

STL

xbZhong

2024-05-10

[本页PDF](#)

迭代器(可以看作指针)

- C++里面可以用auto自动识别迭代器类型
- 不用auto,则需要写完整代码。例如: `vector< int > :: iterator it`
- `for(auto x: nums)`可以利用x来遍历nums, 但不能修改值(其不是迭代器, 因此输出时直接cout << x即可)
 - x是深拷贝的一份数据
- `for(auto &x:nums)`可以利用x来修改容器nums的值
 - 本质上x是指针
- 迭代器可分为迭代器和常量迭代器(常量迭代器适用于模板为const类型的, 即值不可修改)
- 还可分为正向迭代器, 反向迭代器, 双向迭代器, 随机访问迭代器
- 随机访问迭代器只有vector, deque, string有, 他们对+=, ,<=,>=进行了重载
- 支持双向迭代器的有set, map, list, multiset, multimap

string

初始化

- `string();` 初始化一个空字符串 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
- `string(int n, char c);` 使用n个字符初始化 ##### 赋值 `string str1,str2; ##### 直接赋值`
- `str1 = "hello";`
- `str2 = str1; ##### 使用成员函数`
- `str1.assign("hello");`
- `str1.assign("hello",3);` 把hello前三个字符赋值给str1 ##### 字符串拼接
- `str1 += "大帅哥";` 通过对+重载实现字符串拼接
- `str1.append("大帅哥");` 使用成员函数进行拼接
- `str2.append(str1,int pos,int n);` 从str中第pos个字符开始截取n个字符拼接到str2末尾 ##### 查找和替换
- `s.find(string str,int pos);` 从s里的pos开始查找, 未找到返回-1, 找到则返回下标值
- `s.rfind();` 从右向左查找
- `s.replace(int pos,int n,string str);` 将s中从pos开始的n个字符替换成str ##### 字符串比较
- 根据字符的ASCII码进行比较
 - = 返回0
 - > 返回1
 - < 返回-1
- `s.compare(str)` s与str比较 ##### string字符存取
- `str[1]` 类似数组的方式来获取字符
- `s.at(i)` 也可获取字符串 ##### 字符串插入和删除
- `s.insert(pos,str);` 从s中pos位置开始插入str
- `s.erase(pos,n);` 从s中pos位置开始删除n个字符 ##### 获取子串
- `s.substr(pos,n);` 从s中pos位置开始读取n个字符并返回, 返回类型为string ##### 常见接口 > s.size(); > s.copy(); > s.length(); > s.assign(); > s.append(); > s.find(); > s.rfind(); > s.replace(); > s.compare(); > s.at(); > s.insert(); > s.erase(); ## vector(单端数组, 支持动态扩展) 动态扩展: 找更大的内存空间, 拷贝数据, 释放原空间 需要注意的是: 动态扩展后内存地址发生变化, 原有的迭代器会失效 vector的迭代器是支持随机访问的迭代器 常用迭代器: v.begin(),v.end(),v.rbegin(),v.rend().后两个为反向迭代器。.end()和.rend()是指向元素的下一个地址 ##### 初始化
- `vector< int > v;`
- `vector(v.begin(),v.end());` 将v[begin(),end())区间中的元素拷贝。左闭右开 ##### 赋值
- 通过重载=运算符直接对vector直接赋值
- `.assign()` 进行赋值
- 可以用{}对vector进行赋值 ##### 容量和大小
- `.empty();` 非空返回false, 空返回true
- `.capacity()` 返回值>= .size() 返回值
- `.resize(int num,(int elem));` 重新指定容器长度为num, 容器变短, 超出长度的元素被删除, 容器变长, 默认值填充(用elem值填充) ##### 插入和删除
- `.push_back();` 尾插
- `.pop_back();` 尾删
- `.insert(pos,ele);` 迭代器指向位置pos插入ele(pos为迭代器)
- `.erase(pos);` 删除迭代器指向的元素(pos为迭代器) 也支持删除区间元素
- `.clear();` 删除容器中所有元素 ##### 数据读取
- 支持下标访问 例如: `arr[1];`
- `.front();` 返回头部元素;
- `.back();` 返回尾部元素; ##### 互换容器
- 功能: 实现两个容器元素互换
- `.swap(vec);`
- 可以使用swap收缩内存, 防止内存造成浪费 `vector< int >(v).swap(v)` vector< int > (v) 为匿名对象 ##### 预留空间

• .reserve(int len); 预留len个元素的长度，元素不可访问，不初始化(需push_back) 可以减少容器动态扩展的次数 ##### 常见接口 > .begin(),.end(),rbegin(),rend() > .assign(); > .empty(); > .capacity(); > .size(); .resize(); > .push_back(); > .pop_back(); > .insert(); > .erase(); > .front(); > .back(); > .reserve();

deque(双端数组，支持头插，删，尾插，删)

迭代器支持随机访问 其方法和vector差不多

stack(先进后出)

