m_list

Contents

access.c

```c
#include "m_list.h"

int
m_list_first(m_list* list, m_list_elem** out_first)
{
        if (list == NULL || out_first == NULL)
                return M_LIST_E_NULL;

        *out_first = list->first;
        return M_LIST_OK;
}

int
m_list_last(m_list* list, m_list_elem** out_last)
{
        if (list == NULL || out_last == NULL)
                return M_LIST_E_NULL;

        *out_last = list->last;
        return M_LIST_OK;
}

int
```

```c
m_list_nth(m_list* list, uint64_t n, m_list_elem** out_nth)
{
        uint64_t i;
        m_list_elem* runner;

        if (list == NULL || out_nth == NULL)
                return M_LIST_E_NULL;

        if (list->length <= n)
                return M_LIST_E_OUT_OF_BOUNDS;

        if (list->index != NULL) {
                *out_nth = list->index[n];
                return M_LIST_OK;
        }

        runner = list->first;
        for (i = 0; i < n; i++) {
                runner = runner->next;
                if (runner->next == NULL)
                        return M_LIST_E_NULL;
        }

        *out_nth = runner;
        return M_LIST_OK;
}
```

elem.c

```c
#include "m_list.h"

int
m_list_elem_data(m_list_elem* elem, void** out_data)
{
        if (elem == NULL || out_data == NULL)
                return M_LIST_E_NULL;

        *out_data = elem->data;
        return M_LIST_OK;
}

int
m_list_elem_data_size(m_list_elem* elem, size_t* out_size)
{
        if (elem == NULL || out_size == NULL)
                return M_LIST_E_NULL;

        *out_size = elem->size;
        return M_LIST_OK;
}
```

```c
int
m_list_elem_next(m_list_elem* elem, m_list_elem** out_next)
{
        if (elem == NULL || out_next == NULL)
                return M_LIST_E_NULL;

        *out_next = elem->next;
        return M_LIST_OK;
}

int
m_list_elem_prev(m_list_elem* elem, m_list_elem** out_prev)
{
        if (elem == NULL || out_prev == NULL)
                return M_LIST_E_NULL;

        *out_prev = elem->prev;
        return M_LIST_OK;
}
```

general.c

```c
#include <string.h>

#include "m_list.h"

int
m_list_init(m_list* list)
{
        if (list == NULL)
                return M_LIST_E_NULL;

        list->first = NULL;
        list->last = NULL;
        list->length = 0;
        list->index = NULL;

        return M_LIST_OK;
}

int
m_list_length(m_list* list, uint64_t* out_length)
{
        if (list == NULL || out_length == NULL)
                return M_LIST_E_NULL;

        *out_length = list->length;
        return M_LIST_OK;
}

int
```

```
m_list_copy(m_list* list_src, m_list* list_dst, uint8_t copy)
{
	m_list_elem* runner_src;
	m_list_elem* elem;

	if (list_src == NULL || list_dst == NULL)
		return M_LIST_E_NULL;

	runner_src = list_src->first;
	while (runner_src != NULL) {
		elem = malloc(sizeof(m_list_elem));
		elem->size = runner_src->size;

		if (runner_src->copy == M_LIST_COPY_DEEP && copy == M_LIST_COPY_DEEP) {
			elem->copy = copy;
			if (runner_src->data == NULL) {
				elem->data = NULL;
			} else {
				elem->data = malloc(runner_src->size);
				memcpy(elem->data, runner_src->data, runner_src->size);
			}
		}

		if (runner_src->copy == M_LIST_COPY_SHALLOW) {
			elem->data = runner_src->data;
			elem->copy = copy;
		}

		if (list_dst->first == NULL) {
			list_dst->first = elem;
			list_dst->last = elem;
			elem->next = NULL;
			elem->prev = NULL;
		} else {
			list_dst->last->next = elem;
			elem->prev = list_dst->last;
			elem->next = NULL;
			list_dst->last = elem;
		}

		list_dst->length++;
		runner_src = runner_src->next;
	}

	return M_LIST_OK;
}

int
m_list_error_string(int code, const char** out_error_string)
{
	static const char* error_strings[] = {
```

```
                "OK",
                "True",
                "False",
                "One of the objects is NULL",
                "Index out of bounds",
                "No such element is present in the list",
                "Unknown copy method",
                "Unknown insert location",
                "Unknown return code"
        };

        if (out_error_string == NULL)
                return M_LIST_E_NULL;

        if (code < 0 || code > M_LIST_E_MAX) {
                *out_error_string = NULL;
                return M_LIST_E_UNKNOWN_CODE;
        }

        *out_error_string = error_strings[code];
        return M_LIST_OK;
}
```

