STL 中序列式容器剖析

1、向量（vector）

1、概述

vector<T> 容器是包含 T 类型元素的序列容器，和 array<T，N> 容器相似，不同的是 vector<T> 容器的大小可以自动增长，从而可以包含任意数量的元素；因此类型参数 T 不再需要模板参数 N。只要元素个数超出 vector 当前容量，就会自动分配更多的空间。

2、数据结构（物理结构）的定义

vector 所采用的数据结构非常简单：线性连续空间。它以两个迭代器 start 和 finish 分别指向配置得来的连续空间中目前已被使用的范围，并以迭代器 end_of_storage 指向整块连续空间（含备用空间）的尾端：

```
template <class T, class Alloc = alloc>
class vector {
...
protected:
iterator start; // 表示目前使用空间的头
iterator finish; // 表示目前使用空间的尾
iterator end_of_storage; // 表示目前可用空间的尾
...
};
```

为了降低空间配置时的速度成本，vector 实际配置的大小可能比客端需求量更大一些，以备将来可能的扩充。这便是容量（capacity）的观念。换句话说一个 vector 的容量永远大于或等于其大小。一旦容量等于大小，便是满载，下次再有新增元素，整个 vector 就得另觅居所。

运用 start，finish，end_of_storage 三个迭代器，便可轻易提供首尾标示、大小、容量、空容器判断、注标（[ ]）运算子、最前端元素值、最后端元素值…等机能。

3、关键的基本操作的具体实现概述

插入：vector<T>::insert（iterator position，const T& x）；

```
template <class T, class Alloc>
void vector<T, Alloc>::insert_aux(iterator position, const T& x) {
if (finish != end_of_storage) { // 还有备用空间
// 在备用空间起始处建构一个元素，并以 vector 最后一个元素值为其初值。
construct(finish, *(finish - 1));
// 调整水位。
```

```
++finish;
T x_copy = x;
copy_backward(position, finish - 2, finish - 1);
*position = x_copy;
}
else { // 已无备用空间
const size_type old_size = size();
const size_type len = old_size != 0 ? 2 * old_size : 1;
// 以上配置原则：如果原大小为 0，则配置 1（个元素大小）；
// 如果原大小不为 0，则配置原大小的两倍，
// 前半段用来放置原资料，后半段准备用来放置新资料。
iterator new_start = data_allocator::allocate(len); // 实际配置
iterator new_finish = new_start;
try {
// 将原 vector 的内容拷贝到新 vector。
new_finish = uninitialized_copy(start, position, new_start);
// 为新元素设定初值 x
construct(new_finish, x);
// 调整水位。
The Annotated STL Sources
4.2 vector 123
++new_finish;
// 将原 vector 的备用空间中的内容也忠实拷贝过来
new_finish = uninitialized_copy(position, finish, new_finish);
}
catch(...) {
// "commit or rollback" semantics.
destroy(new_start, new_finish);
data_allocator::deallocate(new_start, len);
throw;
}
// 解构并释放原 vector
destroy(begin(), end());
deallocate();
```

```
// 调整迭代器，指向新 vector
start = new_start;
finish = new_finish;
end_of_storage = new_start + len;
}
}
```

在插入操作中，所谓动态增加大小，并不是在原空间之后接续新空间（因为无法保证原空间之后尚有可供配置的空间），而是以原大小的两倍另外配置一块较大空间，然后将原内容拷贝过来，然后才开始在原内容之后建构新元素，并释放原空间。因此，对 vector 的任何操作，一旦引起空间重新配置，指向原 vector 的所有迭代器就都失效了。

4、应用

问题描述

在某次比赛中，评委对每个选手打分，所有评委的分数去掉最高分和最低分后的平均值即为选手得分。现输入评委人数以及每个评委的分数，要求输出该选手的分数。

算法思想

由于每个评委分数按次序输入满足线性关系，因此可选用 vector 解题。首先用 vector 存储每个评委的分数，然后对其排序，去掉头元素与尾元素，即为去掉最低分和最高分。然后对其遍历求和，再除以 vector 大小，即为该选手的平均分。

