The Doubly-Linked List Library

0.2.1

Generated by Doxygen 1.5.8

Mon Jan 24 01:12:46 2011

Contents

Chapter 1

C Data Structure Library: Doubly-Linked List Library

Version:

0.2.1

Author:

Jun Woong (woong.jun at gmail.com)

Date:

last modified on 2011-01-24

1.1 Introduction

This document specifies the Double-Linked List Library which belongs to the C Data Structure Library. The basic structure is from David Hanson's book, "C Interfaces and Implementations." I modified the original implementation to make it more appropriate for my other projects, to speed up operations and to enhance its readibility; for example a prefix is used more strictly in order to avoid the user namespace pollution.

Since the book explains its design and implementation in a very comprehensive way, not to mention the copyright issues, it is nothing but waste to repeat it here, so I finish this document by giving introduction to the library; how to use the facilities is deeply explained in files that define them.

The Doubly-Linked List Library reserves identifiers starting with `dlist_` and `DLIST_`, and imports the Assertion Library (which requires the Exception Handling Library) and the Memory Management Library.

1.2 How to Use The Library

The Doubly-Linked List Library is a typical implementation of a list in which nodes have two pointers to their next and previous nodes; a list with a unidirectional pointer is implemented in the List Library. The storage used to maintain a list itself is managed by the library, but any storage allocated for data stored in nodes should be managed by a user program.

Similarly for other data structure libraries, use of the Doubly-Linked List Library follows this sequence: create, use and destroy. Except for functions to inspect lists, all other functions do one of them in various ways.

As oppsed to a singly-linked list, a doubly-linked list enables its nodes to be accessed randomly. To speed up such accesses, the library is revised from the original version so that a list remembers which node was last accessed. If a request is made to access a node that is next or previous to the remembered node, the library locates it starting from the remembered node. This is from observation that traversing a list from the head or the tail in sequence occurs frequently in many programs and can make a program making heavy use of lists run almost 3 times faster. Therefore, for good performance of your program, it is highly recommended that lists are traversed sequentially whenever possible. Do not forget that the current implementation requires for other types of accesses (that is, any access to a node that is not immediately next or previous to a remembered node) the library to locate the desired node from the head or the tail.

1.3 Boilerplate Code

Using a list starts with creating one. The simplest way to do it is to call dlist_new(). dlist_new() returns an empty list, and if it fails to allocate storage for the list, an exception `mem_exceptfail` is raised rather than returning a null pointer. All functions that allocate storage signals a shortage of memory via the exception; no null pointer returned. There is another function to create a list: dlist_list() that accepts a sequence of data and creates a list containing them in each node.

Once a list has been created, a new node can be inserted in various ways (dlist_add(), dlist_addhead() and dlist_addtail()) and an existing node can be removed from a list also in various ways (dlist_remove(), dlist_remhead() and dlist_remtail()). You can inspect the data of a node (dlist_get()) or replace it with new one (dlist_put()). In addition, you can find the number of nodes in a list (dlist_length()) or can rotate (or shift) a list (dlist_shift()). For an indexing sheme used when referring to an existing node, see dlist_get(). For that used when referring to a position into which a new node inserted, see dlist_add().

dlist_free() destroys a list that is no longer necessary, but note that any storage that is allocated by a user program does not get freed with it; dlist_free() only returns back the storage allocated by the library.

As an example, the following code creates a list and stores input characters into each node until `EOF` encountered. After read, it copies characters in nodes to continuous storage area to construct a string and prints the string.

```
int c;
```

```
char *p, *q;
dlist_t *mylist;

mylist = dlist_new();

while ((c = getc(stdin)) != EOF) {
    MEM_NEW(p);
    *p = c;
    dlist_addtail(mylist, p);
}

n = dlist_length(mylist);

p = MEM_ALLOC(n+1);
for (i = 0; i < n; i++) {
    p = dlist_get(mylist, i);
    q[i] = *p;
    MEM_FREE(p);
}
q[i] = '\0';

dlist_free(&mylist);

puts(q);
```

where MEM_NEW(), MEM_ALLOC() and MEM_FREE() come from the Memory Management Library.