index.c

```
#include "m_list.h"

int
m_list_build_index(m_list* list)
{
        uint64_t i;
        m_list_elem* runner;

        if (list == NULL)
                return M_LIST_E_NULL;

        m_list_drop_index(list);

        list->index = malloc(sizeof(m_list_elem*) * (size_t)list->length);
        i = 0;
        runner = list->first;
        while (runner != NULL) {
                list->index[i] = runner;
                i++;
                runner = runner->next;
        }

        return M_LIST_OK;
}

int
```

```
m_list_drop_index(m_list* list)
{
        if (list == NULL)
                return M_LIST_E_NULL;

        if (list->index != NULL) {
                free(list->index);
                list->index = NULL;
        }

        return M_LIST_OK;
}
```

insert.c

```
#include <string.h>

#include "m_list.h"

int
m_list_insert(m_list* list,
              uint8_t loc,
              m_list_elem* ref,
              uint8_t copy,
              void* data,
              size_t size)
{
        m_list_elem* elem;

        if (list == NULL)
                return M_LIST_E_NULL;

        elem = malloc(sizeof(m_list_elem));
        elem->copy = copy;
        elem->next = NULL;
        elem->prev = NULL;
        elem->size = size;

        if (copy == M_LIST_COPY_DEEP) {
                if (data == NULL) {
                        elem->data = NULL;
                } else {
                        elem->data = malloc(size);
                        memcpy(elem->data, data, size);
                }
        } else if (copy == M_LIST_COPY_SHALLOW) {
                elem->data = data;
        } else {
                free(elem);
                return M_LIST_E_UNKNOWN_COPY;
        }
        if (ref == NULL) {
                if (list->length == 0) {
                        list->first = elem;
                        list->last = elem;
                        list->length = 1;
                        return M_LIST_OK;
                } else {
                        if (elem->copy == M_LIST_COPY_DEEP) {
                                free(elem->data);
                        }
                        free(elem);
                        return M_LIST_E_NULL;
                }
        }

        if (loc == M_LIST_INSERT_AFTER) {
                if (ref == list->last) {
                        elem->next = NULL;
                        elem->prev = ref;
                        ref->next = elem;
                        list->last = elem;
                } else {
                        elem->prev = ref;
                        elem->next = ref->next;
                        ref->next = elem;
                }
        } else if (loc == M_LIST_INSERT_BEFORE) {
                if (ref == list->first) {
                        ref->prev = elem;
                        elem->prev = NULL;
                        elem->next = ref;
                        list->first = elem;
                } else {
                        elem->prev = ref->prev;
                        elem->next = ref;
                        ref->prev = elem;
                }
        } else {
                if (elem->copy == M_LIST_COPY_DEEP) {
                        free(elem->data);
                }
                free(elem);
                return M_LIST_E_UNKNOWN_LOCATION;
        }

        list->length++;
        m_list_drop_index(list);

        return M_LIST_OK;
}
```

```c
int
m_list_append(m_list* list, uint8_t copy, void* data, size_t size)
{
	return m_list_insert(list, M_LIST_INSERT_AFTER, list->last, copy, data, size);
}

int
m_list_prepend(m_list* list, uint8_t copy, void* data, size_t size)
{
	return m_list_insert(list, M_LIST_INSERT_BEFORE, list->first, copy, data, size);
}

int
m_list_generate(m_list* list,
                uint8_t copy,
                void(*gen_fn)(uint64_t, void*, void**, size_t*),
                uint64_t n,
                void* payload)
{
	uint64_t i;
	void* data;
	size_t size;

	if (list == NULL || gen_fn == NULL)
		return M_LIST_E_NULL;

	if (n == 0)
		return M_LIST_OK;

	for (i = 0; i < n; i++) {
		gen_fn(i, payload, &data, &size);
		m_list_append(list, copy, data, size);
	}

	return M_LIST_OK;
}

int
m_list_concat(m_list* list_src, m_list* list_dst)
{
	if (list_src == NULL || list_dst == NULL)
		return M_LIST_E_NULL;

	if (list_dst->length == 0) {
		list_dst->first = list_src->first;
		list_dst->last = list_src->last;
	} else {
		list_dst->last->next = list_src->first;
		list_src->first->prev = list_dst->last;
	}
```

```
        list_dst->length += list_src->length;
        m_list_drop_index(list_dst);

        list_src->first = NULL;
        list_src->last = NULL;
        list_src->length = 0;
        m_list_drop_index(list_src);

        return M_LIST_OK;
}
```

