Εργασία 5: Πίνακες - Δείκτες - Αρχεία

Χρήστος Μαργιώλης - Εργαστηριακό τμήμα 9

Ιανουάριος 2020

Περιεχόμενα

1 Δομή προγραμμάτων και οδηγίες εκτέλεσης

1.1 Εκτέλεση από Unix/Linux/Mac

Λόγω του ότι ορισμένες βιβλιοθήκες που έχω χρησιμοποιήσει δεν είναι συμβατές με τα Windows, τα προγράμματα, και *κυρίως* ο ναρκαλιευτής, είναι περιορισμένα για συστήματα Unix.

```
$ cd path-to-program
$ make
$ make run
$ make run ARGS=txt/data40.txt #fcombinations MONO
$ make clean
```

Προκειμένου να εκτελεστεί ο ναρκαλιευτής χρειάζονται τα παρακάτω dependencies:

- cmake
- ncurses
- SDL2
- SDL2-mixer

1.2 Δομή φακέλων

Το κάθε πρόγραμμα, είναι δομημένο ως εξής: Υπάρχουν πέντε *βασικοί* φάκελοι, καθώς και ένα Makefile στο top directory. Στον φάκελο src βρίσκονται οι πηγαίοι κώδικες, στον include τα header files, στον obj τα object files και στον bin το εκτελέσιμο αρχείο. Στον φάκελο txt υπάρχουν τα text files που διαβάζονται οι γράφονται από το κάθε πρόγραμμα. Το Makefile είναι υπεύθυνο για την μεταγλώττιση όλων των αρχείων μαζί, και την τοποθέτησή τους στους κατάλληλους φακέλους, την εκτέλεση των προγραμμάτων, καθώς και τον καθαρισμό των φακέλων (διαγράφει τα object files και το εκτελέσιμο με την εντολή make clean).

2 combinations - συνδυασμοί

2.1 main.c

```c
#include "combinations.h"

int main(int argc, char **argv)
{
    int *arr, N, x1, x2, y1, y2;

    N = get_n();

    arr = fill_array(N);
    quicksort(arr, 0, N-1);
    x_pair(&x1, &x2);
    y_pair(&y1, &y2);
    print_combs(arr, N, x1, x2, y1, y2);

    free(arr);

    return 0;
}
```

2.2 combinations.c

```c
#include "combinations.h"

int get_n()
{
    int N;

    do
    {
        system("clear||cls");
        printf("N (6 < N <= 49): ");
        scanf("%d", &N);
    } while (N <= 6 || N > 49);

    system("clear||cls");

    return N;
}


void x_pair(int *x1, int *x2)
{
    do
    {
        printf("x1: ");
        scanf("%d", x1);
        printf("x2: ");
        scanf("%d", x2);
    } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
}
```

```
30
31
32 void y_pair(int *y1, int *y2)
33 {
34     do
35     {
36         printf("y1: ");
37         scanf("%d", y1);
38         printf("y2: ");
39         scanf("%d", y2);
40     } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
41 }
42
43
44 void print_combs(int *arr, int N, int x1, int x2, int y1,
       int y2)
45 {
46     int *currComb = (int *)malloc(N * sizeof(int));
47     int *freqArr = (int *)malloc(N *sizeof(int));
48     int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
49
50     if (currComb == NULL)
51     {
52         set_color(BOLD_RED);
53         printf("Error! Not enough memory, exiting...\n");
54         exit(EXIT_FAILURE);
55         set_color(STANDARD);
56     }
57     else
58     {
59         combinations(arr, currComb, freqArr, 0, N-1, 0, &
       printed, &unFrstCond, &unScndCondOnly, x1, x2, y1, y2, N)
       ;
60         print_other(N, unFrstCond, unScndCondOnly, printed,
       arr, freqArr);
61     }
62
63     free(currComb);
64     free(freqArr);
65 }
66
67
68 void combinations(int *arr, int *currComb, int *freqArr, int
        start, int end, int index, int *printed, int *unFrstCond
       , int *unScndCondOnly, int x1, int x2, int y1, int y2,
       int N)
69 {
70     int i, j;
71
72     if (index == COMBSN)
```

```
73      {
74          for (j = 0; j < COMBSN; j++)
75          {
76              if (even_calc(currComb, x1, x2) && sum_comb_calc
    (currComb, y1, y2))
77              {
78                  printf("%d ", *(currComb + j));
79                  if (j == COMBSN - 1)
80                  {
81                      frequency(freqArr, currComb, arr, N);
82                      (*printed)++;
83                      printf("\n");
84                  }
85              }
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, freqArr, i+1, end, index
    +1, printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2,
    N);
96      }
97
98  }
99
100
101 bool even_calc(int *arr, int x1, int x2)
102 {
103     int numEven = 0, i;
104
105     for (i = 0; i < COMBSN; i++)
106         if (*(arr + i) % 2 == 0) numEven++;
107
108     return (numEven >= x1 && numEven <= x2) ? true : false;
109 }
110
111
112 bool sum_comb_calc(int *arr, int y1, int y2)
113 {
114     int sumNums = 0, i;
115
116     for (i = 0; i < COMBSN; i++)
```

```
117         sumNums += *(arr + i);
118
119     return (sumNums >= y1 && sumNums <= y2) ? true : false;
120 }
121
122
123 void frequency(int *freqArr, int *currComb, int *arr, int N)
124 {
125     int pos, i;
126
127     for (i = 0; i < COMBSN; i++)
128     {
129         pos = find_pos(arr, N, *(currComb + i));
130         (*(freqArr + pos))++;
131     }
132 }
133
134
135 long int combinations_count(int N)
136 {
137     return (factorial(N) / (factorial(COMBSN) * factorial(N
    - COMBSN)));
138 }
139
140
141 long double factorial(int num)
142 {
143     int i;
144     long double fac;
145     if (num == 0) return -1;
146     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
147     return fac;
148 }
149
150
151 void print_other(int N, int unFrstCond, int unScndCondOnly,
    int printed,int *arr, int *freqArr)
152 {
153     int i;
154
155     printf("\nTotal number of combinations %d to %d: %ld\n",
     N, COMBSN, combinations_count(N));
156     printf("Number of combinations not satisfying the first
    condition: %d\n", unFrstCond);
157     printf("Number of combinations not satisfying the second
     condition only: %d\n", unScndCondOnly);
158     printf("Printed combinations: %d\n\n", printed);
159
160     for (i = 0; i < N; i++)
161         printf("%d appeared %d times\n", *(arr + i), *(
```

```
    freqArr + i));
162
163 }
```