代码

```
#include <iostream>
#include <iomanip>
#include<vector>
#include<algorithm>
using namespace std;
int main()
{
    int num,n;
    double sum=0;
    while(cin>>num)
   {
       vector<int> score;
       while(num--)
        {
            cin>>n;
```

```
            score.push_back(n);
        }
        sort(score.begin(),score.end());//对 vector 进行排序
        score.erase(score.begin());//下面两个掐头去尾
        score.erase(score.end()-1);//end 指向的是最后一个元素的后一位 所以 end-
1
        int j=score.size();//大小
        for(int i=0;i<j;i++)
            sum=sum+score[i];
        cout<<fixed<<setprecision(2)<<sum/j<<endl;
        sum=0;
    }
    return 0;
}
```

2、列表(list)

1、概述

list<T> 容器模板定义在 list 头文件中，是 T 类型对象的双向链表。

list 容器具有一些 vector 和 deque 容器所不具备的优势，它可以在常规时间内，在序列已知的任何位置插入或删除元素。这是我们使用 list，而不使用 vector 或 deque 容器的主要原因。

list 的缺点是无法通过位置来直接访问序列中的元素，也就是说，不能索引元素。为了访问 list 内部的一个元素，必须一个一个地遍历元素，通常从第一个元素或最后一个元素开始遍历。

2、数据结构（物理结构）的定义

list 不仅是一个双向链表，而且还是一个环状双向链表。所以它只需要一个指标，便可以完整表现整个链表：

```
template <class T, class Alloc = alloc> // 预设使用 alloc 为配置器
class list {
protected:
typedef __list_node<T> list_node;
public:
typedef list_node* link_type;
protected:
link_type node; // 只要一个指标，便可表示整个环状双向链表
```

```
...
};
```

如果让指标 node 指向刻意置于尾端的一个空白节点，node 便能符合 STL 对于「前闭后开」区间的要求，成为 last 迭代器，如图 4-5。这么一来，以下几个函式便都可以轻易完成：

```
iterator begin() { return (link_type)((*node).next); }
iterator end() { return node; }
bool empty() const { return node->next == node; }
size_type size() const {
size_type result = 0;
distance(begin(), end(), result); // 全域函式，第 3 章。
return result;
}
// 取头节点的内容（元素值）。
reference front() { return *begin(); }
// 取尾节点的内容（元素值）。
reference back() { return *(--end()); }
```

3、关键的基本操作的具体实现概述

移除数值相同的连续元素。void list<T, Alloc>::unique()

```
template <class T, class Alloc>
void list<T, Alloc>::unique()
{
    iterator first = begin();
    iterator last = end();
    if (first == last) return; // 空链表，什么都不必做。
    iterator next = first;
    while (++next != last) // 巡访每一个节点
    {
        if (*first == *next) // 如果在此区段中相同的元素
            erase(next); // 移除之
        else
            first = next; // 调整指标
        next = first; // 修正区段范围
    }
```

```
}
```

由于 list 是一个双向环状串行，只要我们把边际条件处理好，那么，在头部或尾部安插元素（push_front 和 push_back），动作几乎是一样的，在头部或尾部移除元素（pop_front 和 pop_back），动作也几乎是一样的。移除（erase）某个迭代器所指元素，只是做一些指标搬移动作而已，并不复杂。

4、应用

问题描述

写一个程序完成以下命令：

new id ——新建一个指定编号为 id 的序列(id<10000)

add id num——向编号为 id 的序列加入整数 num

merge id1 id2——合并序列 id1 和 id2 中的数，并将 id2 清空

unique id——去掉序列 id 中重复的元素

out id ——从小到大输出编号为 id 的序列中的元素，以空格隔开

算法思想

由于命令中的 add 操作每次都会增加序列中的元素，元素数量一直再变化，因此可选则 list 解题。每次发现 add 操作，就新建一个 list，在 add 操作即为 push_back 一个元素。Out 时按序输出即可。

