Bidirectional list

Generated by Doxygen 1.9.4

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 list< T >::iterator Class Reference

`#include <functions.h>`

Public Member Functions

- iterator (std::shared_ptr< listNode< T > > p)
- T & operator* ()
- iterator & operator++ ()
- iterator & operator-- ()
- bool operator== (const iterator &other)
- bool operator!= (const iterator &other)

3.1.1 Detailed Description

template<class T>
class list< T >::iterator

Forward iterator class, used to traverse the list and access the elements stored in it

3.1.2 Constructor & Destructor Documentation

3.1.2.1 iterator()

```
template<class T >
list< T >::iterator::iterator (
            std::shared_ptr< listNode< T > > p )  [inline]
```

Constructor for an iterator object

Parameters

p	- pointer to the node that the iterator will point

3.1.3 Member Function Documentation

3.1.3.1 operator"!=()

```
template<class T >
bool list< T >::iterator::operator!= (
            const iterator & other )  [inline]
```

Inequality operator

Parameters

other	- the other iterator to compare with

Returns

true if the iterators point to the different node, false otherwise

3.1.3.2 operator∗()

```
template<class T >
T & list< T >::iterator::operator* ( )  [inline]
```

Dereference operator

Returns

Returns reference to the element stored in the node

3.1.3.3 operator++()

```
template<class T >
iterator & list< T >::iterator::operator++ ( )  [inline]
```

Increment operator - iterator is moved to the next node in the list

Returns

reference to the next node

3.1.3.4 operator--()

```
template<class T >
iterator & list< T >::iterator::operator-- ( )  [inline]
```

Decrement operator - iterator is moved to the previous node in the list

Returns

reference to the previous node

3.1.3.5 operator==()

```
template<class T >
bool list< T >::iterator::operator== (
            const iterator & other )  [inline]
```

Equality operator

Parameters

other	- the other iterator to compare with

Returns

true if the iterators point to the same node, false otherwise

The documentation for this class was generated from the following file:

- functions.h

3.2 list< T > Class Template Reference

```
#include <functions.h>
```

Classes

- class iterator

Public Member Functions

- list ()
- list (list< T > &otherList)
- list (list< T > &&otherList)
- ~list ()
- void push_back (T elem)
- void push_front (T elem)
- T & back ()
- T & front ()
- bool empty ()
- T pop_back ()
- T pop_front ()
- T pop_specified_position (int position)
- void display ()
- void search (T elem)
- void swap (std::shared_ptr< listNode< T > > first)
- void sort ()
- size_t getSize ()
- void save (std::string fileName)
- void open (std::string fileName)
- void clear ()
- list< T > operator= (list< T > otherList)
- list< T > operator= (list< T > &&otherList) noexcept
- iterator begin ()
- iterator end ()

3.2.1 Detailed Description

template<class T>
class list< T >

Class used as list

3.2.2 Constructor & Destructor Documentation

3.2.2.1 list() [1/3]

```
template<class T >
list< T >::list
```

Default list constructor, creates blank list

3.2.2.2 list() [2/3]

```
template<class T >
list< T >::list (
            list< T > & otherList )
```

Copy constructor, creates a copy of the other list

3.2.2.3 list() [3/3]

```
template<class T >
list< T >::list (
            list< T > && otherList )
```

Move contructor

3.2.2.4 ~list()

```
template<class T >
list< T >::~list  [inline]
```

Default list destructor

3.2.3 Member Function Documentation

3.2.3.1 back()

```
template<class T >
T & list< T >::back
```

Returns contents of the last element in the list

Returns

Reference to the last element in the list

3.2.3.2 begin()

```
template<class T >
iterator list< T >::begin ( )  [inline]
```

Returns an iterator pointing to the first element of the list

Returns

Iterator pointing to the first element of the list

3.2.3.3 clear()

```
template<class T >
void list< T >::clear  [inline]
```

Clears the memory by removing all nodes from the list and resets the head and tail pointers

3.2.3.4 display()

```
template<class T >
void list< T >::display  [inline]
```

Method displays all elements of the list

3.2.3.5 empty()

```
template<class T >
bool list< T >::empty  [inline]
```

Checks if the container is empty

Returns

true if the container is empty, false otherwise

3.2.3.6 end()

```
template<class T >
iterator list< T >::end ( )  [inline]
```

Returns an iterator pointing to the end of the list

Returns

Iterator pointing to the end of the list

3.2.3.7 front()

```
template<class T >
T & list< T >::front
```

Returns contents of the first element in the list

Returns

Reference to the first element in the list

3.2.3.8 getSize()

```
template<class T >
size_t list< T >::getSize ( )  [inline]
```

