LIFT - Lift Is not a Framework or Toolkit

Generated by Doxygen 1.8.6

Mon Oct 5 2015 16:28:10

Contents

Chapter 1

LIFT

LIFT is a collectiong of useful C modules.It isn't a framework or a Toolkit. Most modules are either self-sufficient (other than parts of C standard librarry, or, in some cases, parts of library for a given system (POSIX, Windows...), or depend on few other LIFT modules.

Chapter 2

lift-c

This is a bunch of useful C modules. The name of this kind-of-library, "LIFT", can be interpreted as a recursive acronym: Lift Is not a Framework or a Toolkit (a bunch of useful C modules).

Also, it is a slight pun on a well-known C++ library. This is C, we don't need a *boost* we just need a *lift*. :)

What is there

Well, there is the "master" include file `lift.h`, but, in most cases, you should just include the header of the module that you need.

Chapter 3

File Index

3.1 File List

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

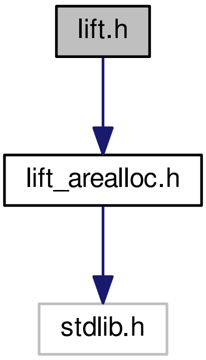
Chapter 4

File Documentation

4.1 lift.h File Reference

```
#include "lift_arealloc.h"
```

Include dependency graph for lift.h:

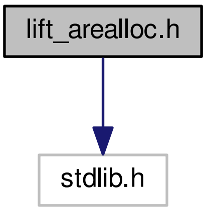


4.2 lift_arealloc.h File Reference

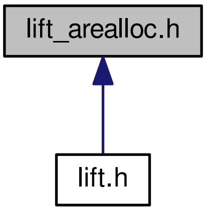
Safe alternative for realloc() for arrays.

```
#include <stdlib.h>
```

Include dependency graph for lift_arealloc.h:



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



Macros

- #define lift_arealloc(ptr, members) lift_arealloc_implementation(&(ptr),(members), sizeof ∗(ptr))

  This macro makes lif_arealloc_implementation() a lot easier to use and less error prone.

Functions

- void ∗ lift_arealloc_implementation (void ∗ptrptr, size_t members, size_t size)

  A safe alternative to realloc() for arrays.

4.2.1 Detailed Description

Safe alternative for realloc() for arrays. Part of LIFT, but can be used on its own - doesn't depend on anything from LIFT.

Author

Srdjan Veljkovic

Copyright

MIT License

4.2.2 Macro Definition Documentation

4.2.2.1 #define lift_arealloc(*ptr,* *members*) lift_arealloc_implementation(&(ptr),(members), sizeof ∗(ptr))

This macro makes lif_arealloc_implementation() a lot easier to use and less error prone.

It is a *good* macro, as it is very simple and doesn't evaluate its arguments more than once.

We fix two usability issues:

1. You may pass a pointer (to a value) instead of a pointer to pointer
2. You may pass a wrong (element) size

Here we accept a pointer, and you can't pass a value. You can, of course pass a pointer to pointer, but, that may be valid input, so we can't reject that.

The size of an element is deduced to be `sizeof *ptr`.

Parameters

ptr	The pointer to reallocate - it will be changed "in place", if need be.
members	The number of members of the new array

Returns

Pointer to the new array or NULL on failure to (re)allocate

Examples:

lift_arealloc_example.c.

4.2.3 Function Documentation

4.2.3.1 void∗ lift_arealloc_implementation (void ∗ *ptrptr,* size_t *members,* size_t *size*)

A safe alternative to realloc() for arrays.

It avoids the problems of overflow (`members *` may overflow) and "leaking" the previously allocated memory in case of failure. In case you're not aware of it, here is the offending code:

```
char *s = malloc(100);
s = realloc(s, 200);
```

If realloc() fails, `s` will now be NULL, and previously malloc()- ated memory is leaked, there is no way to free it now.

The only problem that lift_arealloc() doesn't solve is that passing an invalid pointer (not NULL or "really" allocated) results in undefined behavior.

Warning

Don't forget to pass the address of your pointer, rather than the pointer itself, even though the formal parameter type for is `void*`.

So, to fix the realloc() problem cited above, we would:

```
char *s = malloc(100);
if (NULL == lift_arealloc(&s, 200, sizeof(char)) {
    // handle reallocation failure, but `s` stayed the same
}
```

Note

> The downside is that may simply forget to pass the address of your pointer, and pass the pointer itself, and there is no way that we can detect that. Declaring `ptrptr` to be a `void**` would have actually been worse, as that would require cast if you want to avoid warnings (or even errors) for passing a pointer to, say, `int*`, instead of to `void*`. So, passing something like `3`, because you cast it to `void**` would not be detected.

To help with these usability issues, you should probably use lift_arealloc macro instead of this function.

Remarks

> On detecting overflow or any other invalid usage, it will *not* call realloc and will return NULL and set `errno` to ERANGE. If realloc() returns NULL, `ptrptr` will not be changed. Otherwise, the result of realloc() will be written to `*ptrptr`.

