Федеральное государственное автономное образовательное учреждение высшего образования
Университет ИТМО

Кафедра Вычислительной Техники

Дисциплина: Низкоуровневое программирование

Лабораторная работа №4

Выполнил: **Доморацкий Эридан Алексеевич**

Группа: **P33113**

Преподаватель: **Логинов Иван Павлович**

2020г

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.

Выполнение

```c
// linked_list.h

#pragma once

/* [Int] */
struct list;
typedef struct list list_t;

/* list_create :: Int -> [Int] */
list_t * list_create(int value);

/* list_node_at :: [Int] -> Int -> [Int] */
list_t * list_node_at(list_t * list, unsigned int index);

/* list_free :: [Int] -> IO () */
void list_free(list_t * list);

/* list_length :: [Int] -> Int */
unsigned int list_length(const list_t * list);

/* list_get :: [Int] -> Int -> Int */
int list_get(const list_t * list, unsigned int index);

/* list_sum :: [Int] -> Int */
long list_sum(const list_t * list);

/* list_add_front :: Int -> [Int] -> [Int] */
void list_add_front(int value, list_t ** list);

/* list_add_back :: Int -> [Int] -> [Int] */
void list_add_back(int value, list_t ** list);

// linked_list.c

#include "linked_list.h"

#include <stdlib.h>

#define __list_new (malloc(sizeof(list_t)))
#define __list_foreach(_lst) for (; _lst; _lst = _lst->next)

struct list {
   int value;
   list_t * next;
};

list_t * list_create(int value) {
   list_t * list = __list_new;

   list->value = value;
   list->next = NULL;
   return list;
}

list_t * list_node_at(list_t * list, unsigned int index) {
   while (index-- > 0) {
       if (list == NULL) {
           return NULL;
       }

       list = list->next;
   }

   return list;
}
```

```c
void list_free(list_t * list) {
   list_t * prev = list;

   while ((list = list->next)) {
       free(prev);

       prev = list;
   }

   free(prev);
}

unsigned int list_length(const list_t * list) {
   unsigned int length = 0;

   __list_foreach (list) {
       ++length;
   }

   return length;
}

int list_get(const list_t * list, unsigned int index) {
   const list_t * node = list_node_at((list_t *) list, index);

   if (node == NULL) {
       return 0;
   }

   return node->value;
}

long list_sum(const list_t * list) {
   long sum = 0;

   __list_foreach (list) {
       sum += list->value;
   }

   return sum;
}

void list_add_front(int value, list_t ** list) {
   list_t * new_list;

   if (!list) {
       return;
   }

   new_list = __list_new;
   new_list->value = value;
   new_list->next = *list;
   *list = new_list;
}

void list_add_back(int value, list_t ** list) {
   list_t * last = NULL;
   list_t * current;

   if (!list) {
       return;
   }

   current = *list;
   __list_foreach (current) {
       last = current;
   }

   if (!last) {
       list_add_front(value, list);
       return;
   }

   last->next = list_create(value);
}
```

```c
// main.c

#include <stdio.h>

#include "linked_list.h"

int main() {
   list_t * list = NULL;
   int value;

   while (scanf("%d", &value) == 1) {
       list_add_front(value, &list);
   }

   printf("Sum: %ld\n", list_sum(list));
   list_free(list);
   return 0;
}
```

Вывод

В ходе выполнения данной лабораторной работы была освоена работа со структурами и указателями в языке C, написание макросов на языке препроцессора C, вынесение исходного кода во внешние файлы и использование заголовочных файлов для этого.