代码

```
#include <list>
#include <iostream>
#include <map>
using namespace std;
int main()
{
    int id1, id2, num, n;
    char str[10];
    map< int, list<int> > ml;
    cin >> n;
    while (n--)
    {
        cin >> str;
        switch (str[0])
        {
            case 'n':
```

```
                cin >> id1;
                ml[id1] = list<int>();
                break;
            case 'a':
                cin >> id1 >> num;
                ml[id1].push_back(num);
                break;
             case 'm':
                cin >> id1 >> id2;
                ml[id1].merge(ml[id2]);
                break;
            case 'u':
                cin >> id1; ml[id1].sort()
                ml[id1].unique();break;
            case 'o':
                cin >> id1ml[id1].sort();
                for (auto iter = ml[id1].begin(); iter != ml[id1].end();
            ++iter)
                    cout << *iter << " ";
                cout << endl;
                break;
            }
        }
        return 0;
    }
```

3、双端队列(deque)

1、概述

deque 则是一种双向开口的连续线性空间。所谓双向开口，意思是可以在头尾两端分别做元素的安插和删除动作。它能够快速地随机访问任一个元素，并且能够高效地插入和删除容器的尾部元素。deque 和 vector 的最大差异，一在于 deque 允许于常数时间内对起头端进行元素的安插或移除动作，一在于 deque 没有所谓容量（capacity）观念，因为它是动态地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。换句话说，像 vector 那样「因旧空间不足而重新配置一块更大空间，然后复制元素，再释放旧空间」这样的事情在 deque 是不会发生的。也因此，deque 没有必要提供所谓的空间保留

（reserve）功能。

2、数据结构（物理结构）的定义

deque 除了维护一个指向 map 的指标外，也维护 start, finish 两个迭代器，分别指向第一缓冲区的第一个元素和最后缓冲区的最后一个元素（的下一位置）。此外它当然也必须记住目前的 map 大小。因为一旦 map 所提供的节点不足，就必须重新配置更大的一块 map。

// 见 __deque_buf_size()。BufSize 默认值为 0 的唯一理由是为了闪避某些编译器在处理常数算式（constant expressions）时的 bug。

// 预设使用 alloc 为配置器。

```
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public: // Basic types
typedef T value_type;
typedef value_type* pointer;
typedef size_t size_type;
public: // Iterators
typedef __deque_iterator<T, T&, T*, BufSiz> iterator;
protected: // Internal typedefs
// 元素的指针的指针（pointer of pointer of T）
typedef pointer* map_pointer;
protected:
iterator start;
iterator finish;
map_pointer map;
// Data members
// 表现第一个节点。
// 表现最后一个节点。
// 指向 map，map 是块连续空间，
// 其每个元素都是个指针，指向一个节点（缓冲区）。
size_type map_size; // map 内有多少指标。
...
};
```

有了上述结构，以下数个机能便可轻易完成：

```
public: // Basic accessors
```

```
    iterator begin() { return start; }
    iterator end() { return finish; }
    reference operator[](size_type n) {
    return start[difference_type(n)]; // 唤起 __deque_iterator<>::operator[]
    }
    reference front() { return *start; } // 唤起 __deque_iterator<>::operator*
    reference back() {
    iterator tmp = finish;
    --tmp; // 唤起 __deque_iterator<>::operator--
    return *tmp; // 唤起 __deque_iterator<>::operator*
    // 以上三行何不改为：return *(finish-1);
    // 因为 __deque_iterator<> 没有为 (finish-1) 定义运算子
    }
    // 下行最后有两个 ‘;’，虽奇怪但合乎语法。
    size_type size() const { return finish - start;; }
    // 以上唤起 iterator::operatorsize_
    type max_size() const { return size_type(-1); }
    bool empty() const { return finish == start; }
```

