**Министр науки и высшего образования Российской Федерации**

**Федеральное государственное автономное образовательное учреждение высшего образования**

**«Национальный исследовательский университет ИТМО»**

Факультет информационных технологий и программирования

Лабораторная работа №7

*Кольцевой буфер.*

**Выполнил студент группы № M3116**

Тихонов Александр Дмитриевич

**Подпись:**

**Проверил:**

Повышев Владислав Вячеславович

Санкт-Петербург

2023

Задание:

Реализовать к​ольцевой буфер в виде stl-совместимого контейнера (например, может быть использован с стандартными алгоритмами), обеспеченного итератором произвольного доступа. Реализация не должна использовать ни одни из контейнеров STL.

Буфер должен обладать следующими возможностями:  
1. Вставка и удаление в конец  
2. Вставка и удаление в начало  
3. Вставка и удаление в произвольное место по итератору

4. Доступ в конец, начало

5. Доступ по индексу

6. Изменение капасити

Код реализует класс CircularBuffer, который представляет собой кольцевой буфер с произвольным доступом. Он реализует следующие функциональности:

1. Вставка и удаление элементов в конец.
2. Вставка и удаление элементов в начало.
3. Вставка и удаление элементов по итератору.
4. Доступ к началу и концу буфера.
5. Доступ к элементам по индексу.
6. Изменение емкости буфера.

Рассмотрим основные компоненты класса и их функциональность:

* buffer\_: массив, который содержит элементы кольцевого буфера.
* size\_: размер буфера (количество элементов в буфере + 1).
* start\_: индекс начала буфера.
* end\_: индекс конца буфера.

Класс CircularBuffer реализует методы для работы с буфером:

* push\_back(): добавляет элемент в конец буфера. Если буфер полон, то перезаписывает старый элемент в начале буфера.
* push\_front(): добавляет элемент в начало буфера. Если буфер полон, то перезаписывает старый элемент в конце буфера.
* pop\_front(): удаляет и возвращает элемент из начала буфера. Если буфер пуст, выбрасывается исключение std::out\_of\_range.
* pop\_back(): удаляет и возвращает элемент из конца буфера. Если буфер пуст, выбрасывается исключение std::out\_of\_range.
* empty(): проверяет, пуст ли буфер.
* size(): возвращает текущее количество элементов в буфере.
* capacity(): возвращает емкость буфера (количество элементов, которые могут быть сохранены в буфере).
* set\_capacity(): изменяет емкость буфера. Если новая емкость меньше текущей, то удаляет элементы из начала буфера. Если новая емкость больше текущей, то создает новый массив и копирует элементы из старого буфера в новый.
* operator[]: предоставляет доступ к элементам буфера по индексу.

Класс CircularBuffer также реализует вложенный класс iterator, который предоставляет итератор произвольного доступа для буфера. Итератор реализует стандартные операции для итераторов произвольного доступа, такие как инкремент, декремент, сложение, вычитание, сравнение и доступ к элементам.

Методы begin() и end() класса CircularBuffer возвращают итераторы на начало и конец буфера соответственно.

Методы cbegin() и cend() возвращают константные итераторы на начало и конец буфера, предоставляя доступ к элементам буфера без возможности изменения.

Метод insert() вставляет элемент перед указанным итератором position и возвращает итератор на вставленный элемент. Если буфер полон, выбрасывается исключение std::out\_of\_range. Если переданный итератор больше end(), выбрасывается исключение std::out\_of\_range.

Метод erase() удаляет элемент, на который указывает итератор position, и возвращает итератор на следующий элемент после удаленного. Если переданный итератор равен end(), то возвращает его без изменений.

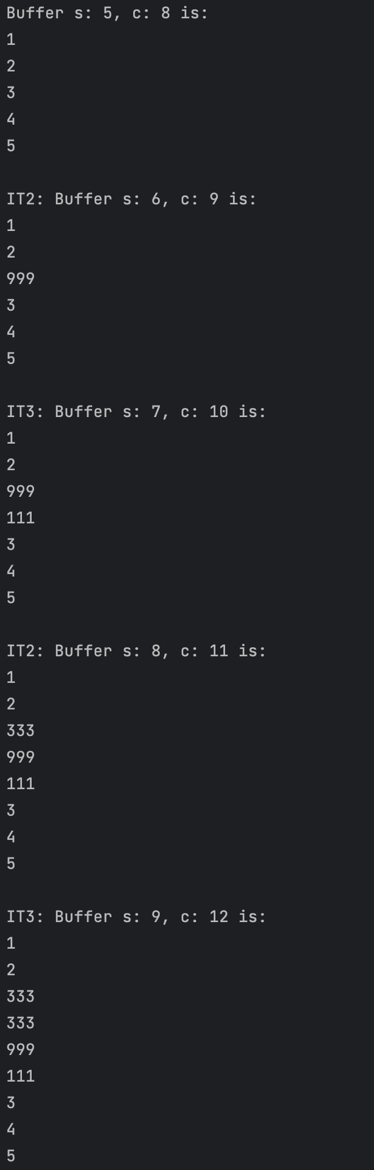
**Main.cpp**

#include "Circular buffer.h"  
  
