：vector数据结构和数组非常相似，也称为**单端数组**；

Vector与普通数组的区别：不同之处在于数组是静态空间，而vector可以**动态扩展**；

动态扩展：并不是在原空间之后续接新空间，而是找更大的内存空间，然后将原数据拷贝到新空间，释放原空间；



Vector容器的迭代器是支持随机访问的迭代器；

1. Vector构造函数

功能描述：创建vector容器

函数原型：

Vector<T> v; // 采用模板实现类实现，默认构造函数

Vector(v.begin(), v.end()); // 将v[begin(),end())区间中的元素拷贝给本身

Vector(n, elem); // 构造函数将n个elem拷贝给本身

Vector(const vector &vec); // 拷贝构造函数

总结：vector的多种构造方式没有可比性，灵活使用即可；

1. Vector赋值操作

功能描述：给vector容器进行赋值

函数原型：vector& operator=(const vector &vec); // 重载等号操作符

Assign(beg, end); // 将[beg, end)区间中的数据拷贝赋值给本身

Assign(n, elem); // 将n个elem拷贝赋值给本身

总结：vector赋值方式 比较简单，使用operator=,或者assign都可以；

1. Vector容量和大小

功能描述：对vector容器的容量和大小操作；

函数原型：

**Empty();** // 判断容器是否为空

**Capacity();** // 容器的容量

**Size();** // 返回容器中元素的个数

**Resize(int num);** // 重新指定容器的长度为num,若容器变长，则以默认值填充新位置；如果容器变短，则末尾超出容器长度的元素被删除；

**Resize(int num, elem);** // 重新指定容器的长度为num，若容器变长，则以elem值填充新位置，如果容器变短，则末尾超出容器长度的元素被删除；

总结：

判断是否为空---empty()

返回元素个数---size()

返回容器容量---capacity()

重新指定大小----resize();

1. Vector插入和删除

功能描述：对vector容器进行插入，删除操作；

函数原型：

Push\_back(ele); // 尾部插入元素ele

Pop\_back(); // 删除最后一个元素

Insert(const\_iterator pos,ele); // 迭代器指向位置pos插入元素

Insert(const\_iterator\_pos,int count,ele) // 迭代器指向位置pos插入count个元素ele;

Erase(const\_iterator pos) // 删除迭代器指向的元素；

Erase(const\_iterator start,const\_iterator end); // 删除迭代器从start到end之间的元素

Clear() ; // 删除容器中所有元素

总结：尾插 --- push\_back

尾删 --- pop\_back

插入 --- insert (位置迭代器)

删除 --- erase (位置迭代器)

