缓存队列与前向迭代器的设计和实现

背景介绍

迭代器是C++ STL的重要设计思想。通过迭代器访问泛型容器的内容，而不是直接通过容器访问，使得相同的算法可以很简单的作用在不同的容器、不同的数据类型上。

有时候为了特定的需求，我们还需要实现自定义功能的容器，比如这里要实现的缓存队列。实现了容器之后，还需要为其编写专属的迭代器，使其能够使用STL已有的算法完成各种功能。

缓存队列倾向于保留最近被使用的数据元素。它的具体功能包括：

1. 每个缓存队列可以存储不超过给定数量的元素，其大小由模板实例化时传入的大小确定；
2. 缓存队列可以通过use进行按值使用，有多个元素具有相同值时，将使用最久未被使用的那个元素；
3. 当队列未满时，可以使用push可以直接向队列中加入元素；而当队列已满时，push操作会先取出最长时间未使用到的元素，再加入进新的元素；
4. 使用pop取出元素时，取出的元素是最长时间没有使用到元素（例如，对一个空队列通过push加入1、2、3，则通过pop取出这三个元素的顺序依次是1、2、3。如果，在push加入1、2、3后，发生了对1的使用，那么通过pop取出这三个元素的顺序将变为2、3、1）；

5. 缓存队列的元素也可以通过下标进行访问，下标的顺序由小到大，表示上次使用时间由远及近（即下标为0的元素是最长时间没有使用的元素）；注意通过下标访问不会改变pop的顺序。

*相比而言，STL的queue只能按照加入的顺序存储和取出元素。

任务要求

请编写一个泛型的缓存队列类CacheQueue，和一个作用在CacheQueue上的前向迭代器类Iter；我们要求前向迭代器遍历的顺序是，从最久没有使用到的元素，逐个遍历到最近使用的元素。为降低难度，我们提供了这两个类的基本框架，请设计类的数据成员，并实现框架中要求的成员函数。

```
#include <stddef.h>

template<typename T,int L> class Iter;//Iter前置声明以定义CacheQueue中的begin, end迭代器

//模板参数T是数据元素的类型，L是CacheQueue可以包含的元素的最大数量
template<typename T,int L>
class CacheQueue
{
public:
    //类型别名
    using iterator=Iter<T,L>;
    using size_type=size_t;
    using difference_type=ptrdiff_t;
    using value_type=T;
    using pointer=T*;
    using reference=T&;

    //CacheQueue的默认构造函数
    CacheQueue();

    //下标访问操作符的重载函数，返回元素的引用
    //下标的顺序由小到大，表示上次使用时间由远及近（即下标为0的元素是最长时间没有使用的元素）
    //例如，队列元素从最久未使用到最近使用的元素依次为2 7 3 4，那么2的偏移量就是0，7的偏移量就是1，依次类推
    reference operator[](size_type idx);
```

```
25
26      //使用值为val的元素，如果有多个元素具有相同的值，则本次使用的是最久未被使用过的元素
27      void use(value_type val);
28
29      //向CacheQueue添加元素
30      //当CacheQueue已满时，先pop出最长时间没有使用的元素，再加入新的元素
31      //返回true表示添加前队列未满，返回false表示添加前队列已满
32      bool push(value_type val);
33
34      //从CacheQueue取出元素
35      //将最长时间没有使用到的元素pop出来
36      //返回true表示取出成功，false表示队列调用pop前为空
37      bool pop();
38
39      //返回迭代器对象begin，即指向最长时间未使用的元素的位置
40      iterator begin();
41
42      //返回迭代器对象end，按照C++标准，end返回的迭代器对象指向最后一个元素的下一个位置，所以容器数据范围实际上是[begin,end)
43      iterator end();
44  private:
45      //...
46  };
```

迭代器记录了当前在访问的是哪一个容器的哪一个位置的元素。

```
1  #include <stddef.h>
2  #include <iterator>
3
4  template<typename T,int L>
5  class Iter
6  {
7  public:
8      //类型别名，不能删除，否则STL不认为该类属于迭代器
9      using container_type=CacheQueue<T,L>;
10     using iterator=Iter<T,L>;
```

```
11      using size_type=size_t;
12      using difference_type=ptrdiff_t;
13      using value_type=T;
14      using pointer=T*;
15      using reference=T&;
16      using iterator_category = std::forward_iterator_tag;//前向迭代器标志
17
18      //迭代器的构造函数
19      //cq是迭代器对应的容器对象，offset是迭代器指向的容器中元素的位置，即该迭代器指向元素的下标
20      Iter(container_type &cq,difference_type offset);
21
22      //重载迭代器操作符
23
24      //判断两个迭代器是否相等，即是否指向了同一个容器的同一个位置
25      bool operator==(const iterator &iter);
26      bool operator!=(const iterator &iter);
27
28      //赋值操作符的重载函数，传入指向同一个容器的迭代器，将当前迭代器对象设置为指向与iter相同的位置，返回当前迭代器的引用
29      iterator& operator=(const iterator &iter);
30
31      //间接访问操作符的重载函数，相当于对指针ptr的间接访问操作*ptr，返回迭代器指向容器数据元素的引用
32      reference operator*();
33
34      //前置自增运算符的重载函数，先使迭代器指向当前容器元素的下一个位置，再返回迭代器的引用
35      iterator &operator++();
36      //后置自增运算符的重载函数，使迭代器指向当前容器元素的下一个位置，但返回的是递增前的迭代器的拷贝
37      iterator operator++(int);
38
39  private:
40      //...
41  };
```

