КПІ ім. Ігоря Сікорського

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт до комп'ютерного практикуму з курсу «Основи програмування»

Прийняв
асистент кафедри ІПІ
Ахаладзе А. Е.
«2» січня 2025 р.

Виконав
студент групи ІП-43
Дутов І. А.

Київ 2025

Комп'ютерний практикум №9

Тема: *Робота з файлами.*

Завдання:

Написати програму, яка виконує наступні дії:

1. Створення файлу.
2. Відкриття вже створеного файлу та завантаження даних із файлу.
3. Видалення файлу.
4. Запис в файл даних, введених з клавіатури користувачем.
5. Зчитування запису(ів) із файлу і виведення їх на екран.
6. Редагування запису з файлу.
7. Впорядкування (на вибір користувача, за зростанням або спаданням) записів в файлі за полями: назва області, площа, кількість населення.
8. Вставка у впорядкований файл записів так, щоб файл залишився впорядкованим.
9. Видалення запису з файлу.

Текст програми

../CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(RegionSimulator3000 C)

include_directories(${CMAKE_SOURCE_DIR})
enable_testing()

set(CMAKE_BUILD_TYPE Debug)
set(CMAKE_C_FLAGS_DEBUG "-g -gdwarf-4")
set(CMAKE_CXX_FLAGS_DEBUG "-g -gdwarf-4")

set(SOURCES
    src/main.c
    src/actions/files.c
    src/actions/misc.c
    src/actions/records.c
    src/actions/regions.c
    src/actions/utils.c
    src/common/fileContext.c
    src/common/sort.c
    src/common/stack.c
    src/io/choices.c
    src/io/number.c
    src/io/string.c
    src/io/utils.c
    src/io/validators.c
    src/menu/default.c
    src/menu/menu.c
)

```

```
add_executable(RegionSimulator3000 ${SOURCES})

target_link_libraries(RegionSimulator3000 m)
```

../src/main.c

```c
#include "common/common.h"
#include "io/io.h"
#include "menu/menu.h"
#include <stdlib.h>

#define APP_FAILURE SUCCESS

int main() {
  MenuNode *menu = initializeDefaultMenu();
  if (!menu)
    return APP_FAILURE;

  FileContext *context = initFileContext();
  if (!context)
    return APP_FAILURE;

  const size_t menuChoiceCount = assignMenuOptions(menu, MENU_START_INDEX);

  NumberRange *choiceRange = initChoiceRange(MENU_START_INDEX, menuChoiceCount)
      ;
  if (!choiceRange)
    return APP_FAILURE;

  printMenu(menu);

  size_t option = 0;
  int actionResult = SUCCESS;

  do {
    printf("\n");
    option = getUserChoice(choiceRange, NULL, NULL);
    if (option == 0) // EOF
      break;

    actionResult = executeMenuActionWithOption(menu, option, context);
    if (actionResult == DISPLAY_MENU)
      printMenu(menu);
  } while (actionResult != EXIT);

  closeFileContext(context);
  free(context);
  context = NULL;

  freeNumberRange(choiceRange);
  choiceRange = NULL;

  freeMenu(menu);
  menu = NULL;

  return SUCCESS;
}
```

../src/common/common.h

```c
#ifndef COMMON_H
#define COMMON_H

#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>

#define SUCCESS 0
#define FAILURE 1
#define NOOP 2
#define DISPLAY_MENU 3
#define EXIT -2 // INFO: to not be equal EOF = -1
#define PIPE 1

typedef struct {
  char *name;
  double area;
  double population;
} Region;

// INPUT
#define INFINITE_LENGTH 0
#define DEFAULT_STRING_LENGTH 16
typedef enum { TYPE_DOUBLE, TYPE_SIZE_T, TYPE_COUNT } NumberType;

typedef struct {
  void *min;
  void *max;
  bool isMinIncluded;
  bool isMaxIncluded;
  char *valueName;
  int (*compare)(const void *, const void *);
  NumberType type;
} NumberRange;

typedef enum {
  LESS,
  LESS_EQUAL,
  GREATER,
  GREATER_EQUAL,
  WITHIN_RANGE
} RangeCheckResult;

typedef bool(ValidationFunc)(void *, va_list);
// FILE
#define SIGNATURE "RegionSimulator3000\n"
#define TEMP "temp.region"
#define MAX_FILENAME_LENGTH 255

typedef struct {
  FILE *file;
  char *filename;
  char *signature;
  size_t signatureSize;
} FileContext;

typedef int (*Action)(FileContext *context);

```

```
59 FileContext *initFileContext();
60 int fillFileContext(FileContext *context, const char *filename,
61                     const char *signature);
62 int closeFileContext(FileContext *context);
63
64 // SORTING
65 #define BIGGER_THAN 1
66 #define LESS_THAN -1
67 #define EQUAL 0
68 #define COMPARATOR_COUNT 3
69 #define ORDER_COUNT 2
70
71 typedef enum { DESCENDING = -1, ASCENDING = 1 } SortOrder;
72 typedef int (*SortComparator)(const Region *, const Region *);
73
74 int compareRegionNames(const Region *region1, const Region *region2);
75 int compareRegionAreas(const Region *region1, const Region *region2);
76 int compareRegionPopulations(const Region *region1, const Region *region2);
77
78 void quickSort(Region **arr, size_t low, size_t high);
79 bool detectSort(Region **regions, size_t regionCount);
80 SortComparator getQuickSortComparator();
81 SortOrder getQuickSortOrder();
82 void setQuickSortOrder(SortOrder newSortOrder);
83 void setQuickSortComparator(SortComparator newComparator);
84
85 extern const char *SORT_BY_CHOICES[COMPARATOR_COUNT];
86 extern const char *SORT_HOW_CHOICES[ORDER_COUNT];
87 extern const SortComparator SORT_COMPARATORS[COMPARATOR_COUNT];
88 extern const SortOrder SORT_ORDERS[ORDER_COUNT];
89
90 // STACK AND MENU
91 typedef struct StackNode {
92   void *data;
93   struct StackNode *next;
94 } StackNode;
95
96 typedef struct {
97   struct StackNode *top;
98   size_t size;
99 } Stack;
100
101 Stack *createStack();
102 void freeStack(Stack *stack);
103 int push(Stack *stack, void *menu);
104 void *pop(Stack *stack);
105 void *peek(Stack *stack);
106
107 typedef struct MenuNode {
108   char *name;
109   struct MenuNode *child;
110   struct MenuNode *sibling;
111   int option;
112   Action action;
113 } MenuNode;
114 #endif
```

../src/common/fileContext.c

```c
#include "../actions/utils.h"
#include "../io/utils.h"
#include "common.h"
#include <stdlib.h>
#include <string.h>

FileContext *initFileContext() {
  FileContext *context = malloc(sizeof(FileContext));
  if (!context) {
    handleErrorMemoryAllocation("file_context");
    return NULL;
  }

  context->file = NULL;
  context->filename = NULL;
  context->signature = NULL;
  context->signatureSize = 0;

  return context;
}

int closeFileContext(FileContext *context) {
  if (context == NULL) {
    handleError("Context_can't_be_NULL_before_closing:_nothing_to_close");
    return FAILURE;
  }

  if (context->file) {
    fclose(context->file);
    context->file = NULL;
  }

  if (context->filename) {
    free(context->filename);
    context->filename = NULL;
  }

  if (context->signature) {
    free(context->signature);
    context->signature = NULL;
  }
  return SUCCESS;
}

int fillFileContext(FileContext *context, const char *filename,
                    const char *signature) {
  context->filename = strdup(filename);
  if (context->filename == NULL) {
    handleErrorMemoryAllocation("filename_string");
    closeFileContext(context);
    return FAILURE;
  }

  context->signature = strdup(signature);
  if (context->signature == NULL) {
    handleErrorMemoryAllocation("signature_string");
    closeFileContext(context);
    return FAILURE;
  }

```

```
  context->signatureSize = strlen(signature);

  bool doesExist = fileExists(filename);
  const char *mode = (doesExist) ? "r+" : "w+";

  context->file = fopen(filename, mode);
  if (!context->file) {
    handleError("Failed to create new file.");
    closeFileContext(context);
    return FAILURE;
  }

  if (!doesExist) { // Writing signature to new file
    if (writeSignature(context) == FAILURE) {
      closeFileContext(context);
      return FAILURE;
    }
  }

  return SUCCESS;
}
```

../src/common/sort.c

```
#include "../common/common.h"
#include "../io/io.h"
#include "../io/utils.h"
#include <string.h>

const char *SORT_BY_CHOICES[COMPARATOR_COUNT] = {
    "by_region_name",
    "by_region_area",
    "by_region_population",
};

const char *SORT_HOW_CHOICES[ORDER_COUNT] = {"ascending", "descending"};

const SortComparator SORT_COMPARATORS[COMPARATOR_COUNT] = {
    compareRegionNames, compareRegionAreas, compareRegionPopulations};

const SortOrder SORT_ORDERS[ORDER_COUNT] = {ASCENDING, DESCENDING};

static SortComparator comparator = NULL;
static SortOrder sortOrder = 0;
int compareRegionNames(const Region *region1, const Region *region2) {
  return compareUtf8Strings(region1->name, region2->name);
}

int compareRegionAreas(const Region *region1, const Region *region2) {
  if (region1->area > region2->area)
    return BIGGER_THAN;
  if (region1->area < region2->area)
    return LESS_THAN;
  return EQUAL;
}

int compareRegionPopulations(const Region *region1, const Region *region2) {
  if (region1->population > region2->population)
    return BIGGER_THAN;
```

```
  if (region1->population < region2->population)
    return LESS_THAN;
  return EQUAL;
}

void swap(Region *a, Region *b) {
  if (a == NULL || b == NULL) {
    return;
  }
  Region temp = *a;
  *a = *b;
  *b = temp;
}

size_t partition(Region **arr, size_t low, size_t high) {
  Region *pivot = arr[high];
  size_t i = low - 1;

  for (size_t j = low; j < high; j++) {
    int comparison = comparator(arr[j], pivot);
    if ((sortOrder == ASCENDING && comparison < 0) ||
        (sortOrder == DESCENDING && comparison > 0)) {
      i++;
      swap(arr[i], arr[j]);
    }
  }

  swap(arr[i + 1], arr[high]);
  return i + 1;
}

void quickSort(Region **arr, size_t low, size_t high) {
  if (low >= high) {
    return;
  }
  size_t pivotIndex = partition(arr, low, high);
  if (pivotIndex > 0) {
    quickSort(arr, low, pivotIndex - 1);
  }
  quickSort(arr, pivotIndex + 1, high);
}

void setQuickSortComparator(SortComparator newComparator) {
  static SortComparator *comparatorPtr = NULL;
  if (!comparatorPtr) {
    comparatorPtr = &comparator;
  }
  *comparatorPtr = newComparator;
}

void setQuickSortOrder(SortOrder newSortOrder) {
  static SortOrder *sortOrderPtr = NULL;
  if (!sortOrderPtr) {
    sortOrderPtr = &sortOrder;
  }
  *sortOrderPtr = newSortOrder;
}

SortComparator getQuickSortComparator() { return comparator; }
SortOrder getQuickSortOrder() { return sortOrder; }
```