清空 --- clear

1. Vector数据存取

功能描述：对vector中的数据的存取操作

函数原型：

At(int idx); // 返回索引idx所指的数据

Operator[]; // 返回索引idx所指的数据

Front(); // 返回容器中第一个数据元素

Back(); // 返回容器中最后一个数据元素

总结：除了用迭代器获取vector容器中的元素，[]和at也可以

Front返回容器第一个元素

Back返回容器最后一个元素

1. Vector互换容器

功能描述：实现两个容器内元素进行互换

函数原型：swap(vec) // 将vec与本身的元素互换

总结：swap可以使两个容器互换，可以达到实用的收缩内存的效果；

文本

描述已自动生成

1. Vector预留空间

功能描述：减少vector在动态扩展容量时的扩展次数

函数原型：reserve(int len); // 容器预留len个元素长度，预留位置不初始化，元素不可访问；

总结：如果数据量较大，可以一开始利用reserve预留空间；

Deque容器

1. deque容器基本概念

功能描述：双端数组，可以对头端进行插入删除操作

Deque与vector的区别：

Vector对于头部的插入删除效率低，数据量越大，效率越低

Deque相对而言，对头部的插入删除速度会比vector快

Vector访问元素时的速度会比deque快，这和两者内部实现有关

图示

描述已自动生成

Deque内部工作原理：

deque内部有个中控器，维护每段缓冲区中的内容，缓冲区中存放真实数据；中控器维护的是每个缓冲区的地址，使得使用deque时向一片连续的内存空间；

表格

描述已自动生成

Deque容器的迭代器也是支持随机访问的；

1. deque构造函数

功能描述：deque容器构造

函数原型：

Deque<T> deqT; // 默认构造形式

Deque(beg,end); // 构造函数将[beg,end)区间中的元素拷贝给本身

Deuqe(n,elem) // 构造函数将n个elem拷贝给本身

Deque(const deque &deq); // 拷贝构造函数

总结：deque容器和vector容器的构造方式几乎一致，灵活使用即可；

1. deque赋值操作

功能描述：给deque容器进行赋值

函数原型：

Deque& operator=(const deque& deq); // 重载等号操作符

Assign(beg, end); // 将[beg, end)区间中的数据拷贝赋值给本身

Assign(n, elem); // 将n个elem拷贝赋值给本身

总结：deque赋值操作也与vector赋值操作相同，需要熟练掌握；

1. deque大小操作

功能描述：对deque容器的大小进行操作

函数原型：

Deque.empty(); // 判断容器是否为空

Deque.size() ; // 返回容器中元素的个数

Deque.resize(num); // 重新指定容器长度为num,若容器变长，则以默认值填充新位置，如果容器变短，则末尾超出容器长度的元素被删除；

Deque.resize(num,elem); // 重新指定容器长度为num,若容器变长，则以elem填充新位置，如果容器变短，则末尾超出容器长度的元素被删除；

