

Εργασία 5: Πίνακες - Δείκτες - Αρχεία

Χρήστος Μαργιώλης - Εργαστηριακό τμήμα 9

Ιανουάριος 2020

Περιεχόμενα

| | | |
|----------|--|-----------|
| 1 | Δομή προγραμμάτων και οδηγίες εκτέλεσης | 1 |
| 1.1 | Εκτέλεση από Linux | 1 |
| 1.2 | Δομή φακέλων | 2 |
| 2 | combinations - συνδυασμοί | 2 |
| 2.1 | main.c | 2 |
| 2.2 | combinations.c | 2 |
| 2.3 | combinations.h | 6 |
| 2.4 | arrhandler.c | 6 |
| 2.5 | arrhandler.h | 8 |
| 2.6 | Περιγραφή υλοποίησης | 8 |
| 3 | kcombinations - συνδυασμοί με K | 8 |
| 3.1 | main.c | 8 |
| 3.2 | kcombinations.c | 9 |
| 3.3 | kcombinations.h | 12 |
| 3.4 | arrhandler.c | 13 |
| 3.5 | arrhandler.h | 14 |
| 3.6 | Περιγραφή υλοποίησης | 15 |
| 4 | fcombinations - συνδυασμοί από αρχείο | 15 |
| 4.1 | main.c | 15 |
| 4.2 | fcombinations.c | 15 |
| 4.3 | fcombinations.h | 19 |
| 4.4 | arrhandler.c | 19 |
| 4.5 | arrhandler.h | 21 |
| 4.6 | Περιγραφή υλοποίησης | 21 |
| 5 | minesweeper - ναρκαλιευτής | 21 |
| 5.1 | main.c | 21 |
| 5.2 | minesweeper.c | 21 |
| 5.3 | minesweeper.h | 28 |
| 5.4 | gameplay.c | 29 |
| 5.5 | gameplay.h | 36 |
| 5.6 | Περιγραφή υλοποίησης | 37 |
| 6 | Διευκρινήσεις | 37 |
| 7 | Εργαλεία | 37 |
| 1 | Δομή προγραμμάτων και οδηγίες εκτέλεσης | |
| 1.1 | Εκτέλεση από Linux | |

```
1 $ cd path-to-program
2 $ make
3 $ make run
4 $ make run ARGS=tst/data.txt #fcombinations ONLY
5 $ make clean #optional
```

1.2 Δομή φακέλων

2 combinations - συνδυασμοί

2.1 main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "combinations.h"
4 #include "arrhandler.h"
5
6
7 int main(int argc, char **argv)
8 {
9     int *arr, N, x1, x2, y1, y2;
10
11     N = get_n();
12
13     arr = fill_array(N);
14     quicksort(arr, 0, N-1);
15     x_pair(&x1, &x2);
16     y_pair(&y1, &y2);
17     print_combs(arr, N, x1, x2, y1, y2);
18
19     free(arr);
20
21     return 0;
22 }
```

2.2 combinations.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include "combinations.h"
5 #include "arrhandler.h"
6
7
8 int get_n()
9 {
10     int N;
11
12     do
13     {
```

```
14     system("clear||cls");
15     printf("N (6 < N <= 49): ");
16     scanf("%d", &N);
17 } while (N <= 6 || N > 49);
18
19     system("clear||cls");
20
21     return N;
22 }
23
24
25 void x_pair(int *x1, int *x2)
26 {
27     do
28     {
29         printf("x1: ");
30         scanf("%d", x1);
31         printf("x2: ");
32         scanf("%d", x2);
33     } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
34 }
35
36
37 void y_pair(int *y1, int *y2)
38 {
39     do
40     {
41         printf("y1: ");
42         scanf("%d", y1);
43         printf("y2: ");
44         scanf("%d", y2);
45     } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
46 }
47
48
49 void print_combs(int *arr, int N, int x1, int x2, int y1,
50                 int y2)
51 {
52     int *currComb = (int *)malloc(N * sizeof(int));
53     int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
54
55     if (currComb == NULL)
56     {
57         printf("Error! Not enough memory, exiting...\n");
58         exit(EXIT_FAILURE);
59     }
60     else
61     {
62         combinations(arr, currComb, 0, N-1, 0, &printed, &
unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
63     }
```

```

62     print_other(N, unFrstCond, unScndCondOnly, printed);
63 }
64
65     free(currComb);
66 }
67
68
69 void combinations(int *arr, int *currComb, int start, int
    end, int index, int *printed, int *unFrstCond, int *
    unScndCondOnly, int x1, int x2, int y1, int y2)
70 {
71     int i, j;
72
73     if (index == COMBSN)
74     {
75         for (j = 0; j < COMBSN; j++)
76         {
77             if (even_calc(currComb, x1, x2) && sum_comb_calc
                (currComb, y1, y2))
78             {
79                 printf("%d ", *(currComb + j));
80                 if (j == COMBSN - 1) { (*printed)++; printf(
                    "\n"); }
81             } // add freq
82         }
83         if (!even_calc(currComb, x1, x2) && sum_comb_calc(
                currComb, y1, y2)) (*unFrstCond)++;
84         if (!sum_comb_calc(currComb, y1, y2)) (*
                unScndCondOnly)++;
85         return;
86     }
87
88     for (i = start; i <= end && end-i+1 >= COMBSN-index; i++)
89     {
90         *(currComb + index) = *(arr + i);
91         combinations(arr, currComb, i+1, end, index+1,
                printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
92     }
93 }
94
95
96 bool even_calc(int *arr, int x1, int x2)
97 {
98     int numEven = 0, i;
99
100    for (i = 0; i < COMBSN; i++)
101        if (*(arr + i) % 2 == 0) numEven++;
102
103    return (numEven >= x1 && numEven <= x2) ? true : false;

