The Table Library

0.2.1

Generated by Doxygen 1.7.6.1

Tue Apr 23 2013 22:32:07

Contents

Chapter 1

C Data Structure Library: Table 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 Table Library which belongs to the C Data Structure Library. The basic structure is from David Hanson's book, "C Interfaces and Implementations." I modifies 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 Table Library reserves identifiers starting with `table_` and `TABLE_`, and imports the Assertion Library (which requires the Exception Handling Library) and the Memory Management Library.

1.2 How to Use The Library

The Table Library implements tables that are similar to arrays except that nothing is imposed on the type of an index to a specific node; it is also known as a "dictionary",

"associative array" or "map" in some languages. A table stores a key and its associated value, and its user is free to put a new key-value pair, to retrieve a value giving its key, to replace a value for a key with a new one, and to get rid of a key-value pair from a table. The storage used to maintain a table itself is managed by the library, but any storage allocated for data stored in tables should be managed by a user program.

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

table_new() that creates an empty table takes three unusual arguments. The first one is a hint for the expected length of the table it will create, and the other two are to specify user-defined functions that perform creation and comparison of hash values used to represent keys. Some important conditions that those functions have to satisfy are described in the function.

1.3 Boilerplate Code

Using a table starts with creating one using table_new(). As explained in table_new(), it is important to provide three arguments properly. If keys to a table are generated by the Hash Library, the second and third arguments can be granted null pointers, which lets internal default functions used for the table. table_new() allocates a storage necessary for a table and if no allocation is possible, an exception is raised instead of returning a failure indicator like a null pointer.

Once a table created, a key-value pair can be added to and removed from a table using table_put() and table_remove(). Adding a pair to a table also entails memory allocation, and thus an exception can be raised. table_get() takes a key and returns its relevant value if any, and table_length() gives the number of key-value pairs in a table, a.k.a. the length of a table.

There are two ways to apply some operations on every pair in a table; table_map() takes a user-defined function and calls it for each of key-value pairs, and table_toarray() converts a table into a dynamic array; the array converted from a table has keys in elements with even indices and values in those with odd ones. Storage for the generated array is allocated by the library (thus, an exception is possible again), but a user program is responsible for releasing the storage when the array is no longer necessary.

table_free() takes a table and releases the storage used to maintain it. Note that any storage allocated by a user program to contain or represent keys and values is not deallocated by the library.

As an example, the following code creates a table (whose expected length is set to 20 and keys are generated by the Hash Library), and uses it to compute the frequencies of different characters in the input. To explain details, it first inspects if the table already has an input character and its frequency in it. If found, the frequency is simply increased. Otherwise, storage to contain the frequency is allocated and put into the table after set properly.

(This is intended to just show an example for using a table; as you (and probably everyone who have learned to program in C) already know, an ordinary array is enough to print the frequencies of different characters that appear in the user input.)

```
int c;
char b;
int *pfrq;
table_t *mytab;
const char *key;
void **pa, **pb;

mytab = table_new(20, NULL, NULL);

while ((c = getchar()) != EOF) {
    b = c;
    key = hash_new(&b, 1);
    if ((pfrq = table_get(mytab, key))) == NULL) {
        MEM_NEW(pfrq);
        *pfrq = 0;
    }
    (*pfrq)++;
    table_put(mytab, key, pfrq);
}

pa = table_toarray(mytab, NULL);
for (pb = pa; *pb; pb += 2) {
    printf("%c: %d\n", *(char *)pb[0], *(int *)pb[1]);
    MEM_FREE(pb[1]);
}
MEM_FREE(pa);

hash_reset();
table_free(&mytab);
```

where hash_new() and hash_reset() come from the Hash Library, and MEM_NEW() and MEM_FREE() from the Memory Management Library.

Things to note include:

- storages for keys and values to be stored into a table should be prepared by a user program, not the library, thus releasing them properly is up to the user program (MEM_FREE() in the `for` loop above and hash_reset() release them);
- since a table has pointers to objects for the frequencies, not their values, increasing values in the objects effectively updates the table; and
- an array generated by table_toarray() has to be deallocated by a user code.