总结：

Deque没有容量的概念

判断是否为空 --- empty

返回元素个数 --- size

重新指定个数 --- resize

1. deque插入和删除

功能描述：向deque容器中插入和删除数据

函数原型：

两端插入操作：

Push\_back(elem) // 在容器尾部添加一个数据

Push\_front(elem) // 在容器头部插入一个数据

Pop\_back(elem) // 删除容器最后一个元素

Pop\_front(elem) // 删除容器第一个元素

指定位置操作：

Insert(pos, elem) // 在pos位置插入一个elem元素的拷贝，返回新数据的位置；

Insert(pos，n, elem) // 在pos位置插入n个elem数据，无返回值

Insert(pos，beg, end) // 在pos位置插入[beg,end)区间的数据，无返回值

Clear() // 清空容器的所有数据

Erase(beg, end) // 删除[beg, end)区间的数据，返回下一个数据的位置

Erase(pos) // 删除pos位置的数据，返回下一个数据的位置；

总结：

插入和删除提供的位置是迭代器！

尾插 --- push\_back

尾删 --- pop\_back

头插 --- push\_front

头删 --- pop\_front

1. deque数据存取

功能描述：对deque中的数据的存取操作

函数原型：

At(int idx); // 返回索引idx所指的数据

Operator[]; // 返回索引idx所指的数据

Front() ; // 返回容器中第一个数据元素

Back(); // 返回容器中最后一个数据元素

总结：除了用迭代器获取deque容器中元素，[]和at也可以

Front返回容器第一个元素

Back返回容器最后一个元素

1. deque排序

功能描述：利用算法实现对deque容器进行排序

函数原型：sort(iterator beg, iterator end); // 对beg和end区间内元素进行排序；

总结：sort算法非常实用，使用时包含头文件algorithm即可；

案例-评委打分

案例描述：有5名选手：选手ABCDE，10个评委分别对每一名选手打分，去除最高分，去除评委中的最低分，取平均分。

stack容器：

基本概念：

Stack是一种先进后出（First In Last Out,FILO）的数据结构，它只有一个出口

图示

描述已自动生成

栈中只有顶端的元素才可以被外界使用，因此栈不允许有遍历行为。

栈中进入数据称为 --- 入栈push

栈中弹出数据称为 --- 出栈pop

常用接口：

功能描述：栈容器常用的对外接口

构造函数：stack<T> stk; // stack采用模板类实现，stack对象的默认构造形式

Stack(const stack &stk) // 拷贝构造函数

赋值操作：

Stack& operator=(const stack &stk); // 重载等号操作符

数据存取：

Push(elem); // 向栈顶添加元素

Pop(); // 从栈顶移除第一个元素

Top() // 返回栈顶元素

大小操作：

Empty(); // 判断堆栈是否为空

Size(); // 返回栈的大小

总结：

入栈 --- push

出栈 --- pop

返回栈顶 --- top

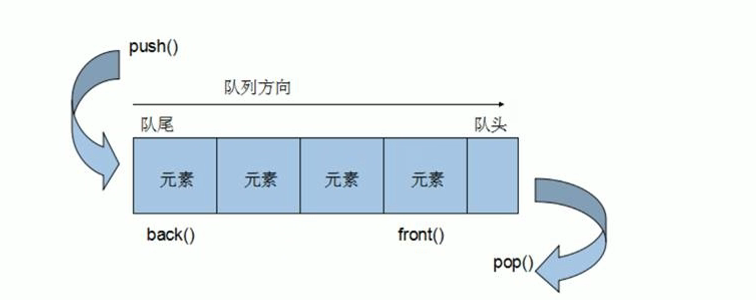
判断栈是否为空 --- empty

返回栈大小 --- size

queue容器：

基本概念：

Queue是一种先进先出（First In First Out，FIFO）的数据结构，它有两个出口；



队列容器允许从一端新增元素，从另一端移除元素；

队列中只有队头和队尾才可以被外界使用，因此队列不允许有遍历行为；

队列中进数据称为 … **入队** push

队列中出数据称为 … **出队** pop

常用接口：

功能描述：栈容器常用的对外接口

构造函数：

Queue<T> que; // queue采用模板类实现，queue对象的默认构造形式

Queue(const queue& que); // 拷贝构造函数

赋值操作：

Queue& operator=(const queue& que); // 重载等号操作符

数据存取：

Push(elem) // 往队尾添加元素

Pop(); // 从队头移除第一个元素

Back(); // 返回最后一个元素

Front(); // 返回第一个元素

大小操作：

Empty(); // 判断堆栈是否为空

Size(); // 返回栈的大小

总结：

入队 --- push

出队 --- pop

返回队头元素 --- front

返回队尾元素 --- back

判断队是否为空 --- empty

返回队列大小 --- size

list容器

基本概念：

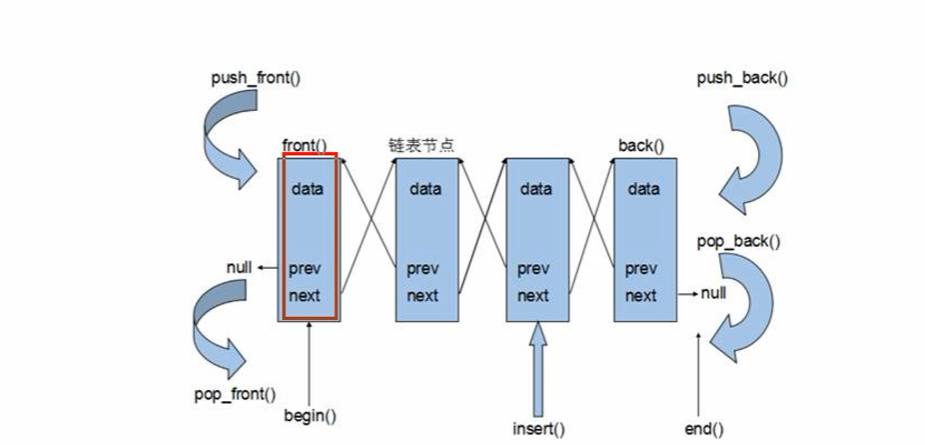
功能：将数据进行链式存储；

**链表**(list)是一种物理存储单元上非连续的存储结构，数据元素的逻辑顺序是通过链表中的指针链接实现的

链表的组成：链表由一系列**结点**组成；

结点的组成：一个是存储数据元素的**数据域**，另一个是存储下一个结点地址的**指针域**；

STL中的链表是一个双向循环链表；



由于链表的存储方式并不是连续的内存空间，因此链表list中的迭代器只支持前移和后移，属于**双向迭代器**；

List的优点：采用动态存储分配，不会造成内存浪费和溢出；

链表执行插入和删除操作十分方便，修改指针即可，不需要移动大量元素；

List的缺点：链表灵活，但是空间(指针域)和时间(遍历)额外耗费较大；