```

```
104 }
105
106
107 bool sum_comb_calc(int *arr, int y1, int y2)
108 {
109     int sumNums = 0, i;
110
111     for (i = 0; i < COMBSN; i++)
112         sumNums += *(arr + i);
113
114     return (sumNums >= y1 && sumNums <= y2) ? true : false;
115 }
116
117
118 int frequency()
119 {
120
121 }
122
123
124 long int combinations_count(int N) // wtf ???????
125 {
126     return (factorial(N) / (factorial(COMBSN) * factorial(N
127 - COMBSN)));
128 }
129
130 long double factorial(int num)
131 {
132     int i;
133     long double fac;
134     if (num == 0) return -1;
135     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
136     return fac;
137 }
138
139
140 void print_other(int N, int unFrstCond, int unScndCondOnly,
141                 int printed)
142 {
143     printf("\nTotal number of combinations %d to %d: %ld\n",
144           N, COMBSN, combinations_count(N));
145     printf("Number of combinations not satisfying the first
146 condition: %d\n", unFrstCond);
147     printf("Number of combinations not satisfying the second
148 condition only: %d\n", unScndCondOnly);
149     printf("Printed combinations: %d\n", printed);
150 }
```

2.3 combinations.h

```
1 #ifndef COMBINATIONS_H
2 #define COMBINATIONS_H
3
4 #include <stdbool.h>
5
6 #define COMBSN 6
7
8 void x_pair(int *, int *);
9 void y_pair(int *, int *);
10
11 void print_combs(int *, int, int, int, int, int);
12 void combinations(int *, int *, int, int, int, int *, int *,
13                  int *, int, int, int, int);
14
15 bool even_calc(int *, int, int);
16 bool sum_comb_calc(int *, int, int);
17
18 int frequency();
19 long int combinations_count(int);
20 long double factorial(int);
21 void print_other(int, int, int, int); // add freq
22 #endif
```

2.4 arrhandler.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "arrhandler.h"
4 #include "combinations.h"
5
6
7 int *fill_array(int N)
8 {
9     int num, i = 0;
10    int *arr = (int *)malloc(N * sizeof(int));
11
12    if (arr == NULL)
13    {
14        printf("Error! Not enough memory, exiting...\n");
15        exit(EXIT_FAILURE);
16    }
17    else
18    {
19        do
20        {
21            printf("arr[%d]: ", i);
22            scanf("%d", &num);
```

```
23
24     if (num >= 1 && num <= 49)
25     {
26         if (i == 0) { *(arr + i) = num; i++; }
27         else
28         {
29             if (!exists_in_array(arr, N, num)) { *(
arr + i) = num; i++; }
30             else printf("Give a different number.\n"
);
31         }
32     }
33     else printf("Give a number in [1, 49].\n");
34     } while (i < N);
35 }
36
37 return arr;
38 }
39
40
41 bool exists_in_array(int *arr, int N, int num)
42 {
43     int *arrEnd = arr + (N - 1);
44     while (arr <= arrEnd && *arr != num) arr++;
45     return (arr <= arrEnd) ? true : false;
46 }
47
48
49 void quicksort(int *arr, int low, int high)
50 {
51     if (low < high)
52     {
53         int partIndex = partition(arr, low, high);
54         quicksort(arr, low, partIndex - 1);
55         quicksort(arr, partIndex + 1, high);
56     }
57 }
58
59
60 int partition(int *arr, int low, int high)
61 {
62     int pivot = *(arr + high);
63     int i = (low - 1), j;
64
65     for (j = low; j <= high - 1; j++)
66         if (*(arr + j) < pivot)
67             swap(arr + ++i, arr + j);
68
69     swap(arr + (i + 1), arr + high);
70     return (i + 1);
```



```
71 }
72
73
74 void swap(int *a, int *b)
75 {
76     int temp = *a;
77     *a = *b;
78     *b = temp;
79 }
```

2.5 arrhandler.h

```
1 #ifndef ARRHANDLER_H
2 #define ARRHANDLER_H
3
4 #include <stdbool.h>
5
6 int *fill_array(int);
7 bool exists_in_array(int *, int, int);
8
9 void quicksort(int *, int, int);
10 int partition(int *, int, int);
11 void swap(int *, int *);
12
13 #endif
```

2.6 Περιγραφή υλοποίησης

3 kcombinations - συνδυασμοί με K

3.1 main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "kcombinations.h"
4 #include "arrhandler.h"
5
6
7 int main(int argc, char **argv)
8 {
9     int *arr, N, K, x1, x2, y1, y2;
10
11     N = get_n();
12     K = get_k(N);
13
14     arr = fill_array(N);
15     quicksort(arr, 0, N-1);
16     x_pair(&x1, &x2);
17     y_pair(&y1, &y2);
18     print_combs(arr, N, K, x1, x2, y1, y2);
```

```
19     free(arr);
20
21     return 0;
22 }
23 }
```

3.2 kcombinations.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include "kcombinations.h"
5  #include "arrhandler.h"
6
7
8  int get_n()
9  {
10     int N;
11
12     do
13     {
14         system("clear||cls");
15         printf("N (6 < N <= 49): ");
16         scanf("%d", &N);
17     } while (N <= 6 || N > 49);
18
19     return N;
20 }
21
22
23 int get_k(int N)
24 {
25     int K;
26
27     do
28     {
29         printf("K (K < N <= 49): ");
30         scanf("%d", &K);
31     } while (K >= N || K > 49);
32
33     system("clear||cls");
34
35     return K;
36 }
37
38
39 void x_pair(int *x1, int *x2)
40 {
41     do
42     {
```

```
43     printf("x1: ");
44     scanf("%d", &x1);
45     printf("x2: ");
46     scanf("%d", &x2);
47 } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
48 }
49
50
51 void y_pair(int *y1, int *y2)
52 {
53     do
54     {
55         printf("y1: ");
56         scanf("%d", &y1);
57         printf("y2: ");
58         scanf("%d", &y2);
59     } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
60 }
61
62
63 void print_combs(int *arr, int N, int K, int x1, int x2, int
64     y1, int y2)
65 {
66     int *currComb = (int *)malloc(N * sizeof(int));
67     int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
68
69     if (currComb == NULL)
70     {
71         printf("Error! Not enough memory, exiting...\n");
72         exit(EXIT_FAILURE);
73     }
74     else
75     {
76         combinations(arr, currComb, 0, N-1, 0, K, &printed,
77             &unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
78         print_other(N, K, unFrstCond, unScndCondOnly,
79             printed);
80     }
81     free(currComb);
82 }
83
84 void combinations(int *arr, int *currComb, int start, int
85     end, int index, int K, int *printed, int *unFrstCond, int
86     *unScndCondOnly, int x1, int x2, int y1, int y2)
87 {
88     int i, j;
89
90     if (index == K)
```

