

**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 ІІІ  
Ахаладзе А. Е.  
«2» січня 2025 р.

Виконав  
студент групи ІІІ-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*

---

```
1 cmake_minimum_required(VERSION 3.10)
2 project(RegionSimulator3000 C)
3
4 include_directories(${CMAKE_SOURCE_DIR})
5 enable_testing()
6
7 set(CMAKE_BUILD_TYPE Debug)
8 set(CMAKE_C_FLAGS_DEBUG "-g_-gdwarf-4")
9 set(CMAKE_CXX_FLAGS_DEBUG "-g_-gdwarf-4")
10
11 set(SOURCES
12     src/main.c
13     src/actions/files.c
14     src/actions/misc.c
15     src/actions/records.c
16     src/actions/regions.c
17     src/actions/utils.c
18     src/common/fileContext.c
19     src/common/sort.c
20     src/common/stack.c
21     src/io/choices.c
22     src/io/number.c
23     src/io/string.c
24     src/io/utils.c
25     src/io/validators.c
26     src/menu/default.c
27     src/menu/menu.c
28 )
29
```

```
30 add_executable(RegionSimulator30000 ${SOURCES})
31
32 target_link_libraries(RegionSimulator30000 m)
```

---

*../src/main.c*

---

```
1 #include "common/common.h"
2 #include "io/io.h"
3 #include "menu/menu.h"
4 #include <stdlib.h>
5
6 #define APP_FAILURE SUCCESS
7
8 int main() {
9     MenuNode *menu = initializeDefaultMenu();
10    if (!menu)
11        return APP_FAILURE;
12
13    FileContext *context = initFileContext();
14    if (!context)
15        return APP_FAILURE;
16
17    const size_t menuChoiceCount = assignMenuOptions(menu, MENU_START_INDEX);
18
19    NumberRange *choiceRange = initChoiceRange(MENU_START_INDEX, menuChoiceCount)
20    ;
21    if (!choiceRange)
22        return APP_FAILURE;
23
24    printMenu(menu);
25
26    size_t option = 0;
27    int actionResult = SUCCESS;
28
29    do {
30        printf("\n");
31        option = getUserChoice(choiceRange, NULL, NULL);
32        if (option == 0) // EOF
33            break;
34
35        actionResult = executeMenuActionWithOption(menu, option, context);
36        if (actionResult == DISPLAY_MENU)
37            printMenu(menu);
38    } while (actionResult != EXIT);
39
40    closeFileContext(context);
41    free(context);
42    context = NULL;
43
44    freeNumberRange(choiceRange);
45    choiceRange = NULL;
46
47    freeMenu(menu);
48    menu = NULL;
49
50    return SUCCESS;
51 }
```

---

../src/common/common.h

---

```
1 #ifndef COMMON_H
2 #define COMMON_H
3
4 #include <stdarg.h>
5 #include <stdbool.h>
6 #include <stdio.h>
7
8 #define SUCCESS 0
9 #define FAILURE 1
10 #define NOOP 2
11 #define DISPLAY_MENU 3
12 #define EXIT -2 // INFO: to not be equal EOF = -1
13 #define PIPE 1
14
15 typedef struct {
16     char *name;
17     double area;
18     double population;
19 } Region;
20
21 // INPUT
22 #define INFINITE_LENGTH 0
23 #define DEFAULT_STRING_LENGTH 16
24 typedef enum { TYPE_DOUBLE, TYPE_SIZE_T, TYPE_COUNT } NumberType;
25
26 typedef struct {
27     void *min;
28     void *max;
29     bool isMinIncluded;
30     bool isMaxIncluded;
31     char *valueName;
32     int (*compare)(const void *, const void *);
33     NumberType type;
34 } NumberRange;
35
36 typedef enum {
37     LESS,
38     LESS_EQUAL,
39     GREATER,
40     GREATER_EQUAL,
41     WITHIN_RANGE
42 } RangeCheckResult;
43
44 typedef bool(ValidationFunc)(void *, va_list);
45 // FILE
46 #define SIGNATURE "RegionSimulator3000\n"
47 #define TEMP "temp.region"
48 #define MAX_FILENAME_LENGTH 255
49
50 typedef struct {
51     FILE *file;
52     char *filename;
53     char *signature;
54     size_t signatureSize;
55 } FileContext;
56
57 typedef int (*Action)(FileContext *context);
58
```

```

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*

---

```

1 #include "../actions/utils.h"
2 #include "../io/utils.h"
3 #include "common.h"
4 #include <stdlib.h>
5 #include <string.h>
6
7 FileContext *initFileContext() {
8     FileContext *context = malloc(sizeof(FileContext));
9     if (!context) {
10         handleErrorMemoryAllocation("file_context");
11         return NULL;
12     }
13
14     context->file = NULL;
15     context->filename = NULL;
16     context->signature = NULL;
17     context->signatureSize = 0;
18
19     return context;
20 }
21
22 int closeFileContext(FileContext *context) {
23     if (context == NULL) {
24         handleError("Context_can't_be_NULL_before_closing:_nothing_to_close");
25         return FAILURE;
26     }
27
28     if (context->file) {
29         fclose(context->file);
30         context->file = NULL;
31     }
32
33     if (context->filename) {
34         free(context->filename);
35         context->filename = NULL;
36     }
37
38     if (context->signature) {
39         free(context->signature);
40         context->signature = NULL;
41     }
42     return SUCCESS;
43 }
44
45 int fillFileContext(FileContext *context, const char *filename,
46                    const char *signature) {
47     context->filename = strdup(filename);
48     if (context->filename == NULL) {
49         handleErrorMemoryAllocation("filename_string");
50         closeFileContext(context);
51         return FAILURE;
52     }
53
54     context->signature = strdup(signature);
55     if (context->signature == NULL) {
56         handleErrorMemoryAllocation("signature_string");
57         closeFileContext(context);
58         return FAILURE;
59     }
60

```

```

61 context->signatureSize = strlen(signature);
62
63 bool doesExist = fileExists(filename);
64 const char *mode = (doesExist) ? "r+" : "w+";
65
66 context->file = fopen(filename, mode);
67 if (!context->file) {
68     handleError("Failed_to_create_new_file.");
69     closeFileContext(context);
70     return FAILURE;
71 }
72
73 if (!doesExist) { // Writing signature to new file
74     if (writeSignature(context) == FAILURE) {
75         closeFileContext(context);
76         return FAILURE;
77     }
78 }
79
80 return SUCCESS;
81 }

```

---

*../src/common/sort.c*

---

```

1 #include "../common/common.h"
2 #include "../io/io.h"
3 #include "../io/utills.h"
4 #include <string.h>
5
6 const char *SORT_BY_CHOICES[COMPARATOR_COUNT] = {
7     "by_region_name",
8     "by_region_area",
9     "by_region_population",
10 };
11
12 const char *SORT_HOW_CHOICES[ORDER_COUNT] = {"ascending", "descending"};
13
14 const SortComparator SORT_COMPARATORS[COMPARATOR_COUNT] = {
15     compareRegionNames, compareRegionAreas, compareRegionPopulations};
16
17 const SortOrder SORT_ORDERS[ORDER_COUNT] = {ASCENDING, DESCENDING};
18
19 static SortComparator comparator = NULL;
20 static SortOrder sortOrder = 0;
21 int compareRegionNames(const Region *region1, const Region *region2) {
22     return compareUtf8Strings(region1->name, region2->name);
23 }
24
25 int compareRegionAreas(const Region *region1, const Region *region2) {
26     if (region1->area > region2->area)
27         return BIGGER_THAN;
28     if (region1->area < region2->area)
29         return LESS_THAN;
30     return EQUAL;
31 }
32
33 int compareRegionPopulations(const Region *region1, const Region *region2) {
34     if (region1->population > region2->population)
35         return BIGGER_THAN;

```

```

36  if (region1->population < region2->population)
37      return LESS_THAN;
38  return EQUAL;
39 }
40
41 void swap(Region *a, Region *b) {
42     if (a == NULL || b == NULL) {
43         return;
44     }
45     Region temp = *a;
46     *a = *b;
47     *b = temp;
48 }
49
50 size_t partition(Region **arr, size_t low, size_t high) {
51     Region *pivot = arr[high];
52     size_t i = low - 1;
53
54     for (size_t j = low; j < high; j++) {
55         int comparison = comparator(arr[j], pivot);
56         if ((sortOrder == ASCENDING && comparison < 0) ||
57             (sortOrder == DESCENDING && comparison > 0)) {
58             i++;
59             swap(arr[i], arr[j]);
60         }
61     }
62
63     swap(arr[i + 1], arr[high]);
64     return i + 1;
65 }
66
67 void quickSort(Region **arr, size_t low, size_t high) {
68     if (low ≥ high) {
69         return;
70     }
71     size_t pivotIndex = partition(arr, low, high);
72     if (pivotIndex > 0) {
73         quickSort(arr, low, pivotIndex - 1);
74     }
75     quickSort(arr, pivotIndex + 1, high);
76 }
77
78 void setQuickSortComparator(SortComparator newComparator) {
79     static SortComparator *comparatorPtr = NULL;
80     if (!comparatorPtr) {
81         comparatorPtr = &comparator;
82     }
83     *comparatorPtr = newComparator;
84 }
85
86 void setQuickSortOrder(SortOrder newSortOrder) {
87     static SortOrder *sortOrderPtr = NULL;
88     if (!sortOrderPtr) {
89         sortOrderPtr = &sortOrder;
90     }
91     *sortOrderPtr = newSortOrder;
92 }
93
94 SortComparator getQuickSortComparator() { return comparator; }
95 SortOrder getQuickSortOrder() { return sortOrder; }

```



