Contents

1 Class Index 1
  1.1 Class List ......................................................... 1

2 Class Documentation 3
  2.1 List<T> Class Template Reference ............................ 3
Chapter 1

Class Index

1.1 Class List

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

List< T > .......................................................... 3
Chapter 2

Class Documentation

2.1 List< T > Class Template Reference

#include <list.h>

Public Member Functions

- List ()
- List (const List< T >& ) throw ( bad_alloc )
- ~List ()
- unsigned size ( ) const
- void clear ( )
- bool empty ( ) const
- void push_back (const T & ) throw ( bad_alloc )
- void push_front (const T & ) throw ( bad_alloc )
- T pop_front ( ) throw ( ListEmpty )
- T pop_back ( ) throw ( ListEmpty )
- T & getFront ( ) const throw ( ListEmpty )
- T & getCurrent ( ) const throw ( ListEmpty )
- T & getBack ( ) const throw ( ListEmpty )
- void insertBeforeCurrent (const T & ) throw ( ListEmpty, bad_alloc )
- void insertAfterCurrent (const T & ) throw ( ListEmpty, bad_alloc )
- T removeCurrent ( ) throw ( ListEmpty )
- void setToCurrent ( ) throw ( ListEmpty )
- void setToBack ( ) throw ( ListEmpty )
- void moveForward ( ) throw ( ListEmpty )
- void moveBackward ( ) throw ( ListEmpty )
- bool find (const T & ) throw ( ListEmpty )
- bool atFront ( ) const throw ( ListEmpty )
- bool atBack ( ) const throw ( ListEmpty )
- const List< T >& operator= (const List< T >& ) throw ( bad_alloc )
2.1.1 Detailed Description

template<typename T> class List<T>

Implementation of a List (p. 3) ADT using a doubly-linked list.

Author:

Mark Maloof
(your name)

Version:

1.0 3/1/05

2.1.2 Constructor & Destructor Documentation

2.1.2.1 template<typename T> List<T>::List ()

Default constructor.

2.1.2.2 template<typename T> List<T>::List (const List<T>& l) throw (
  bad_alloc )

Copy constructor.

Exceptions:

  bad_alloc if memory cannot be allocated.

2.1.2.3 template<typename T> List<T>::~List ()

Class destructor.

2.1.3 Member Function Documentation

2.1.3.1 template<typename T> unsigned List<T>::size () const

Returns the size (i.e., number of elements) of the list.

Returns:

  an unsigned integer indicating the list’s size.

2.1.3.2 template<typename T> void List<T>::clear ()

Removes the elements in the list.
2.1 List<T> Class Template Reference

2.1.3.3  template<typename T> bool List<T>::empty () const

Returns true if the list is empty; returns false otherwise.

Returns:

true if empty; false otherwise.

2.1.3.4  template<typename T> void List<T>::push_back (const T & object)
          throw ( bad_alloc )

Adds the object to the back of the list. After adding, sets current to the new node.

Parameters:

object the object to be added to the back of the list.

Exceptions:

bad_alloc if memory cannot be allocated.

2.1.3.5  template<typename T> void List<T>::push_front (const T & object)
          throw ( bad_alloc )

Adds the object to the front of the list. After adding, sets current to the new node.

Parameters:

object the object to be added to the front of the list.

Exceptions:

bad_alloc if memory cannot be allocated.

2.1.3.6  template<typename T> T List<T>::pop_front () throw ( ListEmpty )

Removes and returns the object at the front of the list. If current points to the front of the list,
then sets current to point to the new front of the list. Otherwise, current is left unchanged.

Returns:

the object at the front of the list.

Exceptions:

ListEmpty if the list is empty.
2.1.3.7 template<typename T> T List<T>::pop_back() throw (ListEmpty)

Removes and returns the object at the back of the list. If current points to the back of the list, then sets current to point to the new back of the list. Otherwise, current is left unchanged.

Returns:

the object at the back of the list.

Exceptions:

ListEmpty if the list is empty.

2.1.3.8 template<typename T> T & List<T>::getFront() const throw (ListEmpty)

Gets, but does not remove, the object at the front of the list. Current is left unchanged.

Returns:

a reference to the object at the front of the list.

Exceptions:

ListEmpty if the list is empty.

2.1.3.9 template<typename T> T & List<T>::getCurrent() const throw (ListEmpty)

Gets, but does not remove, the object pointed to by current.

Returns:

a reference to the object pointed to by current.

Exceptions:

ListEmpty if the list is empty.

2.1.3.10 template<typename T> T & List<T>::getBack() const throw (ListEmpty)

Gets, but does not remove, the object at the back of the list. Current is left unchanged.