```

88     {
89         for (j = 0; j < K; j++)
90         {
91             if (even_calc(currComb, K, x1, x2) &&
92                 sum_comb_calc(currComb, K, y1, y2))
93             {
94                 printf("%d ", *(currComb + j));
95                 if (j == K - 1) { (*printed)++; printf("\n")
96             }
97             } // add freq
98         }
99         if (!even_calc(currComb, K, x1, x2) && sum_comb_calc
100         (currComb, K, y1, y2)) (*unFrstCond)++;
101         if (!sum_comb_calc(currComb, K, y1, y2)) (*
102         unScndCondOnly)++;
103         return;
104     }
105 }
106
107 for (i = start; i <= end && end-i+1 >= K-index; i++)
108 {
109     *(currComb + index) = *(arr + i);
110     combinations(arr, currComb, i+1, end, index+1, K,
111     printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
112 }
113
114 bool even_calc(int *arr, int K, int x1, int x2)
115 {
116     int numEven = 0, i;
117
118     for (i = 0; i < K; i++)
119         if (*(arr + i) % 2 == 0) numEven++;
120
121     return (numEven >= x1 && numEven <= x2) ? true : false;
122 }
123
124 bool sum_comb_calc(int *arr, int K, int y1, int y2)
125 {
126     int sumNums = 0, i;
127
128     for (i = 0; i < K; i++)
129         sumNums += *(arr + i);
130
131     return (sumNums >= y1 && sumNums <= y2) ? true : false;
132 }
133
134 int frequency()

```

```

133 {
134
135 }
136
137
138 long int combinations_count(int N, int K) // wtf ???????
139 {
140     return (factorial(N) / (factorial(K) * factorial(N - K))
141 );
142 }
143
144 long double factorial(int num)
145 {
146     int i;
147     long double fac;
148     if (num == 0) return -1;
149     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
150     return fac;
151 }
152
153
154 void print_other(int N, int K, int unFrstCond, int
155 unScndCondOnly, int printed)
156 {
157     printf("\nTotal number of combinations %d to %d: %ld\n",
158 N, K, combinations_count(N, K));
159     printf("Number of combinations not satisfying the first
160 condition: %d\n", unFrstCond);
161     printf("Number of combinations not satisfying the second
162 condition only: %d\n", unScndCondOnly);
163     printf("Printed combinations: %d\n", printed);
164 }

```

3.3 kcombinations.h

```

1 #ifndef COMBINATIONS_H
2 #define COMBINATIONS_H
3
4 #include <stdbool.h>
5
6 void x_pair(int *, int *);
7 void y_pair(int *, int *);
8
9 void print_combs(int *, int, int, int, int, int, int);
10 void combinations(int *, int *, int, int, int, int, int *,
11 int *, int *, int, int, int, int);
12
13 bool even_calc(int *, int, int, int);
14 bool sum_comb_calc(int *, int, int, int);

```

```

14
15 int frequency();
16 long int combinations_count(int, int);
17 long double factorial(int);
18 void print_other(int, int, int, int, int); // add freq
19
20 #endif

```

3.4 arrhandler.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "arrhandler.h"
4 #include "kcombinations.h"
5
6
7 int *fill_array(int N)
8 {
9     int num, i = 0;
10    int *arr = (int *)malloc(N * sizeof(int));
11
12    if (arr == NULL)
13    {
14        printf("Error! Not enough memory, exiting...\n");
15        exit(EXIT_FAILURE);
16    }
17    else
18    {
19        do
20        {
21            printf("arr[%d]: ", i);
22            scanf("%d", &num);
23
24            if (num >= 1 && num <= 49)
25            {
26                if (i == 0) { *(arr + i) = num; i++; }
27                else
28                {
29                    if (!exists_in_array(arr, N, num)) { *(
arr + i) = num; i++; }
30                    else printf("Give a different number.\n"
);
31                }
32            }
33            else printf("Give a number in [1, 49].\n");
34        } while (i < N);
35    }
36
37    return arr;
38 }

```

```
39
40
41 bool exists_in_array(int *arr, int N, int num)
42 {
43     int *arrEnd = arr + (N - 1);
44     while (arr <= arrEnd && *arr != num) arr++;
45     return (arr <= arrEnd) ? true : false;
46 }
47
48
49 void quicksort(int *arr, int low, int high)
50 {
51     if (low < high)
52     {
53         int partIndex = partition(arr, low, high);
54         quicksort(arr, low, partIndex - 1);
55         quicksort(arr, partIndex + 1, high);
56     }
57 }
58
59
60 int partition(int *arr, int low, int high)
61 {
62     int pivot = *(arr + high);
63     int i = (low - 1), j;
64
65     for (j = low; j <= high - 1; j++)
66         if (*(arr + j) < pivot)
67             swap(arr + ++i, arr + j);
68
69     swap(arr + (i + 1), arr + high);
70     return (i + 1);
71 }
72
73
74 void swap(int *a, int *b)
75 {
76     int temp = *a;
77     *a = *b;
78     *b = temp;
79 }
```

3.5 arrhandler.h

```
1 #ifndef ARRHANDLER_H
2 #define ARRHANDLER_H
3
4 #include <stdbool.h>
5
6 int *fill_array(int);
```

```
7 bool exists_in_array(int *, int, int);
8
9 void quicksort(int *, int, int);
10 int partition(int *, int, int);
11 void swap(int *, int *);
12
13 #endif
```

