LIFT - Lift Is not a Framework or Toolkit

Generated by Doxygen 1.8.6

Mon Jan 25 2016 21:23:47

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 (depend only on parts of C standard library, or, in some cases, parts of standard library for a given system (POSIX, Windows...)), or depend on a few other LIFT modules.

For a reasonably type-safe realloc for arrays (and, in most cases you use realloc for arrays), check out lift_arealloc.h.

For a reasonably type-safe free that also NULL-ifies the pointer, check out lift_free_and_null.h.

For a reasonably type-safe and efficient generic vector, with an interface inspired by C++ STL, check out lift_vec.h.

For a minimalistic, type-safe and efficient generic list, with interface having no resemblense of C++ STL, check out lift_list.h.

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

Todo List

Global lift_vec_resize (var, newsize)
This doesn't handle all cases yet

Chapter 4

File Index

4.1 File List

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

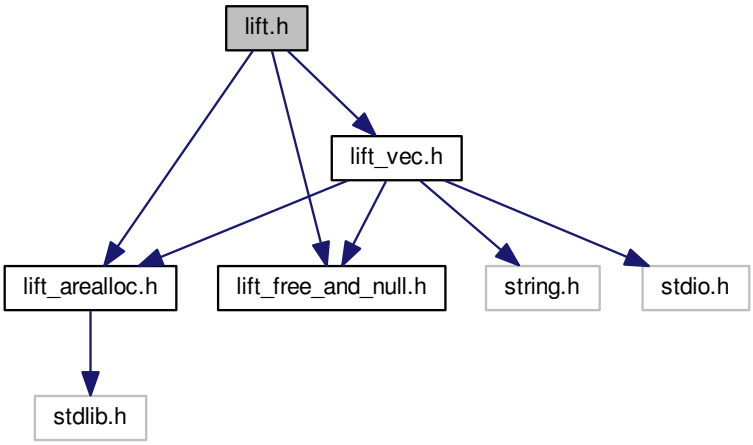
Chapter 5

File Documentation

5.1 lift.h File Reference

```
#include "lift_arealloc.h"
#include "lift_free_and_null.h"
#include "lift_vec.h"
```

Include dependency graph for lift.h:

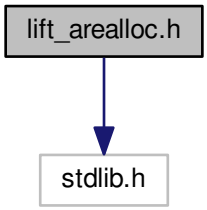


5.2 lift_arealloc.h File Reference

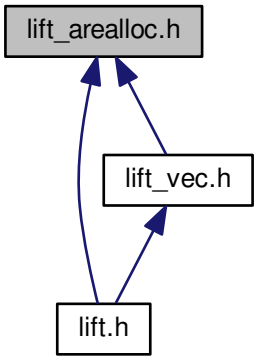
Safe alternative for realloc() for arrays.

```
#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.

5.2.1 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

5.2.2 Macro Definition Documentation

5.2.2.1 #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.

5.2.3 Function Documentation

5.2.3.1 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().

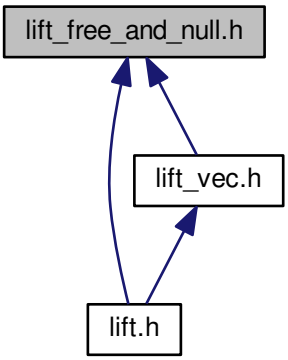
Examples:

lift_arealloc_example.c.

5.3 lift_free_and_null.h File Reference

A safer alternative to free().

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



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().

5.3.1 Detailed Description

A safer alternative to free().

Author

Srdjan Veljkovic

Copyright

MIT license

5.3.2 Macro Definition Documentation

5.3.2.1 #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.

5.3.3 Function Documentation

5.3.3.1 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

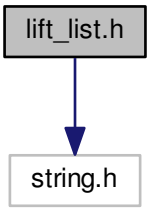
Examples:

lift_free_and_null_example.c.

5.4 lift_list.h File Reference

A doubly linked list generic type for C.

`#include <string.h>`
Include dependency graph for lift_list.h:



Macros

- #define LIFT_DECL_LIST(typ) struct LiftList_##typ { struct LiftList_##typ *next; struct LiftList_##typ *prev; typ data; }

  Declare a list (node) for the given type `typ`.
- #define LIFT_LIST_VAR(typ, var) LIFT_DECL_LIST(typ) var = { NULL, NULL }

  Declare a variable of the list (node) of type `typ`, with the name (symbol) `var`.
- #define LIFT_LIST_INIT(var) memset(&(var), 0, sizeof (var))

  Initializes a list (node) variable with symbol/name `var`.
- #define lift_list_next(var) ((var)->next)

  Gives an iterator (node pointer, really) to the next element of the list node `var`.
- #define lift_list_previous(var) ((var)->prev)

  Gives an iterator (node pointer, really) to the previous element of the list node `var`.
- #define lift_list_has_next(var) (lift_list_next(var) != NULL)

  Predicate giving indication whether there is an element after the given `var` list node.
- #define lift_list_has_previous(var) (lift_list_previous(var) != NULL)

  Predicate giving indication whether there is an element before the given `var` list node.
- #define lift_list_link_before(to_link, before_this) (to_link)->next = (before_this), (to_link)->prev = (before_this)->prev, (before_this)->prev ? (before_this)->prev->next = (to_link) : NULL, (before_this)->prev = (to_link)

  Links the element `to_link` before the `before_this` element, keeping any other elements in the list in the same order.
- #define lift_list_link_after(to_link, after_this) (to_link)->prev = (after_this), (to_link)->next = (after_this)->next, (after_this)->next ? (after_this)->next->prev = (to_link) : NULL, (after_this)->next = (to_link)

  Links the element `to_link` after the element, keeping any other elements in the list in the same order.
- #define lift_list_unlink(var) ((var)->next) ? (var)->next->prev = (var)->prev : NULL, ((var)->prev) ? (var)->prev->next = (var)->next : NULL, (var)->prev = (var)->next = NULL

  Unlinks the list node `var` from the list it is in.
- #define lift_list_unlink_safe(var, head_pointer) ((var) == *(head_pointer)) ? *(head_pointer) = (var)->next : NULL, lift_list_unlink(var)

  Unlinks the list node `var` from the list it is in, with the `head_pointer` being the pointer to the list it is in (may be pointer to `var`).
- #define lift_list_data(var) ((var).data)

  Gives the data of the list element `var`.
- #define LIFT_LIST_FOR_EACH(it, var) for (it = &(var); it != NULL; it = lift_list_next(it))

  A helper macro to execute a for-each loop on the list `var`, accessing by the iterator/pointer `it`.

- #define lift_list_find(var, val, rslt)

  A helper macro to search for the value `val` *in the list* `var`*.*
- #define lift_list_find_if(var, predicate, rslt)

  A helper macro to search for the element position in the list `var` *that satisfies the* `predicate`*.*

5.4.1 Detailed Description

A doubly linked list generic type for C. It is a macro implementation, but some care was taken to make it as safe as possible:

- No variables are declared in the macros themselves
- Most macros are expressions
- There is a reasonable amount of checking done...
- ...and most type errors will be caught, with maybe ugly compiler errors

The only significant problem with this implementation is that *a* lot of parameters in these macros a evaluated more than once. So, you have to be careful not to pass expressions with side effects.

This implementation goes for minimalism. One of the reasons for this is the macro implementation.

Hence, it doesn't do any memory management of the list elements at all. It's up to the user to allocate and deallocate the elements and take care of their lifetime.

This doesn't follow the C++ STL interface. Some other list implementation in LIFT might follow the STL interface, if a way can be found for the implementation to be of similar type safety and (macro) complexity.

Author

Srdjan Veljkovic

Copyright

MIT License

5.4.2 Macro Definition Documentation

5.4.2.1 #define LIFT_DECL_LIST(*typ*) struct LiftList_##typ { struct LiftList_##typ *next; struct LiftList_##typ *prev; typ data; }

Declare a list (node) for the given type `typ`.

A lot of the time you should `typedef` this.

Examples:

lift_list_example.c.

5.4.2.2 #define lift_list_find(*var, val, rslt*)

Value:

