**算法竞赛C++ STL容器、算法、迭代器详解**

**本蒟蒻写这篇分享的目的一个是为了写一个归纳总结方便自己以后随时能够复习还有就是给那些对STL还不是很了解的萌新介绍一下什么是STL以及如何使用STL更高效偷懒地解题。本篇文章将会长期更新，欢迎大家一起监督学习，有错误的地方或者需要补充的欢迎在评论区留言哦~**

**一、STL概念**

**STL（Standard Template Library，标准模板库），是惠普实验室开发的一系列软件的统称。现主要出现在C++中，STL从广义上分为：容器（Container）、算法（Algorithm）和迭代器（Iterator）。STL几乎所有的代码都采用了模板类或者模板函数，这相比传统的由函数和类组成的库来说提供了更好的代码重用机会。**

**二、STL六大组件**

**STL提供了六大组件，彼此之间可以组合套用，这六大组件分别是容器、算法、迭代器、仿函数、适配器、空间配置器。其中，在算法竞赛中用到最多的为容器、算法与迭代器。**

**容器（Container）：STL容器为各种数据结构，如vector、stack、queue、map、set等，用来存放数据，从实现角度来看，STL容器是一种class template。**

**算法（Algorithm）：STL的算法多数定义在<algorithm>头文件中，其中包括了各种常用的算法，如sort、find、copy、reverse等，从实现角度来看，STL算法是一种function template。**

**迭代器（Iterator）：STL迭代器扮演了容器与算法之间的胶合剂，共有五种类型，从实现角度来看，迭代器是一种将opetator\*、opetator->、operator++等指针相关操作予以重载的class template。所有STL容器都附带有自己专属的迭代器，只有容器的设计者才知道如何遍历自己的元素。**

**仿函数（Functor）：行为类似函数，可作为算法的某种策略，从实现角度来看，仿函数是一种重载了operator()的class或者class template。**

**适配器（Adaptor）：一种用来修饰容器或仿函数或迭代器接口的东西。**

**空间配置器（Allocator）：负责空间的配置与管理。从实现角度来看，配置器是一个实现了动态空间配置、空间管理、空间释放的class template。**

**三、STL容器**

**相信很多人学习STL就是为了在比赛中能够更好地装B运用各种数据结构和算法，提高解题速度。确实，使用STL中的容器能够不需要自己手写定义各种数据结构，使用STL中的算法能够不需要自己手写实现各种基本算法，因此本部分对于算法巨巨们是最为重要的一部分，那么STL容器究竟有哪些呢?在做题中该如何使用呢?**