3.6 Περιγραφή υλοποίησης

4 fcombinations - συνδυασμοί από αρχείο

4.1 main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "fcombinations.h"
4 #include "arrhandler.h"
5
6
7 int main(int argc, char **argv)
8 {
9     int N, K;
10    int *arr;
11    int x1, x2, y1, y2;
12
13    read_file(argv);
14
15    return 0;
16 }
```

4.2 fcombinations.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <string.h>
5 #include "fcombinations.h"
6
7 #define COMBSN 6
8
9
10 void read_file(char **argv)
11 {
12     FILE *dataFile = fopen(argv[1], "r");
13
14     if (dataFile == NULL)
15     {
16         printf("Error opening the file, exiting...\n");
17         exit(EXIT_FAILURE);
18     }
```



```
18     }
19     else
20     {
21         printf("Cool\n");
22         // fscanf();
23     }
24
25     fclose(dataFile);
26 }
27
28
29 void x_pair(int *x1, int *x2)
30 {
31     do
32     {
33         printf("x1: ");
34         scanf("%d", x1);
35         printf("x2: ");
36         scanf("%d", x2);
37     } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
38 }
39
40
41 void y_pair(int *y1, int *y2)
42 {
43     do
44     {
45         printf("y1: ");
46         scanf("%d", y1);
47         printf("y2: ");
48         scanf("%d", y2);
49     } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
50 }
51
52
53 void print_combs(int *arr, int N, int x1, int x2, int y1,
54                 int y2)
55 {
56     int *currComb = (int *)malloc(N * sizeof(int));
57     int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
58
59     if (currComb == NULL)
60     {
61         printf("Error! Not enough memory, exiting...\n");
62         exit(EXIT_FAILURE);
63     }
64     else
65     {
66         combinations(arr, currComb, 0, N-1, 0, &printed, &
unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
67     }
68 }
```

```

66     print_other(N, unFrstCond, unScndCondOnly, printed);
67 }
68
69     free(currComb);
70 }
71
72
73 void combinations(int *arr, int *currComb, int start, int
    end, int index, int *printed, int *unFrstCond, int *
    unScndCondOnly, int x1, int x2, int y1, int y2)
74 {
75     int i, j;
76
77     if (index == COMBSN)
78     {
79         for (j = 0; j < COMBSN; j++)
80         {
81             if (even_calc(currComb, x1, x2) && sum_comb_calc(
            currComb, y1, y2))
82             {
83                 printf("%d ", *(currComb + j));
84                 if (j == COMBSN - 1) { (*printed)++; printf(
            "\n"); }
85             } // add freq
86         }
87         if (!even_calc(currComb, x1, x2) && sum_comb_calc(
            currComb, y1, y2)) (*unFrstCond)++;
88         if (!sum_comb_calc(currComb, y1, y2)) (*
            unScndCondOnly)++;
89         return;
90     }
91
92     for (i = start; i <= end && end-i+1 >= COMBSN-index; i++)
93     {
94         *(currComb + index) = *(arr + i);
95         combinations(arr, currComb, i+1, end, index+1,
            printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
96     }
97 }
98
99
100 bool even_calc(int *arr, int x1, int x2)
101 {
102     int numEven = 0, i;
103
104     for (i = 0; i < COMBSN; i++)
105         if (*(arr + i) % 2 == 0) numEven++;
106
107     return (numEven >= x1 && numEven <= x2) ? true : false;

```

```
108 }
109
110
111 bool sum_comb_calc(int *arr, int y1, int y2)
112 {
113     int sumNums = 0, i;
114
115     for (i = 0; i < COMBSN; i++)
116         sumNums += *(arr + i);
117
118     return (sumNums >= y1 && sumNums <= y2) ? true : false;
119 }
120
121
122 int frequency()
123 {
124
125 }
126
127
128 long int combinations_count(int N) // wtf ???????
129 {
130     return (factorial(N) / (factorial(COMBSN) * factorial(N
131 - COMBSN)));
132 }
133
134 long double factorial(int num)
135 {
136     int i;
137     long double fac;
138     if (num == 0) return -1;
139     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
140     return fac;
141 }
142
143
144 void print_other(int N, int unFrstCond, int unScndCondOnly,
145                 int printed)
146 {
147     printf("\nTotal number of combinations %d to %d: %ld\n",
148           N, COMBSN, combinations_count(N));
149     printf("Number of combinations not satisfying the first
150 condition: %d\n", unFrstCond);
151     printf("Number of combinations not satisfying the second
152 condition only: %d\n", unScndCondOnly);
153     printf("Printed combinations: %d\n", printed);
154 }
```

4.3 fcombinations.h

```
1 #ifndef COMBINATIONS_H
2 #define COMBINATIONS_H
3
4 #include <stdbool.h>
5
6 #define COMBSN 6
7
8 void read_file();
9
10 void x_pair(int *, int *);
11 void y_pair(int *, int *);
12
13 void print_combs(int *, int, int, int, int, int);
14 void combinations(int *, int *, int, int, int, int *, int *,
15                  int *, int, int, int, int);
16
17 bool even_calc(int *, int, int);
18 bool sum_comb_calc(int *, int, int);
19
20 int frequency();
21 long int combinations_count(int);
22 long double factorial(int);
23 void print_other(int, int, int, int); // add freq
24 #endif
```

4.4 arrhandler.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "arrhandler.h"
4 #include "fcombinations.h"
5
6
7 int *fill_array(int N)
8 {
9     int num, i = 0;
10    int *arr = (int *)malloc(N * sizeof(int));
11
12    if (arr == NULL)
13    {
14        printf("Error! Not enough memory, exiting...\n");
15        exit(EXIT_FAILURE);
16    }
17    else
18    {
19        do
20        {
```

```
21         printf("arr[%d]: ", i);
22         scanf("%d", &num);
23
24         if (num >= 1 && num <= 49)
25         {
26             if (i == 0) { *(arr + i) = num; i++; }
27             else
28             {
29                 if (!exists_in_array(arr, N, num)) { *(
arr + i) = num; i++; }
30                 else printf("Give a different number.\n"
);
31             }
32         }
33         else printf("Give a number in [1, 49].\n");
34     } while (i < N);
35 }
36
37 return arr;
38 }
39
40
41 bool exists_in_array(int *arr, int N, int num)
42 {
43     int *arrEnd = arr + (N - 1);
44     while (arr <= arrEnd && *arr != num) arr++;
45     return (arr <= arrEnd) ? true : false;
46 }
47
48
49 void quicksort(int *arr, int low, int high)
50 {
51     if (low < high)
52     {
53         int partIndex = partition(arr, low, high);
54         quicksort(arr, low, partIndex - 1);
55         quicksort(arr, partIndex + 1, high);
56     }
57 }
58
59
60 int partition(int *arr, int low, int high)
61 {
62     int pivot = *(arr + high);
63     int i = (low - 1), j;
64
65     for (j = low; j <= high - 1; j++)
66         if (*(arr + j) < pivot)
67             swap(arr + ++i, arr + j);
68 }
```

```
69     swap(arr + (i + 1), arr + high);
70     return (i + 1);
71 }
72
73
74 void swap(int *a, int *b)
75 {
76     int temp = *a;
77     *a = *b;
78     *b = temp;
79 }
```

4.5 arrhandler.h