Note that, before adding a node to a list, unique storage to contain a character is allocated with MEM_NEW() and this storage is returned back by MEM_FREE() while copying characters into an allocated array.

1.4 Future Directions

No future change on this library planned yet.

1.5 Contact Me

Visit http://project.woong.org to get the lastest version of this library. Only a small portion of my homepage (http://www.woong.org) is maintained in English, thus one who is not good at Korean would have difficulty when navigating most of other pages served in Korean. If you think the information you are looking for is on pages written in Korean you cannot read, do not hesitate to send me an email asking for help.

Any comments about the library are welcomed. If you have a proposal or question on the library just email me, and then I will reply as soon as possible.

1.6 Copyright

Chapter 2

File Index

2.1 File List

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

Chapter 3

File Documentation

3.1 dlist.c File Reference

Source for Doubly-Linked List Library.

```
#include <limits.h>
#include <stddef.h>
#include <stdarg.h>
#include "cbl/assert.h"
#include "cbl/memory.h"
#include "dlist.h"
```

Include dependency graph for dlist.c:

Data Structures

- struct **dlist_t**
- struct **dlist_t::dlist_t::node**

Functions

- dlist_t ∗() dlist_new (void)

  creates an empty new list.

- dlist_t ∗() dlist_list (void ∗data,...)

constructs a new list using a given sequence of data.

- void() dlist_free (dlist_t **pdlist)
  destroys a list.
- long() dlist_length (const dlist_t *dlist)
  returns the length of a list.
- void *() dlist_get (dlist_t *dlist, long i)
  retrieves data stored in the `i-th` *node in a list.*
- void *() dlist_put (dlist_t *dlist, long i, void *data)
  replaces data stored in a node with new given data.
- void *() dlist_addtail (dlist_t *dlist, void *data)
  adds a node after the last node.
- void *() dlist_addhead (dlist_t *dlist, void *data)
  adds a new node before the head node.
- void *() dlist_add (dlist_t *dlist, long pos, void *data)
  adds a new node to a specified position in a list.
- void *() dlist_remove (dlist_t *dlist, long i)
  removes a node with a specific index from a list.
- void *() dlist_remtail (dlist_t *dlist)
  removes the last node of a list.
- void *() dlist_remhead (dlist_t *dlist)
  removes the first node from a list.
- void() dlist_shift (dlist_t *dlist, long n)
  shifts a list to right or left.

3.1.1 Detailed Description

Source for Doubly-Linked List Library.

3.1.2 Function Documentation

3.1.2.1 void*() dlist_add (dlist_t * *dlist*, long *pos*, void * *data*)

adds a new node to a specified position in a list.

dlist_add() inserts a new node to a position specified by `pos`. The position is interpreted as follows: (5 nodes assumed)

```
 1   2    3    4    5    6    positive position values
  +-+  +-+  +-+  +-+  +-+
  | |--| |--| |--| |--| |
  +-+  +-+  +-+  +-+  +-+
-5   -4   -3   -2   -1   0    non-positive position values
```

Non-positive positions are useful when to locate without knowing the length of a list. If a list is empty both 0 and 1 are the valid values for a new node. Note that for positive `pos` `pos` - (dlist_length()+1) gives a non-negative value for the same position, and for negative `pos` `pos` + (dlist_length()+1) gives a positive value for the same position.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list to which new node inserted

← ***pos*** position for new node

← ***data*** data for new node

Returns:

data for new node

Here is the call graph for this function:

3.1.2.2 void∗() dlist_addhead (dlist_t ∗ *dlist*, void ∗ *data*)

adds a new node before the head node.

dlist_addhead() inserts a new node before the head node; the new node will be the head node. dlist_addhead() is equivalent to dlist_add() with 1 given for the position.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list to which new node to be inserted

← ***data*** data for new node

Returns:

data for new node

Here is the call graph for this function:

Here is the caller graph for this function:

3.1.2.3 void*() dlist_addtail (dlist_t * *dlist*, void * *data*)

adds a node after the last node.

dlist_addtail() inserts a new node after the last node; the index for the new node will be N if there are N nodes before the insertion. If a list is empty, dlist_addtail() and dlist_addhead() do the same job. dlist_addtail() is equivalent to dlist_add() with 0 given for the position.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list to which new node to be inserted

← ***data*** data for new node

Returns:

data for new node

Here is the caller graph for this function:

3.1.2.4 void() dlist_free (dlist_t ** *pdlist*)

destroys a list.

dlist_free() destroys a list by deallocating storages for each node and for the list itself. After the call the list does not exist (do not confuse this with an empty list). If `pdlist` points to a null pointer, an assertion in dlist_free() fails; it's a checked error.