3、关键的基本操作的具体实现概述

将头元素出队：void pop_front()

```
void pop_front()
{
    if (start.cur != start.last - 1) / /第一缓冲区有一个（或更多）元素
    {
        destroy(start.cur); // 将第一元素解构
        ++start.cur; // 调整指针，相当于排除了第一元素
    }
    else// 第一缓冲区仅有一个元素
        pop_front_aux();// 这里将进行缓冲区的释放工作，有当 start.cur ==
    start.last - 1 时才会被呼叫。
}
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_front_aux()
{
```

```
    destroy(start.cur); // 将第一缓冲区的第一个元素解构。
    deallocate_node(start.first); // 释放第一缓冲区。
    start.set_node(start.node + 1); // 调整 start 的状态，使指向
    start.cur = start.first; // 下一个缓冲区的第一个元素。
}
```

4、应用

问题描述

一副牌面为从 1 到 n 的牌，每次从牌堆顶取一张放桌子上，再取一张牌放到堆底，直到手里没牌，最后桌子上的牌是从 1 到 n 有序，编写程序，输入 n，输出牌堆的顺序数组。

算法思想

由于题中的每次操作都是从堆顶和堆底完成，因此满足 deque 双向开口，且对队头和队尾操作均为常数级。因此此题选用 deque 完成。首先创建一 deque，且元素从 1 到 n，然后不断取出队头，然后到桌子上与放到堆底，将放到桌子上的牌放入数组。最后输出数组即可。

代码

```
#include<iostream>
#include<deque>
using namespace std;
int main()
{
    int n;
    int *a,*a2,i;
    cin>>n;
    a=new int(n+1);
    a2=new int(n+1);
    deque<int> q;
    for(i=1;i<=n;++i)
        q.push_back(i);
    i=1;
    while(!q.empty())
    {
        int t=q.front();
        a[i]=t;
        a2[t]=i;
        i++;
        q.pop_front();
        t=q.front();
        q.pop_front();
        q.push_back(t);
```

```
        }
        for(i=1;i<=n;++i)
        cout<<a2[i]<<" ";
        return 0;
    }
```

————————————————————

原文链接：https://blog.csdn.net/qq_36388776/article/details/82533126

4、栈(stack)

1、概述

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。stack 允许新增元素、移除元素、取得最顶端元素。但除了最顶端外，没有任何其它方法可以存取 stack 的其它元素。换言之 stack 不允许有遍历行为。将元素推入 stack 的动作称为 push，将元素推出 stack 的动作称为 pop。

2、数据结构（物理结构）的定义

```
    template<class  T,class  Cont = deque<T>>
    class  slack  {
public:
    typedef   Cont container_type;
    typedef   typename  Cont::value_type value_type;
    typedef   typename  Cont::size_type  size_type;
    stack();
    explicit stack(const   contalner_type&  cont) ;
    bool   empty  ();
    size_type  size()   const;
    value_ type  &top();
    const value_type  &top();
    void push(const  value_type &x);
    void  pop() ;
protected:
    Cont c;
    };
```

3、关键的基本操作的具体实现概述

压栈: void push(int val)

```
void push(int val){
    node pNew = node new(sizeof(NODE)); // 创建新节点，放到栈顶
    pNew->data = val;
    pNew->pNext = pS->pTop;
    pTop = pNew;    // 栈顶指针指向新元素
}
```

出栈:int pop(int val):

```
int pop(int val){
    if (isEmpty())
        return 0;
    else
    {
        node p = top;
        top = p->pNext;
        if (val != NULL) {
            *val = p->data;
        }
        Delete p; // 释放原来 top 内存
        p = NULL;
        return 1;
    }
}
```

是否为空栈: bool empty()

```
bool empty()
{
    if (top == bottom)
        return 1;
    else
        return 0;
}
```

