The List Library

0.2.1

Generated by Doxygen 1.7.6.1

Tue Apr 23 2013 22:31:56

Contents

Chapter 1

C Data Structure Library: List Library

Version

0.2.1

Author

Jun Woong (woong.jun at gmail.com)

Date

last modified on 2013-04-23

1.1 Introduction

This document specifies the 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 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 List Library reserves identifiers starting with `list_` and `LIST_`, and imports the Assertion Library (which requires the Exception Handling Library) and the Memory - Management Library.

1.2 How to Use The Library

The List Library is a typical implementation of a list in which nodes have one pointer to their next nodes; a list with two pointers to its next and previous nodes is implemented

in the Doubly-Linked 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; the library provides functions to help it.

Similarly for other data structure libraries, use of the 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 doubly-linked list, a singly-linked list does not support random access, thus there are facilities to aid sequential access to a list: list_toarray(), list_map() and LIST_FOREACH(). These facilities help a user to convert a list to an array, call a user-defined function for each node in a list and traverse a list.

As always, if functions that should allocate storage to finish their job fail the allocation, an exception `mem_exceptfail` is raised rather than returning an error indicator like a null pointer.

The following paragraphs describe differences this library has when compared to other data structure libraries.

In general, pointers that library functions take and return point to descriptors for the data structure the library implements. Once an instance of the structure is created, the location of a desciptor for it remains unchanged until destroyed. This property does not hold for pointers that this library takes and returns. Those pointers in this library point to the head node of a list rather than a descriptor for it. Because it can be replaced as a result of operations like adding or removing a node, a user program is obliged to update the pointer variable it passed with a returned one. Functions that accept a list and return a modified list are list_push(), list_pop() and list_reverse().

A null pointer, which is considered invalid in other libraries, is a valid and only representation for an empty list. This means creating a null pointer of the list_t * type in effect creats an empty list. You can freely pass it to any functions in this library and they are guaranteed to work well with it. Because of this, functions to add data to a list can be considered to also create lists; invoking them with a null pointer gives you a list containing the given data. This includes list_list(), list_append(), list_push() and list_copy().

It is considered good to hide implementation details behind an abstract type with only interfaces exposed when designing and implementing a data structure. Exposing its implementation to users often brings nothing beneficial but unnecessary dependency on it. In this implementation, however, the author decided to expose its implementation since its merits trumph demerits; see the book for more discussion on this issue.

1.3 Boilerplate Code

Using a list starts with creating it. If you need just an empty list, declaring a variable of the list_t * type and making it a null pointer is enough. list_list(), list_append(), list_push and list_copy() also create a list by providing a null-terminated sequence of data for each node, combining two lists, pushing a node with a given data to a list and duplicating a list. As noted, you can use a null pointer as arguments for those functions.

Once a list has been created, a new node can be pushed (list_push()) and inspected

(list_pop()). list_pop() pops a node (that is, gets rid of a node with returning the data in it). If you need to handle a list as if it were an array, list_toarray() converts a list to a dynamically-allocated array. You can find the length of the resulting array by calling list_length() or specifying a value used as a terminator (a null pointer in most cases). A function, list_map() and a macro, LIST_FOREACH() also provide a way to access nodes in sequence. list_reverse() reverses a list, which is useful when it is necessary to repeatedly access a list in the reverse order.

list_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; list_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 prints the characters twice by traversing the list and converting it to an array. Since the last input character resides in the head node, the list behaves like a stack, which is the reason list_push() and list_pop() are named so. The list is then reversed and again prints the stored characters by popping nodes; since it is reversed, the order in which character are printed out differs from the former two cases.

```
int c;
int i;
char *p;
void *pv, **a;
list_t *mylist, *iter;

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

LIST_FOREACH(iter, mylist) {
    putchar(*(char *)iter->data);
}
putchar('\n');

a = list_toarray(mylist, NULL);
for (i = 0; a[i] != NULL; i++)
    putchar(*(char *)a[i]);
putchar('\n');
MEM_FREE(a);

mylist = list_reverse(mylist);

while (list_length(mylist) > 0) {
    mylist = list_pop(mylist, &pv);
    putchar(*(char *)pv);
    MEM_FREE(pv);
}
putchar('\n');

list_free(&mylist);
```

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

In this example, the storage for each node is returned back when popping nodes from

the list. If list_map() were used instead to free storage, a call like this:

```
list_map(mylist, mylistfree, NULL);
```

would be used, where a call-back function, mylistfree() is defined as follows:

```
void mylistfree(void **pdata, void *ignored)
{
    MEM_FREE(*pdata);
}
```

1.4 Future Directions

1.4.1 Circular Lists

Making lists circular enables appending a new node to them to be done in a constant time. The current implementation where the last nodes point to nothing makes list_append() take time proportional to the number of nodes in a list, which is, in other words, the time complexity of list_append() is O(N).