2.3 combinations.h

```
1  #ifndef COMBINATIONS_H
2  #define COMBINATIONS_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <stdbool.h>
7
8  #include "arrhandler.h"
9  #include "ccolors.h"
10
11 #define COMBSN 6
12
13 void x_pair(int *, int *);
14 void y_pair(int *, int *);
15
16 void print_combs(int *, int, int, int, int, int);
17 void combinations(int *, int *, int *, int, int, int, int *,
       int *, int *, int, int, int, int, int);
18
19 bool even_calc(int *, int, int);
20 bool sum_comb_calc(int *, int, int);
21
22 void frequency(int *, int *, int *, int);
23 long int combinations_count(int);
24 long double factorial(int);
25 void print_other(int, int, int, int, int *, int *);
26
27 #endif
```

2.4 arrhandler.c

```
1  #include "arrhandler.h"
2
3  int *fill_array(int N)
4  {
5      int num, i = 0;
6      int *arr = (int *)malloc(N * sizeof(int));
7
8      if (arr == NULL)
9      {
10         set_color(BOLD_RED);
11         printf("Error! Not enough memory, exiting...\n");
12         exit(EXIT_FAILURE);
13         set_color(STANDARD);
```

```
14      }
15      else
16      {
17          do
18          {
19              printf("arr[%d]: ", i);
20              scanf("%d", &num);
21
22              if (num >= 1 && num <= 49)
23              {
24                  if (i == 0) { *(arr + i) = num; i++; }
25                  else
26                  {
27                      if (!exists_in_array(arr, N, num)) { *(
    arr + i) = num; i++; }
28                      else printf("Give a different number.\n"
    );
29                  }
30              }
31              else printf("Give a number in [1, 49].\n");
32          } while (i < N);
33      }
34
35      return arr;
36  }
37
38
39  bool exists_in_array(int *arr, int N, int num)
40  {
41      int *arrEnd = arr + (N - 1);
42      while (arr <= arrEnd && *arr != num) arr++;
43      return (arr <= arrEnd) ? true : false;
44  }
45
46
47  void quicksort(int *arr, int low, int high)
48  {
49      if (low < high)
50      {
51          int partIndex = partition(arr, low, high);
52          quicksort(arr, low, partIndex - 1);
53          quicksort(arr, partIndex + 1, high);
54      }
55  }
56
57
58  int partition(int *arr, int low, int high)
59  {
60      int pivot = *(arr + high);
61      int i = (low - 1), j;
```

```
62
63      for (j = low; j <= high - 1; j++)
64          if (*(arr + j) < pivot)
65              swap(arr + ++i, arr + j);
66
67      swap(arr + (i + 1), arr + high);
68      return (i + 1);
69  }
70
71
72  void swap(int *a, int *b)
73  {
74      int temp = *a;
75      *a = *b;
76      *b = temp;
77  }
78
79
80  int find_pos(int *arr, int numIter, int val)
81  {
82      int pos, i;
83
84      for (i = 0; i < numIter; i++)
85          if (val == *(arr + i))
86              pos = i;
87
88      return pos;
89  }
```

