# The Stack Library

0.2.1

Generated by Doxygen 1.5.8

Mon Jan 24 01:13:00 2011

# Contents

# Chapter 1

# C Data Structure Library: Stack 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 Stack 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 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 Stack Library reserves identifiers starting with `stack_` and `STACK_`, and imports the Assertion Library (which requires the Exception Handling Library) and the Memory Management Library.

## 1.2 How to Use The Library

The Stack Library is a typical implementation of a stack based on a liked list. Even if its implementation is very similar to the List Library, the implementation details are hidden behind an abstract type called `stack_t` because, unlike lists, revealing the implementation of a stack hardly brings benefit. The storage used to maintain a stack itself is managed by the library, but any storage allocated for data stored in a stack should be managed by a user program.

Similarly for other data structure libraries, use of the Stack Library follows this sequence: create, use and destroy.

If functions that allocate storage fail memory allocation, an exception `mem_exceptfail` is raised; therefore functions never return a null pointer.

## 1.3 Boilerplate Code

Using a list starts with creating it. There is only one function provided to create a new stack, stack_new(). Calling it returns a new and empty stack.

Once a stack has benn created, you can push data into or pop it from a stack using stack_push() and stack_pop(), respectively. Because popping an empty stack triggers an exception `assert_exceptfail`, calling stack_empty() is recommended to inspect if a stack is empty before applying stack_pop() to it.

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

As an example, the following code creates a stack and pushes input characters into it until `EOF` encountered. After that, it prints the characters by popping the characters and destroy the stack.

```
int c;
char *p;
stack_t *mystack;

mystack = stack_new();
while ((c = getc(stdin)) != EOF) {
    MEM_NEW(p);
    *p = c;
    stack_push(mystack, p);
}

while (!stack_empty(mystack)) {
    p = stack_pop(mystack);
    putchar(*p);
    MEM_FREE(p);
}
putchar('\n');

stack_free(&mystack);
```

where MEM_NEW() and MEM_FREE() come from the Memory Management Li-

brary.

Note that before invoking stack_pop(), the stack is checked whether empty or not by stack_empty() and that when popping characters, the storage allocated for them gets freed.

## 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 stack.c File Reference

Source for Stack Library (CDSL).

```
#include <stddef.h>
#include "cbl/assert.h"
#include "cbl/memory.h"
#include "stack.h"
```

Include dependency graph for stack.c:

### Data Structures

- struct **stack_t**
- struct **stack_t::stack_t::node**

### Functions

- stack_t *() stack_new (void)

  *creates a stack.*

- int() stack_empty (const stack_t *stk)

  *inspects if a stack is empty.*

- void() stack_push (stack_t *stk, void *data)
  *pushes data into a stack.*
- void *() stack_pop (stack_t *stk)
  *pops data from a stack.*
- void() stack_free (stack_t **stk)
  *destroys a stack.*

### 3.1.1 Detailed Description

Source for Stack Library (CDSL).

### 3.1.2 Function Documentation

#### 3.1.2.1 int() stack_empty (const stack_t * *stk*)

inspects if a stack is empty.

stack_empty() inspects if a given stack is empty.

Possible exceptions: assert_exceptfail

Unchecked erros: foreign data structure given for `stk`

**Parameters:**

← ***stk*** stack to inspect

**Returns:**

whether stack is empty or not

**Return values:**

***1*** empty

***0*** not empty

#### 3.1.2.2 void() stack_free (stack_t ** *stk*)

destroys a stack.

stack_free() deallocates all storages for a stack and set the pointer passed through `stk` to a null pointer. Note that stack_free() does not deallocate any storage for the data in the stack to destroy, which must be done by a user.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `stk`

**Warning:**

The storage allocated for data (whose address a stack's node possesses) is never touched; its allocation and deallocation is entirely up to the user.

**Parameters:**

↔ ***stk*** pointer to stack to destroy

**Returns:**

nothing

#### 3.1.2.3 stack_t*() stack_new (void)

creates a stack.

stack_new() creates a new stack and sets its relevant information to the initial.

Possible exceptions: mem_exceptfail

Unchecked errors: none

**Returns:**

created stack

#### 3.1.2.4 void*() stack_pop (stack_t * *stk*)

pops data from a stack.

stack_pop() pops data from a given stack. If the stack is empty, an exception is raised due to the assertion failure, so popping all data without knowing the number of nodes remained in the stack needs to use stack_empty() to decide when to stop.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `stk`

**Parameters:**

↔ ***stk*** stack from which data popped

**Returns:**

data popped from stack

#### 3.1.2.5 void() stack_push (stack_t * *stk*, void * *data*)

pushes data into a stack.

stack_push() pushes data into the top of a stack. There is no explicit limit on the maximum number of data that can be pushed into a stack.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign data structure given for `stk`

**Parameters:**

↔ ***stk*** stack into which given data pushed

← ***data*** data to push

**Returns:**

nothing

## 3.2 stack.h File Reference

Documentation for Stack Library (CDSL).

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

### Typedefs

- typedef struct stack_t stack_t
  *represents a stack.*

### Functions

**stack creating and destroying functions:**

- stack_t * stack_new (void)
  *creates a stack.*
- void stack_free (stack_t **)
  *destroys a stack.*

**data handling functions:**

- void stack_push (stack_t *, void *)
  *pushes data into a stack.*
- void * stack_pop (stack_t *)
  *pops data from a stack.*

**misc. functions:**

- int stack_empty (const stack_t *)
  *inspects if a stack is empty.*

### 3.2.1 Detailed Description

Documentation for Stack Library (CDSL).

Header for Stack Library (CDSL).

### 3.2.2 Function Documentation

#### 3.2.2.1 int stack_empty (const stack_t * *stk*)

inspects if a stack is empty.

stack_empty() inspects if a given stack is empty.

Possible exceptions: assert_exceptfail

Unchecked erros: foreign data structure given for `stk`

**Parameters:**

← ***stk*** stack to inspect

**Returns:**

whether stack is empty or not

**Return values:**

***1*** empty
***0*** not empty

#### 3.2.2.2 void stack_free (stack_t ** *stk*)

destroys a stack.

stack_free() deallocates all storages for a stack and set the pointer passed through `stk` to a null pointer. Note that stack_free() does not deallocate any storage for the data in the stack to destroy, which must be done by a user.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `stk`

**Warning:**

The storage allocated for data (whose address a stack's node possesses) is never touched; its allocation and deallocation is entirely up to the user.

**Parameters:**

↔ ***stk*** pointer to stack to destroy

**Returns:**

nothing

#### 3.2.2.3 stack_t* stack_new (void)

creates a stack.

stack_new() creates a new stack and sets its relevant information to the initial.

Possible exceptions: mem_exceptfail

Unchecked errors: none

**Returns:**

created stack

#### 3.2.2.4 void* stack_pop (stack_t * *stk*)

pops data from a stack.

stack_pop() pops data from a given stack. If the stack is empty, an exception is raised due to the assertion failure, so popping all data without knowing the number of nodes remained in the stack needs to use stack_empty() to decide when to stop.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `stk`

**Parameters:**

↔ ***stk*** stack from which data popped

**Returns:**

data popped from stack

#### 3.2.2.5 void stack_push (stack_t * *stk*, void * *data*)

pushes data into a stack.

stack_push() pushes data into the top of a stack. There is no explicit limit on the maximum number of data that can be pushed into a stack.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign data structure given for `stk`

**Parameters:**

↔ ***stk*** stack into which given data pushed

← ***data*** data to push

**Returns:**

nothing

# Index