1.5 Contact Me

Visit http://code.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, do not hesitate to send me an email to ask for help.

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

1.6 Copyright

For the parts I added or modified, the following applies:

This package is a singly-linked list implementation by Jun Woong. The implementation was written so as to conform with the Standard C published by ISO 9899:1990 and ISO 9899:1999.

Chapter 2

Todo List

Global list_append (list_t *, list_t *)

Improvements are possible and planned:

- the time complexity of the current implementation is O(N) where N indicates the number of nodes in a list. With a circular list, where the next node of the last node set to the head, it is possible for both pushing and appending to be done in a constant time.

Global list_append (list_t *, list_t *)

Improvements are possible and planned:

- the time complexity of the current implementation is O(N) where N indicates the number of nodes in a list. With a circular list, where the next node of the last node set to the head, it is possible for both pushing and appending to be done in a constant time.

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

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

Chapter 5

Data Structure Documentation

5.1 list_t Struct Reference

represents a node in a list.

`#include <list.h>`

Collaboration diagram for list_t:

Data Fields

- void * data
- struct list_t * next

5.1.1 Detailed Description

represents a node in a list.

This implementation for a linked list does not employ a separate data structure for the head or tail node; see the implementation of the Doubly-Linked List Library for what this means. By imposing on its users the responsability to make sure that a list given to the library functions is appropriate for their tasks, it attains simpler implementation.

The detail of struct `list_t` is intentionally exposed to the users (as opposed to be hidden in an opaque type) because doing so is more useful. For example, a user does not need to complicate the code by calling, say, list_push() just in order to make a temporary list node. Declaring it as having the type of `list_t` (as opposed to `list_t *`) is enough. In addition, an `list_t` object can be embedded in a user-created data structure directly.

5.1.2 Field Documentation

5.1.2.1 void* list_t::data

pointer to data

5.1.2.2 struct list_t* list_t::next

pointer to next node

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

- list.h

Chapter 6

File Documentation

6.1 list.c File Reference

Source for List Library (CDSL)

`#include <stdarg.h> #include <stddef.h> #include "cbl/assert.h" #include "cbl/memory.h" #include "list.h"` Include dependency graph for list.c:

Functions

- list_t *() list_push (list_t *list, void *data)

  pushes a new node to a list.
- list_t *() list_list (void *data,...)

  constructs a new list using a sequence of data.
- list_t *() list_append (list_t *list, list_t *tail)

  appends a list to another.
- list_t *() list_copy (const list_t *list)

  duplicates a list.
- list_t *() list_pop (list_t *list, void **pdata)

  pops a node from a list and save its data (pointer) into a pointer object.
- list_t *() list_reverse (list_t *list)

  reverses a list.
- size_t() list_length (const list_t *list)

  counts the length of a list.
- void() list_free (list_t **plist)

  destroys a list.
- void() list_map (list_t *list, void apply(void **, void *), void *cl)

  calls a user-provided function for each node in a list.
- void **() list_toarray (const list_t *list, void *end)

  converts a list to an array.

6.1.1 Detailed Description

Source for List Library (CDSL)

6.1.2 Function Documentation

6.1.2.1 list_t*() list_append (list_t * *list*, list_t * *tail*)

appends a list to another.

list_append() combines two lists by appending `tail` to `list`, which makes the next pointer of the last node in `list` point to the first node of `tail`.

Possible exceptions: mem_exceptfail

Unchecked errors: foreign data structure given for `list` and `tail`

Warning

Do not forget that a null pointer is considered an empty list, not an error.

Parameters

`in,out`	*list*	list to which `tail` appended
`in`	*tail*	list to append to `list`

Returns

appended list whose (pointer) value should be the same as `list`

Todo Improvements are possible and planned:

- the time complexity of the current implementation is O(N) where N indicates the number of nodes in a list. With a circular list, where the next node of the last node set to the head, it is possible for both pushing and appending to be done in a constant time.

