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 № 5

Кольцевой буффер.

Выполнила студентка группы № M3106
Шеин Максим Андреевич
Подпись:

Проверил:
Повышев Владислав Вячеславович

Санкт-Петербург

2020

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. Доступ в конец, начало
4. Доступ по индексу
5. Изменение капасити

Решение(main.cpp)

```
#include <iostream>
#include <algorithm>
#include "CircularBuffer.h"

template <typename T>
void print(CircularBuffer<T>& buf)
{
   for (typename CircularBuffer<T>::Iterator iterator = buf.begin(); iterator != buf.end();
++iterator)
   {
      std::cout << *iterator << std::endl;;
   }
}

int main()
{
   CircularBuffer<int> buf(2);


   buf.addFirst(1);
   buf.addFirst(4);
   buf.addFirst(3);
   buf.addFirst(2);

   for (CircularBuffer<int>::Iterator i = buf.begin(); i < buf.end(); ++i)
   {
      std::cout << *i << " ";
   }
   std::cout << std::endl;

   buf.changeCapacity(5);
   buf.addLast(666);
   for (CircularBuffer<int>::Iterator i = buf.begin(); i < buf.end(); ++i)
   {
      std::cout << *i << " ";
   }
   std::cout << std::endl;
```

```
    buf.addByIndex(1, 9);
    for (CircularBuffer<int>::Iterator i = buf.begin(); i < buf.end(); ++i)
    {
        std::cout << *i << " ";
    }
    std::cout << std::endl;



    CircularBuffer<int>::Iterator it = std::find(buf.begin(), buf.end(), 1);
    it = std::max_element(buf.begin(), buf.end());
    std::cout << *it << std::endl;



    std::sort(buf.begin(), buf.end());
    for (CircularBuffer<int>::Iterator i = buf.begin(); i < buf.end(); ++i)
    {
        std::cout << *i << " ";
    }
    std::cout << std::endl;
}
```