Returns the number of elements in the list

Returns

The number of elements in the list

3.2.3.9 open()

```
template<class T >
void list< T >::open (
            std::string fileName )  [inline]
```

Opens a list from a specified file

Parameters

fileName	The name of the file to open the list from

3.2.3.10 operator=() [1/2]

```
template<class T >
list< T > list< T >::operator= (
            list< T > && otherList )  [inline], [noexcept]
```

Move assignment operator

Parameters

otherList	- the list to move the contents from

Returns

A reference to the current list with the moved contents

3.2.3.11 operator=() [2/2]

```
template<class T >
list< T > list< T >::operator= (
            list< T > otherList )  [inline]
```

Copy assignment operator

Parameters

otherList	- the list to copy the contents from

Returns

A reference to the current list with the copied contents

3.2.3.12 pop_back()

```
template<class T >
T list< T >::pop_back  [inline]
```

Deletes node at the end of the list and returns deleted element

Returns

Deleted element

3.2.3.13 pop_front()

```
template<class T >
T list< T >::pop_front  [inline]
```

Deletes node at the beginning of the list and returns deleted element

Returns

Deleted element

3.2.3.14 pop_specified_position()

```
template<class T >
T list< T >::pop_specified_position (
            int position )  [inline]
```

Method which allows to pop element in a chosen position in the list

Parameters

position	- position of the element to pop

3.2.3.15 push_back()

```
template<class T >
void list< T >::push_back (
            T elem )
```

Method which allows to add element at the end of the list

Parameters

elem	- element to add

3.2.3.16 push_front()

```
template<class T >
void list< T >::push_front (
            T elem )
```

Method which allows to add element at the beginning of the list

Parameters

elem	- element to add

3.2.3.17 save()

```
template<class T >
void list< T >::save (
            std::string fileName )  [inline]
```

Saves the contents of the list to a specified file

Parameters

fileName	The name of the file to save the list to

3.2.3.18 search()

```
template<class T >
void list< T >::search (
            T elem )  [inline]
```

Search for an element in the list

Parameters

elem	- the elements to search for

3.2.3.19 sort()

```
template<class T >
void list< T >::sort  [inline]
```

Sorts the elements in the list (Bubble sort used)

3.2.3.20 swap()

```
template<class T >
void list< T >::swap (
            std::shared_ptr< listNode< T > > first )  [inline]
```

Swaps two adjacent elements

Parameters

first	- the first element to swap

The documentation for this class was generated from the following file:

- functions.h

3.3 listNode< T > Struct Template Reference

```
#include <functions.h>
```

Public Member Functions

- listNode ()=default
- listNode (listNode< T > &other)

Public Attributes

- T data
- std::shared_ptr< listNode< T > > previousNodePtr
- std::shared_ptr< listNode< T > > nextNodePtr

3.3.1 Detailed Description

template<class T>
struct listNode< T >

Struct used to store data, node of 'list' class

3.3.2 Constructor & Destructor Documentation

3.3.2.1 listNode() [1/2]

```
template<class T >
listNode< T >::listNode ( )  [default]
```

Default node constructor, node doesn't hold any data

3.3.2.2 listNode() [2/2]

```
template<class T >
listNode< T >::listNode (
            listNode< T > & other )  [inline]
```

Node constructor, which assigns data and pointers to the previous and the next node

Parameters

other	- we take values from 'other' list node and we assign it to another

3.3.3 Member Data Documentation

3.3.3.1 data

```
template<class T >
T listNode< T >::data
```

Data stored in node

3.3.3.2 nextNodePtr

```
template<class T >
std::shared_ptr<listNode<T> > listNode< T >::nextNodePtr
```

Pointer, which points to the next element of the list

3.3.3.3 previousNodePtr

```
template<class T >
std::shared_ptr<listNode<T> > listNode< T >::previousNodePtr
```

Pointer, which points to the previous element of the list

The documentation for this struct was generated from the following file:

- functions.h

3.4 myException Class Reference

```
#include <functions.h>
```

Public Member Functions

- myException (errorType _errr)
- myException ()
- const char * what ()