6.1.2.2 list_t*() list_copy (const list_t * *list*)

duplicates a list.

list_copy() creates a new list by copying nodes of `list`.

Possible exceptions: mem_exceptfail

Unchecked errors: foreign data structure given for `list`

Warning

Note that the data pointed by nodes in `list` are not duplicated. An empty list results in returning a null pointer, which means an empty list.

Parameters

`in`	*list*	list to duplicate

Returns

duplicated list

6.1.2.3 void() list_free (list_t ** *plist*)

destroys a list.

list_free() destroys a list by deallocating storages for each node. After the call, the list is empty, which means that it makes a null pointer. If `plist` points to a null pointer, list_free() does nothing since it is already empty.

Possible exceptions: none

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

Warning

Note that the type of `plist` is a double pointer to `list_t`.

Parameters

`in,out`	*plist*	pointer to list to destroy

Returns

nothing

6.1.2.4 size_t() list_length (const list_t * *list*)

counts the length of a list.

list_length() counts the number of nodes in `list`.

Possible exceptions: none

Unchecked errors: foreign data structure given for `list`

Parameters

`in`	*list*	list whose nodes counted

Returns

length of list

Here is the caller graph for this function:

6.1.2.5 list_t*() list_list (void * *data*, ...)

constructs a new list using a sequence of data.

list_list() constructs a 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 a 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 list_list() with one argument, a null pointer, is not treated as an error. Such a call requests an empty list, so returned; note that a null pointer is an empty list.

Parameters

in	*data*	data to store in head node of new list
in	...	other data to store in new list

Returns

new list containing given sequence of data

6.1.2.6 void() list_map (list_t * *list*, void *applyvoid* **, *void* *, void * *cl*)

calls a user-provided function for each node in a list.

For each node in a list, list_map() calls a user-provided callback function; it is useful when doing some common task for each node. The pointer given in `cl` is passed to the second parameter of a user callback function, so can be used as a communication channel between the caller of list_map() and the callback. Since the callback has the address of `data` (as opposed to the value of `data`) through the first parameter, it is free to change its content. One of cases where list_map() is useful is to deallocate storages given for `data` of each node. Differently from the original implementation, this library also provides a marco named LIST_FOREACH() that can be used in the similar situation.

Possible exceptions: none (user-provided function may raise some)

Unchecked errors: foreign data structure given for `list`, user callback function doing something wrong (see the warning below)

Warning

Be wraned that modification to a list like pushing and popping a node before finishing list_map() must be done very carefully and highly discouraged. For example, in a callback function popping a node from the same list list_map() is applying to may spoil subsequent tasks depending on which node list_map() is dealing with. It is possible to provide a safer version, but not done because such an operation is not regarded as appropriate to the list.

Parameters

`in,out`	*list*	list with which `apply` called
`in`	*apply*	user-provided function (callback)
`in`	*cl*	passing-by argument to `apply`

Returns

nothing

6.1.2.7 list_t*() list_pop (list_t * *list*, void ** *pdata*)

pops a node from a list and save its data (pointer) into a pointer object.

list_pop() copies a pointer value stored in the head node of `list` to a pointer object pointed to by `pdata` and pops the node. If the list is empty, list_pop() does nothing and just returns `list`. If `pdata` is a null pointer list_pop() just pops without saving any data.

Possible exceptions: none

Unchecked errors: foreign data structure given for `list`

Parameters

`in`	*list*	list from which head node popped
`out`	*pdata*	points to pointer into which data (pointer value) stored

Warning

The return value of list_pop() has to be used to update the variable for the list passed. list_pop() takes a list and returns a modified list.

Returns

list with its head node popped

6.1.2.8 list_t*() list_push (list_t * *list*, void * *data*)

pushes a new node to a list.

list_push() pushes a pointer value `data` to a list `list` with creating a new node. A null pointer given for `list` is considered to indicate an empty list, thus not treated as an error.

Possible exceptions: mem_exceptfail

Unchecked errors: foreign data structure given for `list`

Parameters

`in`	*list*	list to which `data` pushed
`in`	*data*	data to push into `list`

Warning

The return value of list_push() has to be used to update the variable for the list passed. list_push() takes a list and returns a modified list.