2.5 arrhandler.h

```
1  #ifndef ARRHANDLER_H
2  #define ARRHANDLER_H
3
4  #include "combinations.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 int find_pos(int *, int, int);
14
15 #endif
```

2.6 Περιγραφή υλοποιήσης

Το πρόγραμμα αυτό, όπως και τα ακόλουθα 2 προγράμματα, έχουν ως στόχο την εμφάνιση συνδυασμών, εφόσον πληρούν τις προϋποθέσεις του φύλλου εργασίας.

Όσο αφορά το πρόγραμμα όμως, στο main.c αποθηκεύεται η τιμή του N, ο πίνακας N θέσεων που έχει τα στοιχεία που δίνει ο χρήστης, καθώς και τα ζευγάρια x_1, x_2, y_1 και y_2. Από την main, τέλος, καλείται και η συνάρτηση εκτύπωσης των συνδυασμών, από την οποία καλούνται επιπλέον συναρτήσεις προκειμένου να εκτελεστεί σωστά αυτή η λειτουργία.

Στο arrhandler.c γίνεται όλος ο χειρισμός του πίνακα - δηλαδή, η *δυναμική* του δέσμευση, το γέμισμα του, στο όποιο εκτελούνται συναρτήσεις για τον έλεγχο του αν ένας αριθμός κατά την εισαγωγή τιμών έχει ήδη εισαχθεί, και αν τηρούνται τα όρια $[1, 49]$. Επίσης, γίνεται ταξινόμηση του πίνακα με τον αλγόριθμο quicksort, και τέλος, μία επιπλέον λειτουργία, που θα αναλυθεί στην συνέχεια.

Στο combinations.c, γίνονται όλες οι λειτουργίες που αφορούν τους συνδυασμούς, τα ζευγάρια x και y, και τους ελέγχους για το αν τηρούνται οι εξής προϋποθέσεις προκειμένου να εμφανιστεί ένας συνδυασμός 6 αριθμών.

- Το πλήθος των αρτίων αριθμών του συνδυασμού να βρίσκεται στο διάστημα $[x_1, x_2]$
- Το άθροισμα των έξι αριθμών του συνδυασμού να βρίσκεται στο διάστημα $[y_1, y_2]$

Έπειτα, μετριούνται πόσοι συνδυασμοί δεν πληρούσαν τον πρώτο όρο ή μόνο τον δεύτερο όρο, ποιοί συνδυασμοί τυπώθηκαν, καθώς και την συχνότητα εμφάνισης του κάθε στοιχείου στο σύνολο των συνδυασμών που τυπώθηκαν. Για την τελευταία λειτουργία, χρησιμοποιείται η συναρτήση findpos() που βρίσκεται στο arrhandler.c, και ελέγχει για κάθε στοιχείο του τρέχοντως συνδυασμού, ποια θέση έχει στον πίνακα N θέσεων, η οποία αντιστοιχίζεται με έναν τρίτο πίνακα, τον freqArr, στον οποίο αποθηκεύονται ως ακέραιες τιμές η συνχότητες εμφάνισης του κάθε στοιχείου. Ο λόγος που χρειάζεται μια συνάρτηση που βρίσκει την τοποθεσία του κάθε στοιχείου στον πίνακα N θέσεων είναι ώστε να αυξηθεί κατά ένα κάθε φορά ο πίνακας των συχνοτήτων στην κατάλληλη θέση, η οποία όπως προανέφερα, αντιστοιχεί στο στοιχείο του πίνακα N θέσεων, δηλαδή στο στοιχείο που έλεγχεται κάθε φορά κατα την προσπέλαση του πίνακα του τρέχοντως συνδυασμού.

