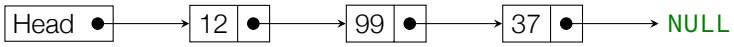
Laborarbeit 4

Abgabe: Mittwoch 11.12.2024 23:59 Uhr

Damit für alle die gleichen Regeln gelten, müssen Ihre Programme standardkonform und ohne Warnungen mit den Compileroptionen (GCC, Clang, MSVS vergleichbar): `-Wall -Wextra -Werror -pedantic -std=c11` ohne Warnungen oder Fehler kompilieren. Vermeiden Sie insbesondere nicht autorisierte Bibliotheken oder Build-Environments. Reichen Sie Ihre Programme (Bitte nur die Quelltext-Dateien) als ZIP Archiv via Opal ein.

Aufgabe 1 (38 Punkte + 9 Zusatzpunkte) In dieser Aufgabe implementieren Sie ein C-Modul `linked_list.c`, das eine einfach verkettete Liste für den Basisdatentyp **`int`** bereitstellt. Verwenden Sie dazu keine C-Arrays, wie bspw. in der letzten Übungsserie für dynamische Arrays. Diesmal sollen Sie Strukturen und Pointer zur Implementierung nutzen. Das ist wie alle anderen Vorgaben zwingend!



Wie Sie sehen benötigen Sie drei verschiedene Elemente: Listenkopf, Listenknoten und Listenende, wobei das Ende durch einen invaliden Zeiger markiert ist. All das soll in den opaken Typ `linked_list` gekapselt werden. Tipp: Erzeugen Sie sich einen internen Datentyp für Listenknoten und einen internen Datentyp für eine Liste welcher dann vom Typ im Header genutzt wird. Achten Sie wie immer auf korrekte Allokation und Freigabe.

Als weitere Vorgabe ist der Header `linked_list.h` zu verwenden und darf nicht verändert werden. Ebenso ist eine Programmdatei `main.c` schon vorhanden und vorgegeben und darf ebenso nicht verändert werden.

Hinweis. Der Compiler (genauer der Linker) muss wissen wo die Symbole der Funktionen sich befinden. Entweder Sie kompilieren immer alles in einem Rutsch: `gcc main.c linked_list.c -o linked_list.exe` oder einzeln mit `gcc -c -o linked_list.o linked_list.c` und dann `gcc main.c linked_list.o -o linked_list.exe`. Auf jeden Fall sollten Sie Ihrer IDE beibringen wo und wie die Dateien zu kompilieren sind, vgl. Folie 131ff.

Erstellen sie die Datei `linked_list.c` und definieren Sie alle vorgegebenen Funktionen.

Je Funktion können Sie 4 Punkte ergattern, 2 Punkte auf den Datentyp, und 9 Zusatzpunkte auf Quelltextqualität (saubere Struktur, gute gewählte Bezeichner, etc.).

linked_list.h

```c
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

#include <stddef.h> // For size_t

/**
 * @brief Error codes for linked list operations.
 */
typedef enum {
    LINKED_LIST_SUCCESS = 0,       /**< Operation succeeded */
    LINKED_LIST_ERR_OUT_OF_BOUNDS, /**< Index out of bounds */
    LINKED_LIST_ERR_ALLOCATION,    /**< Memory allocation failed */
    LINKED_LIST_ERR_NULL_POINTER   /**< Null pointer passed */
} linked_list_error_t;

/**
 * @brief Opaque type for a linked list.
 */
typedef struct linked_list linked_list_t;

/**
 * @brief Creates a new linked list.
 * @return Pointer to the created list, or NULL if allocation fails.
 */
linked_list_t* linked_list_create(void);

/**
 * @brief Frees all resources associated with the linked list.
 * @param list Pointer to the list. Safe to pass NULL.
 */
void linked_list_destroy(linked_list_t* list);
```

```c

/**
 * @brief Appends a value to the end of the list.
 * @param list Pointer to the linked list.
 * @param value Value to append.
 * @return LINKED_LIST_SUCCESS on success, or an error code.
 */
linked_list_error_t linked_list_append(linked_list_t* list, int value);

/**
 * @brief Prepends a value to the beginning of the list.
 * @param list Pointer to the linked list.
 * @param value Value to prepend.
 * @return LINKED_LIST_SUCCESS on success, or an error code.
 */
linked_list_error_t linked_list_prepend(linked_list_t* list, int value);

/**
 * @brief Removes the element at a given index and stores its value.
 * @param list Pointer to the linked list.
 * @param index Index of the element to remove.
 * @param value Pointer to store the removed value.
 * @return LINKED_LIST_SUCCESS on success, or an error code.
 */
linked_list_error_t linked_list_remove_at(linked_list_t* list, size_t index, int* value);

/**
 * @brief Retrieves the value at a given index.
 * @param list Pointer to the linked list.
 * @param index Index of the element to retrieve.
 * @param value Pointer to store the retrieved value.
 * @return LINKED_LIST_SUCCESS on success, or an error code.
 */
linked_list_error_t linked_list_get(const linked_list_t* list, size_t index, int* value);

/**
 * @brief Sets the value at a given index.
 * @param list Pointer to the linked list.
 * @param index Index of the element to update.
 * @param value New value to set.
 * @return LINKED_LIST_SUCCESS on success, or an error code.
 */
linked_list_error_t linked_list_set(linked_list_t* list, size_t index, int value);

/**
 * @brief Returns the number of elements in the list.
 * @param list Pointer to the linked list.
 * @return The number of elements, or 0 on error.
 */
size_t linked_list_size(const linked_list_t* list);

/**
 * @brief Prints all elements in the linked list to the console.
 * @param list Pointer to the linked list.
 * @return LINKED_LIST_SUCCESS on success, or an error code.
 */
linked_list_error_t linked_list_print(const linked_list_t* list);

#endif // LINKED_LIST_H
```

Die Datei `main.c` hat folgenden Inhalt:

main.c

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

#include "linked_list.h"

void handle_error(linked_list_error_t error, const char* message, int terminate) {
  if (error != LINKED_LIST_SUCCESS) {
    fprintf(stderr, "%s: Error code %d\n", message, error);
    if (terminate) {
      exit(EXIT_FAILURE);
    }
  }
}

int main(void) {
  linked_list_t* list = linked_list_create();
  if (!list) {
    handle_error(LINKED_LIST_ERR_ALLOCATION, "Failed to create linked list", EXIT_FAILURE);
  }

  handle_error(linked_list_append(list, 10), "Failed to append 10", 1);
  handle_error(linked_list_append(list, 20), "Failed to append 20", 1);
  handle_error(linked_list_prepend(list, 5), "Failed to prepend 5", 1);

  printf("List size: %zu\n", linked_list_size(list));
  for (size_t i = 0; i < linked_list_size(list); i++) {
    int value;
    handle_error(linked_list_get(list, i, &value), "Failed to get element", 0);
    printf("Element %zu: %d\n", i, value);
  }

  int removed;
  handle_error(linked_list_remove_at(list, 1, &removed), "Failed to remove element at index 1", 0);
  printf("Removed: %d\n", removed);

  for (size_t i = 0; i < linked_list_size(list); i++) {
    int value;
    handle_error(linked_list_get(list, i, &value), "Failed to get element", 0);
    printf("Element %zu: %d\n", i, value);
  }

  linked_list_print(list);
  linked_list_destroy(list);

  return EXIT_SUCCESS;
}
```