**3.1 vector**

**vector又称变长数组，定义在<vector>头文件中，vector容器是动态空间，随着元素的加入，它的内部机制会自动扩充空间以容纳新的元素。因此vector的运用对于内存的合理利用与运用的灵活性有很大的帮助。**

**vector的定义方式：**

**vector<int> v; // 定义一个vector，其中的元素为int类型**

**vector<int> v[N]; // 定义一个vector数组，其中有N个vector**

**vector<int> v(len); // 定义一个长度为len的vector**

**vector<int> v(len, x); // 定义一个长度为len的vector，初始化每个元素为x**

**vector<int> v2(v1); // 用v1给v2赋值，v1的类型为vector**

**vector<int> v2(v1.begin(), v1.begin() + 3); // 将v1中第0~2三个元素赋值给v2**

**vector的常用内置函数：**

**// vector中的常用内置函数**

**vector<int> v = { 1, 2, 3 }; // 初始化vector，v:{1, 2, 3}**

**vector<int>::iterator it = v.begin(); // 定义vector的迭代器，指向begin()**

**v.push\_back(4); // 在vector的尾部插入元素4，v:{1, 2, 3, 4}**

**v.pop\_back(); // 删除vector的最后一个元素，v:{1, 2, 3}**

**// 注意使用lower\_bound()与upper\_bound()函数时vector必须是有序的，upper\_bound()在<algorithm>中**

**lower\_bound(v.begin(), v.end(), 2); // 返回第一个大于等于2的元素的迭代器v.begin() + 1，若不存在则返回v.end()**

**upper\_bound(v.begin(), v.end(), 2); // 返回第一个大于2的元素的迭代器v.begin() + 2，若不存在则返回v.end()**

**v.size(); // 返回vector中元素的个数**

**v.empty(); // 返回vector是否为空，若为空则返回true否则返回false**

**v.front(); // 返回vector中的第一个元素**

**v.back(); // 返回vector中的最后一个元素**

**v.begin(); // 返回vector第一个元素的迭代器**

**v.end(); // 返回vector最后一个元素后一个位置的迭代器**

**v.clear(); // 清空vector**

**v.erase(v.begin()); // 删除迭代器it所指向的元素，即删除第一个元素**

**v.erase(v.begin(), v.begin() + 2); // 删除区间[v.begin(), v.begin() + 2)的所有元素**

**v.insert(v.begin(), 1); // 在迭代器it所指向的位置前插入元素1，返回插入元素的迭代器**

**// 根据下标进行遍历**

**for (int i = 0; i < v.size(); i++)**

**cout << v[i] << ' ';**

**// 使用迭代器遍历**

**for (vector<int>::iterator it = v.begin(); it != v.end(); it++)**

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

**// for\_each遍历(C++11)**

**for (auto x : v)**

**cout << x << ' ';**

**3.2 stack**

**stack又称栈，是一种后进先出（Last In First Out，LIFO）的数据结构，定义在<stack>头文件中，stack容器允许新增元素、移除元素、取得栈顶元素，但是除了最顶端以外，没有任何方法可以存取stack的其它元素，换言之，stack不允许有遍历行为。**

**stack的定义方式：**

**stack<int> stk; // 定义一个stack，其中元素的类型为int**

**stack<int> stk[N]; // 定义一个stack数组，其中有N个stack**

**stack的常用内置函数：**

**// stack中的常用内置函数**

**stack<int> stk;**

**stk.push(x); // 在stack中插入元素x**

**stk.pop(); // 弹出stack的栈顶元素**

**stk.top(); // 返回stack的栈顶元素**

**stk.size(); // 返回stack中元素的个数**

**stk.empty(); // 返回stack是否为空，若为空则返回true否则返回false**

**3.3 string**

**string又称字符串，定义在<string>头文件中。C风格的字符串（以空字符结尾的字符数组）太过复杂难于掌握，因此C++标准库定义了一种string类。string和vector<char>在数据结构、内存管理等方面都是相同的。但是，vector<char>只是单纯的一个“char元素的容器”，而string不仅是一个“char元素的容器”，它还扩展了一些针对字符串的操作，例如string可以使用c\_str()函数转换为C风格的字符串，vector中并未对输入输出流操作符进行重载，因此无法直接对vector<char>进行cin或者cout这样的操作，但是string可以，且vector<char>并不能直接实现字符串的拼接，但是string可以，string中重载了+, +=运算符。**