3.4.1 Detailed Description

Class used in errors generation

3.4.2 Constructor & Destructor Documentation

3.4.2.1 myException() [1/2]

```
myException::myException (
            errorType _errr )  [inline]
```

Error constructor, creates error of given error type

Parameters

_errr	- errorType to be set

3.4.2.2 myException() [2/2]

```
myException::myException ( )  [inline]
```

Default error constructor, creates error of default error type

3.4.3 Member Function Documentation

3.4.3.1 what()

```
const char * myException::what ( )  [inline]
```

Method which translates error types to strings

The documentation for this class was generated from the following file:

- functions.h

3.5 person Class Reference

Public Member Functions

- person ()
- person (std::string _name, int _age)
- ~person ()=default
- void setName (std::string _name)
- std::string getName ()
- void setAge (int _age)
- int getAge ()
- bool operator== (const person &other) const
- bool operator< (const person &other)
- bool operator> (const person &other)

Friends

- std::ostream & operator<< (std::ostream &s, const person &_person)
- std::istream & operator>> (std::istream &in, person &_person)

3.5.1 Constructor & Destructor Documentation

3.5.1.1 person() [1/2]

```
person::person ( )  [inline]
```

Default person constructor

3.5.1.2 person() [2/2]

```
person::person (
            std::string _name,
            int _age )  [inline]
```

Person constructor, assigns name and age to the person

Parameters

_name	The name to assign to the person
_age	The age to assign to the person

3.5.1.3 ~person()

```
person::~person ( )  [default]
```

Default person destructor

3.5.2 Member Function Documentation

3.5.2.1 getAge()

```
int person::getAge ( )
```

Method to get the age of the person

Returns

The age of the person

3.5.2.2 getName()

```
std::string person::getName ( )
```

Method to get the name of the person

Returns

Name of the person

3.5.2.3 operator<()

```
bool person::operator< (
            const person & other )
```

"Lower than" operator

Parameters

other	- the person to compare to

Returns

true if this person's age is less than the other person's age, false otherwise

3.5.2.4 operator==()

```
bool person::operator== (
            const person & other ) const  [inline]
```

Equality operator

Parameters

other	- the person to compare to

Returns

true if the two persons have the same name and age, false otherwise

3.5.2.5 operator>()

```
bool person::operator> (
            const person & other )
```

"Greater than" operator

Parameters

other	- the person to compare to

Returns

true if this person's age is greater than the other person's age, false otherwise

3.5.2.6 setAge()

```
void person::setAge (
            int _age )
```

Method to change the age of the person

Parameters

_age	- new age to assign to the person

3.5.2.7 setName()

```
void person::setName (
            std::string _name )
```

Method which allows to change the name

Parameters

_name	- new name to assign to the person

3.5.3 Friends And Related Function Documentation

3.5.3.1 operator<<

```
std::ostream & operator<< (
            std::ostream & s,
            const person & _person )  [friend]
```

Output stream operator

Parameters

s	- output stream
_person	- person to print

Returns

The output stream

3.5.3.2 operator>>

```
std::istream & operator>> (
            std::istream & in,
            person & _person )  [friend]
```

Input stream operator

Parameters

in	- The input stream
_person	- The person to read

Returns

The input stream

The documentation for this class was generated from the following files:

- functions.h
- functions.cpp

Chapter 4

File Documentation

4.1 bidirectional-list.cpp File Reference

```
#include "functions.h"
```

Functions

- int **main** ()

4.1.1 Detailed Description

Author

Maciej Jarnot (mj300741@student.polsl.pl)

Version

0.1

Date

26.01.2023

4.2 functions.cpp File Reference

```
#include "functions.h"
```

Functions

- std::ostream & operator<< (std::ostream &_stream, const person &_person)
- std::istream & operator>> (std::istream &in, person &_person)
- void gap ()

4.2.1 Detailed Description

Author

Maciej Jarnot (mj300741@student.polsl.pl)

Version

0.1

Date

26.01.2023

4.2.2 Function Documentation

4.2.2.1 gap()

```
void gap ( )
```

Function used in debugging, used just to put some space between outputs

4.2.2.2 operator<<()

```
std::ostream & operator<< (
            std::ostream & _stream,
            const person & _person )
```