3 kcombinations - συνδυασμοί με K

3.1 main.c

```
1  #include "kcombinations.h"
2
3  int main(int argc, char **argv)
4  {
5      int *arr, N, K, x1, x2, y1, y2;
6
7      N = get_n();
8      K = get_k(N);
9
10     arr = fill_array(N);
```

```
11      quicksort(arr, 0, N-1);
12      x_pair(&x1, &x2);
13      y_pair(&y1, &y2);
14      print_combs(arr, N, K, x1, x2, y1, y2);
15
16      free(arr);
17
18      return 0;
19  }
```

3.2 kcombinations.c

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

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

```
     y1, int y2)
{
    int i, j;

    if (index == K)
    {
        for (j = 0; j < K; j++)
        {
            if (even_calc(currComb, K, x1, x2) &&
    sum_comb_calc(currComb, K, y1, y2))
            {
                printf("%d ", *(currComb + j));
                if (j == K - 1)
                {
                    frequency(freqArr, currComb, arr, N);
                    (*printed)++;
                    printf("\n");
                }
            }
        }
        if (!even_calc(currComb, K, x1, x2) && sum_comb_calc
    (currComb, K, y1, y2)) (*unFrstCond)++;
        if (!sum_comb_calc(currComb, K, y1, y2)) (*
    unScndCondOnly)++;
        return;
    }

    for (i = start; i <= end && end-i+1 >= K-index; i++)
    {
        *(currComb + index) = *(arr + i);
        combinations(arr, currComb, freqArr, i+1, end, index
    +1, N, K, printed, unFrstCond, unScndCondOnly, x1, x2, y1
    , y2);
    }
}


bool even_calc(int *arr, int K, int x1, int x2)
{
    int numEven = 0, i;

    for (i = 0; i < K; i++)
        if (*(arr + i) % 2 == 0) numEven++;

    return (numEven >= x1 && numEven <= x2) ? true : false;
}


bool sum_comb_calc(int *arr, int K, int y1, int y2)
{
```

```
127     int sumNums = 0, i;
128 
129     for (i = 0; i < K; i++)
130         sumNums += *(arr + i);
131 
132     return (sumNums >= y1 && sumNums <= y2) ? true : false;
133 }
134 
135 
136 int frequency(int *freqArr, int *currComb, int *arr, int N)
137 {
138     int pos, i;
139 
140     for (i = 0; i < N; i++)
141     {
142         pos = find_pos(arr, N, *(currComb + i));
143         (*(freqArr + pos))++;
144     }
145 }
146 
147 
148 long int combinations_count(int N, int K)
149 {
150     return (factorial(N) / (factorial(K) * factorial(N - K))
    );
151 }
152 
153 
154 long double factorial(int num)
155 {
156     int i;
157     long double fac;
158     if (num == 0) return -1;
159     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
160     return fac;
161 }
162 
163 
164 void print_other(int N, int K, int unFrstCond, int
    unScndCondOnly, int printed, int *arr, int *freqArr)
165 {
166     int i;
167 
168     printf("\nTotal number of combinations %d to %d: %ld\n",
     N, K, combinations_count(N, K));
169     printf("Number of combinations not satisfying the first
    condition: %d\n", unFrstCond);
170     printf("Number of combinations not satisfying the second
     condition only: %d\n", unScndCondOnly);
171     printf("Printed combinations: %d\n\n", printed);
```

```
172
173     for (i = 0; i < N; i++)
174         printf("%d appeared %d times\n", *(arr + i), *(
    freqArr + i));
175 }
```

3.3 kcombinations.h

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

3.4 arrhandler.c

```
1  #include "arrhandler.h"
2
3  int *fill_array(int N)
4  {
5      int num, i = 0;
6      int *arr = (int *)malloc(N * sizeof(int));
7
8      if (arr == NULL)
9      {
10         set_color(BOLD_RED);
11         printf("Error! Not enough memory, exiting...\n");
12         exit(EXIT_FAILURE);
13         set_color(STANDARD);
```

```
14      }
15      else
16      {
17          do
18          {
19              printf("arr[%d]: ", i);
20              scanf("%d", &num);
21
22              if (num >= 1 && num <= 49)
23              {
24                  if (i == 0) { *(arr + i) = num; i++; }
25                  else
26                  {
27                      if (!exists_in_array(arr, N, num)) { *(
    arr + i) = num; i++; }
28                      else printf("Give a different number.\n"
    );
29                  }
30              }
31              else printf("Give a number in [1, 49].\n");
32          } while (i < N);
33      }
34
35      return arr;
36  }
37
38
39  bool exists_in_array(int *arr, int N, int num)
40  {
41      int *arrEnd = arr + (N - 1);
42      while (arr <= arrEnd && *arr != num) arr++;
43      return (arr <= arrEnd) ? true : false;
44  }
45
46
47  void quicksort(int *arr, int low, int high)
48  {
49      if (low < high)
50      {
51          int partIndex = partition(arr, low, high);
52          quicksort(arr, low, partIndex - 1);
53          quicksort(arr, partIndex + 1, high);
54      }
55  }
56
57
58  int partition(int *arr, int low, int high)
59  {
60      int pivot = *(arr + high);
61      int i = (low - 1), j;
```

```
62
63      for (j = low; j <= high - 1; j++)
64          if (*(arr + j) < pivot)
65              swap(arr + ++i, arr + j);
66
67      swap(arr + (i + 1), arr + high);
68      return (i + 1);
69  }
70
71
72  void swap(int *a, int *b)
73  {
74      int temp = *a;
75      *a = *b;
76      *b = temp;
77  }
78
79
80  int find_pos(int *arr, int numIter, int val)
81  {
82      int pos, i;
83
84      for (i = 0; i < numIter; i++)
85          if (val == *(arr + i))
86              pos = i;
87
88      return pos;
89  }
```

