C++

* STL
  + 容器、迭代器、算法
    - 实现对多个类型的数据结构和相关操作
    - 序列容器
      * 序列容器未排序
      * vector
        + 底层数据结构为数组

支持快速随机访问

可以使用== ，!= 比较是否相等

vector底层机制

三个迭代器分别指向vector容器的三个位置

\_Myfirst，起始字节地址

\_Mylast，当前最后一个元素的末尾字节

\_Myend，整个vector容器所占用内存空间的末尾字节

当size==capacity时，需要扩容

重新申请更大的内存空间vector 容器扩容时，不同的编译器申请更多内存空间的量是不同的。以 VS 为例，它会扩容现有容器容量的 50%。

跟操作系统有关

win10+vs ，1.5倍

Linux+G++，2倍

将原有的数据拷贝进来

将旧的内存空间释放

指向原来的指针、引用和迭代器失效

插入操作导致迭代器失效？

插入使得size == capacity

容器需要扩容

旧的内存空间被释放

emplace\_back()和push\_back()

具有push\_back()和emplace\_back()函数功能的容器：deque、list、vector

底层实现机制不同

emplace\_back()

原地构造，省去了拷贝或移动元素的过程，效率更高

push\_back()

先创建元素，然后使用拷贝构造或者移动构造函数在容器末插入元素，最后再销毁之前的元素

只有vector和string有capacity属性

clear()函数的实现机制以及vector堆内存的释放方法

clear只是清空元素，size=0，capacity并不变

什么时候用vector和list

数组和双向链表的区别

vector支持快速随机访问，适合数据量小，对象数据变化不大

list支持快速增删操作，适合对象数据变化频繁，数据量大，增删频繁

数组是连续的内存空间，链表不是连续的

遍历元素的时候，vector更快，因为它是连续内存

扩容方式

vector插入新元素时，如果未超过当前容量，直接在末尾插入元素，如果没有剩余空间，将动态扩容

list每插入一个元素都会分配空间，每删除一个元素都会释放空间，空间利用率高

* + - * deque
        + 底层数据结构是一个中央控制器和多个缓冲区

支持首位快速增删，支持随机访问

* + - * + 双向队列，头尾两端分别做元素的插入删除
        + deque底层实现原理

与vector不同，deque的容器存储空间是由一段一段等长的连续存储空间构成各段空间之间并不一定是连续的，可以位于内存的不同区域

为了管理这些分段连续空间，deque容器用map数组存储各个连续空间的首地址，存的也就是指针

迭代器内部包含四个迭代器

cur指向当前正在遍历的元素

first指向当前连续空间的首地址

last指向当前连续地址的末尾地址

node是一个二级指针，用于指向map中控器数组中存储当前连续地址的首地址

* + - * list
        + 底层数据结构为双向链表

支持快速增删

* + - * stack
        + 底层一般是用list和deque实现，封闭头部即可，不同vector的原因是容量大小有限制，扩容耗时
      * queue
        + 和stack一样，stack和queue其实是适配器，不叫容器，因为是对容器的再封装
      * priority\_queue
        + 底层数据结构一般是vector为底层的容器，堆heap为处理规则来管理底层容器实现

包含头文件#include <queue>

定义：priority\_queue<Type, Container, Functional>Type 就是数据类型，Container 就是容器类型,Functional 就是比较的方式

//升序队列priority\_queue <int,vector<int>,greater<int> > q;//降序队列priority\_queue <int,vector<int>,less<int> >q;

* + - 关联容器
      * 特点
        + 将键和值关联起来，使用键来查找值
        + 可对元素进行快速访问
        + 允许插入元素，但不能指定位置
        + map和set的区别在于键和值是否相同，set中将值作为键，支持交并补运算
        + multimap和multiset就在map和set的基础上，使他们的键可以重复，除此之外基本等同。
      * 红黑树
        + set