1.4 Future Directions

1.4.1 Retaining Sequence

The Table Library offers two functions that scan every key-value pair maintained in tables. These functions visit key-value pairs in order that is dependent on the internal structure of the table, which might force a user to sort them properly if certain sequence is desired. It would be thus helpful to have those functions to retain sequence in which key-value pairs are stored into tables.

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

ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2

Todo List

Global table_map (table_t *table, void apply(const void *key, void **value, void *cl), void *cl)

Improvements are possible and planned:

- it sometimes serves better to call a user callback for key-value pairs in order they are stored.

Global table_toarray (const table_t *, void *)

Improvements are possible and planned:

- it sometimes serves better to call a user callback for key-value pairs in order they are stored.

Global table_toarray (const table_t *, void *)

Improvements are possible and planned:

- it sometimes serves better to call a user callback for key-value pairs in order they are stored.

Chapter 3

File Index

3.1 File List

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

Chapter 4

File Documentation

4.1 table.c File Reference

Source for Table Library (CDSL)

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

Functions

- table_t *() table_new (int hint, int cmp(const void *, const void *), unsigned hash(const void *))

  creates a new table.
- void *() table_get (const table_t *table, const void *key)

  gets data for a key from a table.
- void *() table_put (table_t *table, const void *key, void *value)

  puts a value for a key to a table.
- size_t() table_length (const table_t *table)

  returns the length of a table.
- void() table_map (table_t *table, void apply(const void *key, void **value, void *cl), void *cl)

  calls a user-provided function for each key-value pair in a table.
- void *() table_remove (table_t *table, const void *key)

  removes a key-value pair from a table.
- void **() table_toarray (const table_t *table, void *end)

  converts a table to an array.
- void() table_free (table_t **ptable)

  destroys a table.

4.1.1 Detailed Description

Source for Table Library (CDSL)

4.1.2 Function Documentation

4.1.2.1 void() table_free (table_t ** *ptable*)

destroys a table.

table_free() destroys a table by deallocating the stroage for it and set a given pointer to the null pointer. As always, table_free() does not deallocate storages for values in the table, which must be done by a user.

Possible exceptions: assert_exceptfail

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

Parameters

`in,out`	*ptable*	pointer to table to destroy

Returns

nothing

4.1.2.2 void*() table_get (const table_t * *table,* const void * *key*)

gets data for a key from a table.

table_get() returns a value associated with a key.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `table`

Parameters

`in`	*table*	table in which key-value pair to be found
`in`	*key*	key for which value to be returned

Returns

value for given key or null pointer

Return values

non-null	value found
NULL	value not found

Warning

If the stored value was a null pointer, an ambiguous situation may occur.

4.1.2.3 size_t() table_length (const table_t * *table*)

returns the length of a table.

table_length() returns the length of a table which is the number of key-value pairs in a table.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `table`

Parameters

`in`	*table*	table whose length returned

Returns

length of table

4.1.2.4 void() table_map (table_t * *table*, void *applyconst void *key, void **value, void *cl*, void * *cl*)

calls a user-provided function for each key-value pair in a table.

For each key-value pair in a table, table_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 third parameter of a user callback function, so can be used as a communication channel between the caller of table_map() and the callback. Since the callback has the address of `value` (as opposed to the value of `value`) through the second parameter, it is free to change its content. A macro like LIST_FOREACH() provided in the List Library is not provided for a table.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `table`

Warning

The order in which a user-provided function is called for each key-value pair is unspecified.

Parameters

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

Returns

nothing

Todo Improvements are possible and planned:

- it sometimes serves better to call a user callback for key-value pairs in order they are stored.

4.1.2.5 table_t*() table_new (int *hint*, int *cmp*const void *, *const void *, unsigned *hash*const void *)

creates a new table.

table_new() creates a new table. It takes some information on a table it will create:

- `hint`: an estimate for the size of a table;
- `cmp`: a user-provided function for comparing keys;
- `hash`: a user-provided function for generating a hash value from a key

table_new() determines the size of the internal hash table kept in a table based on `hint`. It never restricts the number of entries one can put into a table, but a better estimate probably gives better performance.