m_list.h

```
#ifndef M_LIST_H
#define M_LIST_H

#include <stdlib.h>
#include <stdint.h>

typedef struct m_list_elem {
        struct m_list_elem* next;
        struct m_list_elem* prev;
        void* data;
        size_t size;
        uint8_t copy;
        char padding[sizeof(void*)-1];
} m_list_elem;

typedef struct m_list {
        m_list_elem* first;
        m_list_elem* last;
        uint64_t length;
        m_list_elem** index;
} m_list;

#define M_LIST_OK                  0
#define M_LIST_TRUE                1
#define M_LIST_FALSE               2
#define M_LIST_E_NULL              3
#define M_LIST_E_OUT_OF_BOUNDS     4
#define M_LIST_E_NOT_PRESENT       5
#define M_LIST_E_UNKNOWN_COPY      6
#define M_LIST_E_UNKNOWN_LOCATION 7
#define M_LIST_E_UNKNOWN_CODE      8
#define M_LIST_E_MAX               8

#define M_LIST_COPY_DEEP    0
#define M_LIST_COPY_SHALLOW 1

#define M_LIST_INSERT_AFTER  0
```

```c
#define M_LIST_INSERT_BEFORE 1

int m_list_init(m_list* list);
int m_list_length(m_list* list, uint64_t* out_length);
int m_list_copy(m_list* list_src, m_list* list_dst, uint8_t copy);
int m_list_error_string(int code, const char** out_error_string);

int m_list_insert(m_list* list, uint8_t loc, m_list_elem* ref, uint8_t copy, void* data, size_t size);
int m_list_append(m_list* list, uint8_t copy, void* data, size_t size);
int m_list_prepend(m_list* list, uint8_t copy, void* data, size_t size);
int m_list_generate(m_list* list, uint8_t copy, void(*gen_fn)(uint64_t, void*, void**, size_t*), uint64
int m_list_concat(m_list* list_src, struct m_list* list_dst);

int m_list_is_empty(m_list* list);
int m_list_remove(m_list* list, m_list_elem* elem);
int m_list_remove_safe(m_list* list, m_list_elem* elem);
int m_list_remove_first(m_list* list);
int m_list_remove_last(m_list* list);
int m_list_remove_all(m_list* list);

int m_list_first(m_list* list, m_list_elem** out_first);
int m_list_nth(m_list* list, uint64_t n, m_list_elem** out_elem);
int m_list_build_index(m_list* list);
int m_list_drop_index(m_list* list);
int m_list_last(m_list* list, m_list_elem** out_last);

int m_list_map(m_list* list, void(*fn)(void*, void*), void* payload);
int m_list_map_ex(m_list* list, void(*fn)(m_list_elem*, uint64_t, void*), void* payload);
int m_list_map2(m_list* list, void(*fn)(void*, void*, void*), void* payload1, void* payload2);

int m_list_join(m_list* list, uint8_t copy, void* data, size_t size);
int m_list_find(m_list* list, int(*fn)(void*, void*), void* key, void** output);
int m_list_filter(m_list* list, int(*fn)(void*, void*), void* payload);
int m_list_zip(m_list* list_a, m_list* list_b, void(*fn)(void*, void*, void*), void* payload);
int m_list_reverse(m_list* list);

int m_list_equal(m_list* list_a, m_list* list_b);

int m_list_match_all(m_list* list, int(*fn)(void*, void*), void* payload);
int m_list_match_any(m_list* list, int(*fn)(void*, void*), void* payload);
int m_list_match_exactly(m_list* list, int(*fn)(void*, void*), uint64_t count, void* payload);
int m_list_match_at_least(m_list* list, int(*fn)(void*, void*), uint64_t count, void* payload);

int m_list_is_sorted(m_list* list, int(*cmp_fn)(void*, void*));
int m_list_sort(m_list* list, int(*cmp_fn)(void*, void*, void*), void*);

int m_list_elem_data(m_list_elem* elem, void** out_data);
int m_list_elem_data_size(m_list_elem* elem, size_t* out_size);
int m_list_elem_next(m_list_elem* elem, m_list_elem** out_next);
int m_list_elem_prev(m_list_elem* elem, m_list_elem** out_prev);
```

```
#endif
```