```c

bool detectSort(Region **regions, size_t regionCount) {
  if (getQuickSortComparator()) {
    return true;
  }
  bool isSorted = true;
  for (size_t comp = 0; comp < COMPARATOR_COUNT; comp++) {
    for (size_t order = 0; order < ORDER_COUNT; order++) {
      for (size_t ri = 1; ri < regionCount; ri++) {
        if ((SORT_COMPARATORS[comp])(regions[ri - 1], regions[ri]) *
              SORT_ORDERS[order] >
            0) { // If order is broken
          isSorted = false;
          break;
        }
      }

      if (isSorted) {
        setQuickSortComparator(SORT_COMPARATORS[comp]);
        setQuickSortOrder(ASCENDING);
        return true;
      }
      isSorted = true;
    }
  }

  return false;
}

void showCurrentSortMethod() {
  SortComparator comp = getQuickSortComparator();
  SortOrder sortOrder = getQuickSortOrder();
  size_t compI = 0;
  size_t orderI = 0;

  for (; compI < COMPARATOR_COUNT; compI++) {
    if (SORT_COMPARATORS[compI] == comp) {
      break;
    }
  }
  for (; orderI < ORDER_COUNT; orderI++) {
    if (SORT_ORDERS[orderI] == sortOrder) {
      break;
    }
  }

  printf("Current sorting method is: \n");
  showSuccess("%s %s", SORT_BY_CHOICES[compI], SORT_HOW_CHOICES[orderI]);
  printf("\n");
}
```

../src/common/stack.c

```c
#include "../common/common.h"
#include "../io/utils.h"
#include <stdio.h>
#include <stdlib.h>

Stack *createStack() {
```

```c
  Stack *stack = (Stack *)malloc(sizeof(Stack));
  if (!stack) {
    handleErrorMemoryAllocation("Stack");
    return NULL;
  }
  stack->size = 0;
  stack->top = NULL;
  return stack;
}

void freeStack(Stack *stack) {
  StackNode *item = stack->top;
  while (item) {
    StackNode *temp = item;
    item = item->next;
    free(temp);
  }
  free(stack);
}

int push(Stack *stack, void *data) {
  StackNode *item = (StackNode *)malloc(sizeof(StackNode));
  if (!item) {
    handleErrorMemoryAllocation("StackNode");
    return FAILURE;
  }
  item->data = data;
  item->next = stack->top;
  stack->top = item;
  stack->size++;
  return SUCCESS;
}

void *pop(Stack *stack) {
  if (stack->size == 0) {
    handleError("Error popping value from the stack");
    return NULL;
  }
  StackNode *temp = stack->top;
  void *data = temp->data;
  stack->top = temp->next;
  free(temp);
  stack->size--;
  return data;
}

void *peek(Stack *stack) {
  if (stack->size == 0) {
    return NULL;
  }
  return stack->top->data;
}
```

../src/menu/menu.h

```c
#ifndef MENU_H

#include "../common/common.h"
#include <stdbool.h>
```

```

#define MENU_START_INDEX 1
MenuNode *createMenuNode(const char *name, Action action);
void addMenuChild(MenuNode *parent, MenuNode *child);
int printMenu(MenuNode *root);
void freeMenu(MenuNode *node);
int executeMenuActionWithOption(MenuNode *menu, int option,
                                FileContext *context);
int assignMenuOptions(MenuNode *node, int start);

MenuNode *initializeDefaultMenu();
#endif // !MENU_H
```

../src/menu/default.c

```
#include "../actions/actions.h"
#include "menu.h"

MenuNode *initializeDefaultMenu() {
  MenuNode *menu = createMenuNode("Menu", NULL);
  if (!menu)
    return NULL;

  MenuNode *file = NULL, *records = NULL, *createFile = NULL, *readFile = NULL
      ,
          *deleteFile = NULL, *createRecords = NULL, *readRecords = NULL,
          *editRecords = NULL, *sortRecords = NULL, *insertRecords = NULL,
          *deleteRecords = NULL, *exitProgram = NULL, *verifySignature = NULL,
          *displayMenu = NULL;

  // File part
  if (!(file = createMenuNode("File", NULL)) ||
      !(createFile = createMenuNode("Create/Select File", actionCreateFile)) ||
      !(readFile = createMenuNode("Read all Records", actionReadFile)) ||
      !(deleteFile =
            createMenuNode("Delete/Deselect File", actionDeleteFile)) ||

      // Record part
      !(records = createMenuNode("Records", NULL)) ||
      !(createRecords = createMenuNode("Create Record", actionCreateRecord)) ||
      !(readRecords = createMenuNode("Read Record", actionReadRecord)) ||
      !(editRecords = createMenuNode("Edit Record", actionEditRecord)) ||
      !(sortRecords = createMenuNode("Sort Records", actionSortRecords)) ||
      !(insertRecords =
            createMenuNode("Insert Record", actionInsertRecordIfSorted)) ||
      !(deleteRecords = createMenuNode("Delete Record", actionDeleteRecord)) ||

      // Misc part
      !(exitProgram = createMenuNode("Exit Program", actionExitProgram)) ||
      !(verifySignature =
            createMenuNode("Verify File Header", actionVerifyFileHeader)) ||
      !(displayMenu =
            createMenuNode("Display Menu Again", actionDisplayMenu))) {
    freeMenu(menu);
    return NULL;
  }

  addMenuChild(menu, file);
  addMenuChild(menu, records);
```

```
  addMenuChild(menu, exitProgram);
  addMenuChild(menu, verifySignature);
  addMenuChild(menu, displayMenu);

  addMenuChild(file, createFile);
  addMenuChild(file, readFile);
  addMenuChild(file, deleteFile);

  addMenuChild(records, createRecords);
  addMenuChild(records, readRecords);
  addMenuChild(records, editRecords);
  addMenuChild(records, sortRecords);
  addMenuChild(records, insertRecords);
  addMenuChild(records, deleteRecords);

  assignMenuOptions(menu, MENU_START_INDEX);

  return menu;
}
```

../src/menu/menu.c

```
#include "menu.h"
#include "../common/common.h"
#include "../io/utils.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TREE_SPACING 2

MenuNode *createMenuNode(const char *name, Action action) {
  MenuNode *node = (MenuNode *)malloc(sizeof(MenuNode));

  if (!node) {
    handleErrorMemoryAllocation("menu_node");
    return NULL;
  }

  node->name = strdup(name);
  if (!node->name) {
    free(node);
    handleErrorMemoryAllocation("menu_node_name");
    return NULL;
  }

  node->action = action;
  node->child = NULL;
  node->sibling = NULL;

  return node;
}

void addMenuChild(MenuNode *parent, MenuNode *child) {
  if (!parent->child) {
    parent->child = child;
    return;
  }

```

```
  MenuNode *temp = parent->child;
  while (temp->sibling) {
    temp = temp->sibling;
  }
  temp->sibling = child;
  child->sibling = NULL;
}

int executeMenuActionWithOption(MenuNode *menu, int option,
                                FileContext *context) {
  Stack *stack = createStack();
  if (!stack) {
    handleErrorMemoryAllocation("stack_for_menu");
    return FAILURE;
  }

  push(stack, menu);
  while (stack->size != 0) {
    MenuNode *cur = pop(stack);
    if (cur->action && cur->option == option) {
      freeStack(stack);
      return cur->action(context);
    }

    if (cur->child) {
      MenuNode *child = cur->child;
      while (child) {
        push(stack, child);
        child = child->sibling;
      }
    }
  }
  freeStack(stack);
  return FAILURE; // specified option not found.
}

int assignMenuOptions(MenuNode *node, int start) {
  if (!node)
    return start - 1;

  int lastItem;
  if (node->action) {
    node->option = start;
    lastItem = assignMenuOptions(node->child, start + 1);
  } else {
    lastItem = assignMenuOptions(node->child, start);
  }

  return assignMenuOptions(node->sibling, lastItem + 1);
}

void freeMenu(MenuNode *node) {
  if (!node)
    return;

  freeMenu(node->child);
  freeMenu(node->sibling);

  free(node->name);
  free(node);
```

```c
}

void printMenuNode(MenuNode *node) {
  if (!node)
    return;

  if (node->action) {
    printf("%d: %s", node->option, node->name);
  } else {
    printf("%s", node->name);
  }
}

int printMenu(MenuNode *root) {
  if (!root) {
    handleError("Root menu node is NULL.");
    return FAILURE;
  }

  Stack *stack = createStack();
  if (!stack) {
    return FAILURE;
  }

  size_t maxLevels = 10;
  bool *hasSiblings = (bool *)malloc(maxLevels * sizeof(bool));
  if (hasSiblings == NULL) {
    handleErrorMemoryAllocation("array of bool for finding menu node siblings");
    freeStack(stack);
    return FAILURE;
  }

  size_t level = 0;
  push(stack, root);

  while (stack->size != 0) {
    MenuNode *cur = pop(stack);

    // Ensure `hasSiblings` array has enough space
    if (level >= maxLevels) {
      maxLevels *= 2;
      bool *temp = (bool *)realloc(hasSiblings, maxLevels * sizeof(bool));
      if (temp == NULL) {
        handleErrorMemoryAllocation("reallocation of menu node siblings array");
        free(hasSiblings);
        freeStack(stack);
        return FAILURE;
      }
      hasSiblings = temp;
    }

    hasSiblings[level] = (cur->sibling != NULL);

    for (size_t i = 0; i < level; i++) {
      if (hasSiblings[i]) {
        printf("%-*s ", TREE_SPACING, " ");
      } else {
        printf(" %-*s ", TREE_SPACING, " ");
      }
    }
```

```
    if (hasSiblings[level]) {
      printf("├");
    } else {
      printf("└");
    }
    printf("─");
    printMenuNode(cur);

    if (cur->child) {
      MenuNode *child = cur->child;

      // Push siblings in reverse order to preserve the order
      Stack *tempStack = createStack();
      if (!tempStack)
        return FAILURE;

      while (child) {
        push(tempStack, child);
        child = child->sibling;
      }

      while (tempStack->size != 0) {
        push(stack, pop(tempStack));
      }

      freeStack(tempStack);
      level++;
    } else if (!cur->sibling) {
      level--; // Go back up if no siblings remain
    }
    printf("\n");
  }

  free(hasSiblings);
  freeStack(stack);
  return SUCCESS;
}
```

../src/actions/actions.h

```
#ifndef ACTIONS_H
#define ACTIONS_H
#include "../common/common.h"

// FILE ACTIONS
int actionCreateFile(FileContext *);
int actionReadFile(FileContext *);
int actionDeleteFile(FileContext *);

// RECORD ACTIONS
int actionCreateRecord(FileContext *);
int actionReadRecord(FileContext *);
int actionEditRecord(FileContext *);
int actionSortRecords(FileContext *);
int actionInsertRecordIfSorted(FileContext *);
int actionDeleteRecord(FileContext *);

// MISC ACTIONS
int actionDisplayMenu();
```

```c
int actionExitProgram();
int actionNotImplemented(FileContext *context);
int actionVerifyFileHeader();