3.5 arrhandler.h

```
1  #ifndef ARRHANDLER_H
2  #define ARRHANDLER_H
3
4  #include "kcombinations.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 υλοποιήσης

Όπως και στο προηγούμενο πρόγραμμα, το combinations, το kcombinations λειτουργεί με τον ίδιο ακριβώς τρόπο, με την διαφορά οτι η σταθερά COMBSN = 6 του προηγούμενου προγράμματος έχει αντικατασταθεί με την μεταβλητή K, η

οποία δίνεται κάθε φορά από τον χρήστη.

4 fcombinations - συνδυασμοί από αρχείο

4.1 main.c

```c
#include "fcombinations.h"

int main(int argc, char **argv)
{
    int *arr, N, x1, x2, y1, y2;
    FILE *dataFile = fopen(*(argv + 1), "r");

    if (dataFile == NULL)
    {
        set_color(BOLD_RED);
        printf("Error! Not enough memory, exiting...\n");
        exit(EXIT_FAILURE);
        set_color(STANDARD);
    }
    else
    {
        N = get_n(dataFile);
        arr = fill_array(N, dataFile);
        quicksort(arr, 0, N-1);
        x_pair(&x1, &x2, dataFile);
        y_pair(&y1, &y2, dataFile);
        print_combs(arr, N, x1, x2, y1, y2);
    }

    fclose(dataFile);
    free(arr);

    return 0;
}
```

4.2 fcombinations.c

```c
#include "fcombinations.h"

int get_n(FILE *dataFile)
{
    int N;

    do
    {
        fscanf(dataFile, "%d\n", &N);
    } while (N <= 6 || N > 49);

```

```
12      return N;
13  }
14
15
16  void x_pair(int *x1, int *x2, FILE *dataFile)
17  {
18      do
19      {
20          fscanf(dataFile, "%d\n", x1);
21          fscanf(dataFile, "%d\n", x2);
22      } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
23  }
24
25
26  void y_pair(int *y1, int *y2, FILE *dataFile)
27  {
28      do
29      {
30          fscanf(dataFile, "%d\n", y1);
31          fscanf(dataFile, "%d\n", y2);
32      } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
33  }
34
35
36  void print_combs(int *arr, int N, int x1, int x2, int y1,
        int y2)
37  {
38      int *currComb = (int *)malloc(N * sizeof(int));
39      int *freqArr = (int *)malloc(N *sizeof(int));
40      int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
41
42      if (currComb == NULL)
43      {
44          set_color(BOLD_RED);
45          printf("Error! Not enough memory, exiting...\n");
46          exit(EXIT_FAILURE);
47          set_color(STANDARD);
48      }
49      else
50      {
51          combinations(arr, currComb, freqArr, 0, N-1, 0, &
        printed, &unFrstCond, &unScndCondOnly, x1, x2, y1, y2, N)
        ;
52          print_other(N, unFrstCond, unScndCondOnly, printed,
        arr, freqArr);
53      }
54
55      free(currComb);
56      free(freqArr);
57  }
```