**string的定义方式：**

**string str; // 定义一个空的字符串**

**string str[N]; // 定义一个string数组，其中有N个string**

**string str(5, 'a'); // 使用5个字符'a'初始化**

**string str("abc"); // 使用字符串初始化**

**string的常用内置函数：**

**// string中的常用内置函数**

**string str("abcabc");**

**str.push\_back('d'); // 在string尾部插入字符，"abcabcd"**

**str.pop\_back(); // 删除string尾部的字符，"abcabc"**

**str.length(); // 返回string中字符的个数**

**str.size(); // 作用与length()相同**

**str.empty(); // 返回string是否为空，若为空返回true否则返回false**

**str.substr(1); // 返回string中从下标为1开始至末尾的子串，"bcabc"**

**str.substr(0, 2); // 返回string中从下标为0开始长度为2的子串，"ab"**

**str.insert(1, 2, 'x'); // 在下标为1的字符前插入2个字符'x'，"axxbcabc"**

**str.insert(1, "yy"); // 在下标为1的字符前插入字符串"yy"，"ayyxxbcabc"**

**str.erase(1, 4); // 删除从位置1开始的4个字符，"abcabc"**

**str.find('b'); // 返回字符'b'在string中第一次出现的位置，返回1，若不存在则返回-1**

**str.find('b', 2); // 返回从位置2开始字符'b'在string中第一次出现的位置，返回4**

**str.find("bc"); // 同上，返回字符串第一次出现的位置，返回1，若不存在则返回-1**

**str.find("bc", 2); // 返回4**

**str.rfind('b'); // 反向查找，原理同上，返回4，若不存在则返回-1**

**str.rfind('b', 3); // 返回1**

**str.rfind("bc"); // 返回4，若不存在则返回-1**

**str.rfind("bc", 3); // 返回1**

**stoi(str); // 返回str的整数形式**

**to\_string(value); // 返回value的字符串形式，value为整型、浮点型等**

**str[0]; // 用下标访问string中的字符**

**cout << (str == str) << endl; // string可比较大小，按字典序**

**string的erase()与remove()函数的用法：**

**// string中erase()与remove()的用法**

**string str1, str2, str3, str4, str5;**

**str1 = str2 = str3 = str4 = str5 = "I love AcWing! It's very funny!";**

**str1.erase(15); // 删除[15,end())的所有元素，"I love AcWing!"**

**str2.erase(6, 11); // 从第6个元素(包括)开始往后删除11个元素，"I love's very funny!"**

**str3.erase(str3.begin() + 2); // 删除迭代器所指的元素，"I ove AcWing! It's very funny!"**

**str4.erase(str4.begin() + 7, str4.end() - 11); // 删除[str4.begin()+7,str4.end()-11)的所有元素，"I love very funny!"**

**str5.erase(remove(str5.begin(), str5.end(), 'n'), str5.end()); // 删除[str5.begin(),str5.end())中所有字符'n'，"I love AcWig! It's very fuy!"**

**3.4 queue/priority\_queue**

**queue又称队列，是一种先进先出（First In First Out，FIFO）的数据结构，定义在<queue>头文件中，queue容器允许从一端（称为队尾）新增元素（入队），从另一端（称为队头）移除元素（出队）。**

**priority\_queue又称优先队列，同样定义在<queue>头文件中，与queue不同的地方在于我们可以自定义其中数据的优先级，优先级高的排在队列前面，优先出队。priority\_queue具有queue的所有特性，包括基本操作，只是在这基础上添加了内部的一个排序，它的本质是用堆实现的，因此可分为小根堆与大根堆，小根堆中较小的元素排在前面，大根堆中较大的元素排在前面。（创建priority\_queue时默认是大根堆！）**

**queue/priority\_queue的定义方式：**

**queue<int> que; // 定义一个queue，其中元素的类型为int**

**queue<int> que[N]; // 定义一个queue数组，其中有N个queue**

**priority\_queue<int> bigHeap; // 定义一个大根堆**

**priority\_queue<int, vector<int>, greater<int> > smallHeap; // 定义一个小根堆**

**queue/priority\_queue的常用内置函数：**

**// queue/priority\_queue中的常用内置函数**

**queue<int> que;**

**priority\_queue<int> bigHeap;**

**que.push(x); // 在queue的队尾插入元素x**

**que.pop(); // 出队queue的队头元素**

**que.front(); // 返回queue的队头元素**

**que.back(); // 返回queue的队尾元素**

**que.size(); // 返回queue中元素的个数**

**que.empty(); // 返回queue是否为空，若为空则返回true否则返回false**

**bigHeap.top(); // 返回priority\_queue的队头元素**

**3.5 deque**

**deque又称双端队列，定义在<deque>头文件中，vector容器是单向开口的连续内存空间，deque则是一种双向开口的连续线性空间。所谓的双向开口，意思是可以在头尾两端分别做元素的插入和删除操作，当然，vector也可以在头尾两端插入元素，但是在其头部进行插入操作效率很低。deque和vector最大的差异一是在于deque允许使用常数项时间在头部进行元素的插入和删除操作，二是在于deque没有容量的概念，因为它是动态的以分段连续空间组合而成，随时可以增加一段新的空间并链接起来。**