```
for (rslt = &(var); rslt != NULL; rslt = lift_list_next(rslt)) {     \
        if (rslt->data == (val)) {                                    \
            break;                                                    \
        }                                                             \
    }
```

A helper macro to search for the value `val` in the list `var`.

The result will be stored in the iterator/pointer `rslt`, that you must provide. On success, it will give a safe iterator/position at which the given value was found. On failure, it will give NULL.

Note

this will only work if the type of the elements that are stored in the list is such that its variables can be compared using the == operator

See Also

lift_list_find_if

Examples:

lift_list_example.c.

5.4.2.3 #define lift_list_find_if(*var, predicate, rslt*)

Value:

```
for (rslt = &(var); rslt != NULL; rslt = lift_list_next(rslt)) {    \
        if (predicate) {                                            \
            break;                                                  \
        }                                                           \
    }
```

A helper macro to search for the element position in the list `var` that satisfies the `predicate`.

The result will be stored in the iterator/pointer `rslt`, that you must provide. On success, it will give a safe iterator/position at which the given value was found. On failure, it will give NULL.

The `predicate` is an expression that can operate on pretty much anything, but is expected to operate on the `rslt`, which will hold the current iterator in the vector. Use like:

```
lift_list_find_if(v, iter->data == 55, iter); // list of integers
lift_list_find_if(v, iter->data->a == 55, iter); // list of structures
```

See Also

lift_list_find

Examples:

lift_list_example.c.

5.4.2.4 #define LIFT_LIST_FOR_EACH(*it, var*) for (it = &(var); it != NULL; it = lift_list_next(it))

A helper macro to execute a for-each loop on the list `var`, accessing by the iterator/pointer `it`.

You have to declare the variable `it`.

Examples:

lift_list_example.c.

5.4.2.5 #define lift_list_has_next(*var*) (lift_list_next(var) != NULL)

Predicate giving indication whether there is an element after the given `var` list node.

Examples:

lift_list_example.c.

5.4.2.6 #define lift_list_has_previous(*var*) (lift_list_previous(var) != NULL)

Predicate giving indication whether there is an element before the given `var` list node.

Examples:

lift_list_example.c.

5.4.2.7 #define LIFT_LIST_INIT(*var*) memset(&(var), 0, sizeof (var))

Initializes a list (node) variable with symbol/name `var`.

This is the default initalization, which initializes the data to all 0.

Warning

The data initializaiton may not be a good value for your type, so you may need to change it after executing this.

Examples:

lift_list_example.c.

5.4.2.8 #define lift_list_unlink(*var*) ((var)->next) ? (var)->next->prev = (var)->prev : NULL, ((var)->prev) ? (var)->prev->next = (var)->next : NULL, (var)->prev = (var)->next = NULL

Unlinks the list node `var` from the list it is in.

Has no effect if `var` is not in any list.

Warning

This may be a problem if `var` is the head of the list, as you may loose information of what element is the head after this. So, only use this if you know it is not a problem for you. Otherwise, use lift_list_unlink_safe()

Examples:

lift_list_example.c.

5.4.2.9 #define lift_list_unlink_safe(*var, head_pointer*) ((var) == *(head_pointer)) ? *(head_pointer) = (var)->next : NULL, lift_list_unlink(var)

Unlinks the list node `var` from the list it is in, with the `head_pointer` being the pointer to the list it is in (may be pointer to `var`).

Has no effect if `var` is not in any list.

If `var` is not in the list whose head is `head_pointer`, it will still unlink itself from the list, without giving any indication that the parameters are incorrect.

Examples:

lift_list_example.c.

5.4.2.10 #define LIFT_LIST_VAR(*typ, var*) LIFT_DECL_LIST(typ) var = { NULL, NULL }

Declare a variable of the list (node) of type `typ`, with the name (symbol) `var`.

Be aware that, while macros will work on this variable, it's type will not be the same as other variables declared with this macro in C99 and newer versions. To solve that, define a type with `typedef LIFT_DEL_LIST(typ)`, then a variable of such type and then use LIFT_LIST_INIT() to initialize it.

See Also

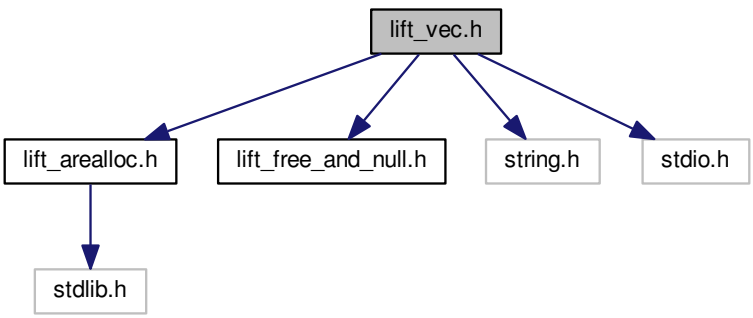
LIFT_LIST_INIT

5.5 lift_vec.h File Reference

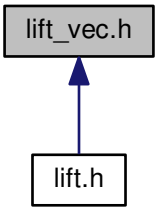
A vector generic type for C.