Решение(CircularBuffer.h)

```
template<class T = unsigned>
class CircularBuffer
{
    T* info;
    unsigned capacity;
    unsigned Size;
    unsigned k;
public:

    class Iterator : public std::iterator<std::random_access_iterator_tag, T>
    {

    private:

        T* iterator;

    public:

        using difference_type = typename std::iterator<std::random_access_iterator_tag,
T>::difference_type;
        Iterator() : iterator(nullptr){}

        explicit Iterator(T* it) : iterator(it){}

        Iterator(const Iterator &other) : iterator(other.iterator){}

        inline Iterator& operator+=(difference_type it)
        {
            iterator += it; return *this;
```

```
}

inline Iterator& operator-=(difference_type it)
{
   iterator -= it; return *this;
}

inline T& operator*() const
{
   return *iterator;
}

inline T* operator->() const
{
   return iterator;
}

inline T& operator[](difference_type i) const
{
   return iterator[i];
}

inline Iterator& operator++()
{
   ++iterator; return *this;
}

inline Iterator& operator--()
{
   --iterator; return *this;
}

inline Iterator operator++(T)
{
   Iterator tmp(*this); ++iterator; return tmp;
}

inline Iterator operator--(T)
{
   Iterator tmp(*this); --iterator; return tmp;
}

inline difference_type operator-(const Iterator& it) const
{
   return iterator - it.iterator;
}

inline Iterator operator+(difference_type it) const
{
   return Iterator(iterator + it);
}
```

```
    inline Iterator operator-(difference_type it) const
    {
        return Iterator(iterator - it);
    }

    friend inline Iterator operator+(difference_type lhs, const Iterator& rhs)
    {
        return Iterator(lhs + rhs.iterator);
    }

    friend inline Iterator operator-(difference_type lhs, const Iterator& rhs)
    {
        return Iterator(lhs - rhs.iterator);
    }

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

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

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

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

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

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

Iterator begin() const
{
    return Iterator(info);
}

Iterator end() const
```

```
    {
        return Iterator(info + Size);
    }

    explicit CircularBuffer(unsigned capacity = 1) : capacity(capacity), Size(0), k(0), info(new
T[capacity])
    {
        for (auto i = 0; i < capacity; ++i)
        {
            info[i] = 0;
        }

        std::cout << "Circular buffer of capacity " << capacity << " has been created.\n";
    }

    std::size_t size()
    {
        return Size;
    }

    void addLast(T x)
    {
        if (k >= capacity)
        {
            k = 0;
        }
        if (Size == capacity)
        {
            info[k] = x;
        }
        else
        {
            T* info_ = new T[Size + 1];
            for (auto i = 0; i < Size; ++i)
            {
                info_[i] = info[i];
            }
            info_[Size] = x;
            delete[] info;
            info = info_;
            ++Size;
        }
        k++;
    }

    void removeLast()
    {
        if (Size == 0)
        {
            throw std::out_of_range("Empty array!");
        }
        T* info_ = new T[Size - 1];
```

```
    for (auto i = 0; i < Size - 1; ++i)
    {
        info_[i] = info[i];
    }
    delete[] info;
    info = info_;
    --Size;
}

void addFirst(T x)
{
    if (Size == capacity)
    {
        T* info_ = new T[Size];
        info_[0] = x;
        for (auto i = 1; i < Size; ++i)
        {
            info_[i] = info[i - 1];
        }
        delete[] info;
        info = info_;
    }
    else
    {
        T* info_ = new T[Size + 1];
        for (auto i = 1; i < Size + 1; ++i)
        {
            info_[i] = info[i - 1];
        }
        info_[0] = x;
        delete[] info;
        info = info_;
        ++Size;
    }
}

void removeFirst()
{
    if (Size == 0)
    {
        throw std::out_of_range("Empty array!");
    }
    T* info_ = new T[Size - 1];
    for (auto i = 0; i < Size; ++i)
    {
        info_[i] = info[i + 1];
    }
    delete[] info;
    info = info_;
    --Size;
}
```

```
void addByIndex(std::size_t index, T x)
{
    if (Size == capacity)
    {
        info[index] = x;
    }
    else
    {
        T* info_ = new T[Size + 1];
        for (auto i = 0; i < index; ++i)
        {
            info_[i] = info[i];
        }
        info_[index] = x;
        for (auto i = index + 1; i < Size + 1; ++i)
        {
            info_[i] = info[i - 1];
        }
        delete[] info;
        info = info_;
        ++Size;
    }
}

void removeByIndex(std::size_t index)
{
    if (Size == 0)
    {
        throw std::out_of_range("Empty array!");
    }
    else
    {
        T* info_ = new T[Size - 1];
        for (auto i = 0; i < index; ++i)
        {
            info_[i] = info[i];
        }
        for (auto i = index; i < Size; ++i)
        {
            info_[i] = info[i + 1];
        }
        delete[] info;
        info = info_;
        --Size;
    }
}

const T& getFirst()
{
    if (Size == 0)
    {
        throw std::out_of_range("Empty array!");
```

```
        }
        return info[0];
    }

    const T& getLast()
    {
        if (Size == 0)
        {
            throw std::out_of_range("Empty array!");
        }
        return info[Size - 1];
    }

    void changeCapacity(unsigned newCapacity)
    {
        if (newCapacity <= capacity)
        {
            throw std::bad_alloc();
        }
        T* info_ = new T[newCapacity];
        for (auto i = 0; i < Size; ++i)
        {
            info_[i] = info[i];
        }
        delete[] info;
        info = info_;
        std::cout << "Capacity changed from " << capacity << " to " << newCapacity << std::endl;
        capacity = newCapacity;
        k = Size;
    }

    const T& operator[](int index)
    {
        if (index < 0 || index > Size - 1)
        {
            throw (std::out_of_range("Index is out of range!"));
        }
        return info[index];
    }
};
```