```cpp
#pragma once

#include <initializer_list>

template<class T>
class cyclic{
private:
    int m_size;
    int m_datalost;
    int m_capacity;
    int m_writePointer;
    int m_readPointer;
    int m_numOfLostItem;
    T* m_pBuffer;

    void incrementPointer(int& pointer) {
        int tempVal = pointer;
        tempVal++;
        pointer = tempVal % m_capacity;
    }

    void incrementSize() {
        if (m_size < m_capacity) {
            m_size++;
        }
        else {
            //do nothing m_size can be at most m_capacity
        }
    }

    void decrementSize() {
        if (m_size > 0) {
            m_size--;
        }
        else {
            //do nothing m_size can be at least 0
        }
    }

public:
    class iterator;

    cyclic(int capacity)
        : m_capacity(capacity), m_writePointer(0), m_readPointer(0),
        m_size(0), m_datalost(false), m_numOfLostItem(0){

        m_pBuffer = new T[m_capacity] {};
    }

    cyclic(std::initializer_list<T> items)
        : m_writePointer(0), m_readPointer(0),
        m_datalost(false), m_numOfLostItem(0) {
        /* if the user wants to start with a defined full buffer it is abit stupid to
        supply this constructor but to exercise it is needed */

        m_size = (int)items.size();
        m_capacity = m_size;
        m_pBuffer = new T[m_capacity];

        for (auto item : items) {
            m_pBuffer[m_writePointer] = item;
            incrementPointer(m_writePointer);
        }
    }

    //continue on next page
    ~cyclic()
    {
        delete[] m_pBuffer;
    }

    void push(T obj) {
        if (m_size == m_capacity) {
            //we will overwite data
            m_datalost = true;
            m_numOfLostItem++;
            incrementPointer(m_readPointer);
        }
        else {
            //buffer has free spaces
            m_datalost = false;
        }
        m_pBuffer[m_writePointer] = obj;
        incrementPointer(m_writePointer);
        incrementSize();
    }

    T& pull() {
        T* retVal = &m_pBuffer[0];

        if (m_size > 0) {
            //check if there is data in the buffer
            retVal = &m_pBuffer[m_readPointer];
            incrementPointer(m_readPointer);
            decrementSize();
        }
        else {
        }
        return *retVal;
    }

    T& read(int index) {
        T* obj = &m_pBuffer[0];
        if (index < m_capacity) {
            obj = &m_pBuffer[index];
        }
        else {
        }
        return *obj;
    }

    iterator begin() { return iterator(0, *this); }
    iterator end() { return iterator(m_capacity, *this); }
    int size() { return m_size; }
    bool datalost() { return m_datalost; }
    int lostDataCount() { return m_numOfLostItem; }
    void clearlostDataCounter() { m_numOfLostItem = 0; }
};
    //continue on next page
```

```
template<class T>
class cyclic<T>::iterator {
private:
    int m_pos;
    cyclic& m_theBuffer;
public:
    iterator(int pos, cyclic& aBuffer) : m_pos(pos), m_theBuffer(aBuffer) {}

    iterator& operator++() {
        //overloads prefix ++ operator ++it
        m_pos++;
        return *this;
    }

    iterator& operator++(int) {
        //overloads postfix ++ operator it++
        m_pos++;
        return *this;
    }

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

    T& operator*() {
        return m_theBuffer.read(m_pos);
    }

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

};
```