调用示例

示例1

```
CacheQueue<int,4> cache_queue;//初始化一个存储数据类型为int的缓存队列
cache_queue.pop()//由于初始时缓存队列为空，故直接返回false
cache_queue.push(1);//向cache_queue添加数据1
cache_queue.push(2);
cache_queue.push(3);
cache_queue.push(4);
cache_queue.use(1);//访问缓存数据1
cache_queue.pop();//由于1刚才被使用过了，所以现在最长时间未使用的数据是2，故将2弹出，并返回true
cache_queue.push(5);
cache_queue.push(6);//由于缓存队列已满，故添加数据6前，先把队列中当前最长时间未使用的数据3弹出，并添加数据6
```

示例2

```
CacheQueue<double,2> cache_queue;//初始化一个存储数据类型为double的缓存队列
cache_queue.push(1.1);
cache_queue.push(2.2);
cache_queue.push(3.3);//1.1被弹出，3.3被放入，同时返回false
cache_queue.push(4.4);//2.2被弹出，4.4被放入，同时返回false
cache_queue[1]=7.7;//4.4被修改为了7.7
*cache_queue.begin()=1.1;//3.3被修改为了1.1，当前缓存队列数据变为了1.1 7.7
```

示例3

```
//初始时，缓存队列cache_queue数据依次是1 2 3 4 5，缓存队列类型是CacheQueue<int,10>

Iter<int,10> iter=cache_queue.begin();//获得缓存队列的begin迭代器，即指向1的迭代器
++iter;//将迭代器iter向前移动一个位置，当前iter指向2
iter++;//将迭代器iter向前移动一个位置，当前iter指向3

//循环打印迭代器指向数据并递增迭代器，直到迭代器指向cache_queue的end迭代器
//结果是"3 4 5 "
while(iter!=cache_queue.end())
{
    cout<<*iter<<' ';//访问iter指向的元素
    iter++;
}

Iter<int,10> second_iter(iter);//通过拷贝构造函数创建新的迭代器对象
Iter<int,10> third_iter(cache_queue,2);//创建一个绑定到cache_queue的迭代器，并指向数据3
third_iter=second_iter;//将second_iter赋值给third_iter
```

数据说明

1. 用于实例化缓存队列的类型T重载了运算符==与!=
2. 用于实例化缓存队列的数值L取值范围$0 \leqslant L \leqslant 100$

注意事项

1. 最长时间未使用和最近使用仅根据use函数的调用确定，其他的访问方式，如operator[]和迭代器不改变CacheQueue中的访问状态
2. 实现的迭代器不应该依赖于任何STL库，通过复用STL迭代器的方式可能无法通过所有样例；
3. 可以任意添加类成员，但不能改变接口定义，也不能删去定义在Iter类中的类型别名，这是STL要求的自定义迭代器特征（即满足iterator_traits特征类）；
4. 请注意处理Iter与CacheQueue的依赖关系，避免重复导入
5. 本次测验没有要求算法的时间复杂度
6. 不要在代码中包含main函数

7. 模板类的定义要放在头文件中以在main中进行实例化，所以你只需要创建以下两个文件，并将其打包成zip压缩包上传

```
XXXX.zip
  |
  |--Iter.h
  |--CacheQueue.h
```

6. 文件编码格式为utf-8

扩展说明（非测验要求内容，可后续阅读）

算法的时间复杂度

你的实现的缓存队列的use或者push，pop操作的时间复杂度可能是$O(n)$或者$O(\lg n)$的，但是作为缓存，非$O(1)$算法实践中实现的意义就很小了，请思考$O(1)$的实现思路

关于迭代器

一旦你按照要求上面的要求编写完容器类和迭代器类，下列C++语句就可以自然调用了

```
//for range语句
for(auto &&val : cq)
{
    cout<<val<<' ';
}

//用迭代器初始化其他容器
set<int> tree(cq.begin(),cq.end());

//...
```

以及能够成功使用STL算法库中的以下接口：

```
//返回[first,last)间第一个使谓词Pred为true的元素的迭代器
find_if(_InIt _First,_InIt _Last,_Pr _Pred);

//将[first,last)中所有和oldval相等的值替换为newval
replace(_FwdIt _First,_FwdIt _Last,const _Ty &_Oldval,const _Ty &_Newval);

//累计[first,last)间的值到val
accumulate(_InIt _First,_InIt _Last,_Ty _Val);

//...
```

可见，按照C规范编写容器和迭代器，将为我们带来很大的便利。
除了上面要求实现的前向迭代器（forward_iterator_tag）外，C还有输入迭代器（input_iterator_tag），输出迭代器（output_iterator_tag），双向迭代器（bidirectional_iterator_tag），随机访问迭代器（random_iterator_tag）等。它们之间存在继承关系，例如随机访问迭代器一定也是一个前向迭代器。不同的STL算法事实上要求了不同的迭代器类型，例如，如果你想使上面定义的缓存队列能够使用sort()函数排序，就需要将定义的迭代器实现为随机访问迭代器。