Output stream operator

Parameters

s	- output stream
_person	- person to print

Returns

The output stream

4.2.2.3 operator>>()

```
std::istream & operator>> (
            std::istream & in,
            person & _person )
```

Input stream operator

Parameters

in	- The input stream
_person	- The person to read

Returns

The input stream

4.3 functions.h File Reference

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <memory>
#include <exception>
#include <iterator>
```

Classes

- class myException
- struct listNode< T >
- class list< T >
- class list< T >::iterator
- class person

Enumerations

- enum class errorType {
  notSwappable , **emptyList** , **nonEmptyList** , **fileNotOpened** ,
  undefined }

Functions

- void gap ()
- template<class T >
  void menu (list< T > l)

Variables

- constexpr auto LOGO

4.3.1 Detailed Description

Author

Maciej Jarnot (mj300741@student.polsl.pl)

Version

0.1

Date

26.01.2023

4.3.2 Enumeration Type Documentation

4.3.2.1 errorType

```
enum class errorType  [strong]
```

Enum class used for error type marking

4.3.3 Function Documentation

4.3.3.1 gap()

```
void gap ( )
```

Function used in debugging, used just to put some space between outputs

4.3.3.2 menu()

```
template<class T >
void menu (
            list< T > l )
```

Function displays a menu of options to the user and allows to interact with it.

The menu options include: 0) Open file 1) Save to file 2) Sort list 3) Size 4) Add element - push_back() 5) Add element - push_front() 6) Display list 7) pop_back() 8) pop_front() 9) Quit 10) Display head 11) Display tail 12) Clear the list

The user's choice is passed to a switch statement where the corresponding action is executed.

Template Parameters

T	- type of elements stored in the list

Parameters

l	- list that the menu will interact with

4.3.4 Variable Documentation

4.3.4.1 LOGO

```
constexpr auto LOGO  [constexpr]
```

Initial value:

```
= R"(

___ _     _ _                _   _                  _   _    _    _
| _ |_)___| (_)_ _ ___ ___| |_(_)___ _ _  ___ _| | | |  (_)__| |_
| _ \ / _` | | ' \_/ -_) _|  _| / _ \ ' \/ _` | | | |__| (_-<  _|
|___/_\__,_|_|_|_| \___\__|\__|_\___/_||_\__,_|_|_| |____|_/__/\__|


)"
```

4.4 functions.h

Go to the documentation of this file.

```
1
9 #pragma once
10
11 #ifndef FUNCTIONS_H
12 #define FUNCTIONS_H
13
14 #include <iostream>
15 #include <fstream>
16 #include <sstream>
17 #include <string>
18 #include <memory>
19 #include <exception>
20 #include <iterator>
21
22 constexpr auto LOGO = R"(
23
24 ___ _     _ _                _   _                  _   _    _    _
25 | _ |_)___| (_)_ _ ___ ___| |_(_)___ _ _  ___ _| | | |  (_)__| |_
26 | _ \ / _` | | ' \_/ -_) _|  _| / _ \ ' \/ _` | | | |__| (_-<  _|
27 |___/_\__,_|_|_|_| \___\__|\__|_\___/_||_\__,_|_|_| |____|_/__/\__|
28
29
30 )";
31
33 enum class errorType
34 {
35     notSwappable,
36     emptyList,
37     nonEmptyList,
38     fileNotOpened,
39     undefined
40 };
41
43 class myException
```