#endif
```

../src/actions/files.c

```c
#include "../common/common.h"
#include "../io/utils.h"
#include "actions.h"
#include "assert.h"
#include "utils.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int actionCreateFile(FileContext *context) {
  if (!context) {
    handleError("Context is NULL");
    return FAILURE;
  }

  if (context->file) {
    handleError("Can't create new file while other is already available.");
    if (actionDeleteFile(context) == FAILURE)
      return FAILURE;
  }

  char *filename = getValidFilename();
  if (!filename)
    return FAILURE;

  bool doesExist = fileExists(filename);
  const char *mode = doesExist ? "r+" : "w+";

  if (fillFileContext(context, filename, SIGNATURE) == FAILURE)
    goto cleanup;

  showSuccess("File was %s successfully.", doesExist ? "selected" : "created");

  if (handleBadFileHeader(context) == FAILURE)
    goto cleanup;

  free(filename);
  return SUCCESS;

cleanup:
  if (filename)
    free(filename);
  return FAILURE;
}

int actionReadFile(FileContext *context) {
  if (handleFileIfNotExist(context) == FAILURE)
    return FAILURE;

  if (handleFileIfEmpty(context) == FAILURE)
    return FAILURE;
```

```
52
53  if (skipFileHeader(context) == FAILURE)
54    return FAILURE;
55
56  Region *region = initRegion();
57
58  if (!region)
59    return FAILURE;
60
61  size_t processedRegionCount = 0;
62  int result = SUCCESS;
63  while ((result = readRegion(region, context)) != EOF) {
64    if (result == FAILURE) {
65      freeRegion(region);
66      region = NULL;
67      region = initRegion();
68      if (!region)
69        return FAILURE;
70      showWarning("Improperly_formatted_region_number_%lu_is_skipped.",
71                  processedRegionCount + 1);
72      processedRegionCount++;
73      continue;
74    }
75
76    processedRegionCount++;
77    printRegion(region, processedRegionCount);
78    freeRegion(region);
79    region = initRegion();
80  }
81
82  freeRegion(region);
83
84  showSuccess("File_was_read_successfully.");
85  return SUCCESS;
86 }
87
88 int actionDeleteFile(FileContext *context) {
89  if (!context) {
90    handleError("Context_is_NULL");
91    return FAILURE;
92  }
93
94  if (!context->file) {
95    handleErrorFileNotSpecified();
96    return FAILURE;
97  }
98
99  if (!askQuestion("Do_you_REALLY_want_to_DELETE_this_file?")) {
100   if (!askQuestion("Maybe_you_want_to_deselect_it?"))
101     return FAILURE;
102   closeFileContext(context);
103   return SUCCESS;
104 }
105
106 fclose(context->file);
107 context->file = NULL;
108
109 if (remove(context->filename) != SUCCESS) {
110   handleError("Error_deleting_file_with_records.");
111   return FAILURE;
```

```
112   }
113
114   closeFileContext(context);
115   setQuickSortComparator(NULL);
116
117   showSuccess("File was deleted successfully.", context->filename);
118   return SUCCESS;
119 }
```

../src/actions/records.c

```
1  #include "../actions/utils.h"
2  #include "../common/common.h"
3  #include "../io/io.h"
4  #include "../io/utils.h"
5  #include "actions.h"
6  #include "utils.h"
7  #include <stdlib.h>
8  #include <string.h>
9
10 int actionCreateRecord(FileContext *context) {
11   if (handleFileIfNotExist(context) == FAILURE)
12     return FAILURE;
13
14   if (skipFileHeader(context) == FAILURE)
15     return FAILURE;
16
17   if (fseek(context->file, 0L, SEEK_END) != SUCCESS) {
18     handleError("Couldn't go to the bottom of file");
19     return FAILURE;
20   }
21
22   Region *region = getRegionFromUser();
23   if (!region)
24     return FAILURE;
25
26   if (writeRegion(region, context) == FAILURE)
27     return FAILURE;
28
29   printRegion(region, 0); // without number
30   freeRegion(region);
31   setQuickSortComparator(NULL); // Sort may be removed
32   showSuccess("Record was appended successfully.");
33
34   return SUCCESS;
35 }
36
37 int actionReadRecord(FileContext *context) {
38   if (handleFileIfNotExist(context) == FAILURE)
39     return FAILURE;
40
41   if (handleFileIfEmpty(context) == FAILURE)
42     return FAILURE;
43
44   if (skipFileHeader(context) == FAILURE)
45     return FAILURE;
46
47   size_t recordNumber = 0;
48   if (readNumberWithValidation(&recordNumber, TYPE_SIZE_T, "Record number",
```

```
                        isNotZero) == EOF)
    return EXIT;

  Region *region = readRegionAt(recordNumber, context);

  if (!region)
    return FAILURE;

  printRegion(region, recordNumber);
  freeRegion(region);

  showSuccess("Record read successfully");
  return SUCCESS;
}

int actionEditRecord(FileContext *context) {
  if (handleFileIfNotExist(context) == FAILURE)
    return FAILURE;

  if (handleFileIfEmpty(context) == FAILURE)
    return FAILURE;

  size_t recordNumber;
  readNumberWithValidation(&recordNumber, TYPE_SIZE_T, "Record number",
                      isNotZero);

  Region *regionBeforeEdited = readRegionAt(recordNumber - 1, context);
  if (!regionBeforeEdited)
    return FAILURE;
  freeRegion(regionBeforeEdited);

  long positionBeforeEdited = ftell(context->file);
  if (positionBeforeEdited < 0) {
    handleError("Failed to move file pointer to position before edited.");
    return FAILURE;
  }

  Region *region = initRegion();
  if (!region)
    return FAILURE;

  int readResult = readRegion(region, context);
  if (readResult != SUCCESS) {
    if (readResult == EOF)
      handleError("File doesn't contain region %zu", recordNumber);
    freeRegion(region);
    return FAILURE;
  }

  int editResult = editRegion(region, recordNumber);
  if (editResult == FAILURE) {
    freeRegion(region);
    return FAILURE;
  }
  if (editResult == NOOP) { // No changes were made
    showSuccess("No changes were made to the region so far.");
    freeRegion(region);
    return SUCCESS;
  }

```

```
  FileContext *temp = createTempFile();
  if (!temp)
    goto cleanup;

  if (copyRegionsBetweenFiles(context, temp) == FAILURE)
    goto cleanup;

  if (fseek(context->file, positionBeforeEdited, SEEK_SET) != SUCCESS) {
    handleError("Couldn't move back to the position before edited record");
    goto cleanup;
  }

  if (writeRegion(region, context) == FAILURE)
    goto cleanup;

  if (skipFileHeader(temp) == FAILURE)
    goto cleanup;
  if (copyRegionsBetweenFiles(temp, context) == FAILURE)
    goto cleanup;

  if (truncateFileFromCurrentPosition(context) == FAILURE)
    goto cleanup;

  if (handleFlushing(context) == FAILURE)
    goto cleanup;

  setQuickSortComparator(NULL); // The sorting might've been removed
  freeRegion(region);
  closeFileContext(temp);
  free(temp);
  temp = NULL;

  if (remove(TEMP) != SUCCESS) {
    handleError("Failed to delete temporary file.");
    return FAILURE;
  }

  showSuccess("Record was edited successfully.");
  return SUCCESS;

cleanup:
  if (temp) {
    closeFileContext(temp);
    free(temp);
    if (remove(TEMP) != SUCCESS) {
      handleError("Failed to delete temporary file.");
    }
  }
  freeRegion(region);
  return FAILURE;
}

int actionSortRecords(FileContext *context) {
  if (handleFileIfNotExist(context) == FAILURE)
    return FAILURE;

  if (handleFileIfEmpty(context) == FAILURE)
    return FAILURE;

  if (skipFileHeader(context) == FAILURE)
```

```
  return FAILURE;

size_t regionCount = 0;
Region **regions = readAllRegions(&regionCount, context);

if (!regions || regionCount == 0) {
  handleError(
      "Record file doesn't contain any regions, or they are malformed!");
  return FAILURE;
}

if (regionCount == 1) {
  showSuccess("The file contains 1 entry. It's already sorted.");
  return SUCCESS;
}

bool isSorting = true;

NumberRange *sortOptionRange = initChoiceRange(1, COMPARATOR_COUNT);
if (!sortOptionRange)
  goto cleanup;

do {
  size_t sortOption = getUserChoice(sortOptionRange, "Choose sorting method",
                                    SORT_BY_CHOICES) -
                      1;
  setQuickSortComparator(SORT_COMPARATORS[sortOption]);

  setQuickSortOrder(
      askQuestion("Do you want to sort ascending?") ? ASCENDING : DESCENDING);

  quickSort(regions, 0, regionCount - 1);

  for (size_t i = 0; i < regionCount; i++) {
    printRegion(regions[i], i + 1);
  }

  if (askQuestion("Do you want to write to the file?"))
    break;

  if (askQuestion("Do you want to sort it different way?"))
    continue;

  if (!askQuestion("Do you even want to continue?"))
    goto cleanup;
} while (isSorting);

freeNumberRange(sortOptionRange);

if (skipFileHeader(context) == FAILURE)
  goto cleanup;

for (size_t i = 0; i < regionCount; i++) {
  int result;
  if ((result = writeRegion(regions[i], context)) == FAILURE) {
    if (result == FAILURE)
      goto cleanup;
  }
}

```

```
  if (handleFlushing(context) == FAILURE)
    goto cleanup;

  showSuccess("Sorted regions were written successfully!");

  for (size_t i = 0; i < regionCount; i++)
    freeRegion(regions[i]);

  free(regions);
  return SUCCESS;

cleanup:
  if (regions)
    for (size_t i = 0; i < regionCount; i++) {
      freeRegion(regions[i]);
    }
  free(regions);
  return FAILURE;
}

int actionInsertRecordIfSorted(FileContext *context) {
  if (handleFileIfNotExist(context) == FAILURE)
    return FAILURE;

  if (handleFileIfEmpty(context) == FAILURE)
    return FAILURE;

  if (skipFileHeader(context) == FAILURE)
    return FAILURE;

  size_t regionCount = 0;
  Region **regions = readAllRegions(&regionCount, context);
  if (regions == NULL)
    return FAILURE;

  if (!detectSort(regions, regionCount)) {
    handleError("The file you provided is not sorted in any regular way.");
    if (askQuestion("Do you want to sort it?")) {
      if (actionSortRecords(context) == FAILURE)
        goto cleanup;
    }
  }

  showCurrentSortMethod();

  Region *newRegion = getRegionFromUser();
  if (!newRegion)
    goto cleanup;

  regionCount++;
  Region **newRegions = realloc(regions, regionCount * sizeof(Region *));
  if (newRegions == NULL) {
    handleErrorMemoryAllocation("extending region array");
    goto cleanup;
  }
  regions = newRegions;

  regions[regionCount - 1] = newRegion;
  quickSort(regions, 0, regionCount - 1);

```

```
  if (skipFileHeader(context) == FAILURE)
    goto cleanup;

  for (size_t i = 0; i < regionCount; i++) {
    if (writeRegion(regions[i], context) == FAILURE)
      goto cleanup;
  }

  for (size_t i = 0; i < regionCount; i++)
    freeRegion(regions[i]);
  free(regions);

  showSuccess("Inserted item successfully");
  return SUCCESS;

cleanup:
  for (size_t i = 0; i < regionCount; i++)
    freeRegion(regions[i]);
  free(regions);
  return FAILURE;
}

int actionDeleteRecord(FileContext *context) {
  if (handleFileIfNotExist(context) == FAILURE)
    return FAILURE;

  if (handleFileIfEmpty(context) == FAILURE)
    return FAILURE;

  if (skipFileHeader(context) == FAILURE)
    return FAILURE;

  size_t recordNumber;
  if (readNumberWithValidation(&recordNumber, TYPE_SIZE_T, "Record number",
                         isNotZero) == EOF)
    return FAILURE;

  Region *regionBeforeDeleted = readRegionAt(recordNumber - 1, context);
  if (!regionBeforeDeleted)
    return FAILURE;

  freeRegion(regionBeforeDeleted);

  long positionBeforeDeleted = ftell(context->file);

  Region *deleteRegion = initRegion();
  if (!deleteRegion)
    return FAILURE;

  if (readRegion(deleteRegion, context) != SUCCESS) {
    freeRegion(deleteRegion);
    return FAILURE;
  }

  printRegion(deleteRegion, recordNumber);
  freeRegion(deleteRegion);

  if (!askQuestion("Do you REALLY want to delete this region?"))
    return FAILURE;

```