4、应用

问题描述

给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串，判断字符串是否有效。

有效字符串需满足：左括号必须用相同类型的右括号闭合。左括号必须以正确的顺序

闭合。注意空字符串可被认为是有效字符串。

譬如："()[]{}"为合法，"(]"、"([)]"均为非法。

算法思想

要做到合法，即为每个括号均有对应的回括，且括号匹配间符合先进后出的顺序。因此我们可以将每个出现的括号压入栈，若碰到回括则出栈，若最后栈为空则合法，否则不合法。

代码

```
bool isValid(string s)
{
    string s1 = "";
    for(int i=0;i<s.length();i++)
    {
        if(s[i]=='('||s[i]=='{'||s[i]=='[')
             s1 = s1+s[i];
        if(s[i]==')'||s[i]=='}'||s[i]==']')
            {
                if(s1.back()=='('&&s[i]==')')
                {
                    s1.pop_back();
                    continue;
                }
                if(s1.back()=='['&&s[i]==']')
                {
                    s1.pop_back();
                    continue;
                }
                if(s1.back()=='{'&&s[i]=='}')
                {
                    s1.pop_back();
                    continue;
                }
                return false;
            }
    }
    if(s1.empty())
        return true;
    else
        return false;
```

5、队列(queue)

1、概述

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

queue 允许新增元素、移除元素、从最底端加入元素、取得最顶端元素。但除了最底端可以加入、最顶端可以取出，没有任何其它方法可以存取 queue 的其它元素。queue 不允许有走访行为。

2、数据结构（物理结构）的定义

```
class queue
{
public:
    queue(int c = 10);
    ~queue();
    bool empty();        //队列的判空
    int size();           //队列的大小
    bool push(T t);       //入队列
    bool pop();           //出队列
    T front();           //队首元素
private:
    int capacity;
    int begin;
    int end;
    T*  queue;
};
```

3、关键的基本操作的具体实现概述

入队：bool queue<T>::push(T t)

```
template<typename T>
bool queue<T>::push(T t)
{
    if (end + 1 % capacity == begin)            //判断队列是否已满
            return false;
    queue[end] = t; end = (end + 1) % capacity;
    return true;
}
```

出队：bool queue<T>::pop()

```
template <typename T>
bool queue<T>::pop()
{
```

```
    if (end == begin)
    {
        return false;
    }
    begin = (begin + 1) % capacity;
    return true;
}
```

4、应用

问题描述

给定一个队列(初始为空)，只有两种操作入队和出队，现给出这些操作请输出最终的队头元素。 操作解释：1 表示入队，2 表示出队。最终队头元素，若最终队空，或队空时有出队操作，输出”impossible!”(不含引号)。

【输入描述】

N(操作个数)

N 个操作(如果是入队则后面还会有一个入队元素)

【样例输入】

3

1 2

2

2

【样例输出】

impossible!

算法思想

因为该题所模拟的操作只有两种：入队和出队。满足 queue 的使用条件，且 queue 执行这两个操作时间复杂度均为 O(1),因此，选用 queue 完成该题。

首先定义一个 queue，然后每次接收到 1 是就再接受一个数，让其入队，接收到 2，则执行 pop()。若队列中无可出队元素，或者执行完所有操作队空，则输出”impossible!”。

代码

```
#include<iostream>
#include<queue>
using namespace std;
queue <int> Line;
int main()
```

```
{
    int Work_Num,Work,Num;
    cin >> Work_Num;
    for (int i=1; i<=Work_Num; i++)
    {
        cin >> Work;
        if (Work == 1)
            cin >> Num; Line.push(Num);
        else
        {
            if (!Line.front())
            {
                cout << "impossible!";
                return 0;
            }
            //需考虑过度出队的情况
            Line.pop();
        }
    }
    if (Line.front() != 0)
        cout << Line.front();
    else
        cout << "impossible!";
}
```

参考文献

[1] FlyingApe, https://blog.csdn.net/qq_35190319/article/details/86518120,判断括号是否合法[DB/OL]

[2] wwxy261, https://blog.csdn.net/wwxy1995/article/details/89155846, C++ STL list 练习[DB/OL]

[3] 叶逸灵_, https://blog.csdn.net/qq_36388776/article/details/82533126, 数组索引_deque_小米面试题[DB/OL]

[4] LOI_Sherlock, https://blog.csdn.net/Hall_Of_Fame_/article/details/75268502, 数组索引_deque_小米面试题[DB/OL]

[5]侯捷. STL 源码剖析[A]：华中科技大学出版社，2002-6-1