A function given to `cmp` should be defined to take two arguments and to return a value less than, equal to or greater than zero to indicate that the first argument is less than, equal to or greater than the second argument, respectively.

A function given to `hash` takes a key and returns a hash value that is to be used as an index for the internal hash table in a table. If the `cmp` function returns zero (which means they are equal) for some two keys, the `hash` function must generate the same value for them.

If a null pointer is given for `cmp` or `hash`, the default comparison or hashing function is used; see defhashCmp() and defhashGen(), in which case keys are assumed to be hash strings generated by the Hash Library. An example follows:

```
table_t *mytable = table_new(hint, NULL, NULL);
int *pi, val1, val2;
...
table_put(hash_string("key1"), &val1);
table_put(hash_string("key2"), &val2);
pi = table_get(hash_string(key1));
assert(pi == &val1);
```

Possible exceptions: mem_exceptfail

Unchecked errors: invalid functions for `cmp` and `hash`

Parameters

in	*hint*	hint for size of hash table
in	*cmp*	user-defined comparison function
in	*hash*	user-defined hash generating function

Returns

new table created

4.1.2.6 void*() table_put (table_t * *table*, const void * *key*, void * *value*)

puts a value for a key to a table.

table_put() replaces an existing value for a key with a new one and returns the previous value. If there is no existing one, the value is saved for the key and returns a null pointer.

Note that both a key and a value are pointers. If values are, say, integers in an application, objects to contain them are necessary to put them into a table.

Possible exceptions: assert_exceptfail, mem_exceptfail

Unchecked errors: foreign data structure given for `table`

Parameters

`in,out`	*table*	table to which value to be stored
`in`	*key*	key for which value to be stored
`in`	*value*	value to store to table

Returns

previous value for key or null pointer

Return values

non-null	previous value
NULL	no previous value found

Warning

If the stored value was a null pointer, an ambiguous situation may occur.

4.1.2.7 void*() table_remove (table_t * *table*, const void * *key*)

removes a key-value pair from a table.

table_remove() gets rid of a key-value pair for a key from a table. Note that table_remove() does not deallocate any storage for the pair to remove, which must be done by an user.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `table`

Parameters

in,out	*table*	table from which key-value pair removed
in	*key*	key for which key-value pair removed

Returns

previous value for given key or null pointer

Return values

non-null	previous value for key
NULL	key not found

Warning

If the stored value is a null pointer, an ambiguous situation may occur.

4.1.2.8 void**() table_toarray (const table_t * *table*, void * *end*)

converts a table to an array.

table_toarray() converts key-value pairs stored in a table to an array. The last element of the constructed array is assigned by `end`, which is a null pointer in most cases. Do not forget deallocate the array when it is unnecessary.

The resulting array is consisted of key-value pairs: its elements with even indices have keys and those with odd indices have corresponding values. For example, the second element of a resulting array has a value corresponding to a key stored in the first element. Note that the end-marker given as `end` has no corresponding value element.

Possible exceptions: assert_exceptfail, mem_exceptfail

Unchecked errors: foreign data structure given for `table`

Warning

The size of an array generated from an empty table cannot be zero, since there is always an end-mark value.
As in table_map(), the order in which an array is created for each key-value pair is unspecified.

Parameters

in	*table*	table for which array generated
in	*end*	end-mark to save in last element of array

Returns

array generated from table

Todo Improvements are possible and planned:

- it sometimes serves better to call a user callback for key-value pairs in order they are stored.

4.2 table.h File Reference

Documentation for Table Library (CDSL)

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

Typedefs

- typedef struct table_t table_t

  represents a table.