```

96
97 bool detectSort(Region **regions, size_t regionCount) {
98     if (getQuickSortComparator()) {
99         return true;
100    }
101    bool isSorted = true;
102    for (size_t comp = 0; comp < COMPARATOR_COUNT; comp++) {
103        for (size_t order = 0; order < ORDER_COUNT; order++) {
104            for (size_t ri = 1; ri < regionCount; ri++) {
105                if ((SORT_COMPARATORS[comp])(regions[ri - 1], regions[ri]) *
106                    SORT_ORDERS[order] >
107                    0) { // If order is broken
108                    isSorted = false;
109                    break;
110                }
111            }
112
113            if (isSorted) {
114                setQuickSortComparator(SORT_COMPARATORS[comp]);
115                setQuickSortOrder(ASCENDING);
116                return true;
117            }
118            isSorted = true;
119        }
120    }
121
122    return false;
123 }
124
125 void showCurrentSortMethod() {
126     SortComparator comp = getQuickSortComparator();
127     SortOrder sortOrder = getQuickSortOrder();
128     size_t compI = 0;
129     size_t orderI = 0;
130
131     for (; compI < COMPARATOR_COUNT; compI++) {
132         if (SORT_COMPARATORS[compI] == comp) {
133             break;
134         }
135     }
136     for (; orderI < ORDER_COUNT; orderI++) {
137         if (SORT_ORDERS[orderI] == sortOrder) {
138             break;
139         }
140     }
141
142     printf("Current_sorting_method_is:\n");
143     showSuccess("%s_%s", SORT_BY_CHOICES[compI], SORT_HOW_CHOICES[orderI]);
144     printf("\n");
145 }

```

---

*../src/common/stack.c*

---

```

1 #include "../common/common.h"
2 #include "../io/utils.h"
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 Stack *createStack() {

```

```

7  Stack *stack = (Stack *)malloc(sizeof(Stack));
8  if (!stack) {
9      handleErrorMemoryAllocation("Stack");
10     return NULL;
11 }
12 stack->size = 0;
13 stack->top = NULL;
14 return stack;
15 }
16
17 void freeStack(Stack *stack) {
18     StackNode *item = stack->top;
19     while (item) {
20         StackNode *temp = item;
21         item = item->next;
22         free(temp);
23     }
24     free(stack);
25 }
26
27 int push(Stack *stack, void *data) {
28     StackNode *item = (StackNode *)malloc(sizeof(StackNode));
29     if (!item) {
30         handleErrorMemoryAllocation("StackNode");
31         return FAILURE;
32     }
33     item->data = data;
34     item->next = stack->top;
35     stack->top = item;
36     stack->size++;
37     return SUCCESS;
38 }
39
40 void *pop(Stack *stack) {
41     if (stack->size == 0) {
42         handleError("Error_popping_value_from_the_stack");
43         return NULL;
44     }
45     StackNode *temp = stack->top;
46     void *data = temp->data;
47     stack->top = temp->next;
48     free(temp);
49     stack->size--;
50     return data;
51 }
52
53 void *peek(Stack *stack) {
54     if (stack->size == 0) {
55         return NULL;
56     }
57     return stack->top->data;
58 }

```

---

*../src/menu/menu.h*

---

```

1  #ifndef MENU_H
2
3  #include "../common/common.h"
4  #include <stdbool.h>

```

```

5
6 #define MENU_START_INDEX 1
7 MenuNode *createMenuNode(const char *name, Action action);
8 void addMenuChild(MenuNode *parent, MenuNode *child);
9 int printMenu(MenuNode *root);
10 void freeMenu(MenuNode *node);
11 int executeMenuActionWithOption(MenuNode *menu, int option,
12                               FileContext *context);
13 int assignMenuOptions(MenuNode *node, int start);
14
15 MenuNode *initializeDefaultMenu();
16 #endif // !MENU_H

```

---

*../src/menu/default.c*

---

```

1 #include "../actions/actions.h"
2 #include "menu.h"
3
4 MenuNode *initializeDefaultMenu() {
5     MenuNode *menu = createMenuNode("Menu", NULL);
6     if (!menu)
7         return NULL;
8
9     MenuNode *file = NULL, *records = NULL, *createFile = NULL, *readFile = NULL
10
11         ,
12         *deleteFile = NULL, *createRecords = NULL, *readRecords = NULL,
13         *editRecords = NULL, *sortRecords = NULL, *insertRecords = NULL,
14         *deleteRecords = NULL, *exitProgram = NULL, *verifySignature = NULL,
15         *displayMenu = NULL;
16
17     // File part
18     if (!(file = createMenuNode("File", NULL)) ||
19         !(createFile = createMenuNode("Create/Select_File", actionCreateFile)) ||
20         !(readFile = createMenuNode("Read_all_Records", actionReadFile)) ||
21         !(deleteFile =
22             createMenuNode("Delete/Deselect_File", actionDeleteFile)) ||
23
24         // Record part
25         !(records = createMenuNode("Records", NULL)) ||
26         !(createRecords = createMenuNode("Create_Record", actionCreateRecord)) ||
27         !(readRecords = createMenuNode("Read_Record", actionReadRecord)) ||
28         !(editRecords = createMenuNode("Edit_Record", actionEditRecord)) ||
29         !(sortRecords = createMenuNode("Sort_Records", actionSortRecords)) ||
30         !(insertRecords =
31             createMenuNode("Insert_Record", actionInsertRecordIfSorted)) ||
32         !(deleteRecords = createMenuNode("Delete_Record", actionDeleteRecord)) ||
33
34         // Misc part
35         !(exitProgram = createMenuNode("Exit_Program", actionExitProgram)) ||
36         !(verifySignature =
37             createMenuNode("Verify_File_Header", actionVerifyFileHeader)) ||
38         !(displayMenu =
39             createMenuNode("Display_Menu_Again", actionDisplayMenu))) {
40         freeMenu(menu);
41         return NULL;
42     }
43
44     addMenuChild(menu, file);
45     addMenuChild(menu, records);

```

```

44 addMenuChild(menu, exitProgram);
45 addMenuChild(menu, verifySignature);
46 addMenuChild(menu, displayMenu);
47
48 addMenuChild(file, createFile);
49 addMenuChild(file, readFile);
50 addMenuChild(file, deleteFile);
51
52 addMenuChild(records, createRecords);
53 addMenuChild(records, readRecords);
54 addMenuChild(records, editRecords);
55 addMenuChild(records, sortRecords);
56 addMenuChild(records, insertRecords);
57 addMenuChild(records, deleteRecords);
58
59 assignMenuOptions(menu, MENU_START_INDEX);
60
61 return menu;
62 }

```

---

*../src/menu/menu.c*

---

```

1 #include "menu.h"
2 #include "../common/common.h"
3 #include "../io/utils.h"
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7
8 #define TREE_SPACING 2
9
10 MenuNode *createMenuNode(const char *name, Action action) {
11     MenuNode *node = (MenuNode *)malloc(sizeof(MenuNode));
12
13     if (!node) {
14         handleErrorMemoryAllocation("menu_node");
15         return NULL;
16     }
17
18     node->name = strdup(name);
19     if (!node->name) {
20         free(node);
21         handleErrorMemoryAllocation("menu_node_name");
22         return NULL;
23     }
24
25     node->action = action;
26     node->child = NULL;
27     node->sibling = NULL;
28
29     return node;
30 }
31
32 void addMenuChild(MenuNode *parent, MenuNode *child) {
33     if (!parent->child) {
34         parent->child = child;
35         return;
36     }
37

```