```
  FileContext *temp = createTempFile();
  if (!temp)
    goto cleanup;

  if (copyRegionsBetweenFiles(context, temp) == FAILURE)
    goto cleanup;

  if (fseek(context->file, positionBeforeDeleted, SEEK_SET) != SUCCESS) {
    handleError("Failed to move to the position before deleted record.");
    goto cleanup;
  }

  if (skipFileHeader(temp) == FAILURE)
    goto cleanup;

  if (copyRegionsBetweenFiles(temp, context) == FAILURE)
    goto cleanup;

  if (truncateFileFromCurrentPosition(context) == FAILURE)
    goto cleanup;

  closeFileContext(temp);
  free(temp);
  if (remove(TEMP) != SUCCESS) {
    handleError("Failed to delete temporary file");
    goto cleanup;
  }

  if (handleFlushing(context) == FAILURE)
    goto cleanup;

  showSuccess("Record was deleted successfully.");
  return SUCCESS;
cleanup:
  if (temp) {
    closeFileContext(temp);
    free(temp);
    if (remove(TEMP) != SUCCESS) {
      handleError("Failed to delete temporary file");
    }
  }
  return FAILURE;
}
```

../src/actions/misc.c

```
#include "../common/common.h"
#include "../io/utils.h"
#include "actions.h"
#include "utils.h"
#include <stdlib.h>

int actionNotImplemented(FileContext *context) {
  handleError("This action isn't implemented yet!");
  return SUCCESS;
}

int actionExitProgram() { return EXIT; }

```

```
int actionVerifyFileHeader() {
  char *filename = getValidFilename();
  if (!filename)
    return FAILURE;

  FileContext *tempContext = initFileContext();
  if (!tempContext)
    goto cleanup;

  if (!fileExists(filename)) {
    handleError("This file doesn't exist, can't check the file header.");
    goto cleanup;
  }

  if (fillFileContext(tempContext, filename, SIGNATURE) == FAILURE)
    goto cleanup;

  if (handleBadFileHeader(tempContext) == FAILURE)
    goto cleanup;

  closeFileContext(tempContext);
  free(tempContext);
  free(filename);
  showSuccess("File \"%s\" does contain file header", filename);
  return SUCCESS;

cleanup:
  free(filename);
  if (tempContext) {
    closeFileContext(tempContext);
    free(tempContext);
  }
  return FAILURE;
}

int actionDisplayMenu() { return DISPLAY_MENU; }
```

../src/actions/regions.c

```
#include "../common/common.h"
#include "../io/io.h"
#include "../io/utils.h"
#include "utils.h"
#include <stdlib.h>
#include <string.h>
#define MAX_SIGNIFICANT_DIGITS 15

static double minArea = 1e-6;
static double maxArea = 510e6;
static const NumberRange areaRange = {
   &minArea, &maxArea, true, true, "Region area", compareDouble, TYPE_DOUBLE};

static double minPopulation = 1e-6;
static double maxPopulation = 1e10;

static const NumberRange populationRange = {
   &minPopulation, &maxPopulation, true, true,
   "Region population", compareDouble, TYPE_DOUBLE};

```

```c
Region *initRegion() {
  Region *region = malloc(sizeof(Region));
  if (!region) {
    handleErrorMemoryAllocation("region");
    return NULL;
  }

  region->name = NULL;
  region->area = 0.0;
  region->population = 0.0;
  return region;
}

void freeRegion(Region *region) {
  if (region->name) {
    free(region->name);
    region->name = NULL;
  }
  if (region)
    free(region);
}

int writeRegion(Region *region, FileContext *context) {
  if (handleFileIfNotExist(context) == FAILURE)
    return FAILURE;

  if (fprintf(context->file, "%s,%.*lg,%.*lg\n", region->name,
          MAX_SIGNIFICANT_DIGITS, region->area, MAX_SIGNIFICANT_DIGITS,
          region->population) < 0) {
    handleError("Failed to write to file %s.", context->filename);
    return FAILURE;
  }

  if (handleFlushing(context) == FAILURE)
    return FAILURE;
  return SUCCESS;
}

int readRegion(Region *region, FileContext *context) {
  if (!region) {
    handleError("Region is NULL in readRegion");
    return FAILURE;
  }
  if (handleFileIfNotExist(context) == FAILURE)
    return FAILURE;

  char *buffer = NULL;

  int result = SUCCESS;
  do {
    if (buffer)
      free(buffer);
    result = readLine(&buffer, DEFAULT_STRING_LENGTH, INFINITE_LENGTH,
                  context->file);
    if (result == EOF) {
      if (buffer) {
        free(buffer);
        buffer = NULL;
      }
      return EOF;
```

```
  }
  if (result == FAILURE) {
    handleError("Failed to read line in file %s.", context->filename);
    goto cleanup;
  }
} while (isLineEmpty(buffer));

// Locate commas
char *comma1 = strchr(buffer, ',');
char *comma2 = comma1 ? strchr(comma1 + 1, ',') : NULL;
char *comma3 = comma2 ? strchr(comma2 + 1, ',') : NULL;

if (!comma1 || !comma2) {
  handleError("Line doesn't contain enough commas.");
  goto cleanup;
}
if (comma3) {
  handleError("Line contains too many commas.");
  goto cleanup;
}

// Extract and allocate name
size_t nameLength = (size_t)(comma1 - buffer);
region->name = calloc(1 + nameLength, sizeof(char));

if (!region->name) {
  handleErrorMemoryAllocation("region name string");
  goto cleanup;
}
strncpy(region->name, buffer, nameLength);
if (!trimWhitespaceUtf8(region->name))
  goto cleanup;

if (region->name[0] == '\0') {
  handleError(
      "Region name became empty string after whitespace was trimmed.");
  goto cleanup;
}

region->name[nameLength] = '\0';

// Parse area
char *endPtr = NULL;
region->area = strtod(comma1 + 1, &endPtr);
if (endPtr == comma1 + 1 || endPtr != comma2) {
  handleError("Region area is not a valid number.");
  goto cleanup;
}

if (validateNumberRange(&region->area, &areaRange) != WITHIN_RANGE)
  goto cleanup;

// Parse population
endPtr = NULL;
region->population = strtod(comma2 + 1, &endPtr);

if (endPtr == comma2 + 1 || *endPtr != '\0') {
  handleError("Region population is not a valid number.");
  goto cleanup;
}
```

```
141
142   if (validateNumberRange(&region->population, &populationRange) !=
143       WITHIN_RANGE)
144     goto cleanup;
145
146   if (buffer)
147     free(buffer);
148   return SUCCESS;
149
150 cleanup:
151   if (buffer) {
152     free(buffer);
153     buffer = NULL;
154   }
155   return FAILURE;
156 }
157
158 Region *getRegionFromUser() {
159   Region *region = initRegion();
160   if (region == NULL)
161     return NULL;
162   region->name = readStringWithFilterUntilValid("Region name (without commas)",
163                                                 INFINITE_LENGTH, ",");
164   if (!region->name) {
165     free(region);
166     return NULL;
167   }
168
169   if (readNumberWithinRangeWithValidation(&region->area, TYPE_DOUBLE,
170                                           &areaRange, areaRange.valueName,
171                                           NULL) == EOF) {
172     free(region);
173     return NULL;
174   }
175
176   if (readNumberWithinRangeWithValidation(
177         &region->population, TYPE_DOUBLE, &populationRange,
178         populationRange.valueName, NULL) == EOF) {
179     free(region);
180     return NULL;
181   }
182
183   return region;
184 }
185
186 void printRegion(Region *region, size_t regionNumber) {
187   printf("\n");
188   if (regionNumber != 0)
189     printf("%zu) ", regionNumber);
190   printf("Name: %s\n", region->name);
191   printf("Area: %.*lg\n", MAX_SIGNIFICANT_DIGITS, region->area);
192   printf("Population: %.*lg\n", MAX_SIGNIFICANT_DIGITS, region->population);
193   printf("\n");
194 }
195
196 Region **readAllRegions(size_t *regionCount, FileContext *context) {
197   if (handleFileIfNotExist(context) == FAILURE)
198     return NULL;
199
200   if (!regionCount) {
```

```
    handleError("Region count is NULL.");
    return NULL;
  }

  size_t count = 0;
  size_t capacity = 4;
  size_t regionSize = capacity * sizeof(Region *);

  Region **regions = malloc(regionSize);
  if (!regions) {
    handleErrorMemoryAllocation("regions array");
    return NULL;
  }

  Region *newRegion = initRegion();
  if (!newRegion) {
    handleErrorMemoryAllocation("new region");
    goto cleanup;
  }

  int result = 0;

  while ((result = readRegion(newRegion, context)) != EOF) {
    if (result == FAILURE)
      goto cleanup;

    if (count == capacity) {
      capacity *= 2;
      Region **newBuffer = realloc(regions, capacity * sizeof(Region *));
      if (!newBuffer) {
        handleErrorMemoryAllocation("regions array resize");
        goto cleanup;
      }
      regions = newBuffer;
    }

    regions[count] = newRegion;
    count++;

    newRegion = initRegion();
    if (!newRegion) {
      handleErrorMemoryAllocation("new region");
      goto cleanup;
    }
  }

  free(newRegion);
  *regionCount = count;
  return regions;

cleanup:
  for (size_t i = 0; i < count; i++)
    freeRegion(regions[i]);
  free(regions);
  if (newRegion)
    freeRegion(newRegion);
  return NULL;
}

int editRegion(Region *region, size_t regionNumber) {
```

```
  char *nameBuffer = NULL;
  double newArea = region->area;
  double newPopulation = region->population;

  printRegion(region, regionNumber);

  if (askQuestion("Do you want to rename this region?")) {
    nameBuffer =
       readStringWithFilterUntilValid("Region name", INFINITE_LENGTH, ",");
    if (!nameBuffer)
      return FAILURE;
  }

  if (askQuestion("Do you want to change region area?"))
    if (readNumberWithinRangeWithValidation(&newArea, TYPE_DOUBLE, &areaRange,
                                    areaRange.valueName, NULL) == EOF)
      return FAILURE;

  if (askQuestion("Do you want to change region population?"))
    if (readNumberWithinRangeWithValidation(
          &newPopulation, TYPE_DOUBLE, &populationRange,
          populationRange.valueName, NULL) == FAILURE)
      return FAILURE;

  if (newArea == region->area && newPopulation == region->population &&
      nameBuffer == NULL)
    return NOOP;

  if (nameBuffer) {
    if (region->name)
      free(region->name);
    region->name = nameBuffer;
  }

  region->area = newArea;
  region->population = newPopulation;

  return SUCCESS;
}

int copyRegionsBetweenFiles(FileContext *source, FileContext *destination) {
  Region *region = initRegion();

  int result;
  while ((result = readRegion(region, source)) != EOF) {
    if (result == FAILURE || writeRegion(region, destination) == FAILURE) {
      freeRegion(region);
      return FAILURE;
    }
    freeRegion(region);
    region = initRegion();
  }

  freeRegion(region);

  return SUCCESS;
}

Region *readRegionAt(size_t position, FileContext *context) {
  if (handleFileIfNotExist(context) == FAILURE)
```

```
    return NULL;

  if (skipFileHeader(context) == FAILURE)
    return NULL;

  Region *region = initRegion();
  if (!region)
    return NULL;

  if (position == 0) {
    return region; // Noop (e.g. already passed nothing)
  }

  size_t i = 0;
  while (i != position) {
    int result = readRegion(region, context);
    if (result == EOF) {
      handleError("File doesn't contain region with such number");
      freeRegion(region);
      return NULL;
    }

    if (result == FAILURE) {
      freeRegion(region);
      region = initRegion();
      i++;
      continue;
    }

    i++;
    if (i == position) {
      return region;
    }

    freeRegion(region);
    region = initRegion();
    if (!region)
      return NULL;
  }

  if (!region->name) { // If nothing was found
    freeRegion(region);
    return NULL;
  }

  return region;

cleanup:
  freeRegion(region);
  return NULL;
}
```

../src/actions/utils.h

```
#ifndef ACTION_UTILS_H
#define ACTION_UTILS_H
#include "../common/common.h"
#include <stddef.h>

```