```
44 {
45     errorType errr;
47 public:
51     myException(errorType _errr) :  errr(_errr) {}
53     myException() :  errr(errorType::undefined) {}
55     const char *what()
56     {
57         switch (errr)
58         {
59         case errorType::notSwappable:
60             return "Node is not swappable!  (Next node doesn't exist!)";
61         case errorType::emptyList:
62             return "List is empty or it has only one element";
63         case errorType::nonEmptyList:
64             return "List is non empty";
65         case errorType::fileNotOpened:
66             return "File could not be opened";
67         default:
68             return "Unknown error.";
69         }
70     }
71 };
72
73 template <class T>
75 struct listNode
76 {
77     T data;
78     std::shared_ptr<listNode<T>> previousNodePtr;
79     std::shared_ptr<listNode<T>> nextNodePtr;
81     listNode() = default;
85     listNode(listNode<T> &other) :  data(other.data), previousNodePtr(other.previousNodePtr),
    nextNodePtr(other.nextNodePtr) {}
86 };
87
88 template <class T>
90 class list
91 {
92 private:
93     std::shared_ptr<listNode<T>> head;
94     std::shared_ptr<listNode<T>> tail;
95     size_t size = 0;
97 public:
99     list();
101     list(list<T> &otherList);
103     list(list<T> &&otherList);
105     ~list();
109     void push_back(T elem);
113     void push_front(T elem);
117     T &back();
121     T &front();
125     bool empty();
129     T pop_back();
133     T pop_front();
137     T pop_specified_position(int position);
139     void display();
143     void search(T elem);
147     void swap(std::shared_ptr<listNode<T>> first);
149     void sort();
153     size_t getSize() { return size; }
157     void save(std::string fileName);
161     void open(std::string fileName);
163     void clear();
168     list<T> operator=(list<T> otherList)
169     {
170         if (&otherList == this)
171         {
172             return *this;
173         }
174         else
175         {
176             if (!empty())
177             {
178
179                 while (head != nullptr)
180                 {
181                     pop_front();
182                 }
183             }
184
185             std::shared_ptr<listNode<T>> pointrr(otherList.head);
186
187             while (pointrr != nullptr)
188             {
189                 push_back(pointrr->data);
190                 pointrr = pointrr->nextNodePtr;
191             }
192             return *this;
```

```
193         }
194     }
199     list<T> operator=(list<T> &&otherList) noexcept
200     {
201         if (this != &otherList)
202         {
203             clear();
204             head = std::move(otherList.head);
205             tail = std::move(otherList.tail);
206             otherList.clear();
207         }
208 
209         return *this;
210     }
211 
213     class iterator
214     {
215     public:
219         iterator(std::shared_ptr<listNode<T>> p) :  ptr(p) {}
223         T &operator*() { return ptr->data; }
227         iterator &operator++()
228         {
229             ptr = ptr->nextNodePtr;
230             return *this;
231         }
236         iterator &operator--()
237         {
238             ptr = ptr->previousNodePtr;
239             return *this;
240         }
245         bool operator==(const iterator &other) { return ptr == other.ptr; }
250         bool operator!=(const iterator &other) { return ptr != other.ptr; }
251 
252     private:
253         std::shared_ptr<listNode<T» ptr;
254     };
255 
259     iterator begin() { return iterator(head); }
263     iterator end() { return iterator(nullptr); }
264 };
265 
266 /*Sample class, used for list class testing*/
267 class person
268 {
269 public:
271     person() :  name("Jan"), age(32){};
276     person(std::string _name, int _age) :  name(_name), age(_age){};
278     ~person() = default;
282     void setName(std::string _name);
286     std::string getName();
290     void setAge(int _age);
294     int getAge();
299     bool operator==(const person &other)const
300 {
301         return name == other.name && age == other.age;
302     }
307     bool operator<(const person &other);
312     bool operator>(const person &other);
313 
319     friend std::ostream &operator«(std::ostream &s, const person &_person);
320 
326     friend std::istream &operator»(std::istream &in, person &_person);
327 
328 private:
329     std::string name;
330     int age;
331 };
332 
334 void gap();
335 
336 template <class T>
337 list<T>::list() :  head(nullptr), tail(nullptr) {}
338 
339 template <class T>
340 list<T>::list(list<T> &otherList)
341 {
342     std::shared_ptr<listNode<T» pointrr(otherList.head);
343 
344     while (pointrr != nullptr)
345     {
346         push_back(pointrr->data);
347         pointrr = pointrr->nextNodePtr;
348     }
349 }
350 
351 template <class T>
352 list<T>::list(list<T> &&otherList)
```