```
1  #ifndef ARRHANDLER_H
2  #define ARRHANDLER_H
3
4  #include <stdbool.h>
5
6  int *fill_array(int);
7  bool exists_in_array(int *, int, int);
8
9  void quicksort(int *, int, int);
10 int partition(int *, int, int);
11 void swap(int *, int *);
12
13 #endif
```

4.6 Περιγραφή υλοποίησης

5 minesweeper - ναρκαλιευτής

5.1 main.c

```
1  #include "minesweeper.h"
2
3  int main(int argc, char **argv)
4  {
5      main_win();
6      start();
7      endwin();
8
9      return 0;
10 }
```

5.2 minesweeper.c

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <time.h>
```

```
4 #include "minesweeper.h"
5 #include "gameplay.h"
6
7 void main_win()
8 {
9     initscr();
10    noecho();
11    cbreak();
12
13    WINDOW *mainWin = newwin(0, 0, 0, 0);
14    box(mainWin, 0, 0);
15    refresh();
16    wrefresh(mainWin);
17    keypad(mainWin, true);
18 }
19
20
21 void start()
22 {
23     int yMax, xMax;
24     int numSettings = 3;
25     getmaxyx(stdscr, yMax, xMax);
26
27     WINDOW *menuWin = newwin(numSettings+2, xMax-10, yMax-7,
28                               5);
29     box(menuWin, 0, 0);
30     refresh();
31     wrefresh(menuWin);
32     keypad(menuWin, true);
33
34     set_mode(menuWin);
35
36     int WIDTH = set_width(menuWin, xMax);
37     int HEIGHT = set_height(menuWin, yMax);
38     int NMINES = set_nmines(menuWin, WIDTH * HEIGHT);
39
40     game_win(WIDTH, HEIGHT, NMINES);
41     getchar();
42 }
43
44 void set_mode(WINDOW *menuWin) // loop
45 {
46     char mode;
47     mvwprintw(menuWin, 1, 1, "Keyboard or text mode (k/t): "
48               );
49     wrefresh(menuWin);
50     scanw("%c", &mode);
51     mvwprintw(menuWin, 1, strlen("Keyboard or text mode (k/t
52               ): ") + 1, "%c", mode);
```

```
51     wrefresh(menuWin);
52     mvwprintw(menuWin, 1, 1, CLEAR); // thanks stefastra &&
53     spyrosROUM!!!! :-DDDD
54     wrefresh(menuWin);
55     switch (mode)
56     {
57         case 'k':
58         case 'K':
59             mvwprintw(menuWin, 2, 1, "Keyboard mode");
60             wrefresh(menuWin);
61             break;
62         case 't':
63         case 'T':
64             mvwprintw(menuWin, 2, 1, "Text mode");
65             wrefresh(menuWin);
66             break;
67         default:
68             break;
69     }
70 }
71
72
73 int set_width(WINDOW *menuWin, int xMax)
74 {
75     int WIDTH;
76
77     do
78     {
79         mvwprintw(menuWin, 1, 1, "Width (Max = %d): ", xMax -
80             12);
81         wrefresh(menuWin);
82         scanw("%d", &WIDTH);
83         mvwprintw(menuWin, 1, strlen("Width (Max = XXX): ")
84             + 1, "%d", WIDTH);
85         wrefresh(menuWin);
86     } while (WIDTH < 5 || WIDTH > xMax - 12);
87
88     return WIDTH;
89 }
90
91 int set_height(WINDOW *menuWin, int yMax)
92 {
93     int HEIGHT;
94
95     do
96     {
97         mvwprintw(menuWin, 2, 1, "Height (Max = %d): ", yMax
98             - 12);
```



```

97         wrefresh(menuWin);
98         scanw("%d", &HEIGHT);
99         mvwprintw(menuWin, 2, strlen("Height (Max = YYY): ")
100 + 1, "%d", HEIGHT);
100         wrefresh(menuWin);
101     } while (HEIGHT < 5 || HEIGHT > yMax - 12);
102
103     return HEIGHT;
104 }
105
106
107 int set_nmines(WINDOW *menuWin, int DIMENSIONS)
108 {
109     int NMINES;
110
111     do
112     {
113         mvwprintw(menuWin, 3, 1, "Mines (Max = %d): ",
114 DIMENSIONS-10); // -10 so the player has a chance to win
114         wrefresh(menuWin);
115         scanw("%d", &NMINES);
116         mvwprintw(menuWin, 3, strlen("Mines (Max = MMMM): ")
117 + 1, "%d", NMINES);
117         wrefresh(menuWin);
118     } while (NMINES < 1 || NMINES > DIMENSIONS-10);
119
120     return NMINES;
121 }
122
123
124 void game_win(int WIDTH, int HEIGHT, int NMINES)
125 {
126     int yMax, xMax;
127     getmaxyx(stdscr, yMax, xMax);
128
129     WINDOW *gameWin = newwin(43, xMax-10, (yMax/2) - 24, 5);
130     // fix 43
131     box(gameWin, 0, 0);
132     refresh();
133     wrefresh(gameWin);
134     keypad(gameWin, true);
135
136     char **dispboard = init_dispboard(gameWin, WIDTH, HEIGHT
137 );
138     char **mineboard = init_mineboard(gameWin, WIDTH, HEIGHT
139 , NMINES);
140
141     selection(gameWin, dispboard, mineboard, WIDTH, HEIGHT);
142
143     free(dispboard);