List有一个重要的性质，插入操作和删除操作都不会造成原有list迭代器的失效，这在vector是不成立的；

总结：STL中**List和Vector是两个最常被使用的容器**，各有优缺点；

构造函数：

功能描述：创建list容器

函数原型：

List<T> lst; // list采用模板类实现，对象的默认构造形式；

List(beg, end); // 构造函数将[beg, end)区间中的元素拷贝给本身

List(n, elem); // 构造函数将n个elem拷贝给本身

List(const list& lst); // 拷贝构造函数

总结：list构造方式同其他几个常用的stl容器，熟练掌握即可；

赋值和交换

功能描述：给list容器进行赋值，以及交换list容器

函数原型：assign(beg, end) // 将[beg,end)区间中的数据拷贝赋值给本身

Assign(n, elem) // 将n个elem拷贝赋值给本身

List& operator=(const list& lst) //重载等号操作符

Swap(lst) // 将lst与本身的元素互换

总结：list赋值和交换操作能够灵活运用即可；

大小操作：

功能描述：对list容器的大小进行操作

函数原型：size() // 返回容器中元素的个数

Empty() // 判断容器是否为空

Resize(num) // 重新指定容器的长度为num,若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除；

Resize(num, elem) // 重新指定容器的长度为num,若容器变长，则以elem填充新位置。如果容器变短，则末尾超出容器长度的元素被删除；

总结：

判断是否为空 --- empty

返回元素个数 --- size

重新指定个数 --- resize

插入和删除：

功能描述：对list容器进行数据的插入和删除

函数原型：

Push\_back(elem) // 在容器尾部加入一个元素

Pop\_back(); // 删除容器中最后一个元素

Push\_front(elem) // 在容器开头插入一个元素

Pop\_front(); // 从容器开头移除第一个元素

Insert(pos,elem) // 在pos位置插elem元素的拷贝，返回新数据的位置

Insert(pos,n,elem) // 在pos位置插入n个elem数据，无返回值

Insert(pos,beg,end) // 在pos位置插入[beg,end)区间的数据，无返回值

Clear() // 移除容器的所有数据

Erase(beg, end) //删除[beg,end)区间的数据，返回下一个数据的位置

Erase(pos) // 删除pos位置的数据，返回下一个数据的位置

Remove(elem) // 删除容器中所有与elem值匹配的元素

总结：

尾插 --- push\_back

尾删 --- pop\_back

头插 --- push\_front

头删 --- pop\_front

插入 --- insert

删除 --- erase

移除 --- remove

清空 --- clear

数据存取

功能描述：对list容器中数据进行存取

函数原型：front() // 返回第一个元素

Back() // 返回最后一个元素

总结：list容器中不可以通过[]或者at方式访问数据

返回第一个元素 --- front

返回最后一个元素 --- back

反转和排序

功能描述：将容器中的元素反转，以及将容器中的数据进行排序

函数原型：reverse(); // 反转链表

Sort(); // 链表排序

总结：反转 --- reverse

排序 --- sort(成员函数)

排序案例

总结：对于自定义数据类型，必须要指定排序规则，否则编译器不知道如何进行排序；

高级排序只是在排序规则上在进行一次逻辑规则制定，并不复杂；

set/multiset 容器

基本概念：

简介：所有元素都会在插入时自动被排序；

本质：set/multiset属于**关联式容器**，底层结构是用**二叉树**实现；

set和multiset区别：

set不允许容器中有重复的元素；

multiset允许容器中有重复的元素；

构造和赋值

功能：创建set容器以及赋值；

构造：

Set<T> st; // 默认构造函数

Set(const set& st); // 拷贝构造函数

赋值：

Set& operator=(const set& st); // 重载等号操作符

总结：set容器插入数据时用insert

Set容器插入数据的数据会自动排序

大小和交换

功能描述：统计set容器大小以及交换set容器

函数原型：size() // 返回容器中元素的数目

Empty() // 判断容器是否为空

Swap() // 交换两个集合容器

总结：

统计大小 --- size

判断是否为空 --- empty

交换容器 --- swap

插入和删除

功能描述：set容器进行插入数据和删除数据

函数原型：

Insert(elem); // 在容器中插入元素

Clear(); // 清楚所有元素

Erase(pos); // 删除pos迭代器所指的元素，返回下一个元素的迭代器

Erase(beg, end); // 删除区间[beg, end)的所有元素，返回下一个元素的迭代器

Erase(elem); // 删除容器中值为elem的元素；

总结：

插入 --- insert

删除 --- erase

清空 --- clear

查找和统计

功能描述：对set容器进行查找数据以及统计数据

函数原型：

find(key) // 查找key是否存在，返回该键的元素的迭代器；若不存在，返回set.end();