```
353 {
354     head = std::move(otherList.head);
355     tail = std::move(otherList.tail);
356     otherList.clear();
357 }
358
359 template <class T>
360 inline list<T>::~list()
361 {
362     clear();
363 }
364
365 template <class T>
366 void list<T>::push_back(T elem)
367 {
368     std::shared_ptr<listNode<T>> tmp(new listNode<T>);
369     (*tmp).data = elem;
370     if (tail == nullptr)
371     {
372         head = tmp;
373         tail = tmp;
374         (*tmp).previousNodePtr = nullptr;
375         (*tmp).nextNodePtr = nullptr;
376     }
377     else
378     {
379         (*tmp).previousNodePtr = tail;
380         (*tail).nextNodePtr = tmp;
381         tail = tmp;
382         tail->nextNodePtr = nullptr;
383     }
384     size++;
385 }
386
387 template <class T>
388 void list<T>::push_front(T elem)
389 {
390     std::shared_ptr<listNode<T>> tmp(new listNode<T>);
391     (*tmp).data = elem;
392     if (tail == nullptr)
393     {
394         head = tmp;
395         tail = tmp;
396         (*tmp).previousNodePtr = nullptr;
397         (*tmp).nextNodePtr = nullptr;
398     }
399     else
400     {
401         (*tmp).nextNodePtr = head;
402         (*head).previousNodePtr = tmp;
403         head = tmp;
404         head->previousNodePtr = nullptr;
405     }
406     size++;
407 }
408
409 template <class T>
410 T &list<T>::back()
411 {
412     if (tail == nullptr)
413         throw myException();
414     return (*tail).data;
415 }
416
417 template <class T>
418 T &list<T>::front()
419 {
420     if (head == nullptr)
421         throw myException();
422     return (*head).data;
423 }
424
425 template <class T>
426 inline bool list<T>::empty()
427 {
428     return !head;
429 }
430
431 template <class T>
432 inline T list<T>::pop_back()
433 {
434     if (tail == nullptr)
435         throw myException();
436     std::shared_ptr<listNode<T>> tmp(tail);
437     T val = tail->data;
438     if (tmp->previousNodePtr != nullptr)
439         tmp->previousNodePtr->nextNodePtr = nullptr;
```

```
440     else
441         head = nullptr;
442     tail = (tmp->previousNodePtr);
443     tmp.reset();
444     return val;
445 }
446
447 template <class T>
448 inline T list<T>::pop_front()
449 {
450     if (head == nullptr)
451         throw myException();
452     std::shared_ptr<listNode<T>> tmp(head);
453     // debug
454     T val = (*tmp).data;
455     if (tmp->nextNodePtr != nullptr)
456         tmp->nextNodePtr->previousNodePtr = nullptr;
457     else
458         tail = nullptr;
459     head = tmp->nextNodePtr;
460     tmp.reset();
461     return val;
462 }
463
464 template <class T>
465 inline T list<T>::pop_specified_position(int position)
466 {
467     if (head == nullptr)
468         throw myException();
469     if (position == 0)
470     {
471         return pop_front();
472     }
473     else if (position == size - 1)
474     {
475         return pop_back();
476     }
477     else if (position > size - 1 || position < 0)
478     {
479         throw myException();
480     }
481     else
482     {
483         std::shared_ptr<listNode<T>> pointrr(head);
484         int elemCounter = 0;
485         while (pointrr != nullptr)
486         {
487             if (position == elemCounter)
488             {
489                 pointrr->previousNodePtr->nextNodePtr = pointrr->nextNodePtr;
490                 pointrr->nextNodePtr->previousNodePtr = pointrr->previousNodePtr;
491                 T val = pointrr->data;
492                 pointrr.reset();
493                 return val;
494             }
495             ++elemCounter;
496             pointrr = pointrr->nextNodePtr;
497         }
498     }
499     throw myException();
500 }
501
502 template <class T>
503 inline void list<T>::display()
504 {
505     if (head == nullptr)
506     {
507         std::cout « "\t/Empty list!/" « std::endl;
508     }
509     else
510     {
511         std::shared_ptr<listNode<T>> pointrr(head);
512
513         while (pointrr != nullptr)
514         {
515             std::cout « pointrr->data « std::endl;
516             pointrr = pointrr->nextNodePtr;
517         }
518     }
519 }
520
521 template <class T>
522 inline void list<T>::search(T elem)
523 {
524     std::shared_ptr<listNode<T>> pointrr(head);
525     int elemCounter = 0;
526     bool elemFound = false;
```