Possible exceptions: assert_exceptfail

Unchecked error: pointer to foreign data structure given for `plist`

Parameters:

↔ ***pdlist*** pointer to list to destroy

Returns:

nothing

3.1.2.5 void*() dlist_get (dlist_t * *dlist*, long *i*)

retrieves data stored in the `i-th` node in a list.

dlist_get() brings and return data in the `i-th` node in a list. The first node has the index 0 and the last has n-1 when there are n nodes in a list.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list from which data is to be retrieved

← ***i*** index for node

Returns:

data retrieved

3.1.2.6 long() dlist_length (const dlist_t * *dlist*)

returns the length of a list.

dlist_length() returns the length of a list, the number of nodes in it.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

← ***dlist*** list whose length to be returned

Returns:

length of list (non-negative)

3.1.2.7 dlist_t*() dlist_list (void * *data*, ...)

constructs a new list using a given sequence of data.

dlist_list() constructs a doubly-linked list whose nodes contain a sequence of data given as arguments; the first argument is stored in the head (first) node, the second argument in the second node, and so on. There should be a way to mark the end of the argument list, which the null pointer is for. Any argument following a null pointer argument is not invalid, but ignored.

Possible exceptions: mem_exceptfail

Unchecked errors: none

Warning:

Calling dlist_list() with one argument, a null pointer, is not treated as an error. Such a call request an empty list as calling dlist_new().

Parameters:

← ***data*** data to store in head node of new list

← **...** other data to store in new list

Returns:

new list containing given sequence of data

Here is the call graph for this function:

3.1.2.8 dlist_t*() dlist_new (void)

creates an empty new list.

dlist_new() creates an empty list and returns it for further use.

Possible exceptions: mem_exceptfail

Unchecked errors: none

Returns:

empty new list

Here is the caller graph for this function:

3.1.2.9 void*() dlist_put (dlist_t * *dlist*, long *i*, void * *data*)

replaces data stored in a node with new given data.

dlist_put() replaces the data stored in the `i-th` node with new given data and retrieves the old data. For indexing, see dlist_get().

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list whose data to be replaced

← ***i*** index for noded

← ***data*** new data for substitution

Returns:

old data

3.1.2.10 void*() dlist_remhead (dlist_t * *dlist*)

removes the first node from a list.

dlist_remhead() removes the first (head) node from a list. dlist_remhead() is equivalent to dlist_remove() with 0 for the position.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list from which first node to be removed

Returns:

data of deleted node

Here is the call graph for this function:

3.1.2.11 void*() dlist_remove (dlist_t * *dlist*, long *i*)

removes a node with a specific index from a list.

dlist_remove() removes the `i-th` node from a list. For indexing, see dlist_get().

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list from which node to be removed

← ***i*** index for node to remove

Returns:

data of removed node

3.1.2.12 void*() dlist_remtail (dlist_t * *dlist*)

removes the last node of a list.

dlist_remtail() removes the last (tail) node of a list. dlist_remtail() is equivalent to dlist_remove() with dlist_length()-1 for the index.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list from which last node to be removed

Returns:

data of deleted node

Here is the caller graph for this function:

3.1.2.13 void() dlist_shift (dlist_t * *dlist*, long *n*)

shifts a list to right or left.

dlist_shift() shifts a list to right or left according to the value of `n`. A positive value indicates shift to right; for example shift by 1 means to make the tail node become the head node. Similarly, a negative value indicates shift to left; for example shift by -1 means to make the head node become the tail node.

