```hpp
#pragma once

/*******************************************************************************
** A doubly linked list
*******************************************************************************/
template <typename Data_t>
class DLinkedList
{
  public:
    class Iterator;                                          // A forward iterator

    DLinkedList();                                           // empty list constructor
    DLinkedList            ( const DLinkedList & original ); // copy constructor
    DLinkedList & operator=( const DLinkedList & rhs      ); // copy assignment
   ~DLinkedList();                                           // destructor


    bool    empty() const;                                   // returns true if list has no items
    void    clear();                                         // remove all elements setting size to zero
    size_t  size() const;                                    // returns the number of elements in the list


    Data_t & front();                                        // return list's front element
    void     prepend( const Data_t & element );              // add element to front of list (aka push_front)
    void     removeFront();                                  // remove element at front of list (aka pop_front)

    Data_t & back();                                         // return list's back element
    void     append( const Data_t & element );               // add element to back of list (aka push_back)
    void     removeBack();                                   // remove element at front of list (aka pop_back)

    Iterator insertBefore( const Iterator & position, const Data_t & element ); // Inserts element into list before the one occupied at position
    Iterator remove      ( const Iterator & position                         ); // Removes from list the element occupied at position

    Iterator begin() const;                                  // Returns an Iterator to the list's front element, nullptr if list is empty
    Iterator end  () const;                                  // Returns an Iterator beyond the list's back element.  Do not dereference this Iterator

    Iterator rbegin() const;                                 // Returns an Iterator to the list's back element, nullptr if list is empty
    Iterator rend  () const;                                 // Returns an Iterator beyond the list's front element.  Do not dereference this Iterator


  private:
    struct Node;

    Node * _head = nullptr;                                   // head of the list
    Node * _tail = nullptr;                                   // tail of the list
    size_t _size = 0;
};




/*******************************************************************************
```

```
53  ** A doubly linked list bidirectional iterator
54  ********************************************************************************/
55  template<typename Data_t>
56  class DLinkedList<Data_t>::Iterator
57  {
58    friend class DLinkedList<Data_t>;
59
60    public:
61      // Compiler synthesized constructors and destructor are fine, just what we
62      // want (shallow copies, no ownership) but needed to explicitly say that
63      // because there is also a user defined constructor
64      Iterator            (                   )    = default;
65      Iterator            ( const Iterator  & )    = default;
66      Iterator            (       Iterator && )    = default;
67      Iterator & operator=( const Iterator  & )    = default;
68      Iterator & operator=(       Iterator && )    = default;
69     ~Iterator            (                   )    = default;
70
71      Iterator( Node * position );                              // Implicit conversion constructor
72
73      // Pre and post Increment operators move the position to the next node in the list
74      Iterator & operator++();                                  // advance the iterator one node (pre -increment)
75      Iterator   operator++( int );                             // advance the iterator one node (post-increment)
76
77      // Pre and post Increment operators move the position to the next node in the list
78      Iterator & operator--();                                  // retreat the iterator one node (pre -decrement)
79      Iterator   operator--( int );                             // retreat the iterator one node (post-decrement)
80
81      Iterator   next     ( size_t delta = 1 ) const;           // Return an iterator delta nodes after this node (this iterator doesn't change)
82      Iterator   operator+( size_t rhs       ) const;           // Return an iterator delta nodes after this node (this iterator doesn't change)
83
84      Iterator   prev     ( size_t delta = 1 ) const;           // Return an iterator delta nodes after this node (this iterator doesn't change)
85      Iterator   operator-( size_t rhs       ) const;           // Return an iterator delta nodes after this node (this iterator doesn't change)
86
87      // Dereferencing and member access operators provide access to data. The
88      // iterator can be constant or non-constant, but the iterator, by
89      // definition, points to a non-constant linked list.
90      Data_t & operator* () const;
91      Data_t * operator->() const;
92
93      // Equality operators
94      bool operator==( const Iterator & rhs ) const;
95      bool operator!=( const Iterator & rhs ) const;
96
97    private:
98      Node * _node = nullptr;
99  };
100
101
102 // Including template definitions here allows a consistent approach of separating interface (header file) from implementation (source file)
103 //
104 // There are three implementations of this interface provided.  One implements the doubly linked list with direct head and tail pointers, one with
```

```
#pragma once

#include <stdexcept>    // length_error, invalid_argument
#include "DLinkedList.hpp"

/*****************************************************************************
** A doubly linked list's node
*****************************************************************************/
template<typename Data_t>
struct DLinkedList<Data_t>::Node
{
  Node( const Data_t & element ) : _data( element ) {}

  Data_t _data;                                          // linked list element value
  Node * _next = nullptr;                                // next item in the list
  Node * _prev = nullptr;                                // previous item in the list
};

template<typename Data_t>
bool DLinkedList<Data_t>::empty() const
{
  return _head->_next == _tail;
  // can also use return (_size == 0);
}

template<typename Data_t>
typename DLinkedList<Data_t>::Iterator DLinkedList<Data_t>::insertBefore( const Iterator & current, const Data_t & element )
{
  if( current == _head )  throw std::invalid_argument( "Attempt to insert before an invalid location" );

  Node * newNode = new Node( element );    // create new node

  newNode->_next = current._node;
  newNode->_prev = current._node->_prev;

  current._node->_prev->_next = newNode;
  current._node->_prev        = newNode;

  ++_size;
  return newNode;
}

template<typename Data_t>
typename DLinkedList<Data_t>::Iterator DLinkedList<Data_t>::remove( const Iterator & current )
{
  if( empty() )                             throw std::length_error    ( "attempt to remove from an empty list" );
  if( current == _head || current == _tail )  throw std::invalid_argument( "Attempt to remove at an invalid location" );

  current._node->_next->_prev = current._node->_prev;
  current._node->_prev->_next = current._node->_next;

  --_size;
```

```
53
54    Iterator returnNode( current._node->_next );    // return the node after the one removed
55    delete current._node;                           // delete what used to be the old node
56    return returnNode;
57  }
58
59  template<typename Data_t>
60  typename DLinkedList<Data_t>::Iterator DLinkedList<Data_t>::begin() const
61  {
62    return Iterator( _head->_next );
63  }
64
65  template<typename Data_t>
66  typename DLinkedList<Data_t>::Iterator DLinkedList<Data_t>::end() const
67  {
68    return Iterator(_tail);
69  }
70
```