Parameters

`in,out`	*ptrptr*	Pointer to pointer to be reallocated. NULL is invalid. If not NULL, and other checks pass, `*ptrptr` will passed to realloc().
`in`	*members*	The number of members of the new array. If the result of multiply with `size` doesn't overflow, that result will be passed to realloc(). Also, if it or is 0, the function may fail.
`in`	*size*	Size of a member of the new array. If the result of multiply with `members` doesn't overflow, that result will be passed to realloc(). Also, if it or is 0, the function may fail.

Returns

> On internal or realloc() failure, will return NULL. Otherwise, will return the result of realloc().

Examples:

> lift_arealloc_example.c.

4.3 lift_free_and_null.h File Reference

A safer alternative to free().

Macros

- #define lift_nfree(ptr) lift_free_and_null(&(ptr)), sizeof *(ptr)

  This macro solves the usability problem with lift_free_and_null().

Functions

- void lift_free_and_null (void *ptrptr)

  An alternative / wrapper to free().

4.3.1 Detailed Description

A safer alternative to free().

Author

> Srdjan Veljkovic

Copyright

MIT license

4.3.2 Macro Definition Documentation

4.3.2.1 #define lift_nfree(*ptr*) lift_free_and_null(&(ptr)), sizeof *(ptr)

This macro solves the usability problem with lift_free_and_null().

It is a *good* macro, as it is simple and does not evaluate its argument more than once.

Here we expect a pointer and you can't pass a variable. Of course, you may pass a pointer to pointer, but that may be valid input.

There is an additional check - you can't pass a void pointer. That means that some strange, but valid code, will not compile. If you have such code, use lift_free_and_null(), but be careful.

Parameters

ptr	The pointer to free (previously allocated by malloc() or realloc()). It will be set to NULL "in place"

Returns

The size of what the `ptr` points to

Examples:

lift_free_and_null_example.c.

4.3.3 Function Documentation

4.3.3.1 void lift_free_and_null (void * *ptrptr*)

An alternative / wrapper to free().

It will NULL the pointer, not just free() it. Thus, you will not have a dangling pointer.

Passing NULL or a pointer to NULL (pointer) will simply be ignored. Otherwise, free() will be called on `*ptrptr` and then it will be NULL-ed.

Warning

You must pass the address of your pointer, not the pointer itself. Since `ptrptr` is of `void*`, it will not detect if you pass the pointer, and we shall have undefined behavior.

Remarks

The advantage of this function versus a pure macro implementation is that we avoid the problem of multiple evaluation in the macro. That should make it easier to find bugs with not passing address of a pointer (but the pointer itself).

We provida macro wrapper in lift_nfree, that solves this usability problem.

Remarks

Declaring `ptrptr` to be of `void **` type would be much worse, as to silence warnings (or maybe errors) one would need to cast to `void**` always, which would enable passing *anything*.

Parameters

`in,out`	*ptrptr*	Pointer to the pointer to free and NULL

Examples:

lift_free_and_null_example.c.

Chapter 5

Example Documentation

5.1 lift_arealloc_example.c

```c
#include "lift_arealloc.h"

#include <stdio.h>
#include <assert.h>


int main()
{
    int *v = NULL;
    if (NULL == lift_arealloc_implementation(&v, 4, sizeof *v)) {
        printf("Failed to allocate memory\n");
        return -1;
    }
    v[0] = v[1] = v[2] = v[3] = 4443;

    if (NULL == lift_arealloc_implementation(&v, 8, sizeof *v)) {
        printf("Failed to re-allocate memory\n");
    return -1;
    }
    if (NULL == lift_arealloc_implementation(&v, (size_t)~0, sizeof *v)) {
        printf("Failed to re-allocate memory, as expected\n");
    }
    v[4] = v[5] = v[6] = v[7] = 443;

    if (NULL == lift_arealloc(v, 6)) {
        printf("Failed to re-allocate memory\n");
    return -1;
    }
    if (NULL == lift_arealloc(v, (size_t)~0)) {
        printf("Failed to re-allocate memory, as expected\n");
    }

    assert(v[5] == 443);

    free(v);

    puts("lift_arealloc() example finished normally");

    return 0;
}
```

5.2 lift_free_and_null_example.c

```c
#include "lift_free_and_null.h"

#include <stdio.h>
#include <stdlib.h>


int main()
{
    char *s = malloc(100);
    printf("s = %p; after malloc()\n", s);
    free(s);
    printf("s = %p; after free()\n", s);
    s = malloc(1000);
    printf("s = %p; after another malloc()\n", s);
    lift_free_and_null(&s);
    printf("s = %p; after lift_free_and_null()\n", s);

    s = malloc(10000);
    printf("s = %p; after yet another malloc()\n", s);
    lift_nfree(s+2);
    printf("s = %p; after lift_nfree()\n", s);

    return 0;
}
```

Index