```
// GENERAL
int handleErrorFileNotSpecified();
int skipFileHeader(FileContext *context);
int writeSignature(FileContext *context);

void showCurrentSortMethod();
int handleBadFileHeader(FileContext *context);

FileContext *createTempFile();

int handleFlushing(FileContext *context);
int truncateFileFromCurrentPosition(FileContext *context);

int handleFileIfEmpty(FileContext *context);
int handleFileIfNotExist(FileContext *context);

char *getValidFilename();
bool fileExists(const char *filename);

// REGIONS
int readRegion(Region *region, FileContext *context);
Region *initRegion();
void freeRegion(Region *region);
void printRegion(Region *region, size_t regionNumber);

Region *getRegionFromUser();
int writeRegion(Region *region, FileContext *context);
int editRegion(Region *region, size_t regionNumber);
int copyRegionsBetweenFiles(FileContext *source, FileContext *destination);
Region *readRegionAt(size_t recordNumber, FileContext *context);

Region **readAllRegions(size_t *regionCount, FileContext *context);
#endif
```

../src/actions/utils.c

```
#include "utils.h"
#include "../common/common.h"
#include "../io/io.h"
#include "../io/utils.h"
#include "actions.h"
#include <errno.h>
#include <linux/limits.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static const char *filenameReject = "<>:\"/\\|?*";
static const char *disallowedNames[] = {".", ".."};

bool fileExists(const char *filename) {
  int result = access(filename, F_OK);
  if (errno == ENOENT ||
     errno == 0) { // if there's error related not to file non-existence
    errno = 0;
  } else {
    handleError("");
  }
  return result == SUCCESS;
```

```
}

bool isFilenameWithinLength(const char *filename) {
  return strlen(filename) <= NAME_MAX;
}

char *getValidFilename() {
  char *filename = readStringWithFilterUntilValid(
     "Name of file with records", MAX_FILENAME_LENGTH, filenameReject);

  if (!filename)
    return NULL;

  if (filename[0] == '\0') {
    handleError("String should have at least one printable character.");
    free(filename);
    return NULL;
  }

  if (findCharsInUtf8String(filename, filenameReject) !=
     getUtf8StringLength(filename)) {
    handleError("String contains one of forbidden symbols: \"%s\"",
                filenameReject);
    free(filename);
    return NULL;
  }
  size_t disallowedNamesLength =
     sizeof(disallowedNames) / sizeof(disallowedNames[0]);
  for (size_t i = 0; i < disallowedNamesLength; i++) {

    if (compareUtf8Strings(filename, disallowedNames[i]) == 0) {
      handleError("Name %s is not allowed in this OS.", disallowedNames[i],
                  filename);
      free(filename);
      return NULL;
    }
  }

  if (hasSuffix(filename, ".c") || hasSuffix(filename, ".h")) {
    handleError("It's forbidden to create or edit files with extensions of C "
                "Programming Language.");
    free(filename);
    return NULL;
  }

  return filename;
}

int handleFileIfNotExist(FileContext *context) {
  if (!context) {
    handleError("Context is NULL");
    return FAILURE;
  }
  if (context->file == NULL) {
    handleErrorFileNotSpecified();
    if (!askQuestion("Do you want to create/select file?"))
      return FAILURE;
    if (actionCreateFile(context) == FAILURE)
      return FAILURE;
  }
```

```c
  return SUCCESS;
}

int handleFileIfEmpty(FileContext *context) {
  if (handleFileIfNotExist(context) == FAILURE)
    return FAILURE;

  if (skipFileHeader(context) == FAILURE)
    return FAILURE;

  char *line = NULL;
  int result = 0;
  while ((result = readLine(&line, DEFAULT_STRING_LENGTH, INFINITE_LENGTH,
                      context->file)) != EOF) {
    if (result == FAILURE) {
      if (line)
        free(line);
      return FAILURE;
    }

    if (!isLineEmpty(line)) {
      free(line);
      return SUCCESS;
    }
    free(line);
  }

  handleError("This file is empty, consider creating some records first.");
  if (!askQuestion("Do you want to create a record?"))
    return FAILURE;
  return actionCreateRecord(context);
}

int handleFlushing(FileContext *context) {
  if (!context->file)
    return FAILURE;

  if (fflush(context->file) == EOF) {
    handleError("Failed to flush the data into %s", context->filename);
    return FAILURE;
  }

  return SUCCESS;
}

// OUTPUT

int handleErrorFileNotSpecified() {
  return handleError("No file with records was chosen. Please, create file "
                "with records or select it.");
}

int handleBadFileHeader(FileContext *context) {
  if (handleFileIfNotExist(context) == FAILURE)
    return FAILURE;

  rewind(context->file);
  char buffer[context->signatureSize + 1];

  if (!fgets(buffer, context->signatureSize + 1, context->file)) {
```

```
144     handleError("Failed to read signature from file!");
145     return FAILURE;
146   }
147
148   if (strcmp(buffer, context->signature) != 0) {
149     handleError("File %s doesn't contain the signature %s!", context->filename,
150                 context->signature);
151     closeFileContext(context);
152     return FAILURE;
153   }
154
155   return SUCCESS;
156 }
157
158 int writeSignature(FileContext *context) {
159   if (!context->file) {
160     handleErrorFileNotSpecified();
161     return FAILURE;
162   }
163
164   if (fprintf(context->file, "%s", context->signature) < 0) {
165     handleError("Failed to write signature to the file %s", context->filename);
166     return FAILURE;
167   }
168
169   if (handleFlushing(context) == FAILURE)
170     return FAILURE;
171
172   return SUCCESS;
173 }
174
175 int skipFileHeader(FileContext *context) {
176   if (handleFileIfNotExist(context) == FAILURE)
177     return FAILURE;
178
179   if (fseek(context->file, context->signatureSize, SEEK_SET) != SUCCESS) {
180     handleError("Couldn't go to the position after signature.");
181     return FAILURE;
182   }
183
184   return SUCCESS;
185 }
186
187 FileContext *createTempFile() {
188   if (fileExists(TEMP)) {
189     if (remove(TEMP) != SUCCESS) {
190       handleError("Failed to remove the old temporary file");
191       return NULL;
192     }
193   }
194
195   FileContext *temp = initFileContext();
196   if (!temp)
197     return NULL;
198
199   if (fillFileContext(temp, TEMP, SIGNATURE) == FAILURE) {
200     closeFileContext(temp);
201     free(temp);
202     return NULL;
203   }
```

```
204
205   return temp;
206 }
207
208 int truncateFileFromCurrentPosition(FileContext *context) {
209   if (handleFileIfNotExist(context) == FAILURE)
210     return FAILURE;
211
212   long position = ftell(context->file);
213
214   if (position == -1) {
215     handleError("Failed to get the position in file.");
216     return FAILURE;
217   }
218
219   int fd = fileno(context->file);
220   if (fd == -1) {
221     handleError("Failed to get the file descriptor.");
222     return FAILURE;
223   }
224
225   if (ftruncate(fd, position) == -1) {
226     handleError("Failed to truncate the file.");
227     return FAILURE;
228   }
229   return SUCCESS;
230 }
```

../src/io/io.h

```
1  #ifndef IO_H
2  #define IO_H
3
4  #include "../common/common.h"
5
6  // NUMBER
7  void printNumberRange(const NumberRange *range);
8  int readNumberWithinRange(void *value, NumberType type,
9                            const NumberRange *range, const char *prompt);
10 int readNumberWithinRangeWithValidation(void *value, NumberType type,
11                                         const NumberRange *range,
12                                         const char *prompt,
13                                         ValidationFunc additionalCheck, ...);
14
15 extern char *numericTypeDescriptions[TYPE_COUNT];
16
17 NumberRange *initializeNumberRange(NumberType type, void *min, void *max,
18                                    bool isMinIncluded, bool isMaxIncluded,
19                                    const char *valueName);
20
21 int readNumberWithValidation(void *value, NumberType type, const char *prompt,
22                              ValidationFunc additionalCheck, ...);
23
24 RangeCheckResult validateNumberRange(const void *value,
25                                      const NumberRange *range);
26
27 int compareDouble(const void *a, const void *b);
28 int compareLongUnsigned(const void *a, const void *b);
29 void freeNumberRange(NumberRange *range);
```

```c
void printNumber(const void *value, NumberType type);

// STRINGS
bool isLineEmpty(const char *line);
bool isValidUtf8(const char *str);
char *trimWhitespaceUtf8(char *str);
size_t getUtf8StringLength(const char *str);
size_t findCharsInUtf8String(const char *str, const char *reject);
char *readStringWithFilterUntilValid(const char *prompt, size_t maxLength,
                                     const char *reject);
bool hasSuffix(const char *filename, const char *suffix);
int compareUtf8Strings(const char *str1, const char *str2);

// CHOICES
size_t getUserChoice(NumberRange *choiceRange, const char *info,
                     const char *choices[]);

NumberRange *initChoiceRange(size_t start, size_t end);

// VALIDATORS
bool isNotZero(void *value, va_list args);
#endif
```

../src/io/number.c

```c
#include "io.h"
#include "utils.h"
#include <errno.h>
#include <float.h>
#include <limits.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

const char *typeDescriptions[TYPE_COUNT] = {"floating-point decimal",
                                            "length of arrays"};

size_t getTypeSize(NumberType type) {
  switch (type) {
  case TYPE_SIZE_T:
    return sizeof(size_t);
  case TYPE_DOUBLE:
    return sizeof(double);
  default:
    handleError("Number type not implemented.");
    return 0;
  }
}

int compareDouble(const void *a, const void *b) {
  if (*(double *)a < *(double *)b)
    return -1;
  if (*(double *)a > *(double *)b)
    return 1;
  return 0;
}

```

```c
int compareSizeT(const void *a, const void *b) {
  if (*(size_t *)a < *(size_t *)b)
    return -1;
  if (*(size_t *)a > *(size_t *)b)
    return 1;
  return 0;
}

NumberRange *initializeNumberRange(NumberType type, void *min, void *max,
                                   bool isMinIncluded, bool isMaxIncluded,
                                   const char *valueName) {
  NumberRange *range = malloc(sizeof(NumberRange));
  if (!range) {
    handleErrorMemoryAllocation("range");
    return NULL;
  }

  range->type = type;
  size_t typeSize = getTypeSize(type);

  range->min = malloc(typeSize);
  if (!range->min) {
    handleErrorMemoryAllocation("number_range_min");
    freeNumberRange(range);
    return NULL;
  }

  range->max = malloc(typeSize);
  if (!range->max) {
    handleError("Memory_allocation_failed_for_range_min/max_values.");
    freeNumberRange(range);
    return NULL;
  }

  switch (type) {
  case TYPE_SIZE_T:
    *(size_t *)range->min = *(size_t *)min;
    *(size_t *)range->max = *(size_t *)max;
    range->compare = compareSizeT;
    break;

  case TYPE_DOUBLE:
    *(double *)range->min = *(double *)min;
    *(double *)range->max = *(double *)max;
    range->compare = compareDouble;
    break;
  default:
    handleError("Unsupported_type_in_range_initialization.");
    freeNumberRange(range);
    return NULL;
  }

  range->isMinIncluded = isMinIncluded;
  range->isMaxIncluded = isMaxIncluded;
  range->valueName = strdup(valueName);
  if (!range->valueName) {
    handleErrorMemoryAllocation("number_range_value_name");
    freeNumberRange(range);
    return NULL;
  }
```