栈不允许遍历，只有栈顶才能被外界访问 ##### 常见接口 > .push(); > .pop(); > .top(); > .size(); > .empty();

queue(先进先出)

队列不允许遍历，只有队头和队尾能被外界访问 队头出，队尾进 ##### 常见接口 > .push(); > .pop(); > .size(); > .empty(); > .front(); > .back();

list(双向循环链表)

链表的存储方式并不是连续的内存空间，list中的迭代器只支持前移和后移，是双向迭代器(不支持[]操作) ##### 赋值和交换 * .assign(); 不支持单个填入，若有需要则.assign(1,10) * .swap(); * 重载=运算符直接赋值 ##### 插入和删除 * .insert(pos,elem); 在pos(迭代器)位置插入elem值、 * .erase(pos); pos为迭代器 * .remove(elem); 删除容器中所有与elem值相同的元素 ##### 反转和排序 * .reverse(); 反转链表； * .sort(); 直接l1.sort(); 支持自定义排序方式 所有不支持随机访问迭代器的容器，不可以使用标准算法 ##### 常见接口 > .assign(); > .swap(); > .push_front(); > .pop_front(); > .push_back(); > .pop_back(); > .empty(); > .size(); .resize(); > .insert(); > .erase(); > .clear(); > .remove(); > .reverse(); > .sort();

set/multiset

- 关联式容器(插入时自动排序)
- 底层结构用二叉树实现 set和multiset区别：
- set不允许有重复元素
- multiset允许有重复元素 ##### 赋值
- 重载=运算符进行赋值 ##### 插入和删除 只有.insert()
- .erase();
 - .erase(pos)可用迭代器删除
 - .erase(elem)可删除指定元素 ##### 查找和统计
- .find(key); 查找key是否存在，存在则返回该元素的迭代器，不存在则返回.end();
- .count(key); 统计元素key的个数 ##### 大小和交换
- 不支持resize ##### pair
- set也可以存储pair类型的数据。set< pair < int , int > >;
- 按照first值大小进行排序 ##### 创建方式
- pair< type , type > p (value1,value2);
- pair< type , type > p = make_pair(value1,value2); ##### 访问
- .first(); 访问pair中第一个元素

- .second(); 访问pair中第二个元素 ##### 仿函数 利用仿函数，可以改变排序规则

```

class mycompare
{
public:
    bool operator() (int v1, int v2)
    {
        return v2 > v1;
    }
};

int main()
{
    set<int, mycompare> s;
    s.insert(20);
    s.insert(10);
    s.insert(50);
    for(auto x = s.begin(); x != s.end(); x++)
        cout << *x << " ";
}

```

> 自定义排序时，要通过创建类来重载运算符，从而更改排序规则 > 不可以通过创建函数来自定义排序规则，因为set里面要存储的是数据类型而不是函数 ##### 常见接口 > .insert();
> .size(); > .empty(); > .swap(); > .clear(); > .erase(); > .find(); > .count();

map/multimap

- map中所有元素都是pair
- pair中第一个元素为key(键值)，起索引作用，第二个元素为value(实值)
- 可用key快速找到value
- map不允许有重复key值元素(value可以重复)
- multimap允许有重复key值
- map按照key进行排序 ##### 初始化
- map< int , int > m; ##### 赋值
- 重载=运算符
- m.insert(pair< int , int >)(key,value); ##### 插入和删除
- .insert();
 - m.insert(pair< int , int >)(key,value);
 - m.insert(make_pair(key,value));
 - m[key] = 20; 若key不存在会自动创建 可以用m[key]访问对应的value值.会覆盖原有数据
- .erase();
 - .erase(key);按照key值删除对应的value
 - .erase(pos);按照迭代器位置删除value
- .erase(); ##### 查找和统计
- .find();返回的是迭代器 ##### 排序操作
- 类似set的操作 ##### 访问操作 可以把map当成一个结构体指针
- s.begin()->first(second);
- (*s.begin()).first(second);
- 通过key访问。m[key]; ##### 常用接口 > .size(); > .empty(); > .swap(); > .clear(); > .insert(); > .erase(); > .find(); > .count();

函数对象(类)

- 重载函数调用操作符的类，其对象也称为函数对象

- 函数对象使用重载的()时, 行为类似函数调用, 也叫仿函数

谓词

- 返回类型为bool类型的仿函数为谓词
- 一元谓词(operator()接收一个参数)
- 二元谓词(operator()接收两个参数)
- 可以用谓词来自定义排序规则

内建函数

要使用**functional**头文件 * 算术仿函数 * 关系仿函数 * 逻辑仿函数

算术仿函数

- plus< T >;加法仿函数
- minus< T >;减法仿函数
- multiplies< T >;乘法仿函数
- divides< T >;除法仿函数
- modulus< T >;取模仿函数
- negate< T >;取反仿函数 只有**negate**是一元运算, 其它都是二元运算 > 例子; plus< int > v; //本质上是类 > v(50,10); //把其当作函数来调用