remove.c

```
#include "m_list.h"

int
m_list_remove_all(m_list* list)
{
        m_list_elem* runner;

        for (runner = list->last; runner != NULL; ) {
                if (runner->copy == M_LIST_COPY_DEEP)
                        free(runner->data);

                runner = runner->prev;

                if (runner != NULL)
                        free(runner->next);
        }

        list->length = 0;
        list->first = NULL;
        list->last = NULL;
        m_list_drop_index(list);

        return M_LIST_OK;
}

static int
handle_edges(m_list* list, m_list_elem* elem)
{
        if (elem == list->first && elem == list->last) {
                list->first = NULL;
                list->last = NULL;
        } else if (elem == list->first) {
                list->first = elem->next;
                list->first->prev = NULL;
        } else if (elem == list->last) {
                list->last = elem->prev;
                list->last->next = NULL;
        } else {
                return 0;
        }

        return 1;
}

int
m_list_remove(m_list* list, m_list_elem* elem)
{
        if (list->length == 0)
                return M_LIST_E_NOT_PRESENT;

        if (!handle_edges(list, elem)) {
                elem->prev->next = elem->next;
                elem->next->prev = elem->prev;
        }

        if (elem->copy == M_LIST_COPY_DEEP)
                free(elem->data);
        free(elem);

        list->length--;
        m_list_drop_index(list);

        return M_LIST_OK;
}

int
m_list_remove_safe(m_list* list, m_list_elem* elem)
{
        m_list_elem* runner;
        uint8_t found;

        found = 0;

        if (list->length == 0)
                return M_LIST_E_NOT_PRESENT;

        if (!handle_edges(list, elem)) {
                for (runner = list->first; runner != NULL; runner = runner->next) {
                        if (elem == runner) {
                                found = 1;
                                break;
                        }
                }
        }

        if (found) {
                if (elem->copy == M_LIST_COPY_DEEP)
                        free(elem->data);
                free(elem);
        } else {
                return M_LIST_E_NOT_PRESENT;
        }

        list->length--;
        m_list_drop_index(list);

        return M_LIST_OK;
}
```

```
int
m_list_remove_first(m_list* list)
{
        m_list_elem* first;
        int retval;

        if ((retval = m_list_first(list, &first)) != M_LIST_OK)
                return retval;

        return m_list_remove(list, first);
}

int
m_list_remove_last(m_list* list)
{
        m_list_elem* last;
        int retval;

        if ((retval = m_list_last(list, &last)) != M_LIST_OK)
                return retval;

        return m_list_remove(list, last);
}
```

algorithm/filter.c

```
#include "m_list.h"

int
m_list_filter(m_list* list, int(*fn)(void*, void*), void* payload)
{
        m_list_elem* runner;
        m_list_elem* next;
        int retval;
        uint64_t removed_count;

        if (list == NULL || fn == NULL)
                return M_LIST_E_NULL;

        removed_count = 0;
        runner = list->first;
        while (runner != NULL) {
                if (fn(runner->data, payload)) {
                        next = runner->next;
                        if ((retval = m_list_remove(list, runner)) != M_LIST_OK)
                                return retval;
                        removed_count++;
                        runner = next;
                } else {
                        runner = runner->next;
                }
        }

        if (removed_count > 0)
                m_list_drop_index(list);

        return M_LIST_OK;
}
```

algorithm/find.c

```
#include "m_list.h"

int
m_list_find(m_list* list, int(*fn)(void*, void*), void* key, void** output)
{
        m_list_elem* runner;

        if (list == NULL || fn == NULL)
                return M_LIST_E_NULL;

        runner = list->first;
        while (runner != NULL) {
                if (fn(runner->data, key)) {
                        if (output != NULL)
                                *output = runner->data;
                        return M_LIST_TRUE;
                }
                runner = runner->next;
        }

        return M_LIST_FALSE;
}
```