count(key): 统计key的元素个数

总结：查找 --- find （返回的是迭代器）

统计 --- count（对于set,结果为0或者1）

Set和multiset区别

学习目标：掌握set和multiset的区别

区别：set不可以插入重复数据，而multiset可以

Set插入数据的同时会返回插入结果，表示插入是否成功

Multiset不会检测数据，因此可以插入重复数据

总结：如果不允许插入重复的数据可以利用set

如果需要插入重复数据利用multiset

Pair使用-pair对组的创建

功能描述：成对出现的数据，利用对组可以返回两个数据

两种创建方式：

Pair<type, type> p(value1, value2);

Pair<type, type> p = make\_pair(value1, value2);

总结：两种方式都可以创建对组，记住一种即可；

内置类型指定排序规则

学习目标：set容器默认排序规则为从小到大，掌握如何改变排序规则

主要技术点：利用仿函数，可以改变排序规则；

总结：利用仿函数可以指定set容器的排序规则

自定义数据类型指定排序规则

总结：对于自定义数据类型，set必须指定排序规则才可以插入数据；

map/multimap容器

map概念：

简介：map中所有元素都是pair;

Pair中第一个元素为key(键值)，起到索引作用，第二个元素为value(实值)

所有元素都会根据元素的键值自动排序

本质：map/multimap属于**关联式容器**，底层结构是用二叉树实现。

优点：可以根据key值快速找到value值

Map和multimap区别：

Map不允许容器中有重复key值元素

Multimap允许容器中有重复key值元素

构造和赋值

功能描述：对map容器进行构造和赋值操作

函数原型：

构造：map<T1, T2> mp; // map默认构造函数

Map(const map& mp); // 拷贝构造函数

赋值；map& operator=(const map& mp); // 重载等号操作符

总结：map中所有元素都是成对出现，插入数据的时候要使用对组

大小和交换

功能描述：统计map容器大小以及交换map容器

函数原型：size(); // 返回容器中元素的数目

Empty(); // 判断容器是否为空

Swap(st); // 交换两个集合容器

总结：统计大小 --- size

判断是否为空 --- empty

交换容器 --- swap

插入和删除

功能描述：map容器进行插入数据和删除数据

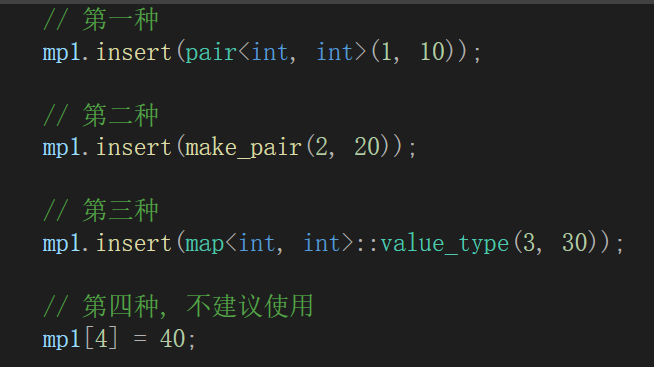
函数原型：insert(elem)；// 在容器中插入元素

Clear(); // 清楚所有元素

Erase(pos); //删除pos迭代器所指的元素，返回下一个元素的迭代器；

Erase(beg,end); // 删除区间[beg,end)的所有元素，返回下一个元素的迭代器；

Erase(key); // 删除容器中值为key的元素



总结：map插入方式很多，记住其一即可

插入 --- insert

删除 --- erase

清空 --- clear

查找和统计

功能描述：对map容器进行查找数据以及统计数据

函数原型：find(key) // 查找key是否存在，若存在，返回该键的元素的迭代器；若不存在，返回set.end();

Count(key) // 统计key的元素个数

总结：查找 --- find (返回的是迭代器)

统计 --- count (对于map,结果为0或者1)