关系仿函数

- bool equal_to< T >;等于
- bool not_equal_to< T >;不等于
- bool greater< T >;大于
- bool greater_equal< T >;大于等于
- bool less< T >;小于
- bool less_equal< T >;小于等于 可以在**sort**里面直接使用

逻辑仿函数

- bool logical_and< T >;逻辑与
- bool logical_or< T >;逻辑或
- bool logical_not< T >;逻辑非

STL常见算法

主要由< **algorithm** > < **functional** >< **numeric** >组成 ##### 常用遍历算法 * **for_each** 遍历容器 * **for_each(beg,end,_func)**; * beg:开始迭代器 end:结束迭代器 _func:函数或者函数对象(遍历时你要进行的操作, 如打印等等) * **_func**是函数则不加括号, 是仿函数则加括号 * **transform** 搬运容器到另一个容器中 * **transform(beg1,end1,beg2,_func)** * beg1:源容器开始迭代器 end1:源容器结束迭代器 beg2:目标容器开始迭代器 _func:函数或者函数对象(一般是对元素作逻辑运算) * 在**transform**时要提前对目标容器开辟空间

常用查找算法

- **find**; 查找元素
 - **find(beg,end,value)**
 - beg: 开始迭代器 end:结束迭代器 value:查找的元素
 - 返回目标元素的迭代器,找不到则返回end
 - 对于自定义数据类型有时还需要重载运算符
- **find_if**; 按条件查找元素
 - **find_if(beg,end,_prec)**
 - beg: 开始迭代器 end:结束迭代器 _prec:谓词(仿函数,返回值为bool类型)
- **adjacent_find**; 查找相邻重复元素
 - **adjacent_find(beg,end)**
 - 会返回相邻元素的第一个位置的迭代器
 - beg:起始迭代器 end:结束迭代器
- **binary_search**; 二分查找法
 - 返回类型为bool类型
 - **binary_search(beg,end,value)**
 - beg:起始迭代器 end:结束迭代器
 - 容器内元素一定要是有序序列
- **count**; 统计元素个数
 - **count(beg,end,value)**
 - beg: 开始迭代器 end:结束迭代器 value:查找的元素
 - 统计自定义数据类型, 重载运算符时对参数加const
- **count_if**; 按条件统计元素个数
 - **count_if(beg,end,_prec)**
 - beg: 开始迭代器 end:结束迭代器 _prec:谓词(仿函数,返回值为bool类型) ##### 常用排序算法
- **sort**; 对容器元素进行排序
 - 可以传仿函数, 函数
- **random_shuffle**; 指定范围内的元素随机调整次序
 - **random_shuffle (lebeg,end,value)**
 - beg: 开始迭代器 end:结束迭代器

- merge; 两个容器元素合并，存储到另一容器中
 - merge(beg1,end1,beg2,end2,dest)
 - 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
 - 两个容器必须是有序的，合并之后还是有序的
- reverse; 反转范围内的指定元素
 - reverse(beg,end)
 - beg: 开始迭代器 end:结束迭代器

常用拷贝和替换算法

- copy; 拷贝元素到另一容器
 - merge(beg,end,dest)
 - beg: 开始迭代器 end:结束迭代器 dest:目标起始迭代器
- replace; 将容器内指定范围的旧元素改成新元素
 - replace(beg,end,oldvalue,newvalue)
 - beg: 开始迭代器 end:结束迭代器 oldvalue: 旧元素 newvalue: 新元素
- replace_if;
 - replace_if(beg,end,_pred,newvalue)
 - beg: 开始迭代器 end:结束迭代器 _pred: 谓词 newvalue: 新元素
- swap;
 - swap(c1,c2)
 - c1: 容器1 c2: 容器2 同种类型的容器才可实现swap

常用算术生成算法

- 头文件为 < numeric >
- accmulate; 计算容器元素累计总和
 - accmulate(beg,end,value)
 - beg: 开始迭代器 end:结束迭代器 value: 起始值
- fill; 向容器中添加元素
 - fill(beg,end,value)
 - beg: 开始迭代器 end:结束迭代器 value: 填充值

常用集合算法

两个集合必须是有序序列 * set_intersection; 求两个容器的交集 * set_intersection(beg1,end1,beg2,end2,dest) * 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 * 目标容器需要提前开辟空间(取较小容器的size) * 最好用一个迭代器接收函数返回值, 否则输出时会输出默认值 * set_union; 求两个容器的并集 * set_union(beg1,end1,beg2,end2,dest) * 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 * 要提前开辟空间(两个容器size相加) * 最好用一个迭代器接收函数返回值(返回值是最后一个元素的位置), 否则输出时会输出默认值 * set_difference; 求两个容器的差集 * set_difference(beg1,end1,beg2,end2,dest) * 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 * 要提前开辟空间(取较大容器的size) * 最好用一个迭代器接收函数返回值(返回值是最后一个元素的位置), 否则输出时会输出默认值 * 谁和谁的差集是有区别的, 例如v1和v2的差集就是v1里面不是交集的部分