```
 95
 96   return range;
 97 }
 98
 99 void freeNumberRange(NumberRange *range) {
100   if (!range)
101     return;
102   if (range->min)
103     free(range->min);
104   if (range->max)
105     free(range->max);
106   if (range->valueName)
107     free(range->valueName);
108   free(range);
109 }
110
111 RangeCheckResult validateNumberRange(const void *value,
112                                      const NumberRange *range) {
113   const char *modal = "should be";
114   if (range->compare(value, range->min) < 0 && range->isMinIncluded) {
115     handleError("%s %s greater or equal to min.", range->valueName, modal);
116     return LESS;
117   }
118
119   if (range->compare(value, range->min) <= 0 && !range->isMinIncluded) {
120     handleError("%s %s greater than min.", range->valueName, modal);
121     return LESS_EQUAL;
122   }
123
124   if (range->compare(value, range->max) > 0 && range->isMaxIncluded) {
125     handleError("%s %s less or equal to max.", range->valueName, modal);
126     return GREATER;
127   }
128
129   if (range->compare(value, range->max) >= 0 && !range->isMaxIncluded) {
130     handleError("%s %s less than max.", range->valueName, modal);
131     return GREATER_EQUAL;
132   }
133
134   return WITHIN_RANGE;
135 }
136
137 void printNumber(const void *value, NumberType type) {
138   switch (type) {
139   case TYPE_SIZE_T:
140     printf("%zu", *(size_t *)value);
141     break;
142   case TYPE_DOUBLE:
143     printf("%.*lg", DBL_DIG, *(double *)value);
144     break;
145   default:
146     handleError("Unknown type");
147     break;
148   }
149 }
150
151 void printNumberRange(const NumberRange *range) {
152   printf("from ");
153   printNumber(range->min, range->type);
154   printf(" to ");
```

```
  printNumber(range->max, range->type);
}

int getMaxCharCountNumber(NumberType type) {
  switch (type) {
  case TYPE_SIZE_T:
    return snprintf(NULL, 0, "%zu", SIZE_MAX);
  case TYPE_DOUBLE:
    return snprintf(NULL, 0, "%.*e", DBL_DIG, DBL_MAX);
  default:
    handleError("No such type found, aborting.");
    return 0;
  }
}

void printCharacterCountRangeNumber(NumberType type) {
  int max = getMaxCharCountNumber(type);
  if (max <= 0) {
    handleError("Failed to find max character count for type");
    return;
  }
  printf("character count for %s ≤ %u", typeDescriptions[type], max);
}

int convertInputToNumber(const char *input, void *value, NumberType type) {
  char *endptr = NULL;
  errno = 0;

  switch (type) {
  case TYPE_SIZE_T: {
    unsigned long long temp = 0;

    if (sizeof(size_t) == sizeof(unsigned long)) {
      temp = strtoul(input, &endptr, 10);
    } else if (sizeof(size_t) == sizeof(unsigned long long)) {
      temp = strtoull(input, &endptr, 10);
    } else if (sizeof(size_t) == sizeof(unsigned int)) {
      temp = strtol(input, &endptr, 10);
    } else {
      handleError("Unsupported size_t size");
      return FAILURE;
    }

    if (errno == ERANGE || temp > SIZE_MAX) {
      handleErrorOverflow();
      return FAILURE;
    }

    *(size_t *)value = (size_t)temp;

    break;
  }

  case TYPE_DOUBLE: {
    double temp = strtod(input, &endptr);
    if (errno == ERANGE) {
      handleErrorOverflow();
      return FAILURE;
    }
    *(double *)value = temp;
```

```
    break;
  }

  default:
    handleError("Unsupported number type");
    return FAILURE;
  }

  // Check if there are any non-number characters left after the number
  if (*endptr != '\0') {
    handleErrorNotNumber();
    return FAILURE;
  }

  return SUCCESS;
}

int readNumber(void *value, NumberType type) {
  int maxCharCount = getMaxCharCountNumber(type);
  if (maxCharCount <= 0) {
    handleError("Failed to find max character count for type");
    return FAILURE;
  }
  char *input = NULL;

  int result = readLine(&input, DEFAULT_STRING_LENGTH, maxCharCount, stdin);

  if (result != SUCCESS)
    return result;

  if (!input)
    return FAILURE;

  if (convertInputToNumber(input, value, type) == FAILURE) {
    free(input);
    input = NULL;
    return FAILURE;
  }

  free(input);
  input = NULL;
  return SUCCESS;
}

int readNumberWithinRange(void *value, NumberType type,
                          const NumberRange *range, const char *prompt) {
  printf("%s (", prompt);
  printNumberRange(range);
  printf("): ");

  int result = readNumber(value, type);
  switch (result) {
  case FAILURE:
    return FAILURE;
  case EOF:
    return EOF;
  }

  if (validateNumberRange(value, range) != WITHIN_RANGE)
    return FAILURE;
```

```c

  return SUCCESS;
}

int readNumberWithValidation(void *value, NumberType type, const char *prompt,
                             ValidationFunc additionalCheck, ...) {
  bool isValid = false;

  va_list args;

  while (!isValid) {
    printf("%s: ", prompt);
    int result = readNumber(value, type);
    if (result == SUCCESS) {
      if (additionalCheck != NULL) {
        va_start(args, additionalCheck);
        isValid = additionalCheck(value, args);
        va_end(args);
      } else {
        isValid = true;
      }
    } else if (result == EOF) {
      return EOF;
    }
  }

  return SUCCESS;
}

int readNumberWithinRangeWithValidation(void *value, NumberType type,
                                        const NumberRange *range,
                                        const char *prompt,
                                        ValidationFunc additionalCheck, ...) {
  bool isValid = false;

  va_list args;

  while (!isValid) {
    int result = readNumberWithinRange(value, type, range, prompt);
    if (result == SUCCESS) {
      if (additionalCheck != NULL) {
        va_start(args, additionalCheck);
        isValid = additionalCheck(value, args);
        va_end(args);
      } else {
        isValid = true;
      }
    } else if (result == EOF) {
      return EOF;
    }
  }

  return SUCCESS;
}
```

../src/io/string.c

```c
#include "io.h"
#include "utils.h"
```

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

// UTF8

// NOTE: The function also advances the pointer to the next character
static int decodeUtf8(const char **str) {
  const unsigned char *s = (const unsigned char *)*str;
  int codepoint = 0;

  if (*s < 0x80) {
    // 1-byte character: 0xxxxxxx
    codepoint = *s++;
  } else if ((*s & 0xE0) == 0xC0) {
    // 2-byte character: 110xxxxx 10xxxxxx
    codepoint = (*s++ & 0x1F) << 6;
    codepoint |= (*s++ & 0x3F);
  } else if ((*s & 0xF0) == 0xE0) {
    // 3-byte character: 1110xxxx 10xxxxxx 10xxxxxx
    codepoint = (*s++ & 0x0F) << 12;
    codepoint |= (*s++ & 0x3F) << 6;
    codepoint |= (*s++ & 0x3F);
  } else if ((*s & 0xF8) == 0xF0) {
    // 4-byte character: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
    codepoint = (*s++ & 0x07) << 18;
    codepoint |= (*s++ & 0x3F) << 12;
    codepoint |= (*s++ & 0x3F) << 6;
    codepoint |= (*s++ & 0x3F);
  } else {
    // Invalid UTF-8
    return -1;
  }

  *str = (const char *)s;
  return codepoint;
}

// Checks string
bool isValidUtf8(const char *str) {
  if (!str)
    return false;

  const char *ptr = str;

  while (*ptr) {
    int codepoint = decodeUtf8(&ptr);
    if (codepoint == -1) {
      handleError("Invalid UTF-8 sequence.");
      return false;
    }
  }

  return true;
}

size_t getUtf8StringLength(const char *str) {
  const char *ptr = str;
  size_t length = 0;
```

```c

  while (*ptr) {
    decodeUtf8(&ptr);
    length++;
  }

  return length;
}

static char *readUtf8Char(const char *source) {
  const char *ptr = source;
  int codepoint = decodeUtf8(&ptr);

  if (codepoint == -1) {
    return NULL; // Invalid UTF-8
  }

  size_t len = ptr - source;
  char *buffer = malloc(len + 1); // +1 for null terminator
  if (!buffer) {
    return NULL;
  }

  memcpy(buffer, source, len);
  buffer[len] = '\0';
  return buffer;
}

size_t findCharsInUtf8String(const char *str, const char *reject) {
  const char *strPtr = str;
  size_t pos = 0;

  while (*strPtr) {
    const char *rejectPtr = reject;
    int codepointStr = decodeUtf8(&strPtr);

    if (codepointStr == -1) {
      handleError("Invalid UTF-8 character encountered.");
      return pos;
    }

    while (*rejectPtr) {
      int codepointReject = decodeUtf8(&rejectPtr);

      if (codepointReject == -1) {
        handleError("Invalid UTF-8 character in reject set.");
        return pos;
      }

      if (codepointStr == codepointReject) {
        return pos;
      }
    }

    pos++;
  }

  return pos;
}

```

```c
int compareUtf8Strings(const char *str1, const char *str2) {
  while (*str1 && *str2) {
    int codepoint1 = decodeUtf8(&str1);
    int codepoint2 = decodeUtf8(&str2);

    if (codepoint1 < 0 || codepoint2 < 0) {
      // Handle invalid UTF-8
      return codepoint1 - codepoint2;
    }

    if (codepoint1 != codepoint2) {
      return codepoint1 - codepoint2;
    }
  }

  // If one string is longer, it's greater
  return *str1 - *str2;
}

bool hasSuffix(const char *word, const char *suffix) {
  size_t wordLength = getUtf8StringLength(word);
  size_t suffixLength = getUtf8StringLength(suffix);

  if (wordLength < suffixLength) {
    return false;
  }

  const char *wordPtr = word;
  const char *suffixPtr = suffix;

  // Move to the possible position for the suffix
  for (int i = 0; i < wordLength - suffixLength; i++)
    decodeUtf8(&wordPtr);

  while (*suffixPtr) {
    int wordCodepoint = decodeUtf8(&wordPtr);
    int suffixCodepoint = decodeUtf8(&suffixPtr);

    if (wordCodepoint != suffixCodepoint)
      return false;
  }

  return true;
}

bool isWhitespace(char c) {
  return (c == ' ' || c == '\t' || c == '\n' || c == '\r' || c == '\v' ||
          c == '\f');
}

bool isLineEmpty(const char *line) {
  if (!line)
    return true;

  for (size_t i = 0; line[i] != '\0'; i++) {
    if (!isWhitespace(line[i]) && line[i] != ' ') {
      return false;
    }
  }
  return true;
```

```
}

char *trimWhitespaceUtf8(char *str) {
  if (!str) {
    handleError("Passed NULL to trimWhitespaceUtf8");
    return NULL;
  }

  char *start = str;
  char *end = str + strlen(str);

  // trim initial
  while (*start && isWhitespace(*start)) {
    start++;
  }

  // trim trailing
  while (end > start && isWhitespace(*(end - 1))) {
    end--;
  }

  *end = '\0';

  size_t trimmedLength = strlen(start);
  memmove(str, start, trimmedLength + 1);

  return str;
}

char *readStringWithFilterUntilValid(const char *prompt, size_t maxLength,
                                     const char *reject) {
  char *input = NULL;
  bool isReadingInput = true;

  do {
    printf("%s: ", prompt);
    int result = readLine(&input, DEFAULT_STRING_LENGTH, maxLength, stdin);

    if (result == FAILURE)
      return NULL;

    if (result == EOF)
      return NULL;

    if (!isValidUtf8(input)) {
      free(input);
      input = NULL;
      continue;
    }

    size_t initialLength = getUtf8StringLength(input);
    trimWhitespaceUtf8(input);

    isReadingInput = false;
  } while (isReadingInput);

  return input;
}
```