```
58
59
60 void combinations(int *arr, int *currComb, int *freqArr, int
       start, int end, int index, int *printed, int *unFrstCond
   , int *unScndCondOnly, int x1, int x2, int y1, int y2,
   int N)
61 {
62     int i, j;
63
64     if (index == COMBSN)
65     {
66         for (j = 0; j < COMBSN; j++)
67         {
68             if (even_calc(currComb, x1, x2) && sum_comb_calc
   (currComb, y1, y2))
69             {
70                 printf("%d ", *(currComb + j));
71                 if (j == COMBSN - 1)
72                 {
73                     frequency(freqArr, currComb, arr, N);
74                     (*printed)++;
75                     printf("\n");
76                 }
77             }
78         }
79         if (!even_calc(currComb, x1, x2) && sum_comb_calc(
   currComb, y1, y2)) (*unFrstCond)++;
80         if (!sum_comb_calc(currComb, y1, y2)) (*
   unScndCondOnly)++;
81         return;
82     }
83
84     for (i = start; i <= end && end-i+1 >= COMBSN-index; i++
   )
85     {
86         *(currComb + index) = *(arr + i);
87         combinations(arr, currComb, freqArr, i+1, end, index
   +1, printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2,
   N);
88     }
89
90 }
91
92
93 bool even_calc(int *arr, int x1, int x2)
94 {
95     int numEven = 0, i;
96
97     for (i = 0; i < COMBSN; i++)
98         if (*(arr + i) % 2 == 0) numEven++;
```

```
99
100     return (numEven >= x1 && numEven <= x2) ? true : false;
101 }
102
103
104 bool sum_comb_calc(int *arr, int y1, int y2)
105 {
106     int sumNums = 0, i;
107
108     for (i = 0; i < COMBSN; i++)
109         sumNums += *(arr + i);
110
111     return (sumNums >= y1 && sumNums <= y2) ? true : false;
112 }
113
114
115 void frequency(int *freqArr, int *currComb, int *arr, int N)
116 {
117     int pos, i;
118
119     for (i = 0; i < COMBSN; i++)
120     {
121         pos = find_pos(arr, N, *(currComb + i));
122         (*(freqArr + pos))++;
123     }
124 }
125
126
127 long int combinations_count(int N)
128 {
129     return (factorial(N) / (factorial(COMBSN) * factorial(N
    - COMBSN)));
130 }
131
132
133 long double factorial(int num)
134 {
135     int i;
136     long double fac;
137     if (num == 0) return -1;
138     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
139     return fac;
140 }
141
142
143 void print_other(int N, int unFrstCond, int unScndCondOnly,
    int printed,int *arr, int *freqArr)
144 {
145     int i;
146
```

```
147     printf("\nTotal number of combinations %d to %d: %ld\n",
         N, COMBSN, combinations_count(N));
148     printf("Number of combinations not satisfying the first
        condition: %d\n", unFrstCond);
149     printf("Number of combinations not satisfying the second
         condition only: %d\n", unScndCondOnly);
150     printf("Printed combinations: %d\n\n", printed);
151
152     for (i = 0; i < N; i++)
153         printf("%d appeared %d times\n", *(arr + i), *(
        freqArr + i));
154
155 }
```

4.3 fcombinations.h

```
1  #ifndef COMBINATIONS_H
2  #define COMBINATIONS_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <stdbool.h>
7  #include <string.h>
8
9  #include "arrhandler.h"
10 #include "ccolors.h"
11
12 #define COMBSN 6
13
14 int get_n(FILE *);
15
16 void x_pair(int *, int *, FILE *);
17 void y_pair(int *, int *, FILE *);
18
19 void print_combs(int *, int, int, int, int, int);
20 void combinations(int *, int *, int *, int, int, int, int *,
       int *, int *, int, int, int, int, int);
21
22 bool even_calc(int *, int, int);
23 bool sum_comb_calc(int *, int, int);
24
25 void frequency(int *, int *, int *, int);
26 long int combinations_count(int);
27 long double factorial(int);
28 void print_other(int, int, int, int, int *, int *);
29
30 #endif
```

4.4 arrhandler.c

```c
#include "arrhandler.h"

int *fill_array(int N, FILE *dataFile)
{
    int num, i = 0;
    int *arr = (int *)malloc(N * sizeof(int));

    if (arr == NULL)
    {
        set_color(BOLD_RED);
        printf("Error! Not enough memory, exiting...\n");
        exit(EXIT_FAILURE);
        set_color(STANDARD);
    }
    else
    {
        do
        {
            fscanf(dataFile, "%d\n", &num);

            if (num >= 1 && num <= 49)
            {
                if (i == 0) { *(arr + i) = num; i++; }
                else if (!exists_in_array(arr, N, num)) { *(arr + i) = num; i++; }
            }
        } while (i < N);
    }

    return arr;
}


bool exists_in_array(int *arr, int N, int num)
{
    int *arrEnd = arr + (N - 1);
    while (arr <= arrEnd && *arr != num) arr++;
    return (arr <= arrEnd) ? true : false;
}


void quicksort(int *arr, int low, int high)
{
    if (low < high)
    {
        int partIndex = partition(arr, low, high);
        quicksort(arr, low, partIndex - 1);
        quicksort(arr, partIndex + 1, high);
    }
}
```

