Министерство 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

Вариант 21.

Кольцевой буфер.

Выполнил студент группы № M3111
Соловьев Михаил Александрович.

Санкт-Петербург
2023

В данном задании необходимо реализовать кольцевой 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:

ringBuffer.h

```
#ifndef COURSE_C___RINGBUFFER_H
#define COURSE_C___RINGBUFFER_H

#include <iostream>
#include <stdexcept>

template <typename T>
class RingBuffer {
private:
    T* buffer;
    size_t capacity;
    size_t size;
    size_t head;
    size_t tail; // index where next element will be inserted

public:
    class Iterator {
    private:
        T* ptr;
        size_t capacity;
        size_t head;
        size_t index;

    public:
        Iterator(T* element, size_t it_capacity, size_t it_head, size_t it_index) {
            ptr = element;
            capacity = it_capacity;
```

```
    index = it_index;
    head = it_head;
}

Iterator& operator++() {        // pre-increment
    ++ptr;
    ++index;
    return *this;
}

Iterator operator++(int) {      // post-increment
    Iterator old = *this;
    ++ptr;
    ++index;
    return old;
}

Iterator& operator--() {        // pre-decrement
    --ptr;
    --index;
    return *this;
}

Iterator operator--(int) {      // post-decrement
    Iterator old = *this;
    --ptr;
    --index;
    return old;
}

Iterator operator+(size_t n) const {        // iterator shift
    return Iterator(ptr + n, capacity, head, index + n);
}

Iterator operator-(size_t n) const {
    return Iterator(ptr - n, capacity, head, index - n);
}

Iterator& operator+=(size_t n) {
    ptr += n;
    index += n;
    return *this;
```

```
    }

    Iterator& operator-=(size_t n) {
        ptr -= n;
        index -= n;
        return *this;
    }

    typename std::iterator_traits<T*>::difference_type operator-(const Iterator&
other) const {
        return index - other.index;     //difference between two iterators in terms of
the number
    }                               // of positions between them.

    bool operator==(const Iterator& other) const {
        return ptr == other.ptr;
    }

    bool operator!=(const Iterator& other) const {
        return ptr != other.ptr;
    }

    bool operator<(const Iterator& other) const {
        return ptr < other.ptr;
    }

    bool operator<=(const Iterator& other) const {
        return ptr <= other.ptr;
    }

    bool operator>(const Iterator& other) const {
        return ptr > other.ptr;
    }

    bool operator>=(const Iterator& other) const {
        return ptr >= other.ptr;
    }

    T& operator*() const {
        return *ptr;
    }
};
```

```
    explicit RingBuffer(size_t capacity) : capacity(capacity), size(0), head(0), tail(0)
{
        buffer = new T[capacity];
    }

    ~RingBuffer() {
        delete[] buffer;
    }

    Iterator insert(Iterator pos, const T& element) {
        if (size == capacity) {
            std::cout << "Buffer is full. Operation of inserting an element denied\n";
            return pos;
        }

        size_t insertIndex = (pos - begin()) % capacity;  // Get the index relative to the
buffer

        // Shift elements to the right to make space for the new element
        for (size_t i = size; i > insertIndex; --i) {
            buffer[(head + i) % capacity] = buffer[(head + i - 1) % capacity];
        }

        buffer[(head + insertIndex) % capacity] = element;
        tail = (tail + 1) % capacity;
        size++;

        return Iterator(buffer + head, capacity, head, insertIndex);
    }

    Iterator erase(Iterator pos) {
        if (size == 0) {
            std::cout << "Buffer is empty. Operation of deleting an element denied\n";
            return pos;
        }

        size_t eraseIndex = (pos - begin()) % capacity;  // Get the index relative to the
buffer

        // Shift elements to the left to remove the element at the specified position
        for (size_t i = eraseIndex; i < size - 1; ++i) {
```

```
            buffer[(head + i) % capacity] = buffer[(head + i + 1) % capacity];
        }

        tail = (tail - 1 + capacity) % capacity;
        size--;

        return Iterator(buffer + head, capacity, head, eraseIndex);
    }

    void push_back(const T& element) {
        if (size == capacity) {
            std::cout << "Buffer is full. Operation of inserting an element denied\n";
            return;
        }

        buffer[tail] = element;
        tail = (tail + 1) % capacity;       // circular behavior
        size++;
    }

    void pop_back() {
        if (size == 0) {
            std::cout << "Buffer is empty. Operation of deleting an element denied\n";
            return;
        }

        tail = (tail - 1 + capacity) % capacity;        // value must stays positive
        size--;
    }

    void push_front(const T& element) {
        if (size == capacity) {
            std::cout << "Buffer is full. Operation of inserting an element denied\n";
            return;
        }

        head = (head - 1 + capacity) % capacity;
        buffer[head] = element;
        size++;
    }

    void pop_front() {
```

```
        if (size == 0) {
            std::cout << "Buffer is empty. Operation of deleting an element denied\n";
            return;
        }

        head = (head + 1) % capacity;
        size--;
    }

    void resize(size_t newCapacity) {
        if (newCapacity < size) {
            throw std::invalid_argument("New capacity cannot be smaller than the
number of elements in the buffer");
        }

        T* newBuffer = new T[newCapacity];

        for (size_t i = 0; i < size; ++i) {
            newBuffer[i] = buffer[(head + i) % capacity];
        }

        delete[] buffer;
        buffer = newBuffer;
        capacity = newCapacity;
        head = 0;
        tail = size;
    }

    T& operator[](size_t index) {
        if (index >= size) {
            throw std::out_of_range("Index out of range");
        }

        return buffer[(head + index) % capacity];
    }

    const T& operator[](size_t index) const {
        if (index >= size) {
            throw std::out_of_range("Index out of range");
        }

        return buffer[(head + index) % capacity];
```

```
    }

    size_t getSize() const {
        return size;
    }

    Iterator begin() {
        return Iterator(buffer + head, capacity, head, 0);
    }

    Iterator end() {
        return Iterator(buffer + tail, capacity, head, size);
    }
};

#endif //COURSE_C___RINGBUFFER_H
```

main.cpp

```
#include <iostream>
#include "ringBuffer.h"

int main() {
    RingBuffer<int> buffer(4);
    buffer.push_back(1);
    buffer.push_back(2);
    buffer.push_back(3);
    buffer.pop_back();      // 1 2
    buffer.pop_front();     // 2
    buffer.push_front(3);       // 3 2

    RingBuffer<int>::Iterator it = buffer.begin() + 1;
    it = buffer.insert(it, 9);     // 3 9 2

    it = buffer.begin() + 2;
    it = buffer.erase(it);      // 3 9

    buffer.push_back(1);
    std::cout << buffer[0] << "\n";


    for (int & el : buffer) {
        std::cout << el << " ";
```

```
    }

    return 0;
}
```

Вывод:
Я реализовал кольцевой буфер и научился работать с итераторами.