```

38  MenuNode *temp = parent->child;
39  while (temp->sibling) {
40      temp = temp->sibling;
41  }
42  temp->sibling = child;
43  child->sibling = NULL;
44 }
45
46 int executeMenuActionWithOptions(MenuNode *menu, int option,
47                                 FileContext *context) {
48     Stack *stack = createStack();
49     if (!stack) {
50         handleErrorMemoryAllocation("stack_for_menu");
51         return FAILURE;
52     }
53
54     push(stack, menu);
55     while (stack->size != 0) {
56         MenuNode *cur = pop(stack);
57         if (cur->action && cur->option == option) {
58             freeStack(stack);
59             return cur->action(context);
60         }
61
62         if (cur->child) {
63             MenuNode *child = cur->child;
64             while (child) {
65                 push(stack, child);
66                 child = child->sibling;
67             }
68         }
69     }
70     freeStack(stack);
71     return FAILURE; // specified option not found.
72 }
73
74 int assignMenuOptions(MenuNode *node, int start) {
75     if (!node)
76         return start - 1;
77
78     int lastItem;
79     if (node->action) {
80         node->option = start;
81         lastItem = assignMenuOptions(node->child, start + 1);
82     } else {
83         lastItem = assignMenuOptions(node->child, start);
84     }
85
86     return assignMenuOptions(node->sibling, lastItem + 1);
87 }
88
89 void freeMenu(MenuNode *node) {
90     if (!node)
91         return;
92
93     freeMenu(node->child);
94     freeMenu(node->sibling);
95
96     free(node->name);
97     free(node);

```

```

98 }
99
100 void printMenuNode(MenuNode *node) {
101     if (!node)
102         return;
103
104     if (node->action) {
105         printf("%d:_%s", node->option, node->name);
106     } else {
107         printf("%s", node->name);
108     }
109 }
110
111 int printMenu(MenuNode *root) {
112     if (!root) {
113         handleError("Root_menu_node_is_NULL.");
114         return FAILURE;
115     }
116
117     Stack *stack = createStack();
118     if (!stack) {
119         return FAILURE;
120     }
121
122     size_t maxLevels = 10;
123     bool *hasSiblings = (bool *)malloc(maxLevels * sizeof(bool));
124     if (hasSiblings == NULL) {
125         handleErrorMemoryAllocation("array_of_bool_for_finding_menu_node_siblings");
126         freeStack(stack);
127         return FAILURE;
128     }
129
130     size_t level = 0;
131     push(stack, root);
132
133     while (stack->size != 0) {
134         MenuNode *cur = pop(stack);
135
136         // Ensure `hasSiblings` array has enough space
137         if (level ≥ maxLevels) {
138             maxLevels *= 2;
139             bool *temp = (bool *)realloc(hasSiblings, maxLevels * sizeof(bool));
140             if (temp == NULL) {
141                 handleErrorMemoryAllocation("reallocation_of_menu_node_siblings_array");
142                 free(hasSiblings);
143                 freeStack(stack);
144                 return FAILURE;
145             }
146             hasSiblings = temp;
147         }
148
149         hasSiblings[level] = (cur->sibling != NULL);
150
151         for (size_t i = 0; i < level; i++) {
152             if (hasSiblings[i]) {
153                 printf("|%-*s_", TREE_SPACING, "_");
154             } else {
155                 printf(" _%-*s_", TREE_SPACING, "_");
156             }
157         }

```

```

158     if (hasSiblings[level]) {
159         printf("|");
160     } else {
161         printf("L");
162     }
163     printf("_");
164     printMenuNode(cur);
165
166     if (cur->child) {
167         MenuNode *child = cur->child;
168
169         // Push siblings in reverse order to preserve the order
170         Stack *tempStack = createStack();
171         if (!tempStack)
172             return FAILURE;
173
174         while (child) {
175             push(tempStack, child);
176             child = child->sibling;
177         }
178
179         while (tempStack->size != 0) {
180             push(stack, pop(tempStack));
181         }
182
183         freeStack(tempStack);
184         level++;
185     } else if (!cur->sibling) {
186         level--; // Go back up if no siblings remain
187     }
188     printf("\n");
189 }
190
191 free(hasSiblings);
192 freeStack(stack);
193 return SUCCESS;
194 }

```

---

## ../src/actions/actions.h

---

```

1 #ifndef ACTIONS_H
2 #define ACTIONS_H
3 #include "../common/common.h"
4
5 // FILE ACTIONS
6 int actionCreateFile(FileContext *);
7 int actionReadFile(FileContext *);
8 int actionDeleteFile(FileContext *);
9
10 // RECORD ACTIONS
11 int actionCreateRecord(FileContext *);
12 int actionReadRecord(FileContext *);
13 int actionEditRecord(FileContext *);
14 int actionSortRecords(FileContext *);
15 int actionInsertRecordIfSorted(FileContext *);
16 int actionDeleteRecord(FileContext *);
17
18 // MISC ACTIONS
19 int actionDisplayMenu();

```

```

20 int actionExitProgram();
21 int actionNotImplemented(FileContext *context);
22 int actionVerifyFileHeader();
23
24 #endif

```

---

*../src/actions/files.c*

---

```

1 #include "../common/common.h"
2 #include "../io/utls.h"
3 #include "actions.h"
4 #include "assert.h"
5 #include "utls.h"
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <string.h>
9
10 int actionCreateFile(FileContext *context) {
11     if (!context) {
12         handleError("Context_is_NULL");
13         return FAILURE;
14     }
15
16     if (context->file) {
17         handleError("Can't_create_new_file_while_other_is_already_available.");
18         if (actionDeleteFile(context) == FAILURE)
19             return FAILURE;
20     }
21
22     char *filename = getValidFilename();
23     if (!filename)
24         return FAILURE;
25
26     bool doesExist = fileExists(filename);
27     const char *mode = doesExist ? "r+" : "w+";
28
29     if (fillFileContext(context, filename, SIGNATURE) == FAILURE)
30         goto cleanup;
31
32     showSuccess("File_was_%s_successfully.", doesExist ? "selected" : "created");
33
34     if (handleBadFileHeader(context) == FAILURE)
35         goto cleanup;
36
37     free(filename);
38     return SUCCESS;
39
40 cleanup:
41     if (filename)
42         free(filename);
43     return FAILURE;
44 }
45
46 int actionReadFile(FileContext *context) {
47     if (handleFileIfNotExist(context) == FAILURE)
48         return FAILURE;
49
50     if (handleFileIfEmpty(context) == FAILURE)
51         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",

```

```

49         isNotZero) = EOF)
50     return EXIT;
51
52     Region *region = readRegionAt(recordNumber, context);
53
54     if (!region)
55         return FAILURE;
56
57     printRegion(region, recordNumber);
58     freeRegion(region);
59
60     showSuccess("Record_read_successfully");
61     return SUCCESS;
62 }
63
64 int actionEditRecord(FileContext *context) {
65     if (handleFileIfNotExist(context) = FAILURE)
66         return FAILURE;
67
68     if (handleFileIfEmpty(context) = FAILURE)
69         return FAILURE;
70
71     size_t recordNumber;
72     readNumberWithValidation(&recordNumber, TYPE_SIZE_T, "Record_number",
73                             isNotZero);
74
75     Region *regionBeforeEdited = readRegionAt(recordNumber - 1, context);
76     if (!regionBeforeEdited)
77         return FAILURE;
78     freeRegion(regionBeforeEdited);
79
80     long positionBeforeEdited = ftell(context->file);
81     if (positionBeforeEdited < 0) {
82         handleError("Failed_to_move_file_pointer_to_position_before_edited.");
83         return FAILURE;
84     }
85
86     Region *region = initRegion();
87     if (!region)
88         return FAILURE;
89
90     int readResult = readRegion(region, context);
91     if (readResult != SUCCESS) {
92         if (readResult == EOF)
93             handleError("File_doesn't_contain_region_%zu", recordNumber);
94         freeRegion(region);
95         return FAILURE;
96     }
97
98     int editResult = editRegion(region, recordNumber);
99     if (editResult == FAILURE) {
100         freeRegion(region);
101         return FAILURE;
102     }
103     if (editResult == NOOP) { // No changes were made
104         showSuccess("No_changes_were_made_to_the_region_so_far.");
105         freeRegion(region);
106         return SUCCESS;
107     }
108 }

```

```

109 FileContext *temp = createTempFile();
110 if (!temp)
111     goto cleanup;
112
113 if (copyRegionsBetweenFiles(context, temp) == FAILURE)
114     goto cleanup;
115
116 if (fseek(context->file, positionBeforeEdited, SEEK_SET) != SUCCESS) {
117     handleError("Couldn't move back to the position before edited record");
118     goto cleanup;
119 }
120
121 if (writeRegion(region, context) == FAILURE)
122     goto cleanup;
123
124 if (skipFileHeader(temp) == FAILURE)
125     goto cleanup;
126 if (copyRegionsBetweenFiles(temp, context) == FAILURE)
127     goto cleanup;
128
129 if (truncateFileFromCurrentPosition(context) == FAILURE)
130     goto cleanup;
131
132 if (handleFlushing(context) == FAILURE)
133     goto cleanup;
134
135 setQuickSortComparator(NULL); // The sorting might've been removed
136 freeRegion(region);
137 closeFileContext(temp);
138 free(temp);
139 temp = NULL;
140
141 if (remove(TEMP) != SUCCESS) {
142     handleError("Failed to delete temporary file.");
143     return FAILURE;
144 }
145
146 showSuccess("Record was edited successfully.");
147 return SUCCESS;
148
149 cleanup:
150 if (temp) {
151     closeFileContext(temp);
152     free(temp);
153     if (remove(TEMP) != SUCCESS) {
154         handleError("Failed to delete temporary file.");
155     }
156 }
157 freeRegion(region);
158 return FAILURE;
159 }
160
161 int actionSortRecords(FileContext *context) {
162     if (handleFileIfNotExist(context) == FAILURE)
163         return FAILURE;
164
165     if (handleFileIfEmpty(context) == FAILURE)
166         return FAILURE;
167
168     if (skipFileHeader(context) == FAILURE)

```