排序

学习目标：map容器默认排序规则为 按照key值进行从小到大排序，掌握如何改变排序规则

主要技术点：利用仿函数，可以改变排序规则；

总结：利用仿函数可以指定map容器的排序规则

对于自定义数据类型，map必须要指定排序规则，同set容器；

STL案例2—员工分组



函数对象-函数对象基本使用

1. 函数对象概念：重载**函数调用操作符**的类，其对象常称为**函数对象**；

函数对象使用重载的（）时，行为类似函数调用，也叫**仿函数**；

本质：函数对象(仿函数)是一个类，不是一个函数

1. 函数对象使用

特点：函数对象在使用时，可以像普通函数那样调用，可以有参数，可以有返回值；

函数对象超出普通函数的概念，函数对象可以有自己的状态；

函数对象可以作为参数传递；

3、总结：仿函数写法非常灵活，可以作为参数进行传递；

谓词：

概念：返回bool类型的仿函数称为**谓词**；

如果operator()接受一个参数，那么叫做一元谓词；

如果operator()接受两个参数，那么叫做二元谓词；

谓词---一元谓词：参数只有一个的谓词，称为一元谓词；

谓词---二元谓词：参数只有两个的谓词，称为二元谓词；

内建函数对象：

STL内建了一些函数对象；

分类：算术仿函数

关系仿函数

逻辑仿函数

用法：这些仿函数所产生的对象，用法和一般函数完全相同；

使用内建函数对象，需要引入头文件#include <functional>

内建函数对象---算术仿函数

功能描述：实现四则运算

其中negate是一元运算，其他都是二元运算

仿函数原型：template <class T> T plus<T> // 加法仿函数

template <class T> T minus<T> // 减法仿函数

template <class T> T multiplies<T> // 乘法仿函数

template <class T> T divides<T> // 除法仿函数

template <class T> T modulus<T> // 取模仿函数

template <class T> T negate<T> // 取反仿函数

总结：使用内建函数对象时，需要引入头文件#include <functional>

内建函数对象---关系仿函数

功能描述：实现关系对比

函数原型：template <class T> bool equal\_to<T> // 等于

template <class T> bool not\_equal\_to<T> // 不等于

template <class T> bool greater<T> // 大于

template <class T> bool greater\_equal<T> // 大于等于

template <class T> bool less<T> // 小于

template <class T> bool less\_equal <T> // 小于等于

总结：关系仿函数中最常用的就是greater<> 大于

内建函数对象---逻辑仿函数

功能描述：实现逻辑运算

函数原型：template <class T> bool logical\_and<T> // 逻辑与

template <class T> bool logical\_or<T> // 逻辑或

template <class T> bool logical\_not<T> // 逻辑非

总结：逻辑仿函数实际应用较少，了解即可；

STL常用算法：

概述：算法主要是由头文件<algorithm> <functional> <numeric>组成；

<algorithm>是所有STL头文件中最大的一个，范围涉及到比较，交换，查找，遍历操作，复制，修改等等；

<functional> 定义了一些模板类，用以声明函数对象

<numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数；

常用遍历算法

---for\_each // 遍历容器

函数原型：for\_each(iterator beg, iterator end,\_func);

Beg:开始迭代器

End:结束迭代器

\_func:函数或者函数对象

总结：for\_each在实际开发中是最常用的遍历算法，需要熟练掌握；

---transform // 搬运容器到另一个容器中

功能描述：搬运容器到另一个容器中

函数原型：transform(iterator beg1,iterator end1, iterator beg2,\_func);

iterator beg1:源容器开始迭代器

iterator end1: 源容器结束迭代器

iterator beg2:目标容器开始迭代器

\_func:函数或者函数对象

总结：搬运的目标容器必须要提前开辟空间，否则无法正常搬运。

常用查找算法

---find // 查找元素

功能描述：查找指定元素，找到返回指定元素的迭代器，找不到返回结束迭代器end();

函数原型：find(iterator beg, iterator end, value);

按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置；

iterator beg：开始迭代器

iterator end：结束迭代器

value：查找的元素

总结：利用find可以在容器中找指定的元素，返回值是迭代器；

---find\_if // 按条件查找元素

功能描述：按条件查找元素

函数原型：find\_if(iterator beg, iterator end, \_Pred);

// 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置；

Beg:开始迭代器

End:结束迭代器

\_Pred函数或者谓词(返回bool类型的仿函数)

---adjacent\_find // 查找相邻重复元素

功能描述：查找相邻重复的元素

函数原型：adjacent\_find(iterator beg, iterator end);