```
#include "lift_arealloc.h"
#include "lift_free_and_null.h"
#include <string.h>
#include <stdio.h>
```

Include dependency graph for lift_vec.h:



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



Macros

- #define LIFT_DECL_VEC(typ) struct { unsigned n; unsigned c; typ *p; }

  Declare a vector for the given type `typ`.
- #define LIFT_VEC_VAR(typ, var) LIFT_DECL_VECT(typ) = { 0, 0, NULL }

  Declare a variable of the vector of type `typ`, with the name (symbol) `var`.
- #define **LIFT_VEC_INIT**(var) memset(&(var), 0, sizeof (var))
- #define lift_vec_begin(var) ((var).p)

  Gives an iterator (pointer, really) to the begining of the vector `var`.
- #define lift_vec_end(var) ((var).p + (var).n)

  Gives an iterator (pointer, really) to the end of the vector `var`.
- #define **LIFT_VEC_VALID_ITERATOR**(var, iter) (((iter) >= lift_vec_begin(var)) && (((iter) <= lift_vec_end(var))))
- #define **LIFT_VEC_SAFE_ITERATOR**(var, iter) (((iter) >= lift_vec_begin(var)) && (((iter) < lift_vec_end(var))))
- #define lift_vec_size(var) ((var).n)

  Returns the number of elements in the vector `var`.
- #define lift_vec_resize(var, newsize)

  Resizes the vector `var` to have `newsize` elements.
- #define lift_vec_capacity(var) ((var).c)

  Returns the current capacity of the vectory `var`.
- #define lift_vec_empty(var) (0 == (var).n)

  Returns whether the vectory `var` is empty (has no elements)
- #define lift_vec_reserve(var, newcap)

  Reserves the memory for vectory `var` to have place for `newcap` elements.
- #define lift_vec_push_back(var, val)

  Pushes the value `val` to the end of vector `var`, increasing the vector's size.
- #define lift_vec_pop_back(var)

  Removes the last element of the vectory `var`, if the vector is not empty.
- #define lift_vec_insert(var, pos, val)

  Inserts the value `val` into the vector `var` at iterator/position `pos`.
- #define lift_vec_erase(var, pos)

  Erases the element of the vector `var` at iterator/position `pos`.
- #define lift_vec_free(var) lift_nfree((var).p), (var).n = (var).c = 0

  Once you're done with the vector, use this macro to free any resources (memory) that was allocated during its use.
- #define lift_vec_front(var) lift_vec_at(var, 0)

Allows you to set element at the front (beginning) of the vector.

- #define lift_vec_front_get(var) (lift_vec_at(var, 0))

  Returns the element at the front (beginning) of the vector.
- #define lift_vec_back(var) lift_vec_at((var), (var).n-1)

  Allows you to set element at the back of the vector (last element).
- #define lift_vec_back_get(var) (lift_vec_at(var, (var).n-1))

  Returns the element at the front (beginning) of the vector.
- #define lift_vec_data(var) (var).p

  Returns the data (array pointer) of the vector `var`, which you can pass to some function that expects such things.
- #define LIFT_VEC_VALID(var) ((var).p != NULL)

  Returns whether the vector `var` is valid.
- #define LIFT_VEC_SAFE(var, idx) (idx < lift_vec_size(var))

  Returns whether it is safe to access the element at index `idx` of the vector `var`.
- #define lift_vec_at(var, idx) assert(LIFT_VEC_SAFE(var,idx)), lift_vec_begin(var)[idx]

  Gives the element at index `idx` of the vector `var`, checking if it is safe to access it.
- #define lift_vec_get(var, idx) (lift_vec_at(var,idx))

  Returns the element at index `idx` of the vector `var`, checking if it is safe to access it.
- #define LIFT_VEC_FOR_EACH_IDX(idx, var) for (idx = 0; idx < lift_vec_size(var); ++idx)

  A helper macro to execute a for-each loop on the vector `var`, accessing by the index `idx`.
- #define LIFT_VEC_FOR_EACH_ITER(it, var) for (it = lift_vec_begin(var); it != lift_vec_end(var); ++it)

  A helper macro to execute a for-each loop on the vector `var`, accessing by the iterator/pointer `it`.
- #define lift_vec_find(var, val, rslt)

  A helper macro to search for the value `val` in the vector `var`.
- #define lift_vec_find_if(var, predicate, rslt)

  A helper macro to search for the element position in the vector `var` that satisfies the `predicate`.
- #define **lift_vec_bsearch**(var, val, rslt, aux)
- #define LIFT_PRINTF(var) printf("\"#var "' lift_vec: size=%d, capacity=%d, data=%p\n", (var).n, (var).c, (var).p)

  A helper macro to printf-out the vector `var`.

5.5.1 Detailed Description