int main() {  
 CircularBuffer<int> buffer(8);  
  
 for (int i = 0; i < 5; ++i) {  
 buffer.push\_back(i+1);  
 }  
  
 std::cout<<"Buffer s: "<<buffer.size()<<", c: "<<buffer.capacity()<<" is:\n";  
 for (int i = 0; i < buffer.size(); ++i) {  
 std::cout<<buffer[i]<<"\n";  
 }  
 std::cout<<"\n";  
  
 auto it2 = buffer.begin()+2;  
 buffer.insert(it2, 999);  
 std::cout<<"IT2: "<<"Buffer s: "<<buffer.size()<<", c: "<<buffer.capacity()<<" is:\n";;  
 for (int i = 0; i < buffer.size(); ++i) {  
 std::cout<<buffer[i]<<"\n";  
 }  
 std::cout<<"\n";  
  
 auto it3 = buffer.begin()+3;  
 buffer.insert(it3, 111);  
 std::cout<<"IT3: "<<"Buffer s: "<<buffer.size()<<", c: "<<buffer.capacity()<<" is:\n"; ;  
 for (int i = 0; i < buffer.size(); ++i) {  
 std::cout<<buffer[i]<<"\n";  
 }  
 std::cout<<"\n";  
  
 it3-it2;  
 buffer.insert(it2, 333);  
 std::cout<<"IT2: "<<"Buffer s: "<<buffer.size()<<", c: "<<buffer.capacity()<<" is:\n";;  
 for (int i = 0; i < buffer.size(); ++i) {  
 std::cout<<buffer[i]<<"\n";  
 }  
 std::cout<<"\n";  
  
  
 buffer.insert(it3, 333);  
 std::cout<<"IT3: "<<"Buffer s: "<<buffer.size()<<", c: "<<buffer.capacity()<<" is:\n";;  
 for (int i = 0; i < buffer.size(); ++i) {  
 std::cout<<buffer[i]<<"\n";  
 }  
 std::cout<<"\n";  
  
  
 return 0;  
}

**Circular buffer.h**

//  
// Created by Тихонов Александр on 01.04.2023.  
//  
  
#ifndef INC\_7\_LAB\_CIRCULAR\_BUFFER\_H  
#define INC\_7\_LAB\_CIRCULAR\_BUFFER\_H  
#include <iostream>  
#include <iterator>  
#include <stdexcept>  
  
template <class T>  
class CircularBuffer {  
private:  
 T \*buffer\_;  
 size\_t size\_;  
 size\_t start\_;  
 size\_t end\_;  
  
public:  
 CircularBuffer(size\_t size) : buffer\_(new T[size+1]), size\_(size+1), start\_(0), end\_(0) {}  
  
 ~CircularBuffer() { delete[] buffer\_; }  
  
 void push\_back(T item){  
 buffer\_[end\_] = item;  
 end\_ = (end\_ + 1) % size\_;  
 if (end\_ == start\_) {  
 start\_ = (start\_ + 1) % size\_;  
 }  
 }  
 void push\_front(T item){  
 start\_ = (start\_ + size\_ - 1) % size\_;  
 buffer\_[start\_] = item;  
 if ((end\_ + 1) % size\_ == start\_) {  
 end\_ = start\_;  
 }  
 }  
  
 T pop\_front() {  
 if (empty()) {  
 throw std::out\_of\_range("Circular buffer is empty");  
 }  
 T item = buffer\_[start\_];  
 start\_ = (start\_ + 1) % size\_;  
 return item;  
 }  
 T pop\_back() {  
 if (empty()) {  
 throw std::out\_of\_range("Circular buffer is empty");  
 }  
 T item = buffer\_[end\_-1];  
 end\_ = (end\_ - 1) % size\_;  
 return item;  
 }  
  
 bool empty() const { return start\_ == end\_; }  
  
 size\_t size() const { return (end\_ + size\_ - start\_) % size\_; }  
 size\_t capacity() const { return size\_-1; }  
  
 void set\_capacity(size\_t new\_size) {  
 if (new\_size == size\_)  
 return;  
 if (new\_size < size\_) {  
 for (int i = 0; i < (size\_ - new\_size); ++i) {  
 this->pop\_front();  
 }  
 } else {  
 T \*new\_buffer = new T[new\_size];  
 size\_t new\_end = 0;  
 size\_t buffer\_size = size();  
  
 for (size\_t i = 0; i < buffer\_size && new\_end < new\_size; i++) {  
 new\_buffer[new\_end++] = buffer\_[(start\_ + i) % size\_];  
 }  
  
 delete[] buffer\_;  
 buffer\_ = new\_buffer;  
 size\_ = new\_size;  
 start\_ = 0;  
 end\_ = new\_end;  
 }  
 }  
  