```

169     return FAILURE;
170
171     size_t regionCount = 0;
172     Region **regions = readAllRegions(&regionCount, context);
173
174     if (!regions || regionCount == 0) {
175         handleError(
176             "Record_file_doesn't_contain_any_regions,_or_they_are_malformed!");
177         return FAILURE;
178     }
179
180     if (regionCount == 1) {
181         showSuccess("The_file_contains_1_entry._It's_already_sorted.");
182         return SUCCESS;
183     }
184
185     bool isSorting = true;
186
187     NumberRange *sortOptionRange = initChoiceRange(1, COMPARATOR_COUNT);
188     if (!sortOptionRange)
189         goto cleanup;
190
191     do {
192         size_t sortOption = getUserChoice(sortOptionRange, "Choose_sorting_method",
193                                           SORT_BY_CHOICES) -
194                               1;
195         setQuickSortComparator(SORT_COMPARATORS[sortOption]);
196
197         setQuickSortOrder(
198             askQuestion("Do_you_want_to_sort_ascending?") ? ASCENDING : DESCENDING);
199
200         quickSort(regions, 0, regionCount - 1);
201
202         for (size_t i = 0; i < regionCount; i++) {
203             printRegion(regions[i], i + 1);
204         }
205
206         if (askQuestion("Do_you_want_to_write_to_the_file?"))
207             break;
208
209         if (askQuestion("Do_you_want_to_sort_it_different_way?"))
210             continue;
211
212         if (!askQuestion("Do_you_even_want_to_continue?"))
213             goto cleanup;
214     } while (isSorting);
215
216     freeNumberRange(sortOptionRange);
217
218     if (skipFileHeader(context) == FAILURE)
219         goto cleanup;
220
221     for (size_t i = 0; i < regionCount; i++) {
222         int result;
223         if ((result = writeRegion(regions[i], context)) == FAILURE) {
224             if (result == FAILURE)
225                 goto cleanup;
226         }
227     }
228

```

```

229 if (handleFlushing(context) == FAILURE)
230     goto cleanup;
231
232 showSuccess("Sorted_regions_were_written_successfully!");
233
234 for (size_t i = 0; i < regionCount; i++)
235     freeRegion(regions[i]);
236
237 free(regions);
238 return SUCCESS;
239
240 cleanup:
241 if (regions)
242     for (size_t i = 0; i < regionCount; i++) {
243         freeRegion(regions[i]);
244     }
245 free(regions);
246 return FAILURE;
247 }
248
249 int actionInsertRecordIfSorted(FileContext *context) {
250     if (handleFileIfNotExist(context) == FAILURE)
251         return FAILURE;
252
253     if (handleFileIfEmpty(context) == FAILURE)
254         return FAILURE;
255
256     if (skipFileHeader(context) == FAILURE)
257         return FAILURE;
258
259     size_t regionCount = 0;
260     Region **regions = readAllRegions(&regionCount, context);
261     if (regions == NULL)
262         return FAILURE;
263
264     if (!detectSort(regions, regionCount)) {
265         handleError("The_file_you_provided_is_not_sorted_in_any_regular_way.");
266         if (askQuestion("Do_you_want_to_sort_it?")) {
267             if (actionSortRecords(context) == FAILURE)
268                 goto cleanup;
269         }
270     }
271
272     showCurrentSortMethod();
273
274     Region *newRegion = getRegionFromUser();
275     if (!newRegion)
276         goto cleanup;
277
278     regionCount++;
279     Region **newRegions = realloc(regions, regionCount * sizeof(Region *));
280     if (newRegions == NULL) {
281         handleErrorMemoryAllocation("extending_region_array");
282         goto cleanup;
283     }
284     regions = newRegions;
285
286     regions[regionCount - 1] = newRegion;
287     quickSort(regions, 0, regionCount - 1);
288

```

```

289 if (skipFileHeader(context) == FAILURE)
290     goto cleanup;
291
292 for (size_t i = 0; i < regionCount; i++) {
293     if (writeRegion(regions[i], context) == FAILURE)
294         goto cleanup;
295 }
296
297 for (size_t i = 0; i < regionCount; i++)
298     freeRegion(regions[i]);
299 free(regions);
300
301 showSuccess("Inserted_item_successfully");
302 return SUCCESS;
303
304 cleanup:
305 for (size_t i = 0; i < regionCount; i++)
306     freeRegion(regions[i]);
307 free(regions);
308 return FAILURE;
309 }
310
311 int actionDeleteRecord(FileContext *context) {
312     if (handleFileIfNotExist(context) == FAILURE)
313         return FAILURE;
314
315     if (handleFileIfEmpty(context) == FAILURE)
316         return FAILURE;
317
318     if (skipFileHeader(context) == FAILURE)
319         return FAILURE;
320
321     size_t recordNumber;
322     if (readNumberWithValidation(&recordNumber, TYPE_SIZE_T, "Record_number",
323                                isNotZero) == EOF)
324         return FAILURE;
325
326     Region *regionBeforeDeleted = readRegionAt(recordNumber - 1, context);
327     if (!regionBeforeDeleted)
328         return FAILURE;
329
330     freeRegion(regionBeforeDeleted);
331
332     long positionBeforeDeleted = ftell(context->file);
333
334     Region *deleteRegion = initRegion();
335     if (!deleteRegion)
336         return FAILURE;
337
338     if (readRegion(deleteRegion, context) != SUCCESS) {
339         freeRegion(deleteRegion);
340         return FAILURE;
341     }
342
343     printRegion(deleteRegion, recordNumber);
344     freeRegion(deleteRegion);
345
346     if (!askQuestion("Do_you REALLY want to delete this region?"))
347         return FAILURE;
348

```

```

349 FileContext *temp = createTempFile();
350 if (!temp)
351     goto cleanup;
352
353 if (copyRegionsBetweenFiles(context, temp) == FAILURE)
354     goto cleanup;
355
356 if (fseek(context->file, positionBeforeDeleted, SEEK_SET) != SUCCESS) {
357     handleError("Failed_to_move_to_the_position_before_deleted_record.");
358     goto cleanup;
359 }
360
361 if (skipFileHeader(temp) == FAILURE)
362     goto cleanup;
363
364 if (copyRegionsBetweenFiles(temp, context) == FAILURE)
365     goto cleanup;
366
367 if (truncateFileFromCurrentPosition(context) == FAILURE)
368     goto cleanup;
369
370 closeFileContext(temp);
371 free(temp);
372 if (remove(TEMP) != SUCCESS) {
373     handleError("Failed_to_delete_temporary_file");
374     goto cleanup;
375 }
376
377 if (handleFlushing(context) == FAILURE)
378     goto cleanup;
379
380 showSuccess("Record_was_deleted_successfully.");
381 return SUCCESS;
382 cleanup:
383 if (temp) {
384     closeFileContext(temp);
385     free(temp);
386     if (remove(TEMP) != SUCCESS) {
387         handleError("Failed_to_delete_temporary_file");
388     }
389 }
390 return FAILURE;
391 }

```

---

*../src/actions/misc.c*

---

```

1 #include "../common/common.h"
2 #include "../io/utlis.h"
3 #include "actions.h"
4 #include "utlis.h"
5 #include <stdlib.h>
6
7 int actionNotImplemented(FileContext *context) {
8     handleError("This_action_isn't_implemented_yet!");
9     return SUCCESS;
10 }
11
12 int actionExitProgram() { return EXIT; }
13

```



```

14 int actionVerifyFileHeader() {
15     char *filename = getValidFilename();
16     if (!filename)
17         return FAILURE;
18
19     FileContext *tempContext = initFileContext();
20     if (!tempContext)
21         goto cleanup;
22
23     if (!fileExists(filename)) {
24         handleError("This_file_doesn't_exist,_can't_check_the_file_header.");
25         goto cleanup;
26     }
27
28     if (fillFileContext(tempContext, filename, SIGNATURE) == FAILURE)
29         goto cleanup;
30
31     if (handleBadFileHeader(tempContext) == FAILURE)
32         goto cleanup;
33
34     closeFileContext(tempContext);
35     free(tempContext);
36     free(filename);
37     showSuccess("File_\"%s\"_does_contain_file_header", filename);
38     return SUCCESS;
39
40 cleanup:
41     free(filename);
42     if (tempContext) {
43         closeFileContext(tempContext);
44         free(tempContext);
45     }
46     return FAILURE;
47 }
48
49 int actionDisplayMenu() { return DISPLAY_MENU; }

```

---

*../src/actions/regions.c*

---

