Университет ИТМО
Кафедра ВТ

Лабораторная работа №4
Низкоуровневое программирование

Выполнил: Федоров Сергей
Группа: P33113

Санкт-Петербург
2020 г.

Задание лабораторной работы:

Реализовать на языке C связный список и методы к нему:

- `list_create` - создание списка
- `list_add_front` - добавление элемента к голове
- `list_add_back` - добавление элемента к хвосту
- `list_free` - очистка памяти от списка
- `list_length` - подсчет длинны списка
- `list_sum` - подсчет суммы элементов
- `list_node_at` - взятие вершины списка на n-ой позиции
- `list_get` - взятие значения элемента списка на n-ой позиции
- `foreach` - применение side-effect по каждому элементу
- `map` - отображение списка в другой по функции
- `map_mut` - отображение списка в самого себя по функции
- `fold_l` - свертка списка по функции и с аккумулятором
- `unfold_l (iterate)` - развертка/генерация списка по правилу
- `save` - сохранение списка в файл (текстовое представление)
- `load` - загрузка списка из файла (текстовое представление)
- `serialize` - сохранение списка в файл (бинарное представление)
- `deserialize` - загрузка списка из файла (бинарное представление)

Выполнение:

```
Lab4/
├── functional.c
├── functional.h
├── linked_list.c
├── linked_list.h
├── list_io.c
├── list_io.h
└── main.c
```

main.c

```c
#include <stdio.h>
#include "linked_list.h"
#include "functional.h"
#include "list_io.h"

void print_val(value a){
    printf("%d\n", a);
}

void print_val_whitespace(value a){
    printf("%d ", a);
}

value sqr(value v){
    return v * v;
}

value cbr(value v){
    return v * v * v;
}

value abs(value v){
    return v > 0 ? v : -v;
}

value times_2(value v){
    return v * 2;
}
```

```c
int main() {

    // PART 1 - PAGE 197

    linked_list list_1 = read_from_console();

    puts("YOUR ARRAY:");
    print_list(list_1);

    puts("SUM OF THE ARRAY:");
    print_val(list_sum(list_1));

    list_free(list_1);

    // PART 2 - PAGE 218

    linked_list list_2 = read_from_console();

    puts("YOUR ARRAY:");
    print_list(list_2);

    puts("PRINTED BY FOREACH ' ':");
    foreach(list_2, &print_val_whitespace);
    puts("");

    puts("PRINTED BY FOREACH '\\n':");
    foreach(list_2, &print_val);

    puts("SQUARES:");
    linked_list list_2_sqr = map(list_2, &sqr);
    print_list(list_2_sqr);

    puts("CUBES:");
    linked_list list_2_cbr = map(list_2, &cbr);
    print_list(list_2_cbr);

    puts("ABS:");
    map_mut(list_2, &abs);
    print_list(list_2);

    puts("POWERS of two:");
    linked_list twos = unfold_l(1, 10, &times_2);
    print_list(twos);

    puts("WRITING LIST TO FILE (TEXT)...");
    const char* file_text = "saved.txt";
    save(list_2, file_text);
    linked_list read_text_list;
    load(&read_text_list, file_text);
    puts("READ THESE NUMBERS BACK:");
    print_list(read_text_list);

    puts("WRITING LIST TO FILE (BIN)...");
    const char* file_bin = "saved.bin";
    serialize(list_2, file_bin);
    linked_list read_bin_list;
    deserialize(&read_bin_list, file_bin);
    puts("READ THESE NUMBERS BACK:");
    print_list(read_bin_list);

    list_free(list_2);
    list_free(list_2_sqr);
    list_free(list_2_cbr);
    list_free(twos);
    list_free(read_text_list);
    list_free(read_bin_list);

    puts("FREED ALL ALLOCATED MEMORY");
    puts("FIN.");
}
```

linked_list.h

```c
#ifndef linked_list_h
#define linked_list_h

#include <stdio.h>

typedef int value;

struct Node {
    struct Node* next_elem;
    value node_value;
};

typedef struct Node* linked_list;

linked_list list_create(value new_value);

linked_list list_add_front(value new_value, const linked_list list);
linked_list list_add_back(value new_value, const linked_list list);

void list_free(const linked_list list); // TODO Implement through foreach

size_t list_length(const linked_list list); // TODO Implement through foreach

value list_sum(const linked_list list); // TODO Implement through foldl

linked_list list_node_at(size_t pos, const linked_list list);
value list_get(size_t pos, const linked_list list);

void print_list(const linked_list); // TODO Implement through foreach

#endif /* linked_list_h */
```

linked_list.c

```c
#include "stdlib.h"
#include "stdio.h"

#include "linked_list.h"
#include "functional.h"

linked_list list_create(value new_value) {
    linked_list node_p = malloc(sizeof(struct Node));
    node_p->node_value = new_value;
    node_p->next_elem = NULL;

    return node_p;
}

void list_free(const linked_list list) {
    linked_list next_node = list;

    while (next_node != NULL) {
        linked_list prev_node = next_node;
        next_node = next_node->next_elem;
        free(prev_node);
    }
}

linked_list list_add_front(value new_value, const linked_list list) {
    linked_list new_node = list_create(new_value);
    new_node->next_elem = list;

    return new_node;
}

linked_list list_add_back(value new_value, const linked_list list) {
    linked_list next_node = list;
    while(next_node->next_elem != NULL) {
        next_node = next_node->next_elem;
    }

    linked_list new_node = list_create(new_value);
    next_node->next_elem = new_node;
```

```
    return list;
}

linked_list list_node_at(size_t pos, const linked_list list) {
    linked_list next_node = list;

    if (pos < 0) return NULL;
    else {
        size_t iter;
        for (iter = 0; iter < pos; iter++) {
            if (next_node->next_elem == NULL) break;
            else next_node = next_node->next_elem;
        }
        if (iter != pos) return NULL;
        else return next_node;
    }
}

value list_get(size_t pos, const linked_list list){
    linked_list desired_node = list_node_at(pos, list);
    if (desired_node == NULL) return -1;
    else return desired_node->node_value;
}


static void print_elem(value v) {
    printf("%d ", v);
}

void print_list(const linked_list list) {
    foreach(list, &print_elem);
    puts("");
}


static value sum(value a, value b){
    return a + b;
}

value list_sum(const linked_list list) {
    return fold_l(list, 0, &sum);
}

size_t list_length(const linked_list list) {
    linked_list next_node = list;

    size_t len = 0;
    while (next_node != NULL) {
        len += 1;
        next_node = next_node->next_elem;
    }

    return len;
}
```

functional.h

```
#ifndef functional_h
#define functional_h

#include "linked_list.h"

// For anonymous functions
#define lambda(l_ret_type, l_arguments, l_body)        \
  ({                                                   \
   l_ret_type l_anonymous_functions_name l_arguments   \
   l_body                                              \
   &l_anonymous_functions_name;                        \
   })

void foreach(const linked_list list, void(*func)(value));

linked_list map(const linked_list list, value(*op)(value));
```

```c
void map_mut(const linked_list list, value(*op)(value));

value fold_l(const linked_list list, value acc, value(*op)(value, value));

linked_list unfold_l(value init, size_t length, value(*op)(value)); // Basically same as
iterate, but more canonical;

#endif /* functional_h */
```

functional.c

```c
#include "stdlib.h"
#include "functional.h"

void foreach(const linked_list list, void(*func)(value)) {
    linked_list next_node = list;
    while (next_node != NULL) {
        func(next_node->node_value);
        next_node = next_node->next_elem;
    }
}

linked_list map(const linked_list list, value(*op)(value)) {
    if (list != NULL) {
        linked_list next_node = list;
        linked_list new_node = list_create(op(next_node->node_value));
        linked_list new_list = new_node;

        next_node = next_node->next_elem;
        while (next_node->next_elem != NULL) {
            new_node = list_add_back(op(next_node->node_value), new_node)->next_elem;
            next_node = next_node->next_elem;
        }; list_add_back(op(next_node->node_value), new_node);

        return new_list;
    } else {
        return NULL;
    }
}

void map_mut(const linked_list list, value(*op)(value)) {
    linked_list next_node = list;
    while (next_node != NULL) {
        next_node->node_value = op(next_node->node_value);
        next_node = next_node->next_elem;
    };
}

value fold_l(const linked_list list, value acc, value(*op)(value, value)) {
    linked_list next_node = list;
    while (next_node != NULL) {
        acc = op(acc, next_node->node_value);
        next_node = next_node->next_elem;
    }

    return acc;
}

linked_list unfold_l(value init, size_t length, value(*op)(value)) {
    linked_list new_list = list_create(init);
    linked_list new_node = new_list;

    size_t iter;
    for (iter = 1; iter < length; iter++) {
        init = op(init);
        new_node = list_add_back(init, new_node)->next_elem;
    }

    return new_list;
}
```

list_io.h

```c
#ifndef list_io_h
#define list_io_h

#include <stdio.h>

#include "linked_list.h"

int save(const linked_list list, const char* filename);
int load(linked_list* list, const char* filename);

int serialize(const linked_list list, const char* filename);
int deserialize(linked_list* list, const char* filename);

linked_list read_from_console(void);
#endif /* list_io_h */
```

list_io.c

```c
#include <stdio.h>
#include <errno.h>

#include "list_io.h"

static int check_file(FILE* file) {
    if (errno || ferror(file)) {
        fclose(file);
        return 0;
    } else return 1;
}

static int close_file(FILE* file) {
    fclose(file);
    if (errno) return 0;
    else return 1;
}

static FILE* open_file(const char* filename, const char* mode) {
    errno = 0;
    FILE* file = fopen(filename, mode);
    if (errno) return NULL;
    else return file;
}

int save(linked_list list, const char* filename) {
    FILE* file = open_file(filename, "w");
    if (file == NULL) return 0;

    linked_list next_node = list;
    while (next_node != NULL) {
        fprintf(file, "%d ", next_node->node_value);
        if (!check_file(file)) return 0;
        next_node = next_node->next_elem;
    }

    return close_file(file);
}

int load(linked_list* list, const char* filename) {
    FILE* file = open_file(filename, "r");
    if (file == NULL) return 0;

    linked_list next_node = NULL , new_list = NULL;

    value read_val;

    while (1) {
        fscanf(file, "%d", &read_val);
        if (feof(file)) break;
        if (!check_file(file)) return 0;


        if (new_list == NULL) {
```

```
            new_list = list_create(read_val);
            next_node = new_list;
        } else {
            next_node = list_add_back(read_val, next_node)->next_elem;
        }
    }

    *list = new_list;
    return close_file(file);
}

int serialize(linked_list list, const char* filename) {
    FILE* file = open_file(filename, "wb");
    if (file == NULL) return 0;

    linked_list next_node = list;
    while (next_node != NULL) {
        fwrite(&next_node->node_value, sizeof(value), 1, file);
        if (!check_file(file)) return 0;
        next_node = next_node->next_elem;
    }

    return close_file(file);
}

int deserialize(linked_list* list, const char* filename) {
    FILE* file = open_file(filename, "rb");
    if (file == NULL) return 0;

    linked_list next_node = NULL , new_list = NULL;

    value read_val;

    while (1) {
        fread(&read_val, sizeof(value), 1, file);
        if (feof(file)) break;
        if (!check_file(file)) return 0;

        if (new_list == NULL) {
            new_list = list_create(read_val);
            next_node = new_list;
        } else {
            next_node = list_add_back(read_val, next_node)->next_elem;
        }
    }

    *list = new_list;
    return close_file(file);
}

linked_list read_from_console() {
    value read_val;

    linked_list new_list = NULL;

    puts("ENTER N:");

    if (scanf("%d", &read_val) != 1) return NULL;

    puts("ENTER LIST NUMBERS:");

    int n = read_val;
    int i;
    for(i = 0; i < n; i++) {
        if (scanf("%d", &read_val) != 1) break;
        else new_list = list_add_front(read_val, new_list);
    }

    return new_list;
}
```

Вывод:

На самом деле не особо много мыслей по поводу данной лабораторной работе, но вот следующая ремарка найдется:

Даже несмотря на то что на языке C приходится писать больше кода чем на других более высокоуровневых языках, а отсутствие некоторых синтаксических возможностей достаточно сильно бьет по читаемости кода и его удобству, все равно при выполнении именно таких задач проявляется сильная сторона данного языка - мы можем с легкостью реализовывать различные алгоритмы и структуры данных, настолько близко и настолько оптимизированно к «машине» насколько возможно, что в свою очередь может быть очень даже полезно.