A vector generic type for C. It is modelled after the C++ (STL) vector. It is a macro implementation, but some care was taken to make it as safe as possible:

- No variables are declared in the macros themselves
- Most macros are expressions
- There is a reasonable amount of checking done...
- ...and most type errors will be caught, with maybe ugly compiler errors

The only significant problem with this implementation is that *a* lot of parameters in these macros a evaluated more than once. So, you have to be careful not to pass expressions with side effects.

Author

Srdjan Veljkovic

Copyright

MIT License

5.5.2 Macro Definition Documentation

5.5.2.1 #define LIFT_DECL_VEC(*typ*) struct { unsigned n; unsigned c; typ *p; }

Declare a vector for the given type `typ`.

A lot of the time you should `typedef` this.

Examples:

lift_vec_example.c.

5.5.2.2 #define lift_vec_back(*var*) lift_vec_at((var), (var).n-1)

Allows you to set element at the back of the vector (last element).

Use like: `lift_vec_end(v) = 43;`.

5.5.2.3 #define lift_vec_begin(*var*) ((var).p)

Gives an iterator (pointer, really) to the begining of the vector `var`.

This is the first element if vector is not empty. If the vector is empty, the only guarantee is that lift_vec_begin(v) == lift_vec_end(v).

Examples:

lift_vec_example.c.

5.5.2.4 #define lift_vec_bsearch(*var, val, rslt, aux*)

Value:

```
for (rslt = lift_vec_begin(var), aux = lift_vec_end(var); rslt < aux; ) { \
        void *probe_ =  rslt + (aux - rslt) / 2;                          \
        if (val == *(rslt + (aux - rslt) / 2)) {                          \
            rslt = probe_;                                                \
            break;                                                        \
        }                                                                 \
        else if (val < *(rslt + (aux - rslt) / 2)) {                      \
            aux = probe_;                                                 \
        }                                                                 \
        else {                                                            \
            rslt = (void*)((char*)probe_ + sizeof *rslt);                 \
        }                                                                 \
    }
```

5.5.2.5 #define lift_vec_end(*var*) ((var).p + (var).n)

Gives an iterator (pointer, really) to the end of the vector `var`.

This points to an element one past the last element of the vector, if vectory is not empty. If the vector is empty, the only guarantee is that lift_vec_begin(v) == lift_vec_end(v).

Examples:

lift_vec_example.c.

5.5.2.6 #define lift_vec_erase(*var, pos*)

Value:

```
(LIFT_VEC_SAFE_ITERATOR(var, pos) ?                                    \
     (                                                                 \
         memmove((pos), (pos) + 1, ((var).n - 1 - ((pos) - lift_vec_begin(var))) * sizeof *(
      var).p), \
         --(var).n,                                                    \
         (pos))                                                        \
     : NULL                                                            \
       )
```

Erases the element of the vector `var` at iterator/position `pos`.

The `pos` has to be a safe iterator - meaning, it can't be the "end". On success, decreases the size of the vector `var` and returns the iterator at which the value `val` was erased. On error, returns NULL.

Examples:

lift_vec_example.c.

5.5.2.7 #define lift_vec_find(*var, val, rslt*)

Value:

```
for (rslt = lift_vec_begin(var); rslt != lift_vec_end(var); ++rslt) { \
        if (*rslt == (val)) {                                          \
            break;                                                     \
        }                                                              \
    }
```

A helper macro to search for the value `val` in the vector `var`.

The result will be stored in the iterator/pointer `rslt`, that you must provide. On success, it will give a safe iterator/position at which the given value was found. On failure, it will give the end iterator.

Note

this will only work if the type of the elements that are stored in the vector is such that its variables can be compared using the == operator

See Also

lift_vec_find_if

Examples:

lift_vec_example.c.

5.5.2.8 #define lift_vec_find_if(*var, predicate, rslt*)

Value:

```
for (rslt = lift_vec_begin(var); rslt != lift_vec_end(var); ++rslt) { \
        if (predicate) {                                               \
            break;                                                     \
        }                                                              \
    }
```

A helper macro to search for the element position in the vector `var` that satisfies the `predicate`.

The result will be stored in the iterator/pointer `rslt`, that you must provide. On success, it will give a safe iterator/position at which the given value was found. On failure, it will give the end iterator.

The `predicate` is an expression that can operate on pretty much anything, but is expected to operate on the `rslt`, which will hold the current iterator in the vector. Use like:

```
lift_vec_find_if(v, *iter == 55, iter); // vector of integers
lift_vec_find_if(v, iter->a == 55, iter); // vector of some structures
```

See Also

lift_vec_find

Examples:

lift_vec_example.c.

5.5.2.9 #define LIFT_VEC_FOR_EACH_IDX(*idx, var*) for (idx = 0; idx < lift_vec_size(var); ++idx)

A helper macro to execute a for-each loop on the vector `var`, accessing by the index `idx`.

You have to declare the variable `idx`.

Examples:

lift_vec_example.c.

5.5.2.10 #define LIFT_VEC_FOR_EACH_ITER(*it, var*) for (it = lift_vec_begin(var); it != lift_vec_end(var); ++it)

A helper macro to execute a for-each loop on the vector `var`, accessing by the iterator/pointer `it`.

You have to declare the variable `it`.

Examples:

lift_vec_example.c.

5.5.2.11 #define lift_vec_free(*var*) lift_nfree((var).p), (var).n = (var).c = 0

Once you're done with the vector, use this macro to free any resources (memory) that was allocated during its use.

After a call to this, vector is invalid.

Examples:

lift_vec_example.c.

5.5.2.12 #define lift_vec_front(*var*) lift_vec_at(var, 0)

Allows you to set element at the front (beginning) of the vector.

Use like: `lift_vec_front(v) = 3;`.

5.5.2.13 #define lift_vec_insert(*var, pos, val*)

Value:

```
(LIFT_VEC_VALID_ITERATOR(var, pos) ?                                    \
     ((((var).n < (var).c) || lift_vec_reserve(var, ((var).c+1))) ?     \
      (memmove((pos)+1, (pos), ((var).n - ((pos) - lift_vec_begin(var))) * sizeof *(var).p),
      \
       *(pos) = val,                                                    \
       ++(var).n,                                                       \
       (pos))                                                           \
      : NULL)                                                           \
     : NULL                                                             \
       )
```

Inserts the value `val` into the vector `var` at iterator/position `pos`.

The `pos` has to be a valid pointer - which includes the "end" pointer (if you pass the end, you will insert at the end of the vector, just like lift_vec_push_back()). On success, increases the size of the vector `var` and returns the iterator at which the value `val` was inserted. On error, returns NULL.

Examples:

lift_vec_example.c.

5.5.2.14 #define lift_vec_pop_back(*var*)

Value:

```
(((var).n > 0) ?                                                        \
     (                                                                  \
         --(var).n,                                                     \
         lift_vec_end(var))                                             \
     : NULL                                                             \
       )
```

Removes the last element of the vectory `var`, if the vector is not empty.

If the vector is not empty, returns NULL.

Examples:

lift_vec_example.c.

5.5.2.15 #define lift_vec_push_back(*var, val*)

Value:

```
((((var).n < (var).c) || lift_vec_reserve(var, ((var).c+1))) ?      \
     (                                  \
         (var).p[(var).n++] = val,                                      \
         0)                                                             \
     : -1                                                               \
       )
```

Pushes the value `val` to the end of vector `var`, increasing the vector's size.

If successful, returns 0, otherwise -1.

Examples:

lift_vec_example.c.

5.5.2.16 #define lift_vec_reserve(*var, newcap*)

Value:

```
((((var).c < (newcap)) && lift_arealloc((var).p, (newcap)+1)) ?     \
     (                                                              \
         (var).c = (newcap),                                        \
         (newcap))                                                  \
     : 0                                                            \
        )
```

Reserves the memory for vectory `var` to have place for `newcap` elements.

5.5.2.17 #define lift_vec_resize(*var, newsize*)

Value:

```
(((var.n < newsize) && lift_vec_reserve(var, newsize)) ?        \
     (                                                          \
         (var).n = (newsize),                                   \
         (newsize))                                             \
     : 0                                                        \
        )
```

Resizes the vector `var` to have `newsize` elements.

If vector needs to be enlarged, the contents of the new elements is not defined.

Todo This doesn't handle all cases yet

5.5.2.18 #define LIFT_VEC_VAR(*typ, var*) LIFT_DECL_VECT(typ) = { 0, 0, NULL }

Declare a variable of the vector of type `typ`, with the name (symbol) `var`.

Be aware that, while macros will work on this variable, it's type will not be the same as other variables declared with this macro in C99 and newer versions. To solve that, define a type with `typedef LIFT_DEL_VEC(typ)`, then a variable of such type and then use LIFT_VEC_INIT() to initialize it.

See Also

LIFT_VEC_INIT

Chapter 6

Example Documentation

6.1 lift_arealloc_example.c

```
/* -*- c-file-style:"stroustrup"; indent-tabs-mode: nil -*- */
#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;
}
```

6.2 lift_free_and_null_example.c

