Reference Manual

Generated by Doxygen 1.3

Thu Mar 31 18:12:54 2005

Contents

Chapter 1

Compound Index

1.1 Compound List

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

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 **setToFront** () 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 ADT using a doubly-linked list.

Author:
Mark Maloof

Version:
1.0 2/25/05

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> 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.

2.1.3.2 template<typename T> bool List< T >::atFront () const throw (ListEmpty)

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

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

Exceptions:
ListEmpty if the list is empty.

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

Removes the elements in the list.

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

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

Returns:
true if empty; false otherwise.

2.1.3.5 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.

2.1.3.6 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.

2.1.3.7 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.

2.1.3.8 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> 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.

2.1.3.10 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.

2.1.3.11 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.

2.1.3.12 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.

2.1.3.13 template<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:
list the list to be copied.

Returns:
a copy of the list.

Exceptions:
bad_alloc if memory cannot be allocated.

2.1.3.14 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.

2.1.3.15 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.

2.1.3.16 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.

2.1.3.17 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.

2.1.3.18 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.

2.1.3.19 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.

2.1.3.20 template<typename T> void List< T >::setToFront () throw (ListEmpty)

Sets current to the first node in the list.

Exceptions:

ListEmpty if the list is empty.

2.1.3.21 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.

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

- list.h

2.2 Vector< T > Class Template Reference

`#include <vector.h>`

Public Member Functions

- **Vector** ()
- **Vector** (const unsigned, const T &=T()) throw (bad_alloc)
- **Vector** (const Vector< T > &) throw (bad_alloc)
- ~**Vector** ()
- bool **empty** () const
- unsigned **size** () const
- unsigned **capacity** () const
- void **clear** ()
- void **resize** (const unsigned, const T &=T()) throw (bad_alloc)
- T & **at** (const unsigned) const throw (VectorEmpty, OutOfBounds)
- void **assign** (const unsigned, const T &) throw (VectorEmpty, OutOfBounds)
- void **push_back** (const T &) throw (bad_alloc)
- void **insert** (const unsigned, const T &) throw (bad_alloc, OutOfBounds)
- void **remove** (const unsigned) throw (VectorEmpty, OutOfBounds)
- T & **operator**[] (const unsigned) const throw (VectorEmpty, OutOfBounds)
- const T & **operator=** (const Vector< T > &) throw (bad_alloc)

2.2.1 Detailed Description

template<typename T> class Vector< T >

Implementation of a resizable Vector ADT using dynamically allocated C-style arrays

Author:
Mark Maloof

Version:
1.0 2/25/05

2.2.2 Constructor & Destructor Documentation

2.2.2.1 template<typename T> Vector< T >::Vector ()

Default constructor.

2.2.2.2 template<typename T> Vector< T >::Vector (const unsigned *sz*, const T & *v* = T()) throw (bad_alloc)

Constructor for initializing a vector to a fixed size and containing a given value.

Exceptions:
bad_alloc if memory cannot be allocated.

2.2.2.3 template<typename T> Vector< T >::Vector (const Vector< T > & *v*) throw (bad_alloc)

Copy constructor.

Exceptions:
bad_alloc if memory cannot be allocated.

2.2.2.4 template<typename T> Vector< T >::~Vector ()

Class destructor.

2.2.3 Member Function Documentation

2.2.3.1 template<typename T> void Vector< T >::assign (const unsigned *i*, const T & *object*) throw (VectorEmpty, OutOfBounds)

Assigns the object to the specified position in the vector.

Parameters:
i the position to be assigned.
object the object to be stored in the vector.

Exceptions:
VectorEmpty if vector is empty.
OutOfBounds if index parameter is out of bounds.

2.2.3.2 template<typename T> T & Vector< T >::at (const unsigned *i*) const throw (VectorEmpty, OutOfBounds)

Returns a reference to the object stored at a given position in the vector.

Parameters:
i the object's location.

Returns:
a reference to the object.

Exceptions:
VectorEmpty if vector is empty.
OutOfBounds if index parameter is out of bounds.

2.2.3.3 template<typename T> unsigned Vector< T >::capacity () const

Returns the capacity of the vector, which is the number of elements that the vector can store before increasing the capacity.

Returns:
an unsigned integer indicating the vector's capacity.

2.2.3.4 template<typename T> void Vector< T >::clear ()

Removes the elements of the vector.

2.2.3.5 template<typename T> bool Vector< T >::empty () const

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

Returns:
true if empty; false otherwise.

2.2.3.6 template<typename T> void Vector< T >::insert (const unsigned *i*, const T & *object*) throw (bad_alloc, OutOfBounds)

Inserts the object at the given position. Increases capacity if necessary.

Parameters:
i the position of insertion.

object the object to be inserted.

Exceptions:
bad_alloc if memory cannot be allocated.

OutOfBounds if index parameter is out of bounds.

2.2.3.7 template<typename T> const Vector< T > & Vector< T >::operator= (const Vector< T > & *v*) throw (bad_alloc)

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

Parameters:
vector the vector to be copied.

Returns:
a copy of the vector.

Exceptions:
bad_alloc if memory cannot be allocated.

2.2.3.8 template<typename T> T & Vector< T >::operator[] (const unsigned *i*) const throw (VectorEmpty, OutOfBounds)

Returns a reference to the object stored at a given position in the vector.

Parameters:
i the object's location.

Returns:
a reference to the object.

Exceptions:

VectorEmpty if vector is empty.

OutOfBounds if index parameter is out of bounds.

2.2.3.9 template<typename T> void Vector< T >::push_back (const T & *object*) throw (bad_alloc)

Adds the object to the end of the vector. Increases capacity if necessary.

Parameters:

object the object to be added to the end of the vector.

Exceptions:

bad_alloc if memory cannot be allocated.

2.2.3.10 template<typename T> void Vector< T >::remove (const unsigned *i*) throw (VectorEmpty, OutOfBounds)

Removes the object stored in the given position.

Parameters:

i the position of removal.

Exceptions:

VectorEmpty if vector is empty.

OutOfBounds if index parameter is out of bounds.

2.2.3.11 template<typename T> void Vector< T >::resize (const unsigned *newSize*, const T & *v* = T()) throw (bad_alloc)

Resizes the vector to its new size. After allocating new memory and copy the contents of old memory, stores the value in any unassigned elements.

Parameters:

newSize the new size of the vector.

v the value for any new, unassigned elements.

Exceptions:

bad_alloc if memory cannot be allocated.

2.2.3.12 template<typename T> unsigned Vector< T >::size () const

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

Returns:

an unsigned integer indicating the vector's size.

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

- vector.h