```

1 #include "../common/common.h"
2 #include "../io/io.h"
3 #include "../io/utils.h"
4 #include "utils.h"
5 #include <stdlib.h>
6 #include <string.h>
7 #define MAX_SIGNIFICANT_DIGITS 15
8
9 static double minArea = 1e-6;
10 static double maxArea = 510e6;
11 static const NumberRange areaRange = {
12     &minArea, &maxArea, true, true, "Region_area", compareDouble, TYPE_DOUBLE};
13
14 static double minPopulation = 1e-6;
15 static double maxPopulation = 1e10;
16
17 static const NumberRange populationRange = {
18     &minPopulation, &maxPopulation, true, true,
19     "Region_population", compareDouble, TYPE_DOUBLE};
20

```

```

21 Region *initRegion() {
22     Region *region = malloc(sizeof(Region));
23     if (!region) {
24         handleErrorMemoryAllocation("region");
25         return NULL;
26     }
27
28     region->name = NULL;
29     region->area = 0.0;
30     region->population = 0.0;
31     return region;
32 }
33
34 void freeRegion(Region *region) {
35     if (region->name) {
36         free(region->name);
37         region->name = NULL;
38     }
39     if (region)
40         free(region);
41 }
42
43 int writeRegion(Region *region, FileContext *context) {
44     if (handleFileIfNotExist(context) == FAILURE)
45         return FAILURE;
46
47     if (fprintf(context->file, "%s,%.*lg,%.*lg\n", region->name,
48                 MAX_SIGNIFICANT_DIGITS, region->area, MAX_SIGNIFICANT_DIGITS,
49                 region->population) < 0) {
50         handleError("Failed_to_write_to_file_%s.", context->filename);
51         return FAILURE;
52     }
53
54     if (handleFlushing(context) == FAILURE)
55         return FAILURE;
56     return SUCCESS;
57 }
58
59 int readRegion(Region *region, FileContext *context) {
60     if (!region) {
61         handleError("Region_is_NULL_in_readRegion");
62         return FAILURE;
63     }
64     if (handleFileIfNotExist(context) == FAILURE)
65         return FAILURE;
66
67     char *buffer = NULL;
68
69     int result = SUCCESS;
70     do {
71         if (buffer)
72             free(buffer);
73         result = readLine(&buffer, DEFAULT_STRING_LENGTH, INFINITE_LENGTH,
74                         context->file);
75         if (result == EOF) {
76             if (buffer) {
77                 free(buffer);
78                 buffer = NULL;
79             }
80             return EOF;

```

```

81     }
82     if (result == FAILURE) {
83         handleError("Failed_to_read_line_in_file_%.s.", context->filename);
84         goto cleanup;
85     }
86 } while (isLineEmpty(buffer));
87
88 // Locate commas
89 char *comma1 = strchr(buffer, ',');
90 char *comma2 = comma1 ? strchr(comma1 + 1, ',') : NULL;
91 char *comma3 = comma2 ? strchr(comma2 + 1, ',') : NULL;
92
93 if (!comma1 || !comma2) {
94     handleError("Line_doesn't_contain_enough_commas.");
95     goto cleanup;
96 }
97 if (comma3) {
98     handleError("Line_contains_too_many_commas.");
99     goto cleanup;
100 }
101
102 // Extract and allocate name
103 size_t nameLength = (size_t)(comma1 - buffer);
104 region->name = calloc(1 + nameLength, sizeof(char));
105
106 if (!region->name) {
107     handleErrorMemoryAllocation("region_name_string");
108     goto cleanup;
109 }
110 strncpy(region->name, buffer, nameLength);
111 if (!trimWhitespaceUtf8(region->name))
112     goto cleanup;
113
114 if (region->name[0] == '\0') {
115     handleError(
116         "Region_name_became_empty_string_after_whitespace_was_trimmed.");
117     goto cleanup;
118 }
119
120 region->name[nameLength] = '\0';
121
122 // Parse area
123 char *endPtr = NULL;
124 region->area = strtod(comma1 + 1, &endPtr);
125 if (endPtr == comma1 + 1 || endPtr != comma2) {
126     handleError("Region_area_is_not_a_valid_number.");
127     goto cleanup;
128 }
129
130 if (validateNumberRange(&region->area, &areaRange) != WITHIN_RANGE)
131     goto cleanup;
132
133 // Parse population
134 endPtr = NULL;
135 region->population = strtod(comma2 + 1, &endPtr);
136
137 if (endPtr == comma2 + 1 || *endPtr != '\0') {
138     handleError("Region_population_is_not_a_valid_number.");
139     goto cleanup;
140 }

```

```

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) {

```

```

201     handleError("Region_count_is_NULL.");
202     return NULL;
203 }
204
205 size_t count = 0;
206 size_t capacity = 4;
207 size_t regionSize = capacity * sizeof(Region *);
208
209 Region **regions = malloc(regionSize);
210 if (!regions) {
211     handleErrorMemoryAllocation("regions_array");
212     return NULL;
213 }
214
215 Region *newRegion = initRegion();
216 if (!newRegion) {
217     handleErrorMemoryAllocation("new_region");
218     goto cleanup;
219 }
220
221 int result = 0;
222
223 while ((result = readRegion(newRegion, context)) != EOF) {
224     if (result == FAILURE)
225         goto cleanup;
226
227     if (count == capacity) {
228         capacity *= 2;
229         Region **newBuffer = realloc(regions, capacity * sizeof(Region *));
230         if (!newBuffer) {
231             handleErrorMemoryAllocation("regions_array_resize");
232             goto cleanup;
233         }
234         regions = newBuffer;
235     }
236
237     regions[count] = newRegion;
238     count++;
239
240     newRegion = initRegion();
241     if (!newRegion) {
242         handleErrorMemoryAllocation("new_region");
243         goto cleanup;
244     }
245 }
246
247 free(newRegion);
248 *regionCount = count;
249 return regions;
250
251 cleanup:
252 for (size_t i = 0; i < count; i++)
253     freeRegion(regions[i]);
254 free(regions);
255 if (newRegion)
256     freeRegion(newRegion);
257 return NULL;
258 }
259
260 int editRegion(Region *region, size_t regionNumber) {

```

```

261 char *nameBuffer = NULL;
262 double newArea = region->area;
263 double newPopulation = region->population;
264
265 printRegion(region, regionNumber);
266
267 if (askQuestion("Do_you_want_to_rename_this_region?")) {
268     nameBuffer =
269         readStringWithFilterUntilValid("Region_name", INFINITE_LENGTH, ",");
270     if (!nameBuffer)
271         return FAILURE;
272 }
273
274 if (askQuestion("Do_you_want_to_change_region_area?"))
275     if (readNumberWithinRangeWithValidation(&newArea, TYPE_DOUBLE, &areaRange,
276                                             areaRange.valueName, NULL) == EOF)
277         return FAILURE;
278
279 if (askQuestion("Do_you_want_to_change_region_population?"))
280     if (readNumberWithinRangeWithValidation(
281         &newPopulation, TYPE_DOUBLE, &populationRange,
282         populationRange.valueName, NULL) == FAILURE)
283         return FAILURE;
284
285 if (newArea == region->area && newPopulation == region->population &&
286     nameBuffer == NULL)
287     return NOOP;
288
289 if (nameBuffer) {
290     if (region->name)
291         free(region->name);
292     region->name = nameBuffer;
293 }
294
295 region->area = newArea;
296 region->population = newPopulation;
297
298 return SUCCESS;
299 }
300
301 int copyRegionsBetweenFiles(FileContext *source, FileContext *destination) {
302     Region *region = initRegion();
303
304     int result;
305     while ((result = readRegion(region, source)) != EOF) {
306         if (result == FAILURE || writeRegion(region, destination) == FAILURE) {
307             freeRegion(region);
308             return FAILURE;
309         }
310         freeRegion(region);
311         region = initRegion();
312     }
313
314     freeRegion(region);
315
316     return SUCCESS;
317 }
318
319 Region *readRegionAt(size_t position, FileContext *context) {
320     if (handleFileIfNotExist(context) == FAILURE)

```