```
527     while (pointrr != nullptr)
528     {
529         if (elem == pointrr->data)
530         {
531             std::cout « pointrr->data « " at [" « elemCounter « "] position." « std::endl;
532             elemFound = true;
533         }
534         ++elemCounter;
535         pointrr = pointrr->nextNodePtr;
536     }
537     if (!elemFound)
538         std::cout « elem « " not found." « std::endl;
539 }
540
541 template <class T>
542 inline void list<T>::swap(std::shared_ptr<listNode<T>> first)
543 {
544     if (first->previousNodePtr == nullptr)
545     {
546         if (first->nextNodePtr == nullptr)
547         {
548             throw myException(errorType::notSwappable);
549         }
550         else
551         {
552             std::shared_ptr<listNode<T» second = first->nextNodePtr;
553             second->previousNodePtr = nullptr;
554             head = second;
555             first->previousNodePtr = second;
556             first->nextNodePtr = second->nextNodePtr;
557             if (second->nextNodePtr != nullptr)
558                 second->nextNodePtr->previousNodePtr = first;
559             second->nextNodePtr = first;
560         }
561     }
562     else
563     {
564         if (first->nextNodePtr == nullptr)
565         {
566             throw myException(errorType::notSwappable);
567         }
568         else
569         {
570             std::shared_ptr<listNode<T» second = first->nextNodePtr;
571             first->previousNodePtr->nextNodePtr = second;
572             second->previousNodePtr = first->previousNodePtr;
573             if (second->nextNodePtr != nullptr)
574             {
575                 second->nextNodePtr->previousNodePtr = first;
576                 first->nextNodePtr = second->nextNodePtr;
577             }
578             else
579             {
580                 first->nextNodePtr = nullptr;
581                 tail = first;
582             }
583             second->nextNodePtr = first;
584             first->previousNodePtr = second;
585         }
586     }
587 }
588
589 template <class T>
590 inline void list<T>::sort()
591 {
592     if ((head != nullptr) || (head != tail))
593     {
594         for (int i = 0; i < size - 1; i++)
595         {
596             auto pointr1 = head;
597             auto pointr2 = pointr1->nextNodePtr;
598             while (pointr2 != nullptr)
599             {
600                 if ((*pointr2).data < (*pointr1).data)
601                     swap(pointr1);
602                 pointr1 = pointr2;
603                 pointr2 = pointr1->nextNodePtr;
604                 // std::cout « "pointr1:  " «(*pointr1).data;
605                 // std::cout « "\npointr2:  " «(*pointr2).data;
606             }
607         }
608     }
609 }
610
611 template <class T>
612 inline void list<T>::save(std::string fileName)
613 {
```

```
614     fileName += ".txt";
615     std::ofstream fileToSave(fileName);
616     if (fileToSave)
617     {
618         std::shared_ptr<listNode<T» pointrr(head);
619 
620         while (pointrr != nullptr)
621         {
622             fileToSave « pointrr->data « std::endl;
623             pointrr = pointrr->nextNodePtr;
624         }
625     }
626     else
627     {
628         throw myException();
629     }
630 }
631 
632 template <class T>
633 inline void list<T>::open(std::string fileName)
634 {
635     if (head == nullptr)
636     {
637         T a;
638         std::ifstream openFile(fileName);
639         std::string line;
640         if (openFile)
641         {
642             while (std::getline(openFile, line))
643             {
644                 std::istringstream ss(line);
645                 ss » a;
646                 push_back(a);
647             }
648         }
649         else
650         {
651             throw myException(errorType::fileNotOpened);
652         }
653     }
654     else
655     {
656         throw myException(errorType::nonEmptyList);
657     }
658 }
659 
660 template <class T>
661 inline void list<T>::clear()
662 {
663     while (head != nullptr)
664     {
665         std::shared_ptr<listNode<T» temp = head;
666         head = head->nextNodePtr;
667         temp->previousNodePtr.reset();
668         temp->nextNodePtr.reset();
669     }
670 
671     tail.reset();
672     size = 0;
673 }
674 
697 template <class T>
698 void menu(list<T> l)
699 {
700     std::cout « LOGO;
701     enum class menuChoice
702     {
703         fileOpen,
704         fileWrite,
705         sortList,
706         size,
707         pushBack,
708         pushFront,
709         display,
710         popBack,
711         popFront,
712         quit,
713         head,
714         tail,
715         clearTheList,
716         popAtSpecifiedPos
717     };
718     std::cout « "------Menu-----\n0) Open file\n1) Save to file\n2) Sort list\n3) Size\n4) Add element -
      push_back()\n5) Add element - push_front()\n6) Display list\n7) pop_back()\n8) pop_front()\n9)
      Quit\n10) Display head\n11) Display tail\n12) Clear the list\n13) Pop at specified position";
719     gap();
720     int choiceNum;
```