```
/* -*- c-file-style:"stroustrup"; indent-tabs-mode: nil -*- */
#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);
    printf("s = %p; after lift_nfree()\n", s);

    return 0;
}
```

6.3 lift_list_example.c

```
/* -*- c-file-style:"stroustrup"; indent-tabs-mode: nil -*- */
#include "lift_list.h"

#include <assert.h>


int main()
{
    typedef LIFT_DECL_LIST(int) listint_t;

    listint_t l;
    listint_t l_2;
    listint_t l_tr;
    listint_t *lhead;
    int data;

    LIFT_LIST_INIT(l);
    LIFT_LIST_INIT(l_2);
    LIFT_LIST_INIT(l_tr);

    assert(!lift_list_has_next(&l));
    assert(!lift_list_has_previous(&l));
    assert(!lift_list_has_next(&l_2));
    assert(!lift_list_has_previous(&l_2));
    assert(!lift_list_has_next(&l_tr));
    assert(!lift_list_has_previous(&l_tr));

    lift_list_data(l) = 1;
    lift_list_data(l_2) = 2;
    lift_list_data(l_tr) = 3;
    assert(lift_list_data(l) == 1);
    assert(lift_list_data(l_2) == 2);
    assert(lift_list_data(l_tr) == 3);

    lift_list_link_before(&l_2, &l);
    assert(!lift_list_has_next(&l));
    assert(lift_list_has_previous(&l));
    assert(lift_list_has_next(&l_2));
    assert(!lift_list_has_previous(&l_2));
    assert(lift_list_next(&l_2) == &l);
    assert(lift_list_previous(&l) == &l_2);
    assert(lift_list_data(l) == 1);
    assert(lift_list_data(l_2) == 2);

    lift_list_unlink(&l_2);
    assert(lift_list_data(l) == 1);
    assert(lift_list_data(l_2) == 2);
    assert(!lift_list_has_next(&l));
    assert(!lift_list_has_previous(&l));
    assert(!lift_list_has_next(&l_2));
    assert(!lift_list_has_previous(&l_2));

    lift_list_link_after(&l_2, &l);
    assert(lift_list_has_next(&l));
    assert(!lift_list_has_previous(&l));
    assert(!lift_list_has_next(&l_2));
    assert(lift_list_has_previous(&l_2));
    assert(lift_list_previous(&l_2) == &l);
    assert(lift_list_next(&l) == &l_2);
    assert(lift_list_data(l) == 1);
    assert(lift_list_data(l_2) == 2);

    lift_list_unlink(&l_2);
    assert(lift_list_data(l) == 1);
    assert(lift_list_data(l_2) == 2);
```

```
    assert(!lift_list_has_next(&l));
    assert(!lift_list_has_previous(&l));
    assert(!lift_list_has_next(&l_2));
    assert(!lift_list_has_previous(&l_2));

    lhead = &l;
    lift_list_link_after(&l_2, &l);
    lift_list_unlink_safe(&l, &lhead);
    assert(lift_list_data(l) == 1);
    assert(lift_list_data(l_2) == 2);
    assert(!lift_list_has_next(&l));
    assert(!lift_list_has_previous(&l));
    assert(!lift_list_has_next(&l_2));
    assert(!lift_list_has_previous(&l_2));
    assert(lhead == &l_2);

    LIFT_LIST_FOR_EACH(lhead, l) {
        assert(lift_list_data(*lhead) == 1);
    }
    lift_list_find(l, 1, lhead);
    assert(lhead == &l);
    lift_list_find(l, 2, lhead);
    assert(lhead == NULL);
    lift_list_find_if(l, lift_list_data(*lhead) == 1, lhead);
    assert(lhead == &l);
    lift_list_find_if(l, lift_list_data(*lhead) == 2, lhead);
    assert(lhead == NULL);

    lift_list_link_after(&l_2, &l);
    data = 1;
    LIFT_LIST_FOR_EACH(lhead, l) {
        assert(lift_list_data(*lhead) == data);
        if (1 == data) {
            data = 2;
        }
    }
    lift_list_find(l, 1, lhead);
    assert(lhead == &l);
    lift_list_find(l, 2, lhead);
    assert(lhead == &l_2);
    lift_list_find(l, 3, lhead);
    assert(lhead == NULL);
    lift_list_find(l_2, 1, lhead);
    assert(lhead == NULL);
    lift_list_find(l_2, 2, lhead);
    assert(lhead == &l_2);
    lift_list_find(l_2, 3, lhead);
    assert(lhead == NULL);
    lift_list_find_if(l, lift_list_data(*lhead) == 1, lhead);
    assert(lhead == &l);
    lift_list_find_if(l, lift_list_data(*lhead) == 2, lhead);
    assert(lhead == &l_2);
    lift_list_find_if(l, lift_list_data(*lhead) == 3, lhead);
    assert(lhead == NULL);
    lift_list_find_if(l_2, lift_list_data(*lhead) == 1, lhead);
    assert(lhead == NULL);
    lift_list_find_if(l_2, lift_list_data(*lhead) == 2, lhead);
    assert(lhead == &l_2);
    lift_list_find_if(l_2, lift_list_data(*lhead) == 3, lhead);
    assert(lhead == NULL);