The absolute value of the shift distance specified by `n` should be equal to or less than the length of a list. For exmple, dlist_shift(..., 7) or dlist_shift(..., -7) is not allowed when there are only 6 nodes in a list. In such a case, dlist_shift(..., 6) or dlist_shift(..., -6) does not have any effect as dlist_shift(..., 0) does not.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Warning:

Note that it is a list itself that dlist_shift() shifts, not the head pointer of a list.

Parameters:

↔ ***dlist*** list to shift

← ***n*** direction and distance of shift

Returns:

nothing

3.2 dlist.h File Reference

Header for Doubly-Linked List Library (CDSL).

This graph shows which files directly or indirectly include this file:

Typedefs

- typedef struct dlist_t dlist_t
  represents a doubly-linked list.

Functions

list creating functions:

- dlist_t * dlist_new (void)
  creates an empty new list.
- dlist_t * dlist_list (void *,...)
  constructs a new list using a given sequence of data.

list destroying functions:

- void dlist_free (dlist_t **)
  destroys a list.

node adding/deleting functions:

- void * dlist_add (dlist_t *, long, void *)
  adds a new node to a specified position in a list.
- void * dlist_addhead (dlist_t *, void *)
  adds a new node before the head node.
- void * dlist_addtail (dlist_t *, void *)
  adds a node after the last node.
- void * dlist_remove (dlist_t *, long)

removes a node with a specific index from a list.

- void * dlist_remhead (dlist_t *)
  removes the first node from a list.
- void * dlist_remtail (dlist_t *)
  removes the last node of a list.

data/information retrieving functions:

- long dlist_length (const dlist_t *)
  returns the length of a list.
- void * dlist_get (dlist_t *, long)
  retrieves data stored in the `i-th` *node in a list.*
- void * dlist_put (dlist_t *, long, void *)
  replaces data stored in a node with new given data.

list handling functions:

- void dlist_shift (dlist_t *, long)
  shifts a list to right or left.

3.2.1 Detailed Description

Header for Doubly-Linked List Library (CDSL).

Documentation for Doubly-Linked List Library (CDSL).

3.2.2 Function Documentation

3.2.2.1 void* dlist_add (dlist_t * *dlist*, long *pos*, void * *data*)

adds a new node to a specified position in a list.

dlist_add() inserts a new node to a position specified by `pos`. The position is interpreted as follows: (5 nodes assumed)

```
 1   2    3    4    5    6    positive position values
  +-+  +-+  +-+  +-+  +-+
  | |--| |--| |--| |--| |
  +-+  +-+  +-+  +-+  +-+
-5   -4   -3   -2   -1   0    non-positive position values
```

Non-positive positions are useful when to locate without knowing the length of a list. If a list is empty both 0 and 1 are the valid values for a new node. Note that for positive

`pos pos` - (dlist_length()+1) gives a non-negative value for the same position, and for negative `pos pos` + (dlist_length()+1) gives a positive value for the same position.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list to which new node inserted
← ***pos*** position for new node
← ***data*** data for new node

Returns:

data for new node

Here is the call graph for this function:

3.2.2.2 void* dlist_addhead (dlist_t * *dlist*, void * *data*)

adds a new node before the head node.

dlist_addhead() inserts a new node before the head node; the new node will be the head node. dlist_addhead() is equivalent to dlist_add() with 1 given for the position.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list to which new node to be inserted
← ***data*** data for new node

Returns:

data for new node

Here is the call graph for this function:

Here is the caller graph for this function:

3.2.2.3 void* dlist_addtail (dlist_t * *dlist*, void * *data*)

adds a node after the last node.

dlist_addtail() inserts a new node after the last node; the index for the new node will be N if there are N nodes before the insertion. If a list is empty, dlist_addtail() and dlist_addhead() do the same job. dlist_addtail() is equivalent to dlist_add() with 0 given for the position.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list to which new node to be inserted

← ***data*** data for new node

Returns:

data for new node

Here is the caller graph for this function:

3.2.2.4 void dlist_free (dlist_t ** *pdlist*)

destroys a list.

dlist_free() destroys a list by deallocating storages for each node and for the list itself. After the call the list does not exist (do not confuse this with an empty list). If `pdlist` points to a null pointer, an assertion in dlist_free() fails; it's a checked error.