Returns:

a reference to the object at the back of the list.

Exceptions:

ListEmpty if the list is empty.
2.1 List\(<\ T\ >\) Class Template Reference

2.1.3.11 template<typename T> void List< T >::insertBeforeCurrent (const T & object) throw ( ListEmpty, bad_alloc )

Inserts the object before the node pointed to by current. Sets current to point to the new node.

Parameters:

- **object** the object to be inserted before the current node.

Exceptions:

- **bad_alloc** if memory cannot be allocated.
- **ListEmpty** if the list is empty.

2.1.3.12 template<typename T> void List< T >::insertAfterCurrent (const T & object) throw ( ListEmpty, bad_alloc )

Inserts the object after the node pointed to by current. Sets current to point to the new node.

Parameters:

- **object** the object to be inserted after the current node.

Exceptions:

- **bad_alloc** if memory cannot be allocated.
- **ListEmpty** if the list is empty.

2.1.3.13 template<typename T> T List< T >::removeCurrent () throw ( ListEmpty )

Removes and returns the object in the node pointed to by current. Sets current to the next node, if possible. Otherwise, it sets current to the previous node.

Returns:

- the object in the current node.

Exceptions:

- **ListEmpty** if the list is empty.

2.1.3.14 template<typename T> void List< T >::setToFirst () throw ( ListEmpty )

Sets current to the first node in the list.

Exceptions:

- **ListEmpty** if the list is empty.
2.1.3.15  template<typename T> void List<T>::setToBack () throw ( ListEmpty )
Sets current to the last node in the list.

Exceptions:

  *ListEmpty* if the list is empty.

2.1.3.16  template<typename T> void List<T>::moveForward () throw ( ListEmpty )
Moves current to the next node in the list. If current points to the end of the list, then current is
left unchanged.

Exceptions:

  *ListEmpty* if the list is empty.

2.1.3.17  template<typename T> void List<T>::moveBackward () throw ( ListEmpty )
Move current to the previous node in the list. If current points to the front of the list, then current
is left unchanged.

Exceptions:

  *ListEmpty* if the list is empty.

2.1.3.18  template<typename T> bool List<T>::find (const T & object) throw ( ListEmpty )
Returns true if the object is found in the list, and sets current to point to the node containing the
found item; Returns false otherwise, leaving current unaltered.

Parameters:

  *object* the object to be found in the list.

Returns:

  true if object is in the list; false otherwise.

Exceptions:

  *ListEmpty* if the list is empty.

2.1.3.19  template<typename T> bool List<T>::atFront () const throw ( ListEmpty )
Returns true if current is at the front of the list; Returns false otherwise.
2.1 List\< T \> Class Template Reference

Returns:

true if at the front of the list; false otherwise.

Exceptions:

ListEmpty if the list is empty.

2.1.3.20 template<

\text{typename T} > bool List\< T \>::atBack () const throw ( 

ListEmpty )

Returns true if current is at the back of the list; Returns false otherwise.

Returns:

true if at the back of the list; false otherwise.

Exceptions:

ListEmpty if the list is empty.

2.1.3.21 template<

\text{typename T} > const List\< T \> & List\< T \>::operator= ( const 

List\< T \> & list ) throw ( bad_alloc )

Returns a deep copy of the list passed in as the parameter.

Parameters:

\text{list} the list to be copied.

Returns:

a copy of the list.

Exceptions:

bad_alloc if memory cannot be allocated.

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

* list.h
Index

~List
   List, 4

atBack
   List, 9

atFront
   List, 8

clear
   List, 4

easy
   List, 4

find
   List, 8

getBack
   List, 6

getCurrent
   List, 6

getFront
   List, 6

insertAfterCurrent
   List, 7

insertBeforeCurrent
   List, 6

List, 3
   ~List, 4
atBack, 9
atFront, 8
clear, 4
easy, 4
find, 8
getBack, 6
getCurrent, 6
getFront, 6
insertAfterCurrent, 7
insertBeforeCurrent, 6
List, 4
moveBackward, 8
moveForward, 8
operator=, 9
pop_back, 5
pop_front, 5
push_back, 5
push_front, 5
removeCurrent, 7
setToBack, 7
setToFront, 7
size, 4

moveBackward
   List, 8

moveForward
   List, 8

operator=
   List, 9

pop_back
   List, 5

pop_front
   List, 5

push_back
   List, 5

push_front
   List, 5

removeCurrent
   List, 7

setToBack
   List, 7

setToFront
   List, 7

size
   List, 4