    lift_list_link_after(&l_tr, &l);
    assert(lift_list_has_next(&l));
    assert(!lift_list_has_previous(&l));
    assert(!lift_list_has_next(&l_2));
    assert(lift_list_has_previous(&l_2));
    assert(lift_list_has_next(&l_tr));
    assert(lift_list_has_previous(&l_tr));
    assert(lift_list_previous(&l_tr) == &l);
    assert(lift_list_next(&l) == &l_tr);
    assert(lift_list_previous(&l_2) == &l_tr);
    assert(lift_list_next(&l_tr) == &l_2);
    assert(lift_list_data(l) == 1);
    assert(lift_list_data(l_2) == 2);
    assert(lift_list_data(l_tr) == 3);

    data = 1;
    LIFT_LIST_FOR_EACH(lhead, l) {
        assert(lift_list_data(*lhead) == data);
        if (1 == data) {
            data = 3;
        }
        else if (3 == data) {
            data = 2;
        }
    }
```

```
    lift_list_unlink(&l_tr);
    assert(lift_list_has_next(&l));
    assert(!lift_list_has_previous(&l));
    assert(!lift_list_has_next(&l_2));
    assert(lift_list_has_previous(&l_2));
    assert(!lift_list_has_next(&l_tr));
    assert(!lift_list_has_previous(&l_tr));
    assert(lift_list_previous(&l_2) == &l);
    assert(lift_list_next(&l) == &l_2);
    assert(lift_list_data(l) == 1);
    assert(lift_list_data(l_2) == 2);
    assert(lift_list_data(l_tr) == 3);

    return 0;
}
```

6.4 lift_vec_example.c

```
/* -*- c-file-style:"stroustrup"; indent-tabs-mode: nil -*- */
#include "lift_vec.h"

#include <assert.h>


int main()
{
    size_t idx;
    typedef LIFT_DECL_VEC(int) vecint_t;

    vecint_t v;
    int *iter;
    LIFT_VEC_INIT(v);

    assert(!LIFT_VEC_VALID(v));
    assert(lift_vec_size(v) == 0);
    assert(lift_vec_empty(v));
    assert(lift_vec_begin(v) == lift_vec_end(v));

    LIFT_PRINTF(v);
    lift_vec_push_back(v,3);
    LIFT_PRINTF(v);
    assert(LIFT_VEC_VALID(v));
    assert(LIFT_VEC_SAFE(v, 0));
    assert(lift_vec_size(v) == 1);
    assert(!lift_vec_empty(v));
    assert(lift_vec_begin(v) + 1 == lift_vec_end(v));
    assert(3 == *lift_vec_begin(v));
    assert(3 == lift_vec_front_get(v));
    assert(3 == lift_vec_back_get(v));
    assert(3 == lift_vec_get(v, 0));

    LIFT_VEC_FOR_EACH_IDX(idx, v) {
        assert(idx == 0);
    }
    LIFT_VEC_FOR_EACH_ITER(iter, v) {
        assert(*iter == 3);
    }

    lift_vec_push_back(v,4);
    LIFT_PRINTF(v);
    assert(LIFT_VEC_VALID(v));
    assert(LIFT_VEC_SAFE(v, 1));
    assert(lift_vec_size(v) == 2);
    assert(!lift_vec_empty(v));
    assert(lift_vec_begin(v) + 2 == lift_vec_end(v));
    assert(3 == *lift_vec_begin(v));
    assert(3 == lift_vec_front_get(v));
    assert(4 == lift_vec_back_get(v));
    assert(3 == lift_vec_get(v, 0));
    assert(4 == lift_vec_get(v, 1));
    LIFT_VEC_FOR_EACH_IDX(idx, v) {
        assert((idx == 0) || (idx == 1));
    }
    idx = 0;
    LIFT_VEC_FOR_EACH_ITER(iter, v) {
        assert((idx == 0) || (idx == 1));
        switch (idx) {
        case 0: assert(*iter == 3); break;
        case 1: assert(*iter == 4); break;
        }
        ++idx;
    }
```

```
    lift_vec_push_back(v,5);
    LIFT_PRINTF(v);
    assert(LIFT_VEC_VALID(v));
    assert(LIFT_VEC_SAFE(v, 2));
    assert(lift_vec_size(v) == 3);
    assert(!lift_vec_empty(v));
    assert(lift_vec_begin(v) + 3 == lift_vec_end(v));
    assert(3 == *lift_vec_begin(v));
    assert(3 == lift_vec_front_get(v));
    assert(5 == lift_vec_back_get(v));
    assert(3 == lift_vec_get(v, 0));
    assert(4 == lift_vec_get(v, 1));
    assert(5 == lift_vec_get(v, 2));
    LIFT_VEC_FOR_EACH_IDX(idx, v) {
        assert((idx == 0) || (idx == 1) || (idx == 2));
    }
    idx = 0;
    LIFT_VEC_FOR_EACH_ITER(iter, v) {
        assert((idx == 0) || (idx == 1) || (idx == 2));
        switch (idx) {
        case 0: assert(*iter == 3); break;
        case 1: assert(*iter == 4); break;
        case 2: assert(*iter == 5); break;
        }
        ++idx;
    }