Possible exceptions: assert_exceptfail

Unchecked error: pointer to foreign data structure given for `plist`

Parameters:

↔ ***pdlist*** pointer to list to destroy

Returns:

nothing

3.2.2.5 void* dlist_get (dlist_t * *dlist*, long *i*)

retrieves data stored in the `i-th` node in a list.

dlist_get() brings and return data in the `i-th` node in a list. The first node has the index 0 and the last has n-1 when there are n nodes in a list.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list from which data is to be retrieved

← ***i*** index for node

Returns:

data retrieved

3.2.2.6 long dlist_length (const dlist_t * *dlist*)

returns the length of a list.

dlist_length() returns the length of a list, the number of nodes in it.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

← ***dlist*** list whose length to be returned

Returns:

length of list (non-negative)

3.2.2.7 dlist_t* dlist_list (void * *data*, ...)

constructs a new list using a given sequence of data.

dlist_list() constructs a doubly-linked list whose nodes contain a sequence of data given as arguments; the first argument is stored in the head (first) node, the second argument in the second node, and so on. There should be a way to mark the end of the argument list, which the null pointer is for. Any argument following a null pointer argument is not invalid, but ignored.

Possible exceptions: mem_exceptfail

Unchecked errors: none

Warning:

Calling dlist_list() with one argument, a null pointer, is not treated as an error. Such a call request an empty list as calling dlist_new().

Parameters:

← ***data*** data to store in head node of new list

← **...** other data to store in new list

Returns:

new list containing given sequence of data

Here is the call graph for this function:

3.2.2.8 dlist_t* dlist_new (void)

creates an empty new list.

dlist_new() creates an empty list and returns it for further use.

Possible exceptions: mem_exceptfail

Unchecked errors: none

Returns:

empty new list

Here is the caller graph for this function:

3.2.2.9 void* dlist_put (dlist_t * *dlist*, long *i*, void * *data*)

replaces data stored in a node with new given data.

dlist_put() replaces the data stored in the `i-th` node with new given data and retrieves the old data. For indexing, see dlist_get().

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list whose data to be replaced

← ***i*** index for noded

← ***data*** new data for substitution

Returns:

old data

3.2.2.10 void* dlist_remhead (dlist_t * *dlist*)

removes the first node from a list.

dlist_remhead() removes the first (head) node from a list. dlist_remhead() is equivalent to dlist_remove() with 0 for the position.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list from which first node to be removed

Returns:

data of deleted node

Here is the call graph for this function:

3.2.2.11 void* dlist_remove (dlist_t * *dlist*, long *i*)

removes a node with a specific index from a list.

dlist_remove() removes the `i-th` node from a list. For indexing, see dlist_get().

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list from which node to be removed

← ***i*** index for node to remove

Returns:

data of removed node

3.2.2.12 void* dlist_remtail (dlist_t * *dlist*)

removes the last node of a list.

dlist_remtail() removes the last (tail) node of a list. dlist_remtail() is equivalent to dlist_remove() with dlist_length()-1 for the index.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Parameters:

↔ ***dlist*** list from which last node to be removed

Returns:

data of deleted node

Here is the caller graph for this function:

3.2.2.13 void dlist_shift (dlist_t * *dlist*, long *n*)

shifts a list to right or left.

dlist_shift() shifts a list to right or left according to the value of `n`. A positive value indicates shift to right; for example shift by 1 means to make the tail node become the head node. Similarly, a negative value indicates shift to left; for example shift by -1 means to make the head node become the tail node.

The absolute value of the shift distance specified by `n` should be equal to or less than the length of a list. For exmple, dlist_shift(..., 7) or dlist_shift(..., -7) is not allowed when there are only 6 nodes in a list. In such a case, dlist_shift(..., 6) or dlist_shift(..., -6) does not have any effect as dlist_shift(..., 0) does not.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `dlist`

Warning:

Note that it is a list itself that dlist_shift() shifts, not the head pointer of a list.

Parameters:

↔ ***dlist*** list to shift

← ***n*** direction and distance of shift

Returns:

nothing

Index