```

321     return NULL;
322
323     if (skipFileHeader(context) == FAILURE)
324         return NULL;
325
326     Region *region = initRegion();
327     if (!region)
328         return NULL;
329
330     if (position == 0) {
331         return region; // Noop (e.g. already passed nothing)
332     }
333
334     size_t i = 0;
335     while (i != position) {
336         int result = readRegion(region, context);
337         if (result == EOF) {
338             handleError("File_doesn't_contain_region_with_such_number");
339             freeRegion(region);
340             return NULL;
341         }
342
343         if (result == FAILURE) {
344             freeRegion(region);
345             region = initRegion();
346             i++;
347             continue;
348         }
349
350         i++;
351         if (i == position) {
352             return region;
353         }
354
355         freeRegion(region);
356         region = initRegion();
357         if (!region)
358             return NULL;
359     }
360
361     if (!region->name) { // If nothing was found
362         freeRegion(region);
363         return NULL;
364     }
365
366     return region;
367
368 cleanup:
369     freeRegion(region);
370     return NULL;
371 }

```

---

*../src/actions/utils.h*

---

```

1 #ifndef ACTION_UTILS_H
2 #define ACTION_UTILS_H
3 #include "../common/common.h"
4 #include <stddef.h>
5

```

```

6 // GENERAL
7 int handleErrorFileNotSpecified();
8 int skipFileHeader(FileContext *context);
9 int writeSignature(FileContext *context);
10
11 void showCurrentSortMethod();
12 int handleBadFileHeader(FileContext *context);
13
14 FileContext *createTempFile();
15
16 int handleFlushing(FileContext *context);
17 int truncateFileFromCurrentPosition(FileContext *context);
18
19 int handleFileIfEmpty(FileContext *context);
20 int handleFileIfNotExist(FileContext *context);
21
22 char *getValidFilename();
23 bool fileExists(const char *filename);
24
25 // REGIONS
26 int readRegion(Region *region, FileContext *context);
27 Region *initRegion();
28 void freeRegion(Region *region);
29 void printRegion(Region *region, size_t regionNumber);
30
31 Region *getRegionFromUser();
32 int writeRegion(Region *region, FileContext *context);
33 int editRegion(Region *region, size_t regionNumber);
34 int copyRegionsBetweenFiles(FileContext *source, FileContext *destination);
35 Region *readRegionAt(size_t recordNumber, FileContext *context);
36
37 Region **readAllRegions(size_t *regionCount, FileContext *context);
38 #endif

```

---

*../src/actions/utils.c*

---

```

1 #include "utils.h"
2 #include "../common/common.h"
3 #include "../io/io.h"
4 #include "../io/utils.h"
5 #include "actions.h"
6 #include <errno.h>
7 #include <linux/limits.h>
8 #include <stdlib.h>
9 #include <string.h>
10 #include <unistd.h>
11
12 static const char *filenameReject = "<>:\\";
13 static const char *disallowedNames[] = {".", ".."};
14
15 bool fileExists(const char *filename) {
16     int result = access(filename, F_OK);
17     if (errno == ENOENT ||
18         errno == 0) { // if there's error related not to file non-existence
19         errno = 0;
20     } else {
21         handleError("");
22     }
23     return result == SUCCESS;

```



```

24 }
25
26 bool isFilenameWithinLength(const char *filename) {
27     return strlen(filename) ≤ NAME_MAX;
28 }
29
30 char *getValidFilename() {
31     char *filename = readStringWithFilterUntilValid(
32         "Name_of_file_with_records", MAX_FILENAME_LENGTH, filenameReject);
33
34     if (!filename)
35         return NULL;
36
37     if (filename[0] == '\0') {
38         handleError("String_should_have_at_least_one_printable_character.");
39         free(filename);
40         return NULL;
41     }
42
43     if (findCharsInUtf8String(filename, filenameReject) ≠
44         getUtf8StringLength(filename)) {
45         handleError("String_contains_one_of_forbidden_symbols:_%s\\",
46             filenameReject);
47         free(filename);
48         return NULL;
49     }
50     size_t disallowedNamesLength =
51         sizeof(disallowedNames) / sizeof(disallowedNames[0]);
52     for (size_t i = 0; i < disallowedNamesLength; i++) {
53
54         if (compareUtf8Strings(filename, disallowedNames[i]) = 0) {
55             handleError("Name_%s_is_not_allowed_in_this_OS.", disallowedNames[i],
56                 filename);
57             free(filename);
58             return NULL;
59         }
60     }
61
62     if (hasSuffix(filename, ".c") || hasSuffix(filename, ".h")) {
63         handleError("It's_forbidden_to_create_or_edit_files_with_extensions_of_C_\"
64             \"Programming_Language.\");
65         free(filename);
66         return NULL;
67     }
68
69     return filename;
70 }
71
72 int handleFileIfNotExist(FileContext *context) {
73     if (!context) {
74         handleError("Context_is_NULL");
75         return FAILURE;
76     }
77     if (context->file = NULL) {
78         handleErrorFileNotSpecified();
79         if (!askQuestion("Do_you_want_to_create/select_file?"))
80             return FAILURE;
81         if (actionCreateFile(context) = FAILURE)
82             return FAILURE;
83     }

```

```

84  return SUCCESS;
85 }
86
87 int handleFileIfEmpty(FileContext *context) {
88     if (handleFileIfNotExist(context) == FAILURE)
89         return FAILURE;
90
91     if (skipFileHeader(context) == FAILURE)
92         return FAILURE;
93
94     char *line = NULL;
95     int result = 0;
96     while ((result = readLine(&line, DEFAULT_STRING_LENGTH, INFINITE_LENGTH,
97                             context->file)) != EOF) {
98         if (result == FAILURE) {
99             if (line)
100                 free(line);
101             return FAILURE;
102         }
103
104         if (!isLineEmpty(line)) {
105             free(line);
106             return SUCCESS;
107         }
108         free(line);
109     }
110
111     handleError("This_file_is_empty,_consider_creating_some_records_first.");
112     if (!askQuestion("Do_you_want_to_create_a_record?"))
113         return FAILURE;
114     return actionCreateRecord(context);
115 }
116
117 int handleFlushing(FileContext *context) {
118     if (!context->file)
119         return FAILURE;
120
121     if (fflush(context->file) == EOF) {
122         handleError("Failed_to_flush_the_data_into_%s", context->filename);
123         return FAILURE;
124     }
125
126     return SUCCESS;
127 }
128
129 // OUTPUT
130
131 int handleErrorFileNotSpecified() {
132     return handleError("No_file_with_records_was_chosen._Please,_create_file_"
133                       "with_records_or_select_it.");
134 }
135
136 int handleBadFileHeader(FileContext *context) {
137     if (handleFileIfNotExist(context) == FAILURE)
138         return FAILURE;
139
140     rewind(context->file);
141     char buffer[context->signatureSize + 1];
142
143     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);

```

```

30 void printNumber(const void *value, NumberType type);
31
32 // STRINGS
33 bool isLineEmpty(const char *line);
34 bool isValidUtf8(const char *str);
35 char *trimWhitespaceUtf8(char *str);
36 size_t getUtf8StringLength(const char *str);
37 size_t findCharsInUtf8String(const char *str, const char *reject);
38 char *readStringWithFilterUntilValid(const char *prompt, size_t maxLength,
39                                     const char *reject);
40 bool hasSuffix(const char *filename, const char *suffix);
41 int compareUtf8Strings(const char *str1, const char *str2);
42
43 // CHOICES
44 size_t getUserChoice(NumberRange *choiceRange, const char *info,
45                  const char *choices[]);
46
47 NumberRange *initChoiceRange(size_t start, size_t end);
48
49 // VALIDATORS
50 bool isNotZero(void *value, va_list args);
51 #endif

```

---

*../src/io/number.c*

---

```

1  #include "io.h"
2  #include "utils.h"
3  #include <errno.h>
4  #include <float.h>
5  #include <limits.h>
6  #include <stdbool.h>
7  #include <stddef.h>
8  #include <stdint.h>
9  #include <stdlib.h>
10 #include <string.h>
11
12 const char *typeDescriptions[TYPE_COUNT] = {"floating-point_decimal",
13                                             "length_of_arrays"};
14
15 size_t getTypeSize(NumberType type) {
16     switch (type) {
17         case TYPE_SIZE_T:
18             return sizeof(size_t);
19         case TYPE_DOUBLE:
20             return sizeof(double);
21         default:
22             handleError("Number_type_not_implemented.");
23             return 0;
24     }
25 }
26
27 int compareDouble(const void *a, const void *b) {
28     if (*(double *)a < *(double *)b)
29         return -1;
30     if (*(double *)a > *(double *)b)
31         return 1;
32     return 0;
33 }
34

```