```

```
141     free(mineboard);
142 }
143
144
145 char **init_dispboard(WINDOW *gameWin, int WIDTH, int HEIGHT
146 )
147 {
148     int i;
149     char **dispboard = (char **)malloc(WIDTH * sizeof(char *
150 ));
151     for (i = 0; i < WIDTH; i++)
152         dispboard[i] = (char *)malloc(HEIGHT);
153
154     if (dispboard == NULL)
155     {
156         mvprintw(1, 1, "Error, not enough memory, exiting...
157 ");
158         exit(EXIT_FAILURE);
159     }
160     else
161     {
162         fill_dispboard(dispboard, WIDTH, HEIGHT);
163         print_board(gameWin, dispboard, WIDTH, HEIGHT);
164         getchar();
165     }
166
167     return dispboard;
168 }
169
170 void fill_dispboard(char **dispboard, int WIDTH, int HEIGHT)
171 {
172     int i, j;
173
174     for (i = 0; i < WIDTH; i++)
175         for (j = 0; j < HEIGHT; j++)
176             dispboard[i][j] = HIDDEN;
177 }
178
179 char **init_mineboard(WINDOW *gameWin, int WIDTH, int HEIGHT
180 , int NMINES)
181 {
182     int i;
183     char **mineboard = (char **)malloc(WIDTH * sizeof(char *
184 ));
185     for (i = 0; i < WIDTH; i++)
186         mineboard[i] = (char *)malloc(HEIGHT);
187
188     if (mineboard == NULL)
189     {
```

```
186         mvprintw(1, 1, "Error, not enough memory, exiting...
187     ");
188     exit(EXIT_FAILURE);
189 }
190 else
191 {
192     place_mines(mineboard, WIDTH, HEIGHT, NMINES);
193     add_adj(mineboard, WIDTH, HEIGHT);
194     fill_spaces(mineboard, WIDTH, HEIGHT, NMINES);
195 }
196
197 return mineboard;
198 }
199
200 void place_mines(char **mineboard, int WIDTH, int HEIGHT,
201 int NMINES)
202 {
203     int i, wRand, hRand;
204
205     srand(time(NULL));
206
207     for (i = 0; i < NMINES; i++)
208     {
209         wRand = rand() % WIDTH;
210         hRand = rand() % HEIGHT;
211         mineboard[wRand][hRand] = MINE;
212     }
213 }
214
215 void add_adj(char **mineboard, int WIDTH, int HEIGHT)
216 {
217     int i, j;
218
219     for (i = 0; i < WIDTH; i++)
220         for (j = 0; j < HEIGHT; j++)
221             if (!is_mine(mineboard, i, j))
222                 mineboard[i][j] = adj_mines(mineboard, i, j,
223 WIDTH, HEIGHT) + '0';
224 }
225
226 bool is_mine(char **mineboard, int row, int col)
227 {
228     return (mineboard[row][col] == MINE) ? true : false;
229 }
230
231 bool outof_bounds(int row, int col, int WIDTH, int HEIGHT)
232 {
```

```

233     return (row < 0 || row > WIDTH-1 || col < 0 || col >
234           HEIGHT-1) ? true : false;
235 }
236
237
238 int8_t adj_mines(char **mineboard, int row, int col, int
239                WIDTH, int HEIGHT)
240 {
241     int8_t numAdj = 0;
242
243     if (!outof_bounds(row, col - 1, WIDTH, HEIGHT)    &&
244         mineboard[row][col-1] == MINE) numAdj++; // North
245     if (!outof_bounds(row, col + 1, WIDTH, HEIGHT)    &&
246         mineboard[row][col+1] == MINE) numAdj++; // South
247     if (!outof_bounds(row + 1, col, WIDTH, HEIGHT)    &&
248         mineboard[row+1][col] == MINE) numAdj++; // East
249     if (!outof_bounds(row - 1, col, WIDTH, HEIGHT)    &&
250         mineboard[row-1][col] == MINE) numAdj++; // West
251     if (!outof_bounds(row + 1, col - 1, WIDTH, HEIGHT) &&
252         mineboard[row+1][col-1] == MINE) numAdj++; // North-East
253     if (!outof_bounds(row - 1, col - 1, WIDTH, HEIGHT) &&
254         mineboard[row-1][col-1] == MINE) numAdj++; // North-West
255     if (!outof_bounds(row + 1, col + 1, WIDTH, HEIGHT) &&
256         mineboard[row+1][col+1] == MINE) numAdj++; // South-East
257     if (!outof_bounds(row - 1, col + 1, WIDTH, HEIGHT) &&
258         mineboard[row-1][col+1] == MINE) numAdj++; // South-West
259
260     return numAdj;
261 }
262
263 void fill_spaces(char **mineboard, int WIDTH, int HEIGHT,
264                int NMINES)
265 {
266     int i, j;
267
268     for (i = 0; i < WIDTH; i++)
269         for (j = 0; j < HEIGHT; j++)
270             if (mineboard[i][j] != MINE && mineboard[i][j] =
271                 = '0')
272                 mineboard[i][j] = '-';
273 }
274
275 void print_board(WINDOW *gameWin, char **mineboard, int
276                WIDTH, int HEIGHT)
277 {
278     int i, j;
279

```

```

270     for (i = 0; i < WIDTH; i++)
271     {
272         for (j = 0; j < HEIGHT; j++)
273         {
274             mvwaddch(gameWin, j + 1, i + 1, mineboard[i][j])
275         }
276         wrefresh(gameWin);
277     }
278 }
279
280
281 void filewrite(char **mineboard, int WIDTH, int HEIGHT, int
    hitRow, int hitCol)
282 {
283     int i, j;
284     FILE *mnsOut = fopen("mnsout.txt", "w");
285
286     if (mnsOut == NULL)
287     {
288         mvprintw(1, 1, "Error opening file, exiting...");
289         exit(EXIT_FAILURE);
290     }
291     else
292     {
293         fprintf(mnsOut, "Mine hit at position (%d, %d)\n\n",
            hitRow, hitCol);
294         fprintf(mnsOut, "Board overview\n\n");
295
296         for (i = 0; i < WIDTH; i++) // fix inversion
297         {
298             for (j = 0; j < HEIGHT; j++)
299                 fprintf(mnsOut, "%c ", mineboard[i][j]);
300             fprintf(mnsOut, "\n");
301         }
302
303         mvprintw(1, 1, "Session written to file");
304         refresh();
305     }
306
307     fclose(mnsOut);
308 }

```

5.3 minesweeper.h

```

1  #ifndef MINESWEEPER_H
2  #define MINESWEEPER_H
3
4  #if defined linux || defined __unix__
5  #include <ncurses.h>

```

```

6 #elif defined _WIN32 || defined _WIN64
7 #include <pdccurses.h>
8 #include <stdint.h>
9 #endif
10
11 #include <stdbool.h>
12
13 #define HIDDEN '#'
14 #define MINE '*'
15 #define CLEAR "
16
17 void main_win();
18 void start();
19 void set_mode(struct _win_st*);
20
21 int set_width(struct _win_st*, int);
22 int set_height(struct _win_st*, int);
23 int set_nmines(struct _win_st*, int);
24
25 void game_win(int, int, int);
26 char **init_dispboard(struct _win_st*, int, int);
27 void fill_dispboard(char **, int, int);
28 char **init_mineboard(struct _win_st*, int, int, int);
29 void place_mines(char **, int, int, int);
30 void add_adj(char **, int, int);
31 bool is_mine(char **, int, int);
32 bool outof_bounds(int, int, int, int);
33 int8_t adj_mines(char **, int, int, int, int);
34 void fill_spaces(char **, int, int, int);
35
36 void print_board(struct _win_st*, char **, int, int);
37 void filewrite(char **, int, int, int, int);
38
39 #endif