algorithm/join.c

```
#include <string.h>

#include "m_list.h"

int
m_list_join(m_list* list, uint8_t copy, void* data, size_t size)
{
        m_list_elem* runner;
        m_list_elem* elem;

        if (list == NULL)
                return M_LIST_E_NULL;

        if (list->first == NULL)
```

```
                return M_LIST_OK;

        runner = list->first;
        while (runner->next != NULL) {
                elem = malloc(sizeof(m_list_elem));
                elem->copy = copy;
                elem->next = runner->next;
                elem->prev = runner;

                if (copy == M_LIST_COPY_DEEP) {
                        if (data == NULL) {
                                elem->data = NULL;
                        } else {
                                elem->data = malloc(size);
                                memcpy(elem->data, data, size);
                        }
                } else if (copy == M_LIST_COPY_SHALLOW) {
                        elem->data = data;
                } else {
                        free(elem);
                        return M_LIST_E_UNKNOWN_COPY;
                }

                runner->next->prev = elem;
                runner->next = elem;
                list->length++;
                runner = elem->next;
        }

        m_list_drop_index(list);

        return M_LIST_OK;
}
```

algorithm/map.c

```
#include "m_list.h"

int
m_list_map(m_list* list, void(*fn)(void*, void*), void* payload)
{
        m_list_elem* runner;

        if (list == NULL || fn == NULL)
                return M_LIST_E_NULL;

        for (runner = list->first; runner != NULL; runner = runner->next)
                fn(runner->data, payload);

        return M_LIST_OK;
}
```

```c
int
m_list_map_ex(m_list* list, void(*fn)(m_list_elem*, uint64_t, void*), void* payload)
{
	m_list_elem* runner;
	uint64_t i;

	if (list == NULL || fn == NULL)
		return M_LIST_E_NULL;

	for (runner = list->first, i = 0; runner != NULL; runner = runner->next, i++)
		fn(runner, i, payload);

	return M_LIST_OK;
}

int
m_list_map2(m_list* list, void(*fn)(void*, void*, void*), void* payload1, void* payload2)
{
	m_list_elem* runner;

	if (list == NULL || fn == NULL)
		return M_LIST_E_NULL;

	for (runner = list->first; runner != NULL; runner = runner->next)
		fn(runner->data, payload1, payload2);

	return M_LIST_OK;
}
```

algorithm/reverse.c

```c
#include "m_list.h"

static void
swap_elem(m_list_elem** a, m_list_elem** b)
{
	m_list_elem* tmp;

	tmp = *a;
	*a = *b;
	*b = tmp;
}

int
m_list_reverse(m_list* list)
{
	m_list_elem* runner;

	if (list == NULL)
		return M_LIST_E_NULL;
```

```
        if (list->length < 2)
                return M_LIST_OK;

        runner = list->first;
        while (runner != NULL) {
                swap_elem(&runner->next, &runner->prev);
                runner = runner->prev;
        }
        swap_elem(&list->first, &list->last);

        return M_LIST_OK;
}
```

algorithm/sort.c

```
#include "m_list.h"

static m_list_elem*
merge_sort(m_list_elem* first,
           uint64_t length,
           int(*cmp_fn)(void*, void*, void*),
           void* payload)
{
        m_list_elem* a;
        m_list_elem* b;
        m_list_elem* e;
        m_list_elem* tail;
        uint64_t segment_size;
        uint64_t a_size;
        uint64_t b_size;
        uint64_t i;

        for (segment_size = 1; segment_size < length; segment_size *= 2) {
                a = first;
                first = NULL;
                tail = NULL;

                while (a != NULL) {
                        b = a;
                        a_size = 0;
                        for (i = 0; i < segment_size; i++) {
                                a_size++;
                                b = b->next;
                                if (b == NULL)
                                        break;
                        }

                        b_size = segment_size;

                        while (a_size > 0 || (b_size > 0 && b != NULL)) {
                                if (a_size == 0) {
                                        e = b; b = b->next; b_size--;
                                } else if (b_size == 0 || b == NULL) {
                                        e = a; a = a->next; a_size--;
                                } else if (cmp_fn(a->data, b->data, payload) <= 0) {
                                        e = a; a = a->next; a_size--;
                                } else {
                                        e = b; b = b->next; b_size--;
                                }

                                if (tail != NULL)
                                        tail->next = e;
                                else
                                        first = e;

                                e->prev = tail;
                                tail = e;
                        }
                        a = b;
                }
                tail->next = NULL;
        }

        return first;
}

int
m_list_sort(m_list* list,
            int(*cmp_fn)(void*, void*, void*),
            void* payload)
{
        if (list == NULL || cmp_fn == NULL)
                return M_LIST_E_NULL;

        if (list->length < 2)
                return M_LIST_OK;

        list->first = merge_sort(list->first, list->length, cmp_fn, payload);
        return M_LIST_OK;
}
```

algorithm/zip.c

```
#include "m_list.h"

int
m_list_zip(m_list* list_a, m_list* list_b, void(*fn)(void*, void*, void*), void* payload)
{
        m_list_elem* runner_a;
        m_list_elem* runner_b;
        if (list_a == NULL || list_b == NULL || fn == NULL)
                return M_LIST_E_NULL;

        runner_a = list_a->first;
        runner_b = list_b->first;
        while (runner_a != NULL && runner_b != NULL) {
                fn(runner_a->data, runner_b->data, payload);
                runner_a = runner_a->next;
                runner_b = runner_b->next;
        }

        return M_LIST_OK;
}
```

predicate/equal.c

```
#include <string.h>

#include "m_list.h"

int
m_list_equal(m_list* list_a, m_list* list_b)
{
        m_list_elem* a;
        m_list_elem* b;

        if (list_a == NULL || list_b == NULL)
                return M_LIST_E_NULL;

        if (list_a->length != list_b->length)
                return M_LIST_FALSE;

        a = list_a->first;
        b = list_b->first;
        while (a != NULL || b != NULL) {
                if (a->copy != b->copy)
                        return M_LIST_FALSE;

                if (a->size != b->size)
                        return M_LIST_FALSE;

                if (a->copy == M_LIST_COPY_SHALLOW)
                        if (a->data != b->data)
                                return M_LIST_FALSE;

                if (a->copy == M_LIST_COPY_DEEP)
                        if (memcmp(a->data, b->data, a->size) != 0)
                                return M_LIST_FALSE;

                a = a->next;
                b = b->next;
```

```c
        }

        return M_LIST_TRUE;
}
```

predicate/is_empty.c

```c
#include "m_list.h"

int
m_list_is_empty(m_list* list)
{
        if (list->length == 0)
                return M_LIST_TRUE;
        else
                return M_LIST_FALSE;
}
```

predicate/is_sorted.c

```c
#include "m_list.h"

int
m_list_is_sorted(m_list* list, int(*cmp_fn)(void*, void*))
{
        m_list_elem* runner;

        if (list == NULL || cmp_fn == NULL)
                return M_LIST_E_NULL;

        if (list->length < 2)
                return M_LIST_TRUE;

        runner = list->first;
        while (runner->next != NULL) {
                if (cmp_fn(runner->data, runner->next->data) > 0)
                        return M_LIST_FALSE;

                runner = runner->next;
        }

        return M_LIST_TRUE;
}
```

predicate/match.c

```c
#include "m_list.h"

int
m_list_match_all(m_list* list, int(*fn)(void*, void*), void* payload)
```

```
{
        return m_list_match_exactly(list, fn, list->length, payload);
}

int
m_list_match_any(m_list* list, int(*fn)(void*, void*), void* payload)
{
        return m_list_match_at_least(list, fn, 1, payload);
}

int
m_list_match_exactly(m_list* list,
                     int(*fn)(void*, void*),
                     uint64_t count,
                     void* payload)
{
        uint64_t matched;
        m_list_elem* runner;

        if (list == NULL || fn == NULL)
                return M_LIST_E_NULL;

        if (count > list->length)
                return M_LIST_E_OUT_OF_BOUNDS;

        matched = 0;
        runner = list->first;
        while (runner != NULL) {
                if (fn(runner->data, payload)) {
                        matched++;
                        if (matched > count)
                                return M_LIST_FALSE;
                }
                runner = runner->next;
        }

        if (matched == count)
                return M_LIST_TRUE;
        else
                return M_LIST_FALSE;
}

int
m_list_match_at_least(m_list* list,
                      int(*fn)(void*, void*),
                      uint64_t count,
                      void* payload)
{
        uint64_t matched;
        uint64_t visited;
        m_list_elem* runner;
        if (list == NULL || fn == NULL)
                return M_LIST_E_NULL;

        if (count > list->length)
                return M_LIST_E_OUT_OF_BOUNDS;

        matched = 0;
        visited = 0;
        runner = list->first;
        while (runner != NULL) {
                if (fn(runner->data, payload))
                        matched++;

                if (matched == count)
                        return M_LIST_TRUE;

                visited++;
                if (matched + (list->length - visited) < count)
                        return M_LIST_FALSE;

                runner = runner->next;
        }

        return M_LIST_FALSE;
}
```