**deque的定义方式：**

**deque<int> deq; // 定义一个deque，其中的元素为int类型**

**deque<int> deq[N]; // 定义一个deque数组，其中有N个deque**

**deque<int> deq(len); // 定义一个长度为len的deque**

**deque<int> deq(len, x); // 定义一个长度为len的deque，初始化每个元素为x**

**deque<int> deq2(deq1); // 用deq1给v2赋值，deq2的类型为deque**

**deque<int> deq2(deq1.begin(), deq1.begin() + 3); // 将deq1中第0~2三个元素赋值给deq2**

**deque的常用内置函数：**

**//deque中的常用内置函数**

**deque<int> deq = { 1, 2, 3 }; // 初始化vector，v:{1, 2, 3}**

**deque<int>::iterator it = deq.begin(); // 定义vector的迭代器，指向begin()**

**deq.push\_back(4); // 在deque的尾部插入元素4，v:{1, 2, 3, 4}**

**deq.pop\_back(); // 删除deque的尾部元素，v:{1, 2, 3}**

**deq.push\_front(4); // 在deque的头部插入元素4，v:{4, 1, 2, 3}**

**deq.pop\_front(); // 删除deque的头部元素，v:{1, 2, 3}**

**deq.size(); // 返回deque中元素的个数**

**deq.empty(); // 返回deque是否为空，若为空则返回true否则返回false**

**deq.front(); // 返回deque中的第一个元素**

**deq.back(); // 返回deque中的最后一个元素**

**deq.begin(); // 返回deque第一个元素的迭代器**

**deq.end(); // 返回deque最后一个元素后一个位置的迭代器**

**deq.clear(); // 清空deque**

**deq.erase(deq.begin()); // 删除迭代器it所指向的元素，即删除第一个元素**

**deq.erase(deq.begin(), deq.begin() + 2); // 删除区间[v.begin(), v.begin() + 2)的所有元素**

**deq.insert(deq.begin(), 1); // 在迭代器it所指向的位置前插入元素1，返回插入元素的迭代器**

**// 根据下标进行遍历**

**for (int i = 0; i < deq.size(); i++)**

**cout << deq[i] << ' ';**

**// 使用迭代器遍历**

**for (deque<int>::iterator it = deq.begin(); it != deq.end(); it++)**

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

**// for\_each遍历(C++11)**

**for (auto x : deq)**

**cout << x << ' ';**

**3.6 map/multimap**

**map/multimap又称映射，定义在<map>头文件中，map和multimap的底层实现机制都是红黑树。map的功能是能够将任意类型的元素映射到另一个任意类型的元素上，并且所有的元素都会根据元素的键值自动排序。map所有的元素都是pair，同时拥有键值和实值（即(key, value)对），key被视为键值，value被视为实值，map不允许两个元素有相同的键值。multimap和map的操作类似，唯一区别是multimap的键值允许重复。**

**map/multimap的定义方式：**

**map<string, int> mp; // 定义一个将string映射成int的map**

**map<string, int> mp[N]; // 定义一个map数组，其中有N个map**

**multimap<string, int> mulmp; // 定义一个将string映射成int的multimap**

**multimap<string, int> mulmp[N]; // 定义一个multimap数组，其中有N个multimap**

**map/multimap的常用内置函数：**

**// map/multimap中的常用内置函数**

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

**mp["abc"] = 3; // 将"abc"映射到3**

**mp["ab"]++; // 将"ab"所映射的整数++**

**mp.insert(make\_pair("cd", 2)); // 插入元素**

**mp.insert({ "ef", 5 }); // 同上**

**mp.size(); // 返回map中元素的个数**

**mp.empty(); // 返回map是否为空，若为空返回true否则返回false**

**mp.clear(); // 清空map**

**mp.erase("ef"); // 清除元素{"ef", 5}**

**mp["abc"]; // 返回"abc"映射的值**

**mp.begin(); // 返回map第一个元素的迭代器**

**mp.end(); // 返回map最后一个元素后一个位置的迭代器**

**mp.find("ab"); // 返回第一个键值为"ab"的迭代器，若不存在则返回mp.end()**

**mp.find({ "abc", 3 }); // 返回元素{"abc", 3}的迭代器，若不存在则返回mp.end()**

**mp.count("abc"); // 返回第一个键值为"abc"的元素数量1，由于map元素不能重复因此count返回值只有0或1**

**mp.count({ "abc", 2 }); // 返回第一个键值为"abc"的元素数量1，注意和find不一样，count只判断第一个键值**

**mp.lower\_bound("abc"); // 返回第一个键值大于等于"abc"的元素的迭代器，{"abc", 3}**

**mp.upper\_bound("abc"); // 返回第一个键值大于"abc"的元素的迭代器，{"cd", 2}**

**// 使用迭代器遍历**

**for (map<string, int>::iterator it = mp.begin(); it != mp.end(); it++)**

**cout << (\*it).first << ' ' << (\*it).second << endl;**

**// for\_each遍历(C++11)**

**for (auto x : mp)**

**cout << x.first << ' ' << x.second << endl;**

**// 扩展推断范围的for\_each遍历(C++17)**

**for (auto &[k, v] : mp)**

**cout << k << ' ' << v << endl;**

**3.7 set/multiset**

**set/multiset又称集合，定义在<set>头文件中。set的特性是所有元素都会根据元素的键值自动被排序，set的元素不像map那样可以同时拥有键值和实值，set的元素既是键值又是实值，set不允许两个元素有相同的键值，因此总结来说就是set中的元素是有序且不重复的。multiset的特性和用法和set完全相同，唯一的区别在于multiset允许有重复元素，set和multiset的底层实现都是红黑树。**