有序，不重复

不允许迭代器修改元素的值，constance\_iterators

键等于值，保存的是两份元素

* + - * + multiset

有序，可以重复

* + - * + map

有序，键不重复

与set的区别

map是键值对，键起到索引的作用set的键就是值，值就是键

set的迭代器是const的，不允许修改元素的值（键值相同）map允许修改值，但不允许修改键

map和set是根据键来排序，确保有序性的如果修改了键的话，就需要先删除键，再调节平衡，再插入键，再调节平衡。破坏了map和set的结构，导致迭代器失效。

map支持下标操作，set不支持下标操作

自定义排序方式

* + - * + multimap

有序，键可以重复

* + - * 哈希表
        + unordered\_map

无序，不重复

哈希冲突解决办法

线性探测法

开地址法

STL采用

每个表格维护一个list

bucket数量选择

分支主题

再散列法

二次探测法

公共溢出区

* + - * + unordered\_multimap

无序，可重复

* + - * + unordered\_set

无序，不重复

* + - * + unordered\_multiset

无序，可重复

* + - 算法
      * sort（iterator begin() ,iterator end() ）// 排序默认増序
      * reverse(iterator begin(), iterator end() ) //反转
      * iterator find (const value\_type& val) const; //查找，set和map中可以使用此函数，找不到则返回end()
      * replace(iterator begin(), iterator end(), oldValue,newValue)//替换元素
      * swap(container c1,container c2) // 交换两个容器的元素
    - 组件
    - 迭代器
      * 作用
        + 把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部结构而到达循环遍历的效果
        + 不是指针，是类模板，表现得像指针，重载了指针的一些操作符，是一个可遍历STL容器内全部或部分元素的对象
        + 用于提供访问容器中对象的方法，而又不需要暴露该对象的内部表示，相当于智能指针，容器提供迭代器，算法使用迭代器
      * 分类
        + 输入输出迭代器

重载自加运算符++，支持对象序列的读写，只能使用一次

* + - * + 前向迭代器

用来遍历容器，支持输入输出迭代器的功能，还能进行对象的访问存储，可以重用

例如queue中的迭代器就是前向迭代器

* + - * + 双向迭代器

重载运算符--，使得它能够反向遍历容器，并包含前向迭代器的功能

例如list包含的是双向迭代器

* + - * + 随机迭代器

重载+，-，!= ,==,运算符，可以对容器内任意数据进行访问，包含了以上所有迭代器的功能

例如vector包含的迭代器就是随机迭代器

* + - * 不能使用迭代器的容器
        + stack
        + queue
        + priority\_queue
      * 迭代器失效问题
        + 数组形vector、deque

扩容后原有的迭代器失效

erase操作，删除元素

删除当前的iterator会使后面所有元素的iterator都失效。这是因为顺序容器内存是连续分配（分配一个数组作为内存），删除一个元素导致后面所有的元素会向前移动一个位置

erase方法可以返回下一个有效的iterator

insert和erase操作，都会使得删除点和插入点之后的元素挪位置，所以，插入点和删除掉之后的迭代器全部失效

* + - * + 链表形list

使用了不连续分配的内存，删除运算使指向删除位置的迭代器失效，但是不会失效其他迭代器

erase操作返回下一个有效的迭代器

* + - * + 树形map、set

底层是红黑树，插入、删除一个结点不会对其他结点造成影响

单个节点在内存中的地址没有变化，变化的是各节点之间的指向关系

erase迭代器只是被删元素的迭代器失效，但是返回值为void，所以要采用erase(iter++)的方式删除迭代器

erase前记录下一个元素的迭代器即可

插入不会使得任何迭代器失效

* + - 分配器allocator
      * 用于封装STL容器在内存管理上的底层细节
        + 其内存的配置和释放

new分两个阶段

调用operator new配置内存

调用对象构造函数初始化对象

delete分两个阶段

调用对象的析构函数

调用operator delete 释放内存

* + - * + 为了精密分工，STL allocator将两个阶段操作区分开来

内存配置有alloc::allocate()负责

内存释放由alloc::deallocate()负责

对象析构由destroy()负责

对象构造由construct()负责

* + - * + 为了提升内存管理的效率，减少申请小内存造成的内存碎片问题

两级配置器

分配的空间大于128B时，会使用第一级空间配置器

直接使用malloc（）,realloc（）、free()函数

小于128B时，将使用第二级空间配置器

内存池技术，通过空闲链表来管理内存

* + - 泛型编程
    - 类型萃取
    - vector/string
      * reserve
        + 改变当前容器的容量（capacity）

不会修改元素，reserve(n)

n大于原来的capacity，重新分配一块大小为n的内存空间，然后再拷贝，销毁之前的内存

n小于元素个数size，使得capacity=size，不会删除元素

* + - * resize
        + 改变容器的元素个数（size）

会修改元素，resize(n)

n大于原来的size，会增加n-size个元素，默认为0

n小size，使得元素减少为n，删除了size-n个元素，capacity不变

* + - * clear
        + 清空元素，使size=0，但capacity不变
      * size/length
        + 底层相同