```cpp
#include <iostream>
#include "Ex04.h"



// Do not modify
template <class T>
mydeque<T>::mydeque() {
    size  = 0;
    first = last = nullptr;
}


// Do not modify
template <class T>
int mydeque<T>::getSize() {
    return size;
}


// (There was an ERROR in this function: the conditional operation in the 'if'
// was != instead of ==)

// Do not modify
template <class T>
void mydeque<T>::print_front() {
    if (size == 0)
        std::cout << "The deque is empty." << std::endl;
    else
        std::cout <<    first->content     << std::endl;
}


// Do not modify
template <class T>
void mydeque<T>::print() {
    if (size == 0)
        std::cout << "The deque is empty." << std::endl;
    else {
        std::cout << "The deque has size " << size << " :" << std::endl;
        Node<T> * current = first;
        while (current != nullptr) {
            cout << " " << current->content << std::endl;
            current = current->next;
        }
    }
}


// Do not modify
template <class T>
void mydeque<T>::push_when_empty(T v) {
    Node<T> * node = new Node<T>;
    node->content  = v;
    node->next     = nullptr;
    first = last   = node;
    size  = 1;
}
```

```
59
60  ////////////////////////////////////////////////////////////////////////////
61
62  // Exercise 4 (a) Check and correct if necessary
63
64  /*template <class T>
65  mydeque<T>::~mydeque() {
66      Node<T> * current = first;
67      while (current != nullptr) {
68          delete current;
69          Node<T> * next = current->next;   // ERROR
70          current = next;                   // ERROR
71      }
72      std::cout << "Destructor completed" << std::endl;
73  }*/
74
75  // ERROR:
76  // Node deallocated before the address to the next node is read.
77
78  // CORRECTED CODE:
79  // I preferred to use a service method to perform the deallocation
80  // recursively.
81
82  template <class T>
83  mydeque<T>::~mydeque() {
84      if (first) {
85          empty(first);
86          std::cout << "Destructor completed" << std::endl;
87      }
88      else
89          std::cout << "Nothing to destroy" << std::endl;
90  }
91
92  template <class T>
93  void mydeque<T>::empty(Node<T> * n){
94      if (n->next)
95          empty(n->next);
96      delete n;
97  }
98
99
100
101 // Exercise 4 (b) Implement this function
102
103 template<class T>
104 void mydeque<T>::push_back(T v) {
105     Node<T> * node = new Node<T>;
106     if (size == 0)
107         push_when_empty(v);
108     else {
109         node->content = v;
110         node->next = nullptr;
111         last->next = node;
112         last = node;
113         size++;
114     }
115 }
116
```

```cpp

// Exercise 4 (c) Implement this function

template<class T>
void mydeque<T>::push_front(T v) {
    Node<T> * node = new Node<T>;
    if (size == 0)
        push_when_empty(v);
    else {
        node->content = v;
        node->next = first;
        first = node;
        size++;
    }
}



// Exercise 4 (d) Check and correct if necessary

/*template<class T>
void mydeque<T>::print_back() {
    if (size != 0) {                          // ERROR 1
        cout << "The deque is empty.\n";
    }
    else {
        cout << first->content << "\n";       // ERROR 2
    }
}*/

// 1. The check for empty deque is size == 0
// 2. The back of the deque is the last element, not the first.


template<class T>
void mydeque<T>::print_back() {
    if (size == 0)
        std::cout << "The deque is empty." << std::endl;
    else
        std::cout << last->content << std::endl;
}
```

```
175
176 // Exercise 4 (e) Complete body of last else
177
178 // The old 2nd element becomes the 1st
179 // The old 1st element is deleted
180 // Be careful not to delete the 1st element after reassigning it
181
182 template<class T>
183 bool mydeque<T>::pop_front() {
184     if (size == 0) {
185         return false;
186     }
187     else if (size == 1) {
188         delete first;
189         first = nullptr;
190         last  = nullptr;
191         size  = 0;
192         return true;
193     }
194     else {
195         auto next = first->next;
196         delete first;
197         first = next;
198         size--;
199     }
200 }
201
202
203 //Do not modify
204 template class mydeque<int>;
```