../src/io/choices.c

```c
#include "io.h"
#include <stdlib.h>

#include <stddef.h>
#include <stdio.h>

#define DEFAULT_CHOICE_PROMPT "Choose option to select"
void showChoices(const char **choices, int choice_count) {
  for (int i = 0; i < choice_count; i++) {
    printf("%d. %s\n", i + 1, choices[i]);
  }
  printf("\n");
}

size_t getUserChoice(NumberRange *choiceRange, const char *info,
                     const char *choices[]) {

  if (info)
    printf("%s:\n", info);
  if (choices)
    showChoices(choices, *(int *)choiceRange->max);

  size_t start = 1;
  size_t end = *(size_t *)choiceRange->max;

  size_t choice;
  if (readNumberWithinRangeWithValidation(&choice, TYPE_SIZE_T, choiceRange,
                                          DEFAULT_CHOICE_PROMPT, NULL) == EOF) {
    return 0;
  }

  return choice;
}

NumberRange *initChoiceRange(size_t start, size_t end) {
  NumberRange *range =
      initializeNumberRange(TYPE_SIZE_T, &start, &end, true, true, "choice");
  if (!range)
    return NULL;
  return range;
}
```

../src/io/validators.c

```c
#include "io.h"
#include "utils.h"
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>

// INFO: Only for size_t for now
bool isNotZero(void *value, va_list args) {
  size_t val = *(size_t *)value;
  if (val == 0) {
    handleError("Value can't be 0!");
    return false;
  }
```

```
14   return true;
15 }
```

../src/io/utils.h

```
1  #ifndef UTILS_H
2  #define UTILS_H
3
4  #define RED "\033[31m"
5  #define GREEN "\033[32m"
6  #define YELLOW "\033[33m"
7  #define RESET "\033[0m"
8
9  #include <stdbool.h>
10 #include <stdio.h>
11
12 bool askQuestion(const char *format, ...);
13 void discardTillNewline();
14 int readLine(char **buffer, size_t initialLength, size_t maxLength,
15           FILE *stream);
16
17 void clearLastLines(int count);
18 int handleError(const char *format, ...);
19 int handleErrorMemoryAllocation(const char *reason);
20 int handleErrorOverflow();
21 int handleErrorOverlength(size_t maxCharCount);
22 int handleErrorNotNumber();
23 int showWarning(const char *format, ...);
24 int showSuccess(const char *format, ...);
25
26 #endif
```

../src/io/utils.c

```
1  #include "utils.h"
2  #include "../common/common.h"
3  #include <stdarg.h>
4  #include <stdbool.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8
9  // INPUT
10
11 void discardTillNewline() {
12   int c;
13   while ((c = getchar()) != '\n' && c != EOF)
14     ;
15 }
16
17 bool isAgree() {
18   char choice;
19
20   printf("Enter␣'y'␣if␣you␣agree.␣Otherwise␣press␣whatever...␣");
21
22   choice = getchar();
23   if (choice == '\n')
```

```
24    return false;
25
26  discardTillNewline();
27
28  if (choice == 'y') {
29    return true;
30  }
31
32  return false;
33 }
34
35 bool askQuestion(const char *format, ...) {
36  va_list args;
37  va_start(args, format);
38  printf("\n" YELLOW);
39  vprintf(format, args);
40  printf(RESET "\n");
41  va_end(args);
42
43  return isAgree();
44 }
45
46 int readLine(char **buffer, size_t initialLength, size_t maxLength,
47           FILE *stream) {
48  if (buffer == NULL || stream == NULL || initialLength == 0) {
49    handleError("Invalid arguments to readLine.");
50    return FAILURE;
51  }
52
53  size_t bufferSize = initialLength;
54  char *input = (char *)malloc(bufferSize * sizeof(char));
55  if (input == NULL) {
56    handleErrorMemoryAllocation("input buffer");
57    return FAILURE;
58  }
59
60  size_t filledBufferSize = 0;
61  bool isCompleted = false;
62
63  while (!isCompleted) {
64    if (!fgets(input + filledBufferSize, bufferSize - filledBufferSize,
65             stream)) {
66      if (feof(stream)) {
67        free(input);
68        return EOF;
69      }
70
71      free(input);
72      handleError("Failed to read input.");
73      return FAILURE;
74    } else {
75      filledBufferSize += strlen(input + filledBufferSize);
76      if (input[filledBufferSize - 1] == '\n') {
77        isCompleted = true;
78        input[filledBufferSize - 1] = '\0'; // Remove newline character
79      }
80    }
81
82    if (!isCompleted && filledBufferSize == bufferSize - 1) {
83      size_t newBufferSize = bufferSize * 2;
```

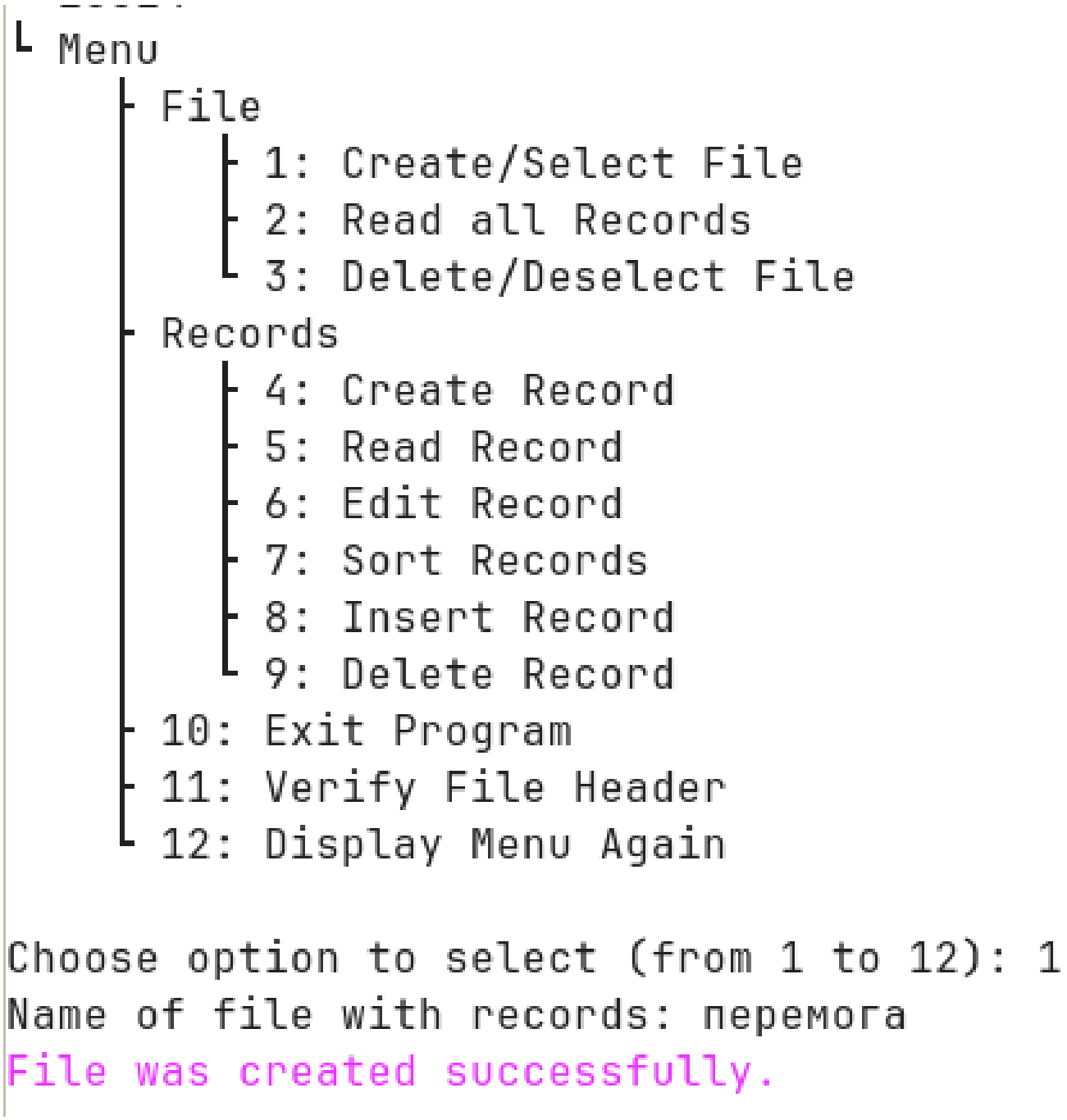
```c
    if (maxLength != INFINITE_LENGTH && newBufferSize > maxLength) {
      newBufferSize = maxLength;
    }

    if (newBufferSize <= bufferSize) {
      discardTillNewline();
      handleErrorOverlength(maxLength);
      free(input);
      return FAILURE;
    }

    char *newBuffer = realloc(input, newBufferSize * sizeof(char));
    if (newBuffer == NULL) {
      free(input);
      handleError("Failed to expand input buffer.");
      return FAILURE;
    }

    input = newBuffer;
    bufferSize = newBufferSize;
  }

  if (maxLength != INFINITE_LENGTH && filledBufferSize >= maxLength) {
    discardTillNewline();
    handleErrorOverlength(maxLength);
    free(input);
    return FAILURE;
  }
 }

 *buffer = input; // Final buffer contains the input
 return SUCCESS;
}

// OUTPUT
#include <errno.h>

#define MOVE_UP "\033[F"
#define CLEAR_LINE "\033[2K"

#define ERROR_CLEAR_LINE_COUNT 3

void clearLastLines(int count) {
 for (int i = 0; i < count; i++) {
   printf(MOVE_UP CLEAR_LINE);
 }
}

void continueAfterError() {
 printf("Press Enter to continue: ");
 fflush(stdout);
 discardTillNewline(); // Getting any input from user
 // NOTE: Remove this line when showcasing via screenshotting
 clearLastLines(ERROR_CLEAR_LINE_COUNT);
}

int handleError(const char *format, ...) {
 va_list args;
 va_start(args, format);

```

```
144   printf("\n" RED "ERROR!␣");
145   vprintf(format, args);
146   if (errno != 0) {
147     perror("");
148     errno = 0;
149   }
150   printf(RESET "\n");
151 
152   va_end(args);
153   continueAfterError();
154   return PIPE;
155 }
156 
157 int handleErrorOverflow() {
158   return handleError("Number␣is␣outside␣the␣allowed␣values␣of␣type!");
159 }
160 
161 int warnNotPrecise(int maxSignificantDigits) {
162   return showWarning(
163       "The␣number␣of␣significant␣digits␣exceeds␣max␣allowed␣significant␣digits␣"
164       "count␣%d,␣so␣some␣calculations␣may␣be␣imprecise.",
165       maxSignificantDigits);
166 }
167 
168 int handleErrorOverlength(size_t maxCharCount) {
169   return handleError("The␣string␣length␣should␣be␣>␣0␣and␣<␣%llu",
170                      maxCharCount);
171 }
172 
173 int handleErrorMemoryAllocation(const char *sufferer) {
174   return handleError("Failed␣to␣allocate␣memory␣for␣%s.", sufferer);
175 }
176 
177 int handleErrorNotNumber() {
178   return handleError(
179       "Value␣should␣be␣a␣number␣and␣not␣contain␣any␣additional␣characters!");
180 }
181 
182 int showWarning(const char *format, ...) {
183   va_list args;
184   va_start(args, format);
185 
186   printf("\n" YELLOW "ATTENTION!␣");
187   vprintf(format, args);
188   printf(RESET "\n");
189 
190   va_end(args);
191   return PIPE;
192 }
193 
194 int showSuccess(const char *format, ...) {
195   va_list args;
196   va_start(args, format);
197 
198   printf(GREEN);
199   vprintf(format, args);
200   printf(RESET "\n");
201 
202   va_end(args);
203   return PIPE;
```

```
204 }
```

