```
#pragma once
 1
 2
 3 #include <initializer_list>
 4
 5
   template<class T>
 6
   class cyclic{
 7
   private:
 8
        int m_size;
 9
        int m_datalost;
10
        int m_capacity;
11
        int m_writePointer;
12
        int m_readPointer;
13
        int m_numOfLostItem;
14
        T* m_pBuffer;
15
16
        void incrementPointer(int& pointer) {
17
            int tempVal = pointer;
18
            tempVal++;
19
            pointer = tempVal % m_capacity;
20
21
22
        void incrementSize() {
23
            if (m_size < m_capacity) {</pre>
24
                m_size++;
25
            }
26
            else {
27
                //do nothing m_size can be at most m_capacity
28
            }
29
        }
30
        void decrementSize() {
31
            if (m_size > 0) {
32
33
                m_size--;
34
            }
35
            else {
                //do nothing m size can be at least 0
36
37
            }
38
        }
39
40
    public:
41
        class iterator;
42
43
        cyclic(int capacity)
44
            : m_capacity(capacity), m_writePointer(0), m_readPointer(0),
45
            m_size(0), m_datalost(false), m_numOfLostItem(0){
46
            m_pBuffer = new T[m_capacity] {};
47
        }
48
49
50
        cyclic(std::initializer_list<T> items)
51
            : m_writePointer(0), m_readPointer(0),
52
            m_datalost(false), m_numOfLostItem(0) {
53
            /* if the user wants to start with a defined full buffer it is abit stupid to
54
            supply this constructor but to exercise it is needed */
55
56
            m_size = (int)items.size();
57
            m capacity = m size;
            m_pBuffer = new T[m_capacity];
58
59
            for (auto item : items) {
60
                m pBuffer[m writePointer] = item;
61
62
                incrementPointer(m_writePointer);
63
            }
64
        }
65
        //continue on next page
66
```

```
67
         ~cyclic()
 68
         {
 69
             delete[] m_pBuffer;
 70
         }
 71
 72
         void push(T obj) {
 73
             if (m_size == m_capacity) {
 74
                 //we will overwite data
 75
                 m_datalost = true;
 76
                 m numOfLostItem++;
 77
                 incrementPointer(m_readPointer);
             }
 78
 79
             else {
 80
                 //buffer has free spaces
 81
                 m_datalost = false;
 82
             m pBuffer[m writePointer] = obj;
 83
             incrementPointer(m_writePointer);
 84
             incrementSize();
 85
 86
 87
         T& pull() {
 88
             T* retVal = &m_pBuffer[0];
 89
 90
 91
             if (m_size > 0) {
 92
                 //check if there is data in the buffer
 93
                 retVal = &m_pBuffer[m_readPointer];
 94
                 incrementPointer(m readPointer);
 95
                 decrementSize();
 96
             }
             else {
 97
 98
             }
             return *retVal;
 99
100
         }
101
         T& read(int index) {
102
103
             T* obj = &m_pBuffer[0];
             if (index < m_capacity) {</pre>
104
105
                 obj = &m pBuffer[index];
             }
106
             else {
107
108
109
             return *obj;
110
         }
111
112
         iterator begin() { return iterator(0, *this); }
113
         iterator end() { return iterator(m_capacity, *this); }
114
         int size() { return m_size; }
115
         bool datalost() { return m_datalost; }
116
         int lostDataCount() { return m numOfLostItem; }
         void clearlostDataCounter() { m_numOfLostItem = 0; }
117
118 };
119
         //continue on next page
120
121
122
123
124
125
126
127
128
129
130
131
132
```

```
133 template<class T>
134 class cyclic<T>::iterator {
135 private:
136
         int m_pos;
137
         cyclic& m_theBuffer;
138
    public:
         iterator(int pos, cyclic& aBuffer) : m_pos(pos), m_theBuffer(aBuffer) {}
139
140
141
         iterator& operator++() {
             //overloads prefix ++ operator ++it
142
143
             m_pos++;
144
             return *this;
145
146
147
         iterator& operator++(int) {
             //overloads postfix ++ operator it++
148
149
             m pos++;
150
             return *this;
151
         }
152
153
         iterator& operator--() {
154
             m_pos--;
155
             return *this;
156
         }
157
        T& operator*() {
158
159
             return m_theBuffer.read(m_pos);
160
         }
161
162
         bool operator!=(const iterator& other) const {
163
             return m_pos != other.m_pos;
164
         }
165
166 };
```