// 查找**相邻重复**元素，返回相邻元素的第一个位置的迭代器；

// beg 开始迭代器

// end 结束迭代器

总结：面试题中如果出现查找相邻重复元素，记得使用STL中adjacent\_find算法；

---binary\_search // 二分查找法

功能描述：查找指定元素是否存在

函数原型：bool binary\_search(iterator beg, iterator end, value);

//查找指定的元素，查找 返回true, 否则返回false;

//注意：在**无序序列**中不可用

//beg 开始迭代器

//end 结束迭代器

// value 查找的元素

总结：二分查找法查找效率很高，值得注意的是查找的容器中元素必须是有序序列；

---count // 统计元素个数

功能描述：统计元素个数

函数原型：count(iterator beg, iterator end, value);

// 统计元素出现次数

//iterator beg：开始迭代器

//iterator end：结束迭代器

//value：统计的元素

总结：统计自定义数据类型时候，需要配合重载operator==

--- count \_if // 按条件统计元素个数

功能描述：按条件统计元素个数

函数原型：count\_if(iterator beg, iterator end, \_Pred)

//按条件统计元素出现次数

// beg 开始迭代器

// end 结束迭代器

// \_Pred谓词

常用排序算法

学习目标：掌握常用的排序算法

算法简介：sort // 对容器内元素进行排序

random\_shuffle // 洗牌，指定范围内的元素随机调整次数；

merge: // 容器元素合并，并存储到另一容器中；

reverse: // 反转指定范围的元素；

sort:

功能描述：对容器内元素进行排序；

函数原型：sort(iterator beg, iterator end, \_Pred);

// 按值查找元素，找到返回指定位置的迭代器，找不到返回结束迭代器位置；

// beg 开始迭代器

// end 结束迭代器

// \_Pred 谓词

总结：sort属于开发中最常用的算法之一，需熟练掌握；

random\_shuffle:

功能描述：指定范围内的元素随机调整次序；

函数原型：random\_shuffle(iterator beg, iterator end);

// 指定范围内的元素随机调整次序；

// beg 开始迭代器

// end 结束迭代器

总结：random\_shuffle 洗牌算法比较实用，使用时记得加随机数种子；

merge:

功能描述：两个容器元素合并，并存储到另一容器中；

函数原型：merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);

// 容器元素合并，并存储到另一容器中

// 注意：两个容器必须是**有序的**

// 指定范围内的元素随机调整次序；

// beg1 容器1开始迭代器

// end1 容器1结束迭代器

// beg2 容器2开始迭代器

// end2 容器2结束迭代器

// dest 目标容器开始迭代器

总结：merge合并的两个容器必须是有序序列；

reverse:

功能描述：将容器内元素进行比较；

函数原型：reverse(iterator beg, iterator end);

// 反转指定范围的元素；

// beg 开始迭代器

// end 结束迭代器

总结：reverse反转区间内的元素，面试题可能涉及到；

常用拷贝和替换算法

学习目标：掌握常用的拷贝和替换算法

算法简介：copy // 容器内指定范围的元素拷贝到另一容器中

Replace // 将容器内指定范围的旧元素修改为新元素

Replace\_if // 容器内指定范围满足条件的元素替换为新元素

Swap // 互换两个容器的元素

copy ：

功能描述：容器内指定范围的元素拷贝到另一容器中

函数原型：copy(iterator beg, iterator end, iterator dest);

// 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置；

// beg 开始迭代器

// end 结束迭代器

// dest 目标起始迭代器

总结：利用copy算法在拷贝时，目标容器记得提前开辟空间；

Replace：

功能描述：将容器内指定范围的旧元素修改为新元素

函数原型：replace(iterator beg, iterator end, oldvalue, newvalue);

// 将区间内旧元素 替换成 新元素；

// beg 开始迭代器

// end 结束迭代器

// oldvalue 旧元素

// newvalue 新元素

总结：replace会替换区间内满足条件的元素；

Replace\_if：

功能描述：将区间内满足条件的元素，替换为指定元素

函数原型：replace\_if(iterator beg, iterator end, \_pred, newvalue);

// 按条件替换元素，满足条件的替换成 指定元素；

// beg 开始迭代器

// end 结束迭代器

// \_pred 谓词

// newvalue 替换的新元素

总结：replace\_if按条件查找，可以利用仿函数灵活筛选满足的条件；

Swap：

功能描述：互换两个容器的元素

函数原型：swap(container c1, container c2);