```

5.4 gameplay.c

```

1 #include <stdlib.h>
2 #include <string.h>
3 #include <time.h>
4 #include "minesweeper.h"
5 #include "gameplay.h"
6
7 void main_win()
8 {
9     initscr();
10    noecho();
11    cbreak();
12

```

```
13 WINDOW *mainWin = newwin(0, 0, 0, 0);
14 box(mainWin, 0, 0);
15 refresh();
16 wrefresh(mainWin);
17 keypad(mainWin, true);
18 }
19
20
21 void start()
22 {
23     int yMax, xMax;
24     int numSettings = 3;
25     getmaxyx(stdscr, yMax, xMax);
26
27     WINDOW *menuWin = newwin(numSettings+2, xMax-10, yMax-7,
28                               5);
29     box(menuWin, 0, 0);
30     refresh();
31     wrefresh(menuWin);
32     keypad(menuWin, true);
33
34     set_mode(menuWin);
35
36     int WIDTH = set_width(menuWin, xMax);
37     int HEIGHT = set_height(menuWin, yMax);
38     int NMINES = set_nmines(menuWin, WIDTH * HEIGHT);
39
40     game_win(WIDTH, HEIGHT, NMINES);
41     getchar();
42 }
43
44 void set_mode(WINDOW *menuWin) // loop
45 {
46     char mode;
47     mvwprintw(menuWin, 1, 1, "Keyboard or text mode (k/t): "
48 );
49     wrefresh(menuWin);
50     scanw("%c", &mode);
51     mvwprintw(menuWin, 1, strlen("Keyboard or text mode (k/t
52 ): ") + 1, "%c", mode);
53     wrefresh(menuWin);
54     mvwprintw(menuWin, 1, 1, CLEAR); // thanks stefastra &&
55     spyrosROUM!!!! :-DDDD
56     wrefresh(menuWin);
57
58     switch (mode)
59     {
60         case 'k':
61         case 'K':
```

```
59         mvwprintw(menuWin, 2, 1, "Keyboard mode");
60         wrefresh(menuWin);
61         break;
62     case 't':
63     case 'T':
64         mvwprintw(menuWin, 2, 1, "Text mode");
65         wrefresh(menuWin);
66         break;
67     default:
68         break;
69 }
70 }
71
72
73 int set_width(WINDOW *menuWin, int xMax)
74 {
75     int WIDTH;
76
77     do
78     {
79         mvwprintw(menuWin, 1, 1, "Width (Max = %d): ", xMax-
80 12);
81         wrefresh(menuWin);
82         scanw("%d", &WIDTH);
83         mvwprintw(menuWin, 1, strlen("Width (Max = XXX): ")
84 + 1, "%d", WIDTH);
85         wrefresh(menuWin);
86     } while (WIDTH < 5 || WIDTH > xMax - 12);
87
88     return WIDTH;
89 }
90
91 int set_height(WINDOW *menuWin, int yMax)
92 {
93     int HEIGHT;
94
95     do
96     {
97         mvwprintw(menuWin, 2, 1, "Height (Max = %d): ", yMax
98 -12);
99         wrefresh(menuWin);
100         scanw("%d", &HEIGHT);
101         mvwprintw(menuWin, 2, strlen("Height (Max = YYY): ")
102 + 1, "%d", HEIGHT);
103         wrefresh(menuWin);
104     } while (HEIGHT < 5 || HEIGHT > yMax - 12);
105
106     return HEIGHT;
107 }
```



```

105
106
107 int set_nmines(WINDOW *menuWin, int DIMENSIONS)
108 {
109     int NMINES;
110
111     do
112     {
113         mvwprintw(menuWin, 3, 1, "Mines (Max = %d): ",
114 DIMENSIONS-10); // -10 so the player has a chance to win
115         wrefresh(menuWin);
116         scanw("%d", &NMINES);
117         mvwprintw(menuWin, 3, strlen("Mines (Max = MMMM): ")
118 + 1, "%d", NMINES);
119         wrefresh(menuWin);
120     } while (NMINES < 1 || NMINES > DIMENSIONS-10);
121
122     return NMINES;
123 }
124
125 void game_win(int WIDTH, int HEIGHT, int NMINES)
126 {
127     int yMax, xMax;
128     getmaxyx(stdscr, yMax, xMax);
129
130     WINDOW *gameWin = newwin(43, xMax-10, (yMax/2) - 24, 5);
131     // fix 43
132     box(gameWin, 0, 0);
133     refresh();
134     wrefresh(gameWin);
135     keypad(gameWin, true);
136
137     char **dispboard = init_dispboard(gameWin, WIDTH, HEIGHT
138 );
139     char **mineboard = init_mineboard(gameWin, WIDTH, HEIGHT
140 , NMINES);
141
142     selection(gameWin, dispboard, mineboard, WIDTH, HEIGHT);
143
144     free(dispboard);
145     free(mineboard);
146 }
147
148 char **init_dispboard(WINDOW *gameWin, int WIDTH, int HEIGHT
149 )
150 {
151     int i;
152     char **dispboard = (char **)malloc(WIDTH * sizeof(char *