**set/multiset的定义方式：**

**set<int> st; // 定义一个set，其中的元素类型为int**

**set<int> st[N]; // 定义一个set数组，其中有N个set**

**multiset<int> mulst; // 定义一个multiset**

**multiset<int> mulst[N]; // 定义一个multiset数组，其中有N个multiset**

**set/multiset的常用内置函数：**

**// set/multiset中的常用内置函数**

**set<int> st;**

**st.insert(5); // 插入元素5**

**st.insert(6); // 同上**

**st.insert(7); // 同上**

**st.size(); // 返回set中元素的个数**

**st.empty(); // 返回set是否为空，若为空返回true否则返回false**

**st.erase(6); // 清除元素6**

**st.begin(); // 返回set第一个元素的迭代器**

**st.end(); // 返回set最后一个元素后一个位置的迭代器**

**st.clear(); // 清空set**

**st.find(5); // 返回元素5的迭代器，若不存在则返回st.end()**

**st.count(5); // 返回元素5的个数1，由于set元素不会重复，因此count返回值只有0或1**

**st.lower\_bound(5); // 返回第一个键值大于等于5的元素的迭代器，返回元素5的迭代器**

**st.upper\_bound(5); // 返回第一个键值大于5的元素的迭代器，返回元素7的迭代器**

**// 使用迭代器遍历**

**for (set<int>::iterator it = st.begin(); it != st.end(); it++)**

**cout << (\*it) << ' ';**

**// for\_each遍历(C++11)**

**for (auto x : st)**

**cout << x << ' ';**

**3.8 unordered\_map/unordered\_set**

**unordered\_map/unordered\_set分别定义在<unordered\_map>与<unordered\_set>头文件中，内部采用的是hash表结构，拥有快速检索的功能。与map/set相比最大的区别在于unordered\_map/unordered\_set中的元素是无序的，增删改查的时间复杂度为O(1)**

