Университет ИТМО

Факультет ПИиКТ

Низкоуровневое программирование

Лабораторная работа №4

Выполнила: Наумова Н.А.

Группа P33022

Преподаватель: Лукьянов Н.М.

Санкт-Петербург
2020 г.

Задание:

10.6 Assignment: Linked List

10.6.1 Assignment

The program accepts an arbitrary number of integers through `stdin`. What you have to do is

1. Save them all in a **linked list** *in reverse order*.
2. Write a function to compute the sum of elements in a linked list.
3. Use this function to compute the sum of elements in the saved list.
4. Write a function to output the n-th element of the list. If the list is too short, signal about it.
5. Free the memory allocated for the linked list.

You need to learn to use

- Structural types to encode the linked list itself.
- The `EOF` constant. Read the section "Return value" of the `man scanf`.

You can be sure that

- The input does not contain anything but integers separated by whitespaces.
- All input numbers can be contained into `int` variables.

Following is the recommended list of functions to implement:

- `list_create` – accepts a number, returns a pointer to the new linked list node.
- `list_add_front` – accepts a number and a pointer to a pointer to the linked list. Prepends the new node with a number to the list.

For example: a list (1,2,3), a number 5, and the new list is (5,1,2,3).

- `list_add_back`, adds an element to the end of the list. The signature is the same as `list_add_front`.
- `list_get` gets an element by index, or returns 0 if the index is outside the list bounds.
- `list_free` frees the memory allocated to all elements of list.
- `list_length` accepts a list and computes its length.
- `list_node_at` accepts a list and an index, returns a pointer to `struct list`, corresponding to the node at this index. If the index is too big, returns `NULL`.
- `list_sum` accepts a list, returns the sum of elements.

These are some additional requirements:

- All pieces of logic that are used more than once (or those which can be conceptually isolated) should be abstracted into functions and reused.
- The exception to the previous requirement is when the performance drop is becoming crucial because code reusage is changing the algorithm in a radically ineffective way. For example, you can use the function `list_at` to get the n-th element of a list in a loop to calculate the sum of all elements. However, the former needs to pass through the whole list to get to the element. As you increase n, you will pass the same elements again and again.

11.7.2 Assignment

The input contains an arbitrary number of integers.

1. Save these integers in a linked list.
2. Transfer all functions written in previous assignment into separate `.h` and `c` files. Do not forget to put an include guard!
3. Implement `foreach`; using it, output the initial list to `stdout` twice: the first time, separate elements with spaces, the second time output each element on the new line.
4. Implement `map`; using it, output the squares and the cubes of the numbers from list.
5. Implement `foldl`; using it, output the sum and the minimal and maximal element in the list.
6. Implement `map_mut`; using it, output the modules of the input numbers.
7. Implement `iterate`; using it, create and output the list of the powers of two (first 10 values: 1, 2, 4, 8, ...).
8. Implement a function `bool save(struct list* lst, const char* filename);`, which will write all elements of the list into a text file `filename`. It should return `true` in case the write is successful, `false` otherwise.
9. Implement a function `bool load(struct list** lst, const char* filename);`, which will read all integers from a text file filename and write the saved list into `*lst`. It should return `true` in case the write is successful, `false` otherwise.

11.7 Assignment: Higher-Order Functions and Lists

11.7.1 Common Higher-Order Functions

In this assignment, we are going to implement several higher-order functions on linked lists, which should be familiar to those used to functional programming paradigm.

These functions are known under the names `foreach`, `map`, `map_mut`, and `foldl`.

- `foreach` accepts a pointer to the list start and a function (which returns void and accepts an `int`). It launches the function on each element of the list.
- `map` accepts a function f and a list. It returns a new list containing the results of the f applied to all elements of the source list. The source list is not affected.

For example, $f(x) = x + 1$ will map the list (1, 2, 3) into (2, 3, 4).

- `map_mut` does the same but changes the source list.
- `foldl` is a bit more complicated. It accepts:
  - The accumulator starting value.
  - A function $f(x, a)$.
  - A list of elements.