Введені та одержані результати

Створення файлу



Створення записів

```
Choose option to select (from 1 to 12): 4
Region name (without commas): Київ
Region area (from 1e-06 to 510000000): 839
Region population (from 1e-06 to 10000000000): 2.952e6

Name: Київ
Area: 839
Population: 2952000

Record was appended successfully.

Choose option to select (from 1 to 12): 4
Region name (without commas): Одеса
Region area (from 1e-06 to 510000000): 162.42
Region population (from 1e-06 to 10000000000): 992874

Name: Одеса
Area: 162.42
Population: 992874

Record was appended successfully.

Choose option to select (from 1 to 12): 4
Region name (without commas): Україна
Region area (from 1e-06 to 510000000): 603628
Region population (from 1e-06 to 10000000000): 38e6

Name: Україна
Area: 603628
Population: 38000000

Record was appended successfully.
```

Зчитування всіх записів

```
Choose option to select (from 1 to 12): 2

1) Name: Київ
Area: 839
Population: 2952000


2) Name: Одеса
Area: 162.42
Population: 992874


3) Name: Україна
Area: 603628
Population: 38000000

File was read successfully.
```

Видалення запису

```
Choose option to select (from 1 to 12): 9
Record number: 4

4) Name: Україна
Area: 603628
Population: 38000000


Do you REALLY want to delete this region?
Enter 'y' if you agree. Otherwise press whatever... y
Record was deleted successfully.

Choose option to select (from 1 to 12): 2

1) Name: Харків
Area: 310
Population: 1402000


2) Name: Львівська область
Area: 21823.7
Population: 2515000


3) Name: Київ
Area: 839
Population: 2952000
```

Редагування запису

```
Choose option to select (from 1 to 12): 6
Record number: 2

2) Name: Одеса
Area: 162.42
Population: 992874


Do you want to rename this region?
Enter 'y' if you agree. Otherwise press whatever... y
Region name: Харків

Do you want to change region area?
Enter 'y' if you agree. Otherwise press whatever... y
Region area (from 1e-06 to 510000000): 310

Do you want to change region population?
Enter 'y' if you agree. Otherwise press whatever... y
Region population (from 1e-06 to 10000000000): 1.402e6
Record was edited successfully.
```

Зчитування запису

```
Choose option to select (from 1 to 12): 5
Record number: 2

2) Name: Харків
Area: 310
Population: 1402000
```

Впорядкування записів

```
Choose option to select (from 1 to 12): 7
Choose sorting method:
1. by region name
2. by region area
3. by region population

Choose option to select (from 1 to 3): 1

Do you want to sort ascending?
Enter 'y' if you agree. Otherwise press whatever... y

1) Name: Київ
Area: 839
Population: 2952000


2) Name: Україна
Area: 603628
Population: 38000000


3) Name: Харків
Area: 310
Population: 1402000


Do you want to write to the file?
Enter 'y' if you agree. Otherwise press whatever... n

Do you want to sort it different way?
Enter 'y' if you agree. Otherwise press whatever... y
```

```
Choose option to select (from 1 to 12): 7
Choose sorting method:
1. by region name
2. by region area
3. by region population

Choose option to select (from 1 to 3): 1

Do you want to sort ascending?
Enter 'y' if you agree. Otherwise press whatever... n

1) Name: Харків
Area: 310
Population: 1402000


2) Name: Україна
Area: 603628
Population: 38000000


3) Name: Київ
Area: 839
Population: 2952000


Do you want to write to the file?
Enter 'y' if you agree. Otherwise press whatever... n
```

```
Do you want to sort it different way?
Enter 'y' if you agree. Otherwise press whatever... y
Choose sorting method:
1. by region name
2. by region area
3. by region population

Choose option to select (from 1 to 3): 2

Do you want to sort ascending?
Enter 'y' if you agree. Otherwise press whatever... n

1) Name: Україна
Area: 603628
Population: 38000000


2) Name: Київ
Area: 839
Population: 2952000


3) Name: Харків
Area: 310
Population: 1402000


Do you want to write to the file?
Enter 'y' if you agree. Otherwise press whatever... n

Do you want to sort it different way?
Enter 'y' if you agree. Otherwise press whatever... y
```

```
Do you want to sort it different way?
Enter 'y' if you agree. Otherwise press whatever... y
Choose sorting method:
1. by region name
2. by region area
3. by region population

Choose option to select (from 1 to 3): 3

Do you want to sort ascending?
Enter 'y' if you agree. Otherwise press whatever... y

1) Name: Харків
Area: 310
Population: 1402000


2) Name: Київ
Area: 839
Population: 2952000


3) Name: Україна
Area: 603628
Population: 38000000


Do you want to write to the file?
Enter 'y' if you agree. Otherwise press whatever... y
Sorted regions were written successfully!
```

Вставка у впорядкований файл

```
Choose option to select (from 1 to 12): 8
Current sorting method is:
by region population ascending

Region name (without commas): Львівська область
Region area (from 1e-06 to 510000000): 21823.7
Region population (from 1e-06 to 10000000000): 2.515e6
Inserted item successfully

Choose option to select (from 1 to 12): 2

1) Name: Харків
Area: 310
Population: 1402000


2) Name: Львівська область
Area: 21823.7
Population: 2515000


3) Name: Київ
Area: 839
Population: 2952000


4) Name: Україна
Area: 603628
Population: 38000000

File was read successfully.
```

Видалення файлу

```
Choose option to select (from 1 to 12): 3

Do you REALLY want to DELETE this file?
Enter 'y' if you agree. Otherwise press whatever... y
File was deleted successfully.
```

Файл без сигнатури

```
Choose option to select (from 1 to 12): 1
Name of file with records: errors
File was selected successfully.

ERROR! File errors doesn't contain the signature RegionSimulator3000
!
Press Enter to continue:
```

Неправильне ім'я файла

```
Choose option to select (from 1 to 12): 1
Name of file with records: .

ERROR! Name . is not allowed in this OS.
Press Enter to continue:

Choose option to select (from 1 to 12): 1
Name of file with records: ..

ERROR! Name .. is not allowed in this OS.
Press Enter to continue:

Choose option to select (from 1 to 12): 1
Name of file with records: main.c

ERROR! It's forbidden to create or edit files with extensions of C Programming Language.
Press Enter to continue:

Choose option to select (from 1 to 12): 1.

ERROR! Value should be a number and not contain any additional characters!
Press Enter to continue:
Choose option to select (from 1 to 12): 1
Name of file with records: ....huhahahah.h

ERROR! It's forbidden to create or edit files with extensions of C Programming Language.
Press Enter to continue:

Choose option to select (from 1 to 12): 1
Name of file with records: ../new

ERROR! String contains one of forbidden symbols: "<>:"/\|?*"
Press Enter to continue:

Choose option to select (from 1 to 12): 1
Name of file with records: jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj
jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj
jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj

ERROR! The string length should be > 0 and < 255
Press Enter to continue:
```

Неправильно відформатований вміст файла

```
RegionSimulator3000
skdljff,1,22,2333
           ,233,5.5
sdfl,23sdf,342




sdf,342,3.5kjj
sdf,,2
sdf,2,
new,1.2,2.3
```

```
Choose option to select (from 1 to 12): 1
Name of file with records: errors
File was selected successfully.

Choose option to select (from 1 to 12): 2

ERROR! Line contains too many commas.
Press Enter to continue:

ATTENTION! Improperly formatted region number 1 is skipped.

ERROR! Region name became empty string after whitespace was trim
Press Enter to continue:

ATTENTION! Improperly formatted region number 2 is skipped.

ERROR! Region area is not a valid number.
Press Enter to continue:

ATTENTION! Improperly formatted region number 3 is skipped.

ERROR! Region population is not a valid number.
Press Enter to continue:

ATTENTION! Improperly formatted region number 4 is skipped.

ERROR! Region area is not a valid number.
Press Enter to continue:

ATTENTION! Improperly formatted region number 5 is skipped.

ERROR! Region population is not a valid number.
Press Enter to continue:

ATTENTION! Improperly formatted region number 6 is skipped.

7) Name: new
Area: 1.2
Population: 2.3

File was read successfully.
```

Використання невідсортованого файлу для вставки

```
Choose option to select (from 1 to 12): 8

ERROR! No file with records was chosen. Please, create file with records or select it.
Press Enter to continue:

Do you want to create/select file?
Enter 'y' if you agree. Otherwise press whatever... y
Name of file with records: EC
File was selected successfully.

ERROR! The file you provided is not sorted in any regular way.
Press Enter to continue:

Do you want to sort it?
Enter 'y' if you agree. Otherwise press whatever... y
Choose sorting method:
1. by region name
2. by region area
3. by region population

Choose option to select (from 1 to 3): 3

Do you want to sort ascending?
Enter 'y' if you agree. Otherwise press whatever... y

1) Name: Repubblika ta' Malta
Area: 316
Population: 563443


2) Name: État du Luxembourg
Area: 2586
Population: 672050


3) Name: Κυπρια
Area: 9251
Population: 933505


4) Name: Eesti Vabariik
Area: 45227
Population: 1374687
```

```
24) Name: Україна
Area: 603628
Population: 43000000


25) Name: Reino de España
Area: 504030
Population: 48610458


26) Name: Repubblica Italiana
Area: 301338
Population: 58989749


27) Name: République française
Area: 640679
Population: 68401997


28) Name: Bundesrepublik Deutschland
Area: 357021
Population: 83445000


29) Name: European Union
Area: 4233262
Population: 449206579


Do you want to write to the file?
Enter 'y' if you agree. Otherwise press whatever... y
Sorted regions were written successfully!
Current sorting method is:
by region population ascending

Region name (without commas): Status Civitatis Vaticanae
Region area (from 1e-06 to 510000000): 0.49
Region population (from 1e-06 to 10000000000): 764
```

Читання, редагування, видалення записів, що не існують

```
Choose option to select (from 1 to 12): 2

ERROR! No file with records was chosen. Please, create file with records or select it.
Press Enter to continue: n

Do you want to create/select file?
Enter 'y' if you agree. Otherwise press whatever... n

Choose option to select (from 1 to 12): 3

ERROR! No file with records was chosen. Please, create file with records or select it.
Press Enter to continue:

Choose option to select (from 1 to 12): 4

ERROR! No file with records was chosen. Please, create file with records or select it.
Press Enter to continue:

Do you want to create/select file?
Enter 'y' if you agree. Otherwise press whatever... n

Choose option to select (from 1 to 12): 5

ERROR! No file with records was chosen. Please, create file with records or select it.
Press Enter to continue:

Do you want to create/select file?
Enter 'y' if you agree. Otherwise press whatever... n

Choose option to select (from 1 to 12): 7

ERROR! No file with records was chosen. Please, create file with records or select it.
Press Enter to continue:

Do you want to create/select file?
Enter 'y' if you agree. Otherwise press whatever... n

Choose option to select (from 1 to 12): 9

ERROR! No file with records was chosen. Please, create file with records or select it.
Press Enter to continue:

Do you want to create/select file?
Enter 'y' if you agree. Otherwise press whatever... n
```

Висновки: Програма працює коректно. Програма вирішує поставлене завдання.