    lift_vec_pop_back(v);
    LIFT_PRINTF(v);
    assert(LIFT_VEC_VALID(v));
    assert(LIFT_VEC_SAFE(v, 1));
    assert(lift_vec_size(v) == 2);
    assert(!lift_vec_empty(v));
    assert(lift_vec_begin(v) + 2 == lift_vec_end(v));
    assert(3 == *lift_vec_begin(v));
    assert(3 == lift_vec_front_get(v));
    assert(4 == lift_vec_back_get(v));
    assert(3 == lift_vec_get(v, 0));
    assert(4 == lift_vec_get(v, 1));
    LIFT_VEC_FOR_EACH_IDX(idx, v) {
        assert((idx == 0) || (idx == 1));
    }
    idx = 0;
    LIFT_VEC_FOR_EACH_ITER(iter, v) {
        assert((idx == 0) || (idx == 1));
        switch (idx) {
        case 0: assert(*iter == 3); break;
        case 1: assert(*iter == 4); break;
        }
        ++idx;
    }

    lift_vec_push_back(v,6);
    LIFT_PRINTF(v);
    assert(LIFT_VEC_VALID(v));
    assert(LIFT_VEC_SAFE(v, 2));
    assert(lift_vec_size(v) == 3);
    assert(!lift_vec_empty(v));
    assert(lift_vec_begin(v) + 3 == lift_vec_end(v));
    assert(3 == *lift_vec_begin(v));
    assert(3 == lift_vec_front_get(v));
    assert(6 == lift_vec_back_get(v));
    assert(3 == lift_vec_get(v, 0));
    assert(4 == lift_vec_get(v, 1));
    assert(6 == lift_vec_get(v, 2));
    LIFT_VEC_FOR_EACH_IDX(idx, v) {
        assert((idx == 0) || (idx == 1) || (idx == 2));
    }
    idx = 0;
    LIFT_VEC_FOR_EACH_ITER(iter, v) {
        assert((idx == 0) || (idx == 1) || (idx == 2));
        switch (idx) {
        case 0: assert(*iter == 3); break;
        case 1: assert(*iter == 4); break;
        case 2: assert(*iter == 6); break;
        }
        ++idx;
    }

    lift_vec_insert(v, lift_vec_begin(v) + 2, 5);
    LIFT_PRINTF(v);
    assert(LIFT_VEC_VALID(v));
    assert(LIFT_VEC_SAFE(v, 3));
    assert(lift_vec_size(v) == 4);
    assert(!lift_vec_empty(v));
    assert(lift_vec_begin(v) + 4 == lift_vec_end(v));
    assert(3 == *lift_vec_begin(v));
```

```
    assert(3 == lift_vec_front_get(v));
    assert(6 == lift_vec_back_get(v));
    assert(3 == lift_vec_get(v, 0));
    assert(4 == lift_vec_get(v, 1));
    assert(5 == lift_vec_get(v, 2));
    assert(6 == lift_vec_get(v, 3));
    LIFT_VEC_FOR_EACH_IDX(idx, v) {
        assert(idx <= 3);
    }
    idx = 0;
    LIFT_VEC_FOR_EACH_ITER(iter, v) {
        assert(idx <= 3);
        switch (idx) {
        case 0: assert(*iter == 3); break;
        case 1: assert(*iter == 4); break;
        case 2: assert(*iter == 5); break;
        case 3: assert(*iter == 6); break;
        }
        ++idx;
    }

    lift_vec_erase(v, lift_vec_begin(v)+1);
    LIFT_PRINTF(v);
    assert(LIFT_VEC_VALID(v));
    assert(LIFT_VEC_SAFE(v, 2));
    assert(lift_vec_size(v) == 3);
    assert(!lift_vec_empty(v));
    assert(lift_vec_begin(v) + 3 == lift_vec_end(v));
    assert(3 == *lift_vec_begin(v));
    assert(3 == lift_vec_front_get(v));
    assert(6 == lift_vec_back_get(v));
    assert(3 == lift_vec_get(v, 0));
    assert(5 == lift_vec_get(v, 1));
    assert(6 == lift_vec_get(v, 2));
    LIFT_VEC_FOR_EACH_IDX(idx, v) {
        assert((idx == 0) || (idx == 1) || (idx == 2));
    }
    idx = 0;
    LIFT_VEC_FOR_EACH_ITER(iter, v) {
        assert((idx == 0) || (idx == 1) || (idx == 2));
        switch (idx) {
        case 0: assert(*iter == 3); break;
        case 1: assert(*iter == 5); break;
        case 2: assert(*iter == 6); break;
        }
        ++idx;
    }

    lift_vec_find(v, 5, iter);
    assert(iter == lift_vec_begin(v) + 1);
    assert(*iter == 5);
    lift_vec_find(v, 3, iter);
    assert(iter == lift_vec_begin(v));
    assert(*iter == 3);
    lift_vec_find(v, 6, iter);
    assert(iter == lift_vec_begin(v) + 2);
    assert(*iter == 6);
    lift_vec_find(v, 555, iter);
    assert(iter == lift_vec_end(v));

    lift_vec_find_if(v, *iter == 5, iter);
    assert(iter == lift_vec_begin(v) + 1);
    assert(*iter == 5);
    lift_vec_find_if(v, *iter == 555, iter);
    assert(iter == lift_vec_end(v));

    lift_vec_pop_back(v);
    lift_vec_pop_back(v);
    lift_vec_pop_back(v);
    assert(LIFT_VEC_VALID(v));
    assert(lift_vec_size(v) == 0);
    assert(lift_vec_empty(v));
    assert(lift_vec_begin(v) == lift_vec_end(v));

    lift_vec_free(v);
    assert(!LIFT_VEC_VALID(v));
    LIFT_PRINTF(v);

    return 0;
}
```

Index