**（map/set增删改查的时间复杂度为O(logn)），但是不支持lower\_bound()/upper\_bound()函数。**

**unordered\_map/unordered\_set的定义方式：**

**unordered\_set<int> st; // 定义一个unordered\_set，其中的元素类型为int**

**unordered\_set<int> st[N]; // 定义一个unordered\_set数组，其中有N个unordered\_set**

**unordered\_map<int, int> mp; // 定义一个unordered\_map**

**unordered\_map<int, int> mp[N]; // 定义一个unordered\_map数组，其中有N个unordered\_map**

**unordered\_map/unordered\_set的常用内置函数：**

**// unordered\_map/unordered\_set中的常用内置函数**

**unordered\_set<int> st;**

**unordered\_map<int, int> mp;**

**st.insert(5); // 插入元素5**

**st.insert(6); // 同上**

**st.insert(7); // 同上**

**st.size(); // 返回unordered\_set中元素的个数**

**st.empty(); // 返回unordered\_set是否为空，若为空返回true否则返回false**

**st.erase(6); // 清除元素6**

**st.find(5); // 返回元素5的迭代器，若不存在则返回st.end()**

**st.count(5); // 返回元素5的个数，由于unordered\_set元素不会重复，因此count返回值只有0或1**

**st.begin(); // 返回unordered\_set第一个元素的迭代器**

**st.end(); // 返回unordered\_set最后一个元素后一个位置的迭代器**

**st.clear(); // 清空unordered\_set**

**mp.insert(make\_pair(1, 2)); // 插入元素{1, 2}**

**mp.insert({ 3, 4 }); // 同上**

**mp.size(); // 返回unordered\_map中元素的个数**

**mp.empty(); // 返回unordered\_map是否为空，若为空返回true否则返回false**

**mp.erase(3); // 清除元素{3, 4}**

**mp.find(1); // 返回第一个键值为1的迭代器，若不存在则返回mp.end()**

**mp.count(1); // 返回第一个键值为1的元素数量，由于unordered\_map元素不能重复因此count返回值只有0或1**

**mp.begin(); // 返回unordered\_map第一个元素的迭代器**

**mp.end(); // 返回unordered\_map最后一个元素后一个位置的迭代器**

**mp.clear(); // 清空unordered\_map**

**// 使用迭代器遍历**

**for (unordered\_set<int>::iterator it = st.begin(); it != st.end(); it++)**

**cout << (\*it) << ' ';**

**// for\_each遍历(C++11)**

**for (auto x : st)**

**cout << x << ' ';**

**// 使用迭代器遍历**

**for (unordered\_map<int, int>::iterator it = mp.begin(); it != mp.end(); it++)**

**cout << (\*it).first << ' ' << (\*it).second << endl;**

**// for\_each遍历(C++11)**

**for (auto x : mp)**

**cout << x.first << ' ' << x.second << endl;**

**// 扩展推断范围的for\_each遍历(C++17)**

**for (auto &[k, v] : mp)**

**cout << k << ' ' << v << endl;**

**四、STL算法**

**C++标准库定义了一组泛型算法，之所以称为泛型指的是它们可以操作在多种容器上，不但可以作用于标准库类型，还可以用在内置数组类型甚至其它类型的序列上。泛型算法定义在<algorithm>头文件中，标准库还定义了一组泛化的算术算法（Generalized Numeric Algorithm），定义在<numeric>头文件中。使用方法如下：**

**#include <iostream>**

**#include <algorithm>**

**#include <numeric>**

**using namespace std;**

**int main()**

**{**

**// 使用STL容器时将数组指针改为迭代器即可**

**int a[5] = { 1, 2, 3, 4, 5 };**

**int b[5] = { 0 };**

**// 排序算法**

**sort(a, a + 5); // 将区间[0, 5)内元素按字典序从小到大排序**

**sort(a, a + 5, greater<int>()); // 将区间[0, 5)内元素按字典序从大到小排序**

**reverse(a, a + 5); // 将区间[0, 5)内元素翻转**

**nth\_element(a, a + 3, a + 5); // 将区间[0, 5)中第a + 3个数归位，即将第3大的元素放到正确的位置上，该元素前后的元素不一定有序**

**// 查找与统计算法**

**find(a, a + 5, 3); // 在区间[0, 5)内查找等于3的元素，返回迭代器，若不存在则返回end()**

**binary\_search(a, a + 5, 2); // 二分查找区间[0, 5)内是否存在元素2，若存在返回true否则返回false**

**count(a, a + 5, 3); // 返回区间[0, 5)内元素3的个数**

**// 可变序列算法**

**copy(a, a + 2, a + 3); // 将区间[0, 2)的元素复制到以a+3开始的区间，即[3, 5)**

**replace(a, a + 5, 3, 4); // 将区间[0, 5)内等于3的元素替换为4**

**fill(a, a + 5, 1); // 将1写入区间[0, 5)中(初始化数组函数)**

**unique(a, a + 5); // 将相邻元素间的重复元素全部移动至末端，返回去重之后数组最后一个元素之后的地址**

**remove(a, a + 5, 3); // 将区间[0, 5)中的元素3移至末端，返回新数组最后一个元素之后的地址**

**// 排列算法**

**next\_permutation(a, a + 5); // 产生下一个排列{ 1, 2, 3, 5, 4 }**

**prev\_permutation(a, a + 5); // 产生上一个排列{ 1, 2, 3, 4, 5 }**

**// 前缀和算法**

**partial\_sum(a, a + 5, b); // 计算数组a在区间[0, 5)内的前缀和并将结果保存至数组b中，b = { 1, 3, 6, 10, 15 }**

**// 差分算法(感谢willem248同学的补充)**

**adjacent\_difference(a, a + 5, b); // 计算数组a区间[0, 5)内的差分并将结果保存至数组b中，b = { 1, 1, 1, 1, 1 }**

**adjacent\_difference(a, a + 5, b, plus<int>()); // 计算相邻两元素的和，b = { 1, 3, 5, 7, 9 }**

**adjacent\_difference(a, a + 5, b, multiplies<int>()); // 计算相邻两元素的乘积，b = { 1, 2, 6, 12, 20 }**

**return 0;**

**}**

**作者：AsanoSaki**

**链接：https://www.acwing.com/blog/content/10558/**

**来源：AcWing**

**著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。**