Functions

table creating/destroying functions:

- table_t * **table_new** (int, int(const void *, const void *), unsigned(const void *))
- void table_free (table_t **)

  destroys a table.

data/information retrieving functions:

- size_t table_length (const table_t *t)

  returns the length of a table.
- void * table_put (table_t *, const void *, void *)

  puts a value for a key to a table.
- void * table_get (const table_t *, const void *)

  gets data for a key from a table.
- void * table_remove (table_t *, const void *)

  removes a key-value pair from a table.

table handling functions:

- void **table_map** (table_t *, void(const void *, void **, void *), void *)
- void ** table_toarray (const table_t *, void *)

  converts a table to an array.

4.2.1 Detailed Description

Documentation for Table Library (CDSL) Header for Table Library (CDSL)

4.2.2 Function Documentation

4.2.2.1 void table_free (table_t ** *ptable*)

destroys a table.

table_free() destroys a table by deallocating the stroage for it and set a given pointer to the null pointer. As always, table_free() does not deallocate storages for values in the table, which must be done by a user.

Possible exceptions: assert_exceptfail

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

Parameters

`in,out`	*ptable*	pointer to table to destroy

Returns

nothing

4.2.2.2 void* table_get (const table_t * *table,* const void * *key*)

gets data for a key from a table.

table_get() returns a value associated with a key.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `table`

Parameters

`in`	*table*	table in which key-value pair to be found
`in`	*key*	key for which value to be returned

Returns

value for given key or null pointer

Return values

non-null	value found
NULL	value not found

Warning

If the stored value was a null pointer, an ambiguous situation may occur.

4.2.2.3 size_t table_length (const table_t * *table*)

returns the length of a table.

table_length() returns the length of a table which is the number of key-value pairs in a table.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `table`

Parameters

`in`	*table*	table whose length returned

Returns

length of table

4.2.2.4 void* table_put (table_t * *table*, const void * *key*, void * *value*)

puts a value for a key to a table.

table_put() replaces an existing value for a key with a new one and returns the previous value. If there is no existing one, the value is saved for the key and returns a null pointer.

Note that both a key and a value are pointers. If values are, say, integers in an application, objects to contain them are necessary to put them into a table.

Possible exceptions: assert_exceptfail, mem_exceptfail

Unchecked errors: foreign data structure given for `table`

Parameters

`in,out`	*table*	table to which value to be stored
`in`	*key*	key for which value to be stored
`in`	*value*	value to store to table

Returns

previous value for key or null pointer

Return values

non-null	previous value
NULL	no previous value found

Warning

If the stored value was a null pointer, an ambiguous situation may occur.

4.2.2.5 void* table_remove (table_t * *table*, const void * *key*)

removes a key-value pair from a table.

table_remove() gets rid of a key-value pair for a key from a table. Note that table_remove() does not deallocate any storage for the pair to remove, which must be done by an user.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `table`

Parameters

`in,out`	*table*	table from which key-value pair removed
`in`	*key*	key for which key-value pair removed

Returns

previous value for given key or null pointer

Return values

non-null	previous value for key
NULL	key not found

Warning

If the stored value is a null pointer, an ambiguous situation may occur.

4.2.2.6 void** table_toarray (const table_t * *table*, void * *end*)

converts a table to an array.

table_toarray() converts key-value pairs stored in a table to an array. The last element of the constructed array is assigned by `end`, which is a null pointer in most cases. Do not forget deallocate the array when it is unnecessary.

The resulting array is consisted of key-value pairs: its elements with even indices have keys and those with odd indices have corresponding values. For example, the second element of a resulting array has a value corresponding to a key stored in the first element. Note that the end-marker given as `end` has no corresponding value element.

Possible exceptions: assert_exceptfail, mem_exceptfail

Unchecked errors: foreign data structure given for `table`

Warning

The size of an array generated from an empty table cannot be zero, since there is always an end-mark value.
As in table_map(), the order in which an array is created for each key-value pair is unspecified.

Parameters

in	*table*	table for which array generated
in	*end*	end-mark to save in last element of array

Returns

array generated from table

Todo Improvements are possible and planned:

- it sometimes serves better to call a user callback for key-value pairs in order they are stored.