```

35 int compareSizeT(const void *a, const void *b) {
36     if (*(size_t *)a < *(size_t *)b)
37         return -1;
38     if (*(size_t *)a > *(size_t *)b)
39         return 1;
40     return 0;
41 }
42
43 NumberRange *initializeNumberRange(NumberType type, void *min, void *max,
44                                     bool isMinIncluded, bool isMaxIncluded,
45                                     const char *valueName) {
46     NumberRange *range = malloc(sizeof(NumberRange));
47     if (!range) {
48         handleErrorMemoryAllocation("range");
49         return NULL;
50     }
51
52     range->type = type;
53     size_t typeSize = getTypeSize(type);
54
55     range->min = malloc(typeSize);
56     if (!range->min) {
57         handleErrorMemoryAllocation("number_range_min");
58         freeNumberRange(range);
59         return NULL;
60     }
61
62     range->max = malloc(typeSize);
63     if (!range->max) {
64         handleError("Memory_allocation_failed_for_range_min/max_values.");
65         freeNumberRange(range);
66         return NULL;
67     }
68
69     switch (type) {
70     case TYPE_SIZE_T:
71         *(size_t *)range->min = *(size_t *)min;
72         *(size_t *)range->max = *(size_t *)max;
73         range->compare = compareSizeT;
74         break;
75
76     case TYPE_DOUBLE:
77         *(double *)range->min = *(double *)min;
78         *(double *)range->max = *(double *)max;
79         range->compare = compareDouble;
80         break;
81     default:
82         handleError("Unsupported_type_in_range_initialization.");
83         freeNumberRange(range);
84         return NULL;
85     }
86
87     range->isMinIncluded = isMinIncluded;
88     range->isMaxIncluded = isMaxIncluded;
89     range->valueName = strdup(valueName);
90     if (!range->valueName) {
91         handleErrorMemoryAllocation("number_range_value_name");
92         freeNumberRange(range);
93         return NULL;
94     }

```

```

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_");

```

```

155     printNumber(range->max, range->type);
156 }
157
158 int getMaxCharCountNumber(NumberType type) {
159     switch (type) {
160     case TYPE_SIZE_T:
161         return snprintf(NULL, 0, "%zu", SIZE_MAX);
162     case TYPE_DOUBLE:
163         return snprintf(NULL, 0, "%.*e", DBL_DIG, DBL_MAX);
164     default:
165         handleError("No_such_type_found,_aborting.");
166         return 0;
167     }
168 }
169
170 void printCharacterCountRangeNumber(NumberType type) {
171     int max = getMaxCharCountNumber(type);
172     if (max ≤ 0) {
173         handleError("Failed_to_find_max_character_count_for_type");
174         return;
175     }
176     printf("character_count_for_%s≤%u", typeDescriptions[type], max);
177 }
178
179 int convertInputToNumber(const char *input, void *value, NumberType type) {
180     char *endptr = NULL;
181     errno = 0;
182
183     switch (type) {
184     case TYPE_SIZE_T: {
185         unsigned long long temp = 0;
186
187         if (sizeof(size_t) == sizeof(unsigned long)) {
188             temp = strtoul(input, &endptr, 10);
189         } else if (sizeof(size_t) == sizeof(unsigned long long)) {
190             temp = strtoull(input, &endptr, 10);
191         } else if (sizeof(size_t) == sizeof(unsigned int)) {
192             temp = strtol(input, &endptr, 10);
193         } else {
194             handleError("Unsupported_size_t_size");
195             return FAILURE;
196         }
197
198         if (errno == ERANGE || temp > SIZE_MAX) {
199             handleErrorOverflow();
200             return FAILURE;
201         }
202
203         *(size_t *)value = (size_t)temp;
204
205         break;
206     }
207
208     case TYPE_DOUBLE: {
209         double temp = strtod(input, &endptr);
210         if (errno == ERANGE) {
211             handleErrorOverflow();
212             return FAILURE;
213         }
214         *(double *)value = temp;

```



```

215     break;
216 }
217
218 default:
219     handleError("Unsupported_number_type");
220     return FAILURE;
221 }
222
223 // Check if there are any non-number characters left after the number
224 if (*endptr != '\0') {
225     handleErrorNotNumber();
226     return FAILURE;
227 }
228
229 return SUCCESS;
230 }
231
232 int readNumber(void *value, NumberType type) {
233     int maxCharCount = getMaxCharCountNumber(type);
234     if (maxCharCount ≤ 0) {
235         handleError("Failed_to_find_max_character_count_for_type");
236         return FAILURE;
237     }
238     char *input = NULL;
239
240     int result = readLine(&input, DEFAULT_STRING_LENGTH, maxCharCount, stdin);
241
242     if (result ≠ SUCCESS)
243         return result;
244
245     if (!input)
246         return FAILURE;
247
248     if (convertInputToNumber(input, value, type) = FAILURE) {
249         free(input);
250         input = NULL;
251         return FAILURE;
252     }
253
254     free(input);
255     input = NULL;
256     return SUCCESS;
257 }
258
259 int readNumberWithinRange(void *value, NumberType type,
260                           const NumberRange *range, const char *prompt) {
261     printf("%s_", prompt);
262     printNumberRange(range);
263     printf("):_");
264
265     int result = readNumber(value, type);
266     switch (result) {
267     case FAILURE:
268         return FAILURE;
269     case EOF:
270         return EOF;
271     }
272
273     if (validateNumberRange(value, range) ≠ WITHIN_RANGE)
274         return FAILURE;

```

```

275
276     return SUCCESS;
277 }
278
279 int readNumberWithValidation(void *value, NumberType type, const char *prompt,
280                             ValidationFunc additionalCheck, ...) {
281     bool isValid = false;
282
283     va_list args;
284
285     while (!isValid) {
286         printf("%s:_", prompt);
287         int result = readNumber(value, type);
288         if (result == SUCCESS) {
289             if (additionalCheck != NULL) {
290                 va_start(args, additionalCheck);
291                 isValid = additionalCheck(value, args);
292                 va_end(args);
293             } else {
294                 isValid = true;
295             }
296         } else if (result == EOF) {
297             return EOF;
298         }
299     }
300
301     return SUCCESS;
302 }
303
304 int readNumberWithinRangeWithValidation(void *value, NumberType type,
305                                         const NumberRange *range,
306                                         const char *prompt,
307                                         ValidationFunc additionalCheck, ...) {
308     bool isValid = false;
309
310     va_list args;
311
312     while (!isValid) {
313         int result = readNumberWithinRange(value, type, range, prompt);
314         if (result == SUCCESS) {
315             if (additionalCheck != NULL) {
316                 va_start(args, additionalCheck);
317                 isValid = additionalCheck(value, args);
318                 va_end(args);
319             } else {
320                 isValid = true;
321             }
322         } else if (result == EOF) {
323             return EOF;
324         }
325     }
326
327     return SUCCESS;
328 }

```

---

*../src/io/string.c*

---

```

1 #include "io.h"
2 #include "utils.h"

```

```

3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <strings.h>
7
8 // UTF8
9
10 // NOTE: The function also advances the pointer to the next character
11 static int decodeUtf8(const char **str) {
12     const unsigned char *s = (const unsigned char *)*str;
13     int codepoint = 0;
14
15     if (*s < 0x80) {
16         // 1-byte character: 0xxxxxxx
17         codepoint = *s++;
18     } else if ((*s & 0xE0) == 0xC0) {
19         // 2-byte character: 110xxxxx 10xxxxxx
20         codepoint = (*s++ & 0x1F) << 6;
21         codepoint |= (*s++ & 0x3F);
22     } else if ((*s & 0xF0) == 0xE0) {
23         // 3-byte character: 1110xxxx 10xxxxxx 10xxxxxx
24         codepoint = (*s++ & 0x0F) << 12;
25         codepoint |= (*s++ & 0x3F) << 6;
26         codepoint |= (*s++ & 0x3F);
27     } else if ((*s & 0xF8) == 0xF0) {
28         // 4-byte character: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
29         codepoint = (*s++ & 0x07) << 18;
30         codepoint |= (*s++ & 0x3F) << 12;
31         codepoint |= (*s++ & 0x3F) << 6;
32         codepoint |= (*s++ & 0x3F);
33     } else {
34         // Invalid UTF-8
35         return -1;
36     }
37
38     *str = (const char *)s;
39     return codepoint;
40 }
41
42 // Checks string
43 bool isValidUtf8(const char *str) {
44     if (!str)
45         return false;
46
47     const char *ptr = str;
48
49     while (*ptr) {
50         int codepoint = decodeUtf8(&ptr);
51         if (codepoint == -1) {
52             handleError("Invalid_UTF-8_sequence.");
53             return false;
54         }
55     }
56
57     return true;
58 }
59
60 size_t getUtf8StringLength(const char *str) {
61     const char *ptr = str;
62     size_t length = 0;

```