```
50
51
52 int partition(int *arr, int low, int high)
53 {
54     int pivot = *(arr + high);
55     int i = (low - 1), j;
56
57     for (j = low; j <= high - 1; j++)
58         if (*(arr + j) < pivot)
59             swap(arr + ++i, arr + j);
60
61     swap(arr + (i + 1), arr + high);
62     return (i + 1);
63 }
64
65
66 void swap(int *a, int *b)
67 {
68     int temp = *a;
69     *a = *b;
70     *b = temp;
71 }
72
73
74 int find_pos(int *arr, int numIter, int val)
75 {
76     int pos, i;
77
78     for (i = 0; i < numIter; i++)
79         if (val == *(arr + i))
80             pos = i;
81
82     return pos;
83 }
```

4.5 arrhandler.h

```
1 #ifndef ARRHANDLER_H
2 #define ARRHANDLER_H
3
4 #include "fcombinations.h"
5
6 int *fill_array(int, FILE *);
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 int find_pos(int *, int, int);
```

```
14
15 #endif
```

4.6 Περιγραφή υλοποιήσης

Το πρόγραμμα αυτό, επίσης όπως και τα δύο προηγούμενα προγράμματα, λειτουργεί με την ίδια λογική ακριβώς, με την διαφορά οτι όλα τα δεδομένα διαβάζονται από αρχείο, το οποίο σημαίνει ότι σε αυτό το πρόγραμμα δεν υπάρχουν όλες οι printf() - scanf() που ζητάνε από τον χρήστη να εισάγει δεδομένα, εφόσον έχουν αντικατασταθεί από την fscanf() η οποία θα διαβάσει τα δεδομένα από αρχείο.

5 minesweeper - ναρκαλιευτής

5.1 main.c

```
1  #include "minesweeper.h"
2  #include <pthread.h>
3
4  int main(int argc, char **argv)
5  {
6      main_win();
7      options_menu();
8
9      int yMax, xMax;
10     WINDOW *menuWin = menu_win(&yMax, &xMax);
11     int COLS = set_cols(menuWin, xMax);
12     int ROWS = set_rows(menuWin, yMax);
13     int NMINES = set_nmines(menuWin, COLS*ROWS);
14
15     WINDOW *gameWin = game_win(COLS, ROWS, NMINES);
16     char **dispboard = init_dispboard(gameWin, COLS, ROWS);
17     char **mineboard = init_mineboard(gameWin, COLS, ROWS,
       NMINES);
18
19     pthread_t audioThread;
20     long threadID = 1;
21     pthread_create(&audioThread, NULL, play_audio, (void *)
       threadID);
22     play_minesweeper(gameWin, dispboard, mineboard, COLS,
       ROWS, NMINES);
23
24     pthread_cancel(audioThread);
25     free(dispboard);
26     free(mineboard);
27     endwin();
28
29     return 0;
30 }
```

5.2 minesweeper.c

```c
#include "minesweeper.h"

char **init_dispboard(WINDOW *gameWin, int COLS, int ROWS)
{
    int i;
    char **dispboard = (char **)malloc(ROWS * sizeof(char *));
    for (i = 0; i < ROWS; i++)
        dispboard[i] = (char *)malloc(COLS);

    if (dispboard == NULL)
    {
        mvprintw(1, 1, "Error, not enough memory, exiting...");
        exit(EXIT_FAILURE);
    }
    else fill_dispboard(dispboard, COLS, ROWS);

    return dispboard;
}


void fill_dispboard(char **dispboard, int COLS, int ROWS)
{
    int i, j;

    for (i = 0; i < ROWS; i++)
        for (j = 0; j < COLS; j++)
            dispboard[i][j] = BLANK;
}


char **init_mineboard(WINDOW *gameWin, int COLS, int ROWS, int NMINES)
{
    int i;
    char **mineboard = (char **)malloc(ROWS * sizeof(char *));
    for (i = 0; i < ROWS; i++)
        mineboard[i] = (char *)malloc(COLS);

    if (mineboard == NULL)
    {
        mvprintw(1, 1, "Error, not enough memory, exiting...");
        exit(EXIT_FAILURE);
    }
    else
```