```
721     std::cin » choiceNum;
722     auto choice = static_cast<menuChoice>(choiceNum);
723     while (choice != menuChoice::quit)
724     {
725         switch (choice)
726         {
727         case menuChoice::fileOpen:
728         {
729 
730             std::string fileName;
731             std::cout « "Enter file name:\n";
732             std::cin » fileName;
733             try
734             {
735                 l.open(fileName);
736                 l.display();
737             }
738             catch (myException e)
739             {
740                 std::cerr « e.what() « '\n';
741             }
742         }
743 
744         break;
745         case menuChoice::fileWrite:
746         {
747             std::string fileName;
748             std::cout « "Enter file name:\n";
749             std::cin » fileName;
750             try
751             {
752                 l.save(fileName);
753                 std::cout « "File saved\n";
754             }
755             catch (myException e)
756             {
757                 std::cerr « e.what() « '\n';
758             }
759         }
760         break;
761         case menuChoice::sortList:
762             try
763             {
764                 l.sort();
765                 std::cout « "List sorted\n";
766                 l.display();
767             }
768             catch (myException e)
769             {
770                 std::cerr « e.what() « '\n';
771             }
772             break;
773         case menuChoice::size:
774             try
775             {
776                 std::cout « "Size of the list:  " « l.getSize() « "\n";
777             }
778             catch (myException e)
779             {
780                 std::cerr « e.what() « '\n';
781             }
782             break;
783         case menuChoice::pushBack:
784             try
785             {
786                 T elemToPush;
787                 std::cout « "Enter data to push:\n";
788                 std::cin » elemToPush;
789                 std::cout « "\n";
790                 l.push_back(elemToPush);
791                 l.display();
792             }
793             catch (myException e)
794             {
795                 std::cerr « e.what() « '\n';
796             }
797             break;
798         case menuChoice::pushFront:
799             try
800             {
801                 T elemToPush;
802                 std::cout « "Enter data to push:\n";
803                 std::cin » elemToPush;
804                 std::cout « "\n";
805                 l.push_front(elemToPush);
806                 l.display();
807             }
```

```
808             catch (myException e)
809             {
810                 std::cerr « e.what() « '\n';
811             }
812             break;
813         case menuChoice::display:
814             try
815             {
816                 l.display();
817             }
818             catch (myException e)
819             {
820                 std::cerr « e.what() « '\n';
821             }
822             break;
823         case menuChoice::popBack:
824             try
825             {
826                 T elem = l.pop_back();
827                 std::cout « "Element deleted:  " « elem « "\n";
828                 l.display();
829             }
830             catch (myException e)
831             {
832                 std::cerr « e.what() « '\n';
833             }
834             break;
835         case menuChoice::popFront:
836             try
837             {
838                 T elem = l.pop_front();
839                 std::cout « "Element deleted:  " « elem « "\n";
840                 l.display();
841             }
842             catch (myException e)
843             {
844                 std::cerr « e.what() « '\n';
845             }
846             break;
847         case menuChoice::head:
848             try
849             {
850                 std::cout « "Head:  " « l.front() « "\n";
851             }
852             catch (myException e)
853             {
854                 std::cerr « e.what() « '\n';
855             }
856             break;
857         case menuChoice::tail:
858             try
859             {
860                 std::cout « "Tail:  " « l.back() « "\n";
861             }
862             catch (myException e)
863             {
864                 std::cerr « e.what() « '\n';
865             }
866             break;
867         case menuChoice::clearTheList:
868             try
869             {
870                 l.clear();
871                 l.display();
872             }
873             catch (myException e)
874             {
875                 std::cerr « e.what() « '\n';
876             }
877             break;
878         case menuChoice::popAtSpecifiedPos:
879             try
880             {
881                 int pos;
882                 std::cout « "Enter position to delete:  \n";
883                 std::cin » pos;
884                 std::cout « "Element deleted:  " « l.pop_specified_position(pos) « "\n";
885                 l.display();
886             }
887             catch (myException e)
888             {
889                 std::cerr « e.what() « '\n';
890             }
891             break;
892         default:
893             break;
894         }
```

```
895         gap();
896         std::cin » choiceNum;
897         choice = static_cast<menuChoice>(choiceNum);
898     }
899 }
900
901 #endif /* FUNCTIONS_H */
```

Index