```

63
64 while (*ptr) {
65     decodeUtf8(&ptr);
66     length++;
67 }
68
69 return length;
70 }
71
72 static char *readUtf8Char(const char *source) {
73     const char *ptr = source;
74     int codepoint = decodeUtf8(&ptr);
75
76     if (codepoint == -1) {
77         return NULL; // Invalid UTF-8
78     }
79
80     size_t len = ptr - source;
81     char *buffer = malloc(len + 1); // +1 for null terminator
82     if (!buffer) {
83         return NULL;
84     }
85
86     memcpy(buffer, source, len);
87     buffer[len] = '\0';
88     return buffer;
89 }
90
91 size_t findCharsInUtf8String(const char *str, const char *reject) {
92     const char *strPtr = str;
93     size_t pos = 0;
94
95     while (*strPtr) {
96         const char *rejectPtr = reject;
97         int codepointStr = decodeUtf8(&strPtr);
98
99         if (codepointStr == -1) {
100             handleError("Invalid_UTF-8_character_encountered.");
101             return pos;
102         }
103
104         while (*rejectPtr) {
105             int codepointReject = decodeUtf8(&rejectPtr);
106
107             if (codepointReject == -1) {
108                 handleError("Invalid_UTF-8_character_in_reject_set.");
109                 return pos;
110             }
111
112             if (codepointStr == codepointReject) {
113                 return pos;
114             }
115         }
116
117         pos++;
118     }
119
120     return pos;
121 }
122

```

```

123 int compareUtf8Strings(const char *str1, const char *str2) {
124     while (*str1 && *str2) {
125         int codepoint1 = decodeUtf8(&str1);
126         int codepoint2 = decodeUtf8(&str2);
127
128         if (codepoint1 < 0 || codepoint2 < 0) {
129             // Handle invalid UTF-8
130             return codepoint1 - codepoint2;
131         }
132
133         if (codepoint1 != codepoint2) {
134             return codepoint1 - codepoint2;
135         }
136     }
137
138     // If one string is longer, it's greater
139     return *str1 - *str2;
140 }
141
142 bool hasSuffix(const char *word, const char *suffix) {
143     size_t wordLength = getUtf8StringLength(word);
144     size_t suffixLength = getUtf8StringLength(suffix);
145
146     if (wordLength < suffixLength) {
147         return false;
148     }
149
150     const char *wordPtr = word;
151     const char *suffixPtr = suffix;
152
153     // Move to the possible position for the suffix
154     for (int i = 0; i < wordLength - suffixLength; i++)
155         decodeUtf8(&wordPtr);
156
157     while (*suffixPtr) {
158         int wordCodepoint = decodeUtf8(&wordPtr);
159         int suffixCodepoint = decodeUtf8(&suffixPtr);
160
161         if (wordCodepoint != suffixCodepoint)
162             return false;
163     }
164
165     return true;
166 }
167
168 bool isWhitespace(char c) {
169     return (c == '_' || c == '\t' || c == '\n' || c == '\r' || c == '\v' ||
170            c == '\f');
171 }
172
173 bool isLineEmpty(const char *line) {
174     if (!line)
175         return true;
176
177     for (size_t i = 0; line[i] != '\0'; i++) {
178         if (!isWhitespace(line[i]) && line[i] != '_') {
179             return false;
180         }
181     }
182     return true;

```

```

183 }
184
185 char *trimWhitespaceUtf8(char *str) {
186     if (!str) {
187         handleError("Passed_NULL_to_trimWhitespaceUtf8");
188         return NULL;
189     }
190
191     char *start = str;
192     char *end = str + strlen(str);
193
194     // trim initial
195     while (*start && isWhitespace(*start)) {
196         start++;
197     }
198
199     // trim trailing
200     while (end > start && isWhitespace(*(end - 1))) {
201         end--;
202     }
203
204     *end = '\0';
205
206     size_t trimmedLength = strlen(start);
207     memmove(str, start, trimmedLength + 1);
208
209     return str;
210 }
211
212 char *readStringWithFilterUntilValid(const char *prompt, size_t maxLength,
213                                     const char *reject) {
214     char *input = NULL;
215     bool isReadingInput = true;
216
217     do {
218         printf("%s:_", prompt);
219         int result = readLine(&input, DEFAULT_STRING_LENGTH, maxLength, stdin);
220
221         if (result == FAILURE)
222             return NULL;
223
224         if (result == EOF)
225             return NULL;
226
227         if (!isValidUtf8(input)) {
228             free(input);
229             input = NULL;
230             continue;
231         }
232
233         size_t initialLength = getUtf8StringLength(input);
234         trimWhitespaceUtf8(input);
235
236         isReadingInput = false;
237     } while (isReadingInput);
238
239     return input;
240 }

```

---

../src/io/choices.c

---

```
1 #include "io.h"
2 #include <stdlib.h>
3
4 #include <stddef.h>
5 #include <stdio.h>
6
7 #define DEFAULT_CHOICE_PROMPT "Choose_option_to_select"
8 void showChoices(const char **choices, int choice_count) {
9     for (int i = 0; i < choice_count; i++) {
10         printf("%d. %s\n", i + 1, choices[i]);
11     }
12     printf("\n");
13 }
14
15 size_t getUserChoice(NumberRange *choiceRange, const char *info,
16                     const char *choices[]) {
17
18     if (info)
19         printf("%s:\n", info);
20     if (choices)
21         showChoices(choices, *(int *)choiceRange->max);
22
23     size_t start = 1;
24     size_t end = *(size_t *)choiceRange->max;
25
26     size_t choice;
27     if (readNumberWithinRangeWithValidation(&choice, TYPE_SIZE_T, choiceRange,
28                                           DEFAULT_CHOICE_PROMPT, NULL) == EOF) {
29         return 0;
30     }
31
32     return choice;
33 }
34
35 NumberRange *initChoiceRange(size_t start, size_t end) {
36     NumberRange *range =
37         initializeNumberRange(TYPE_SIZE_T, &start, &end, true, true, "choice");
38     if (!range)
39         return NULL;
40     return range;
41 }
```

---

../src/io/validators.c

---

```
1 #include "io.h"
2 #include "utils.h"
3 #include <stdarg.h>
4 #include <stdbool.h>
5 #include <stddef.h>
6
7 // INFO: Only for size_t for now
8 bool isNotZero(void *value, va_list args) {
9     size_t val = *(size_t *)value;
10     if (val == 0) {
11         handleError("Value_can't_be_0!");
12         return false;
13     }
14 }
```

```
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;

```

```

84     if (maxLength ≠ INFINITE_LENGTH && newBufferSize > maxLength) {
85         newBufferSize = maxLength;
86     }
87
88     if (newBufferSize ≤ bufferSize) {
89         discardTillNewline();
90         handleErrorOverlength(maxLength);
91         free(input);
92         return FAILURE;
93     }
94
95     char *newBuffer = realloc(input, newBufferSize * sizeof(char));
96     if (newBuffer = NULL) {
97         free(input);
98         handleError("Failed_to_expand_input_buffer.");
99         return FAILURE;
100     }
101
102     input = newBuffer;
103     bufferSize = newBufferSize;
104 }
105
106 if (maxLength ≠ INFINITE_LENGTH && filledBufferSize ≥ maxLength) {
107     discardTillNewline();
108     handleErrorOverlength(maxLength);
109     free(input);
110     return FAILURE;
111 }
112 }
113
114 *buffer = input; // Final buffer contains the input
115 return SUCCESS;
116 }
117
118 // OUTPUT
119 #include <errno.h>
120
121 #define MOVE_UP "\033[F"
122 #define CLEAR_LINE "\033[2K"
123
124 #define ERROR_CLEAR_LINE_COUNT 3
125
126 void clearLastLines(int count) {
127     for (int i = 0; i < count; i++) {
128         printf(MOVE_UP CLEAR_LINE);
129     }
130 }
131
132 void continueAfterError() {
133     printf("Press_Enter_to_continue:");
134     fflush(stdout);
135     discardTillNewline(); // Getting any input from user
136     // NOTE: Remove this line when showcasing via screenshotting
137     clearLastLines(ERROR_CLEAR_LINE_COUNT);
138 }
139
140 int handleError(const char *format, ...) {
141     va_list args;
142     va_start(args, format);
143

```

```

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;

```

**Введені та одержані результати***Створення файлу*

```
-----
L Menu
  |
  | File
  |   |
  |   | 1: Create/Select File
  |   | 2: Read all Records
  |   | 3: Delete/Deselect File
  |   |
  |   Records
  |     |
  |     | 4: Create Record
  |     | 5: Read Record
  |     | 6: Edit Record
  |     | 7: Sort Records
  |     | 8: Insert Record
  |     | 9: Delete Record
  |     |
  | 10: Exit Program
  | 11: Verify File Header
  | 12: Display Menu Again
```

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

Name of file with records: перемога

File was created successfully.

### *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 100000000000): 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 100000000000): 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 100000000000): 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 5100000000): 310

Do you want to change region population?

Enter 'y' if you agree. Otherwise press whatever... y

Region population (from 1e-06 to 100000000000): 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 5100000000): 21823.7

Region population (from 1e-06 to 100000000000): 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:
```

[illegible]

```
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
sdf1,23sdf,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: Купрія

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 100000000000): 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.