 T &operator[](size\_t index) {  
 if (index >= size()) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return buffer\_[(start\_ + index) % size\_];  
 }  
 const T &operator[](size\_t index) const {  
 if (index >= size()) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return buffer\_[(start\_ + index) % size\_];  
 }  
  
 class iterator {  
 private:  
 CircularBuffer\* buffer\_;  
 size\_t index\_;  
 public:  
 size\_t GetIndex\_(){ return index\_;}  
  
 iterator(CircularBuffer\* buffer, size\_t index) : buffer\_(buffer), index\_(index) {}  
  
 //operators here  
 T& operator\*() { return (\*buffer\_)[index\_]; }  
 const T& operator\*() const { return (\*buffer\_)[index\_]; }  
  
 T\* operator->() { return &(\*buffer\_)[index\_]; }  
 const T\* operator->() const { return &(\*buffer\_)[index\_]; }  
  
 iterator &operator++() {  
 index\_++;  
 return \*this;  
 }  
  
 iterator operator++(int) {  
 iterator tmp = \*this;  
 ++(\*this);  
 return tmp;  
 }  
  
 iterator &operator--() {  
 index\_--;  
 return \*this;  
 }  
  
 iterator operator--(int) {  
 iterator tmp = \*this;  
 --(\*this);  
 return tmp;  
 }  
  
 iterator operator+(ptrdiff\_t n) const { return iterator(buffer\_, index\_ + n); }  
  
 iterator operator-(ptrdiff\_t n) const {  
 return iterator(buffer\_, index\_ - n);  
 }  
  
 iterator &operator+=(ptrdiff\_t n) {  
 index\_ += n;  
 return \*this;  
 }  
  
 iterator &operator-=(ptrdiff\_t n) {  
 index\_ -= n;  
 return \*this;  
 }  
  
 std::ptrdiff\_t operator-(const iterator &other) const {  
 return index\_ - other.index\_;  
 }  
  
 bool operator==(const iterator &other) const {  
 return buffer\_ == other.buffer\_ && index\_ == other.index\_;  
 }  
  
 bool operator!=(const iterator &other) const {  
 return !(\*this == other);  
 }  
  
 bool operator<(const iterator &other) const {  
 return index\_ < other.index\_;  
 }  
  
 bool operator>(const iterator &other) const {  
 return index\_ > other.index\_;  
 }  
  
 bool operator<=(const iterator &other) const {  
 return !(\*this > other);  
 }  
  
 bool operator>=(const iterator &other) const {  
 return !(\*this < other);  
 }  
  
 T &operator[](ptrdiff\_t n) {  
 return \*(\*this + n);  
 }  
  
 const T &operator[](ptrdiff\_t n) const { return \*(\*this + n); }  
 };  
  
 iterator begin() { return iterator(this, 0); }  
 iterator end() { return iterator(this, size()); }  
  
 const iterator cbegin() const { return iterator(const\_cast<CircularBuffer \*>(this), 0); }  
 const iterator cend() const { return iterator(const\_cast<CircularBuffer\*>(this), size()); }  
  
 iterator insert(iterator position, const T &value) {  
 if (size() == capacity()) {  
 throw std::out\_of\_range("Buffer is full");  
 }  
  
 if (position > end()){  
 throw std::out\_of\_range("Not correct iterator");  
 }  
  
 size\_t index = position.GetIndex\_();  
 size\_t new\_end = (end\_ + 1) % capacity();  
  
 for (size\_t i = new\_end; i != index; i = (i + capacity() - 1) % capacity()) {  
 buffer\_[i] = buffer\_[(i + capacity() - 1) % capacity()];  
 }  
  
 buffer\_[index] = value;  
 end\_ = new\_end;  
 size\_++;  
  
 return iterator(this, index);  
 }  
 iterator erase(iterator position) {  
 if (position == end()) {  
 return position;  
 }  
  
 iterator next\_it = position;  
 ++next\_it;  
 while (next\_it != end()) {  
 \*position = \*next\_it;  
 position = next\_it;  
 ++next\_it;  
 }  
  
 end\_ = (end\_ + size\_ - 1) % size\_;  
 return iterator(this, position.GetIndex\_());  
 }  
};  
#endif //INC\_7\_LAB\_CIRCULAR\_BUFFER\_H

**Программа работает и выдает правильный ответ:**

****