```

```
));  
149     for (i = 0; i < WIDTH; i++)  
150         dispboard[i] = (char *)malloc(HEIGHT);  
151  
152     if (dispboard == NULL)  
153     {  
154         mvprintw(1, 1, "Error, not enough memory, exiting...  
");  
155         exit(EXIT_FAILURE);  
156     }  
157     else  
158     {  
159         fill_dispboard(dispboard, WIDTH, HEIGHT);  
160         print_board(gameWin, dispboard, WIDTH, HEIGHT);  
161         getchar();  
162     }  
163  
164     return dispboard;  
165 }  
166  
167 void fill_dispboard(char **dispboard, int WIDTH, int HEIGHT)  
168 {  
169     int i, j;  
170  
171     for (i = 0; i < WIDTH; i++)  
172         for (j = 0; j < HEIGHT; j++)  
173             dispboard[i][j] = HIDDEN;  
174 }  
175  
176  
177 char **init_mineboard(WINDOW *gameWin, int WIDTH, int HEIGHT  
178 , int NMINES)  
179 {  
180     int i;  
181     char **mineboard = (char **)malloc(WIDTH * sizeof(char *  
182 ));  
183     for (i = 0; i < WIDTH; i++)  
184         mineboard[i] = (char *)malloc(HEIGHT);  
185  
186     if (mineboard == NULL)  
187     {  
188         mvprintw(1, 1, "Error, not enough memory, exiting...  
");  
189         exit(EXIT_FAILURE);  
190     }  
191     else  
192     {  
193         place_mines(mineboard, WIDTH, HEIGHT, NMINES);  
194         add_adj(mineboard, WIDTH, HEIGHT);  
195         fill_spaces(mineboard, WIDTH, HEIGHT, NMINES);  
196     }
```

```
194     }
195
196     return mineboard;
197 }
198
199
200 void place_mines(char **mineboard, int WIDTH, int HEIGHT,
201                 int NMINES)
202 {
203     int i, wRand, hRand;
204
205     srand(time(NULL));
206
207     for (i = 0; i < NMINES; i++)
208     {
209         wRand = rand() % WIDTH;
210         hRand = rand() % HEIGHT;
211         mineboard[wRand][hRand] = MINE;
212     }
213 }
214
215 void add_adj(char **mineboard, int WIDTH, int HEIGHT)
216 {
217     int i, j;
218
219     for (i = 0; i < WIDTH; i++)
220         for (j = 0; j < HEIGHT; j++)
221             if (!is_mine(mineboard, i, j))
222                 mineboard[i][j] = adj_mines(mineboard, i, j,
223                 WIDTH, HEIGHT) + '0';
224 }
225
226 bool is_mine(char **mineboard, int row, int col)
227 {
228     return (mineboard[row][col] == MINE) ? true : false;
229 }
230
231 bool outof_bounds(int row, int col, int WIDTH, int HEIGHT)
232 {
233     return (row < 0 || row > WIDTH-1 || col < 0 || col >
234             HEIGHT-1) ? true : false;
235 }
236
237
238 int8_t adj_mines(char **mineboard, int row, int col, int
239                 WIDTH, int HEIGHT)
240 {
```

```

240     int8_t numAdj = 0;
241
242     if (!outof_bounds(row, col - 1, WIDTH, HEIGHT)    &&
mineboard[row][col-1]    == MINE) numAdj++; // North
243     if (!outof_bounds(row, col + 1, WIDTH, HEIGHT)    &&
mineboard[row][col+1]    == MINE) numAdj++; // South
244     if (!outof_bounds(row + 1, col, WIDTH, HEIGHT)    &&
mineboard[row+1][col]    == MINE) numAdj++; // East
245     if (!outof_bounds(row - 1, col, WIDTH, HEIGHT)    &&
mineboard[row-1][col]    == MINE) numAdj++; // West
246     if (!outof_bounds(row + 1, col - 1, WIDTH, HEIGHT) &&
mineboard[row+1][col-1]  == MINE) numAdj++; // North-East
247     if (!outof_bounds(row - 1, col - 1, WIDTH, HEIGHT) &&
mineboard[row-1][col-1]  == MINE) numAdj++; // North-West
248     if (!outof_bounds(row + 1, col + 1, WIDTH, HEIGHT) &&
mineboard[row+1][col+1]  == MINE) numAdj++; // South-East
249     if (!outof_bounds(row - 1, col + 1, WIDTH, HEIGHT) &&
mineboard[row-1][col+1]  == MINE) numAdj++; // South-West
250
251     return numAdj;
252 }
253
254
255 void fill_spaces(char **mineboard, int WIDTH, int HEIGHT,
int NMINES)
256 {
257     int i, j;
258
259     for (i = 0; i < WIDTH; i++)
260         for (j = 0; j < HEIGHT; j++)
261             if (mineboard[i][j] != MINE && mineboard[i][j] =
= '0')
262                 mineboard[i][j] = '-';
263 }
264
265
266 void print_board(WINDOW *gameWin, char **mineboard, int
WIDTH, int HEIGHT)
267 {
268     int i, j;
269
270     for (i = 0; i < WIDTH; i++)
271     {
272         for (j = 0; j < HEIGHT; j++)
273         {
274             mvwaddch(gameWin, j + 1, i + 1, mineboard[i][j])
;
275             wrefresh(gameWin);
276         }
277     }

```

```

278 }
279
280
281 void filewrite(char **mineboard, int WIDTH, int HEIGHT, int
    hitRow, int hitCol)
282 {
283     int i, j;
284     FILE *mnsOut = fopen("mnsout.txt", "w");
285
286     if (mnsOut == NULL)
287     {
288         mvprintw(1, 1, "Error opening file, exiting...");
289         exit(EXIT_FAILURE);
290     }
291     else
292     {
293         fprintf(mnsOut, "Mine hit at position (%d, %d)\n\n",
            hitRow, hitCol);
294         fprintf(mnsOut, "Board overview\n\n");
295
296         for (i = 0; i < WIDTH; i++) // fix inversion
297         {
298             for (j = 0; j < HEIGHT; j++)
299                 fprintf(mnsOut, "%c ", mineboard[i][j]);
300             fprintf(mnsOut, "\n");
301         }
302
303         mvprintw(1, 1, "Session written to file");
304         refresh();
305     }
306
307     fclose(mnsOut);
308 }

```

5.5 gameplay.h

```

1  #ifndef GAMEPLAY_H
2  #define GAMEPLAY_H
3
4  #if defined linux || defined __unix__
5  #include <ncurses.h>
6  #elif defined _WIN32 || defined _WIN64
7  #include <pdcurses.h>
8  #include <stdint.h>
9  #endif
10
11 #include <stdbool.h>
12
13 void selection(struct _win_st*, char **, char **, int, int);
14 bool transfer(char **, char **, int, int);

```

```
15 void reveal(struct _win_st*, char **, int, int);
16 void game_over(struct _win_st*, char **, int, int);
17
18 #endif
```

5.6 Περιγραφή υλοποίησης

6 Διευκρινήσεις

7 Εργαλεία

- Editors: Visual Studio Code, Vim
- OS: Arch Linux
- Shell: zsh
- Συγγραφή: L^AT_EX