Returns

modified list

6.1.2.9 list_t*() list_reverse (list_t * *list*)

reverses a list.

list_reverse() reverses a list, which eventually makes its first node the last and vice versa.

Possible exceptions: none

Unchecked errors: foreign data structure given for `list`

Parameters

`in`	*list*	list to reverse

Warning

The return value of list_reverse() has to be used to update the variable for the list passed. list_reverse() takes a list and returns a reversed list.

Returns

reversed list

6.1.2.10 void**() list_toarray (const list_t * *list,* void * *end*)

converts a list to an array.

list_toarray() converts a list to an array whose elements correspond to the data stored in nodes of the list. This is useful when, say, sorting a list. A function like array_tolist() is not necessary because it is easy to construct a list scanning elements of an array, for example:

```
for (i = 0; i < ARRAY_SIZE; i++)
    list = list_push(list, array[i]);
```

The last element of the constructed array is assigned by `end`, which is a null pointer in most cases. Do not forget to deallocate the array when it is unnecessary.

Possible exceptions: mem_exceptfail

Unchecked errors: foreign data structure given for `list`

Warning

The size of an array generated for an empty list is not zero, since there is always an end-mark value.

Parameters

`in`	*list*	list to convert to array
`in`	*end*	end-mark to save in last element of array

Returns

array converted from list

Here is the call graph for this function:

6.2 list.h File Reference

Documentation for List Library (CDSL)

`#include <stddef.h>` Include dependency graph for list.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct list_t

  represents a node in a list.

Defines

- #define LIST_FOREACH(pos, list) for ((pos) = (list); (pos); (pos)=(pos)->next)

  iterates for each node of a list

Functions

list creating functions:

- list_t * list_list (void *,...)
  constructs a new list using a sequence of data.
- list_t * list_append (list_t *, list_t *)
  appends a list to another.
- list_t * list_push (list_t *, void *)
  pushes a new node to a list.
- list_t * list_copy (const list_t *)
  duplicates a list.

data/information retrieving functions:

- list_t * list_pop (list_t *, void **)
  pops a node from a list and save its data (pointer) into a pointer object.
- void ** list_toarray (const list_t *, void *)
  converts a list to an array.
- size_t list_length (const list_t *)
  counts the length of a list.

list destroying functions:

- void list_free (list_t **)
  destroys a list.

list handling functions:

- void **list_map** (list_t *, void(void **, void *), void *)
- list_t * list_reverse (list_t *)
  reverses a list.

6.2.1 Detailed Description

Documentation for List Library (CDSL) Header for List Library (CDSL)

6.2.2 Define Documentation

6.2.2.1 #define LIST_FOREACH(*pos, list*) for ((pos) = (list); (pos); (pos)=(pos)->next)

iterates for each node of a list

LIST_FOREACH() macro is useful when doing some task for every node of a list. For example, the following example deallocates storages for `data` of each node in a list named `list` and destroy `list` itself using list_free():

```
list_t *t;

LIST_FOREACH(t, list)
{
    MEM_FREE(t->data);
}
list_free(list);
```

The braces enclosing the call to MEM_FREE are optional in this case as you may omit them in ordinary iterative statements. After the loop, list is untouched and t becomes indeterminate (if the loop finishes without jumping out of it, it should be a null pointer). There are cases where LIST_FOREACH() is more convenient than list_map() but the latter is recommended.

Warning

Be wraned that modification to a list like pushing and popping a node before finishing the loop must be done very carefully and highly discouraged. For example, pushing a new node with t may invalidate the internal state of the list, popping a node with list may invalidate t thus subsequent tasks depending on which node t points to and so on. It is possible to provide a safer version of LIST_FOREACH() as done by Linux kernel's list implementation, but not done by this implementation for such an operation is not regarded as appropriate to the list.

6.2.3 Function Documentation

6.2.3.1 list_t* list_append (list_t * *list*, list_t * *tail*)

appends a list to another.

list_append() combines two lists by appending tail to list, which makes the next pointer of the last node in list point to the first node of tail.

Possible exceptions: mem_exceptfail

Unchecked errors: foreign data structure given for list and tail

Warning

Do not forget that a null pointer is considered an empty list, not an error.

Parameters

in,out	*list*	list to which tail appended
in	*tail*	list to append to list

Returns

appended list whose (pointer) value should be the same as list

Todo Improvements are possible and planned:

- the time complexity of the current implementation is O(N) where N indicates the number of nodes in a list. With a circular list, where the next node of the last node set to the head, it is possible for both pushing and appending to be done in a constant time.