// 互换两个容器的元素

// c1 容器1

// c2 容器2

总结：swap交换容器时，注意交换的容器要同种类型；

常用算术生成算法

学习目标：掌握常用的算术生成算法

注意：算术生成算法属于小型算法，使用时包含的头文件为#include <numeric>

算法简介：accumulate // 计算容器元素累计总和

Fill // 向容器中添加元素

Accumulate：

功能描述：计算区间内 容器元素累计总和

函数原型：accumulate(iterator beg, iterator end, value);

// 计算容器元素累计总和

//beg开始迭代器

//end结束迭代器

//value起始值

总结：accumulate使用时头文件注意是numeric，这个算法很实用；

Fill   
功能描述：向容器中填充指定的元素

函数原型：fill(iterator beg, iterator end, value);

// 向容器中填充元素

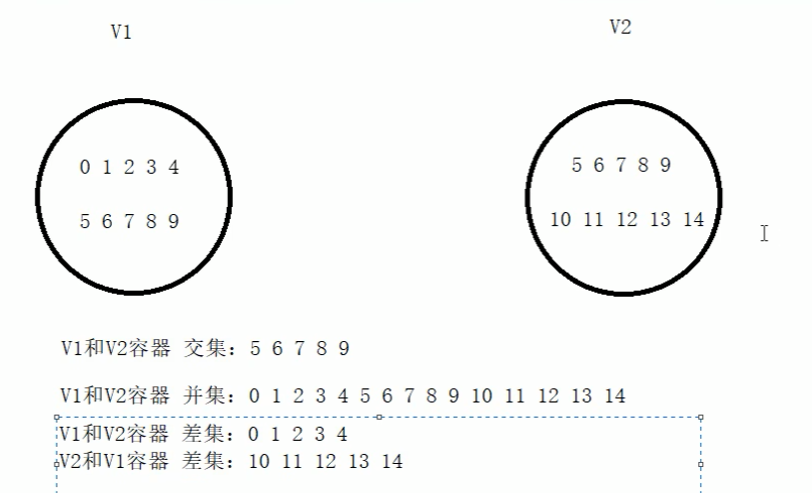
//beg 开始迭代器

//end 结束迭代器

//value 填充的值

总结：利用fill可以将容器区间内元素填充为 指定的值；

常用集合算法



学习目标：掌握常用的集合算法

算法简介：set\_intersection // 求两个容器的交集

Set\_union // 求两个容器的并集

Set\_difference // 求两个集合的差集

set\_intersection：

功能描述：求两个容器的交集

函数交集：set\_intersection(interator beg1, iterator end1, iterator beg2, iterator end2,iterator dest);

// 求两个集合的交集

// 注意：两个集合必须是有序序列

Beg1:容器1开始迭代器

end1:容器1结束迭代器

Beg2:容器2开始迭代器

End2:容器2结束迭代器

dest:目标容器开始迭代器

总结：求交集的两个集合必须是有序序列；

目标容器开辟空间需要从**两个容器中取最小值**；

Set\_intersection返回值即是交集中最后一个元素的位置；

Set\_union // 求两个容器的并集

功能描述：求两个集合的并集

函数原型：set\_union(interator beg1, iterator end1, iterator beg2, iterator end2,iterator dest);

// 求两个集合的并集

// 注意：**两个集合必须是有序序列**

beg1:容器1开始迭代器

end1:容器1结束迭代器

beg2:容器2开始迭代器

end2:容器2结束迭代器

dest:目标容器开始迭代器

总结：求并集的两个集合必须是有序序列；

目标容器开辟空间需要**两个容器相加**；

Set\_union返回值即是并集中最后一个元素的位置；

Set\_difference // 求两个集合的差集

功能描述：求两个集合的差集

函数原型：set\_difference(interator beg1, iterator end1, iterator beg2, iterator end2,iterator dest);

// 求两个集合的差集

// 注意：**两个集合必须是有序序列**

beg1:容器1开始迭代器

end1:容器1结束迭代器

beg2:容器2开始迭代器

end2:容器2结束迭代器

dest:目标容器开始迭代器

总结：求差集的两个集合必须是有序序列；

目标容器开辟空间需要从**两个容器取较大值**；

Set\_difference返回值即是差集中最后一个元素的位置；