It returns a value of the same type as the accumulator, computed in the following way:

1. We launch f on accumulator and the first element of the list. The result is the new accumulator value a'.
2. We launch f on a' and the second element in list. The result is again the new accumulator value a''.
3. We repeat the process until the list is consumed. In the end the final accumulator value is the final result.

For example, let's take $f(x, a) = x * a$. By launching `foldl` with the accumulator value 1 and this function we will compute the product of all elements in the list.

- `iterate` accepts the initial value s, list length n, and function f. It then generates a list of length n as follows:

$$\left[s, f(s), f(f(s)), f(f(f(s)))\ldots\right]$$

The functions described above are called **higher-order functions**, because they do accept other functions as arguments. Another example of such a function is the array sorting function `qsort`.

```
void qsort( void *base,
            size_t nmemb,
            size_t size,
            int (*compar)(const void *, const void *));
```

It accepts the array starting address base, elements count `nmemb`, size of individual elements `size`, and the comparator function `compar`. This function is the decision maker which tells which one of the given elements should be closer to the beginning of the array.

10. Save the list into a text file and load it back using the two functions above. Verify that the save and load are correct.
11. Implement a function `bool serialize(struct list* lst, const char* filename);`, which will write all elements of the list into a *binary* file filename. It should return `true` in case the write is successful, `false` otherwise.
12. Implement a function `bool deserialize(struct list** lst, const char* filename);`, which will read all integers from a *binary* file filename and write the saved list into *lst. It should return `true` in case the write is successful, `false` otherwise.
13. Serialize the list into a binary file and load it back using two functions above. Verify that the serialization and deserialization are correct.
14. Free all allocated memory.

You will have to learn to use

- Function pointers.
- `limits.h` and constants from it. For example, in order to find the minimal element in an array, you have to use `foldl` with the maximal possible `int` value as an accumulator and a function that returns a minimum of two elements.
- The `static` keyword for functions that you only want to use in one module.

You are guaranteed, that

- Input stream contains only integer numbers separated by whitespace characters.
- All numbers from input can be contained as `int`.

It is probably wise to write a separate function to read a list from `FILE`.

The solution takes about 150 lines of code, not counting the functions, defined in the previous assignment.

Код:

linked_list.h

```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

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

struct LinkedList {
   int64_t value;
   struct LinkedList *next;
};

struct LinkedList *list_create(int64_t first_val);
void list_add_front(struct LinkedList **head, int64_t value);
void list_add_back(struct LinkedList **head, int64_t value);
int64_t list_get_at(struct LinkedList *head, size_t index);
void list_free(struct LinkedList *head);
size_t list_length(struct LinkedList *head);
struct LinkedList *list_node_at(struct LinkedList *head, size_t index);
int64_t list_sum(struct LinkedList *head);
#endif
```

linked_list.c

```
#include "./linked_list.h"
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>

struct LinkedList *list_create(int64_t first_val) {
   struct LinkedList *head = NULL;
   list_add_front(&head, first_val);
   return head;
}

void list_add_front(struct LinkedList **head, int64_t value) {
   struct LinkedList *tmp = (struct LinkedList *) malloc(sizeof(struct
LinkedList));
   tmp->value = value;
   tmp->next = (*head);
   (*head) = tmp;
}

void list_add_back(struct LinkedList **head, int64_t value) {
   if (*head == NULL) {
       list_add_front(head, value);
       return;
   }
   struct LinkedList *current = (*head);
   while (current->next != NULL) {
       current = current->next;
   }
   struct LinkedList *newNode = (struct LinkedList *) malloc(sizeof(struct
LinkedList));
   newNode->next = NULL;
   newNode->value = value;
   current->next = newNode;
}
```

```
int64_t list_get_at(struct LinkedList *head, size_t index) {
   head = list_node_at(head, index);
   if (NULL == head)
       return 0;
   else
       return head->value;
}

void list_free(struct LinkedList *head) {
   if (head == NULL)
       return;
   list_free(head->next);
   free(head);
}

size_t list_length(struct LinkedList *head) {
   size_t length = 0;
   while (head->next != NULL) {
       length++;
       head = head->next;
   }
   return length;
}

struct LinkedList *list_node_at(struct LinkedList *head, size_t index) {
   for (size_t i = index; i > 0; i--) {
       if (head == NULL)
           return NULL;
       head = head->next;
   }
   return head;
}

int64_t list_sum(struct LinkedList *head) {
   int64_t sum = 0;
   while (head != NULL) {
       sum += head->value;
       head = head->next;
   }
   return sum;
}
```

higher_order_func.h

```
#include "./linked_list.h"
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

#ifndef H_HIGHER_ORDER_FUNC
#define H_HIGHER_ORDER_FUNC

void foreach(struct LinkedList *list, void (*func)(int64_t));
struct LinkedList *map(struct LinkedList *origin, int64_t (*func)(int64_t));
void map_mut(struct LinkedList **origin, int64_t (*func)(int64_t));
int64_t foldl(int64_t accum, struct LinkedList *list, int64_t
(*func)(int64_t, int64_t));
struct LinkedList *iterate(int64_t s, size_t n, int64_t (*func)(int64_t));

bool save(struct LinkedList *list, const char *filename);
bool load(struct LinkedList **list, const char *filename);
```

```
bool serialize(struct LinkedList *list, const char *filename);
bool deserialize(struct LinkedList **list, const char *filename);
#endif
```

higher_order_func.c

```
#include "./linked_list.h"
#include "./higher_order_func.h"
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <inttypes.h>

void foreach(struct LinkedList *list, void (*function)(int64_t)) {
   while (list != NULL) {
       function(list->value);
       list = list->next;
   }
}

struct LinkedList *map(struct LinkedList *origin, int64_t
(*function)(int64_t)) {
   struct LinkedList *head = NULL;
   while (origin != NULL) {
       list_add_front(&head, function(origin->value));
       origin = origin->next;
   }
   return head;
}

void map_mut(struct LinkedList **origin, int64_t (*func)(int64_t)) {
   if ((*origin) == NULL)
       return;
   (*origin)->value = func((*origin)->value);
   map_mut(&(*origin)->next, func);
   return;
}

int64_t foldl(int64_t accum, struct LinkedList *list, int64_t
(*func)(int64_t, int64_t)) {
   while (list != NULL) {
       accum = func(list->value, accum);
       list = list->next;
   }
   return accum;
}

struct LinkedList *iterate(int64_t s, size_t n, int64_t (*func)(int64_t)) {
   struct LinkedList *list = list_create(s);
   n--;
   for (unsigned int i = 0; i < n; i++) {
       s = func(s);
       list_add_back(&list, s);
   }
   return list;
}


bool save(struct LinkedList *list, const char *filename) {
```

```
    FILE *file;
    file = fopen(filename, "w");
    if (file == NULL) {
        return false;
    }
    while (list != NULL) {
        fprintf(file, "%"PRId64" ", list->value);
        list = list->next;
    }
    fclose(file);
    return true;
}

bool load(struct LinkedList **list, const char *filename) {
    FILE *file;
    int64_t container;
    if ((file = fopen(filename, "r")) == NULL) {
        return false;
    }
    while (fscanf(file, "%"PRId64" ", &container) != EOF) {
        list_add_back(list, container);
    }
    fclose(file);
    return true;
}

bool serialize(struct LinkedList *list, const char *filename) {
    FILE *bin;
    if ((bin = fopen(filename, "wb")) == NULL) {
        return false;
    }
    while (list != NULL) {
        fwrite(&(list->value), sizeof(int64_t), 1, bin);
        list = list->next;
    }
    fclose(bin);
    return true;
}

bool deserialize(struct LinkedList **list, const char *filename) {
    FILE *bin;
    int64_t container;
    if ((bin = fopen(filename, "rb")) == NULL) {
        return false;
    }
    while (fread(&container, sizeof(int64_t), 1, bin) != 0) {
        list_add_back(list, container);
    }
    fclose(bin);
    return true;
}
```

main.c

```
#include "./linked_list.h"
#include "./higher_order_func.h"
#include <stdint.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <inttypes.h>
```

```
void print_space(int64_t);
void print_newline(int64_t);
int64_t square(int64_t);
int64_t cube(int64_t);
int64_t sum(int64_t x, int64_t a);
int64_t timesThree(int64_t x);
struct LinkedList *init(void);
struct LinkedList *iterate_test(void);
void save_load_test(struct LinkedList **list);
void file_bin_test(struct LinkedList **list);
void basic_test(struct LinkedList *list);

enum status {
   Ok, Serialize_error, Deserialize_error, Save_error, Loading_error
};

int main() {
   struct LinkedList *list = init();
   basic_test(list);
   struct LinkedList *iter = iterate_test();
   save_load_test(&iter);
   file_bin_test(&iter);
   list_free(list);
   list_free(iter);
   return Ok;
}

struct LinkedList *init(void) {
   int64_t digit;
   struct LinkedList *list;
   printf("Write digits separated by a space >>> \n");
   if (scanf("%" PRId64 "", &digit) != EOF) {
       list = list_create(digit);
       while (scanf("%" PRId64 "", &digit) != EOF) {
           list_add_front(&list, digit);
       }
   }
   return list;
}

void save_load_test(struct LinkedList **iter) {
   puts("Process saving in file...");
   if (!save(*iter, "aaaa")) {
       puts("Save error.");
       exit(Save_error);
   }
   puts("Save completed.");
   list_free(*iter);
   *iter = NULL;

   puts("Process loading from file...");
   if (!load(iter, "aaaa")) {
       puts("Load error.");
       exit(Loading_error);
   }
   printf("Load is completed, your list is:\n ");
   foreach(*iter, print_space);
   puts("");

}
```

```
void file_bin_test(struct LinkedList **iter) {
   puts("Process serialization...");
   if (!serialize(*iter, "./serialization.bin")) {
       puts("Serialization error");
       exit(Serialize_error);
   }
   puts("Serialization complete.");

   list_free(*iter);
   *iter = NULL;

   puts("Process deserialization...");
   if (!deserialize(iter, "./serialization.bin")) {
       puts("Deserialization error.");
       exit(Deserialize_error);
   }
   printf("Deserialization complete, your list is: \n");
   foreach(*iter, print_space);
   puts("");
}

struct LinkedList *iterate_test(void) {
   struct LinkedList *iter = iterate(1, 10, timesThree);
   puts("Iteration completed: ");
   foreach(iter, print_space);
   puts("");
   return iter;
}

void basic_test(struct LinkedList *list) {
   int64_t second_list = list_get_at(list, 2);
   size_t len = list_length(list);
   struct LinkedList *second_node = list_node_at(list, 2);
   int64_t sum = list_sum(list);
   //check length and sum
   printf("length = %lu, sum = %"PRId64"\n", len, sum);
   if ((second_list != 0) && (second_node != NULL)) {
       printf("list_get(2) = %"PRId64", list_node_at(2) = %"PRId64"\n",
              second_list,
              second_node->value);
   } else {
       puts("Error in list_node_at or list_get: element does not exist.");
   }

   //check foreach
   puts("Foreach with spaces");
   foreach(list, print_space);
   puts("\nForeach with new line");
   foreach(list, print_newline);

   //check map
   struct LinkedList *cubes_map = map(list, cube);
   puts("Map cubes is: ");
   foreach(cubes_map, print_space);
   puts("");
   list_free(cubes_map);

   //check map_mut
   map_mut(&list, imaxabs);
   puts("Map mut abs is: ");
```

```
    foreach(list, print_space);
    puts("");

    //check foldl
    int64_t su = foldl(0, list, square);
    printf("foldl sum is: %"PRId64"\n", su);
}

void print_space(int64_t i) { printf("%" PRId64 " ", i); }
void print_newline(int64_t i) { printf("%" PRId64 "\n", i); }
int64_t square(int64_t x) { return x * x; }
int64_t cube(int64_t x) { return x * x * x; }
int64_t sum(int64_t x, int64_t a) { return x + a; }
int64_t timesThree(int64_t x) { return 3 * x; }
int64_t multiply(int64_t x, int64_t a) { return x * a; }
```

Вывод: выполнив эту лабораторную работу, я реализовала связный список и функции более высокого порядка на нем.