6.2.3.2 list_t* list_copy (const list_t * *list*)

duplicates a list.

list_copy() creates a new list by copying nodes of `list`.

Possible exceptions: mem_exceptfail

Unchecked errors: foreign data structure given for `list`

Warning

Note that the data pointed by nodes in `list` are not duplicated. An empty list results in returning a null pointer, which means an empty list.

Parameters

`in`	*list*	list to duplicate

Returns

duplicated list

6.2.3.3 void list_free (list_t ** *plist*)

destroys a list.

list_free() destroys a list by deallocating storages for each node. After the call, the list is empty, which means that it makes a null pointer. If `plist` points to a null pointer, list_free() does nothing since it is already empty.

Possible exceptions: none

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

Warning

Note that the type of `plist` is a double pointer to `list_t`.

Parameters

`in,out`	*plist*	pointer to list to destroy

Returns

nothing

6.2.3.4 size_t list_length (const list_t * *list*)

counts the length of a list.

list_length() counts the number of nodes in `list`.

Possible exceptions: none

Unchecked errors: foreign data structure given for `list`

Parameters

in	*list*	list whose nodes counted

Returns

length of list

Here is the caller graph for this function:

6.2.3.5 list_t* list_list (void * *data,* ...)

constructs a new list using a sequence of data.

list_list() constructs a 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 a 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 list_list() with one argument, a null pointer, is not treated as an error. Such a call requests an empty list, so returned; note that a null pointer is an empty list.

Parameters

in	*data*	data to store in head node of new list
in	...	other data to store in new list

Returns

new list containing given sequence of data

6.2.3.6 list_t* list_pop (list_t * *list*, void ** *pdata*)

pops a node from a list and save its data (pointer) into a pointer object.

list_pop() copies a pointer value stored in the head node of `list` to a pointer object pointed to by `pdata` and pops the node. If the list is empty, list_pop() does nothing and just returns `list`. If `pdata` is a null pointer list_pop() just pops without saving any data.

Possible exceptions: none

Unchecked errors: foreign data structure given for `list`

Parameters

`in`	*list*	list from which head node popped
`out`	*pdata*	points to pointer into which data (pointer value) stored

Warning

The return value of list_pop() has to be used to update the variable for the list passed. list_pop() takes a list and returns a modified list.

Returns

list with its head node popped

6.2.3.7 list_t* list_push (list_t * *list*, void * *data*)

pushes a new node to a list.

list_push() pushes a pointer value `data` to a list `list` with creating a new node. A null pointer given for `list` is considered to indicate an empty list, thus not treated as an error.

Possible exceptions: mem_exceptfail

Unchecked errors: foreign data structure given for `list`

Parameters

`in`	*list*	list to which `data` pushed
`in`	*data*	data to push into `list`

Warning

The return value of list_push() has to be used to update the variable for the list passed. list_push() takes a list and returns a modified list.

Returns

modified list

6.2.3.8 list_t* list_reverse (list_t * *list*)

reverses a list.

list_reverse() reverses a list, which eventually makes its first node the last and vice versa.

Possible exceptions: none

Unchecked errors: foreign data structure given for `list`

Parameters

`in`	*list*	list to reverse

Warning

The return value of list_reverse() has to be used to update the variable for the list passed. list_reverse() takes a list and returns a reversed list.

Returns

reversed list

6.2.3.9 void** list_toarray (const list_t * *list*, void * *end*)

converts a list to an array.

list_toarray() converts a list to an array whose elements correspond to the data stored in nodes of the list. This is useful when, say, sorting a list. A function like array_tolist() is not necessary because it is easy to construct a list scanning elements of an array, for example:

```
for (i = 0; i < ARRAY_SIZE; i++)
    list = list_push(list, array[i]);
```

The last element of the constructed array is assigned by `end`, which is a null pointer in most cases. Do not forget to deallocate the array when it is unnecessary.

Possible exceptions: mem_exceptfail

Unchecked errors: foreign data structure given for `list`

Warning

The size of an array generated for an empty list is not zero, since there is always an end-mark value.

Parameters

in	*list*	list to convert to array
in	*end*	end-mark to save in last element of array

Returns

array converted from list

Here is the call graph for this function: