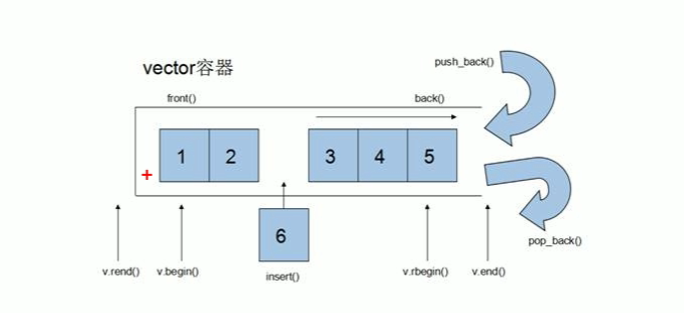
Vector容器

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

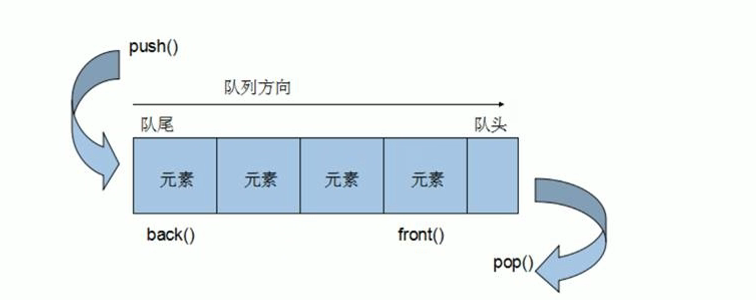
判断栈是否为空 --- empty

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：

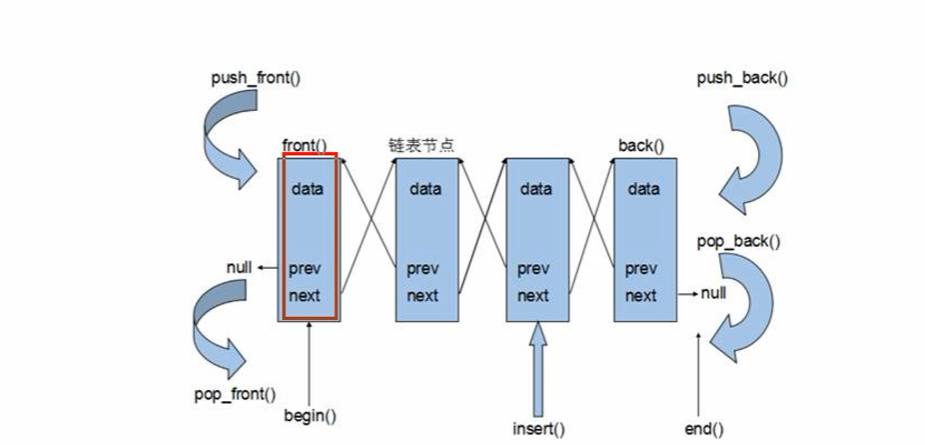
功能：将数据进行链式存储；

**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

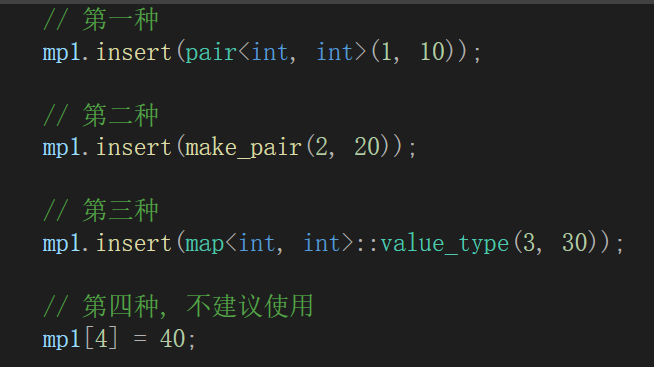
函数原型：insert(elem)；// 在容器中插入元素

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

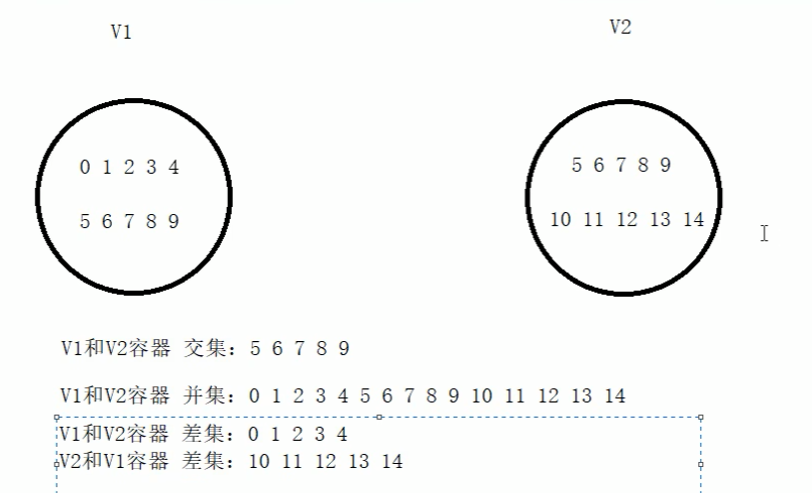
//beg 开始迭代器

//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；