```
44      {
45          place_mines(mineboard, COLS, ROWS, NMINES);
46          add_adj(mineboard, COLS, ROWS);
47          fill_spaces(mineboard, COLS, ROWS, NMINES);
48      }
49
50      return mineboard;
51  }
52
53
54  void place_mines(char **mineboard, int COLS, int ROWS, int
        NMINES)
55  {
56      int i, wRand, hRand;
57
58      srand(time(NULL));
59
60      for (i = 0; i < NMINES; i++)
61      {
62          wRand = rand() % ROWS;
63          hRand = rand() % COLS;
64          mineboard[wRand][hRand] = MINE;
65      }
66  }
67
68
69  void add_adj(char **mineboard, int COLS, int ROWS)
70  {
71      int i, j;
72
73      for (i = 0; i < ROWS; i++)
74          for (j = 0; j < COLS; j++)
75              if (!is_mine(mineboard, i, j))
76                  mineboard[i][j] = adj_mines(mineboard, i, j,
         COLS, ROWS) + '0';
77  }
78
79
80  bool is_mine(char **mineboard, int row, int col)
81  {
82      return (mineboard[row][col] == MINE) ? true : false;
83  }
84
85  bool outof_bounds(int row, int col, int COLS, int ROWS)
86  {
87      return (row < 0 || row > ROWS-1 || col < 0 || col > COLS
        -1) ? true : false;
88  }
89
90
```

```
91
92 int8_t adj_mines(char **mineboard, int row, int col, int
       COLS, int ROWS)
93 {
94     int8_t numAdj = 0;
95
96     if (!outof_bounds(row, col - 1, COLS, ROWS)       &&
       mineboard[row][col-1]    == MINE) numAdj++; // North
97     if (!outof_bounds(row, col + 1, COLS, ROWS)       &&
       mineboard[row][col+1]    == MINE) numAdj++; // South
98     if (!outof_bounds(row + 1, col, COLS, ROWS)       &&
       mineboard[row+1][col]    == MINE) numAdj++; // East
99     if (!outof_bounds(row - 1, col, COLS, ROWS)       &&
       mineboard[row-1][col]    == MINE) numAdj++; // West
100    if (!outof_bounds(row + 1, col - 1, COLS, ROWS)  &&
       mineboard[row+1][col-1]  == MINE) numAdj++; // North-East
101    if (!outof_bounds(row - 1, col - 1, COLS, ROWS)  &&
       mineboard[row-1][col-1]  == MINE) numAdj++; // North-West
102    if (!outof_bounds(row + 1, col + 1, COLS, ROWS)  &&
       mineboard[row+1][col+1]  == MINE) numAdj++; // South-East
103    if (!outof_bounds(row - 1, col + 1, COLS, ROWS)  &&
       mineboard[row-1][col+1]  == MINE) numAdj++; // South-West
104
105    return numAdj;
106 }
107
108
109 void fill_spaces(char **mineboard, int COLS, int ROWS, int
       NMINES)
110 {
111     int i, j;
112
113     for (i = 0; i < ROWS; i++)
114         for (j = 0; j < COLS; j++)
115             if (mineboard[i][j] != MINE && mineboard[i][j] =
       = '0')
116                 mineboard[i][j] = '-';
117 }
```

5.3 minesweeper.h

```
1 #ifndef MINESWEEPER_H
2 #define MINESWEEPER_H
3
4 #if defined linux || defined __unix__
5 #include <ncurses.h>
6 #endif
7
8 #include <stdlib.h>
9 #include <string.h>
```

```
#include <time.h>

#include "settings.h"
#include "gameplay.h"
#include "navigation.h"
#include "outputs.h"
#include "wins.h"
#include "audio.h"

#define BLANK ' '
#define MINE '*'
#define CLEAR "                                    "
#define ENTER '\n'
#define OPEN_LOWER 'o'
#define OPEN_UPPER 'O'
#define FLAG 'F'
#define FLAG_LOWER 'f'
#define FLAG_UPPER 'F'
#define DEFUSE_LOWER 'g'
#define DEFUSE_UPPER 'G'
#define DEFUSED 'D'
#define PAUSE_AUDIO 'p'
#define VOLUME_UP '+'
#define VOLUME_DOWN '-'
#define QUIT 'q'

char **init_dispboard(struct _win_st*, int, int);
void fill_dispboard(char **, int, int);
char **init_mineboard(struct _win_st*, int, int, int);

void place_mines(char **, int, int, int);
void add_adj(char **, int, int);
bool is_mine(char **, int, int);
bool outof_bounds(int, int, int, int);
int8_t adj_mines(char **, int, int, int, int);
void fill_spaces(char **, int, int, int);

#endif
```

5.4 gameplay.c

```
#include "gameplay.h"

void play_minesweeper(WINDOW *gameWin, char **dispboard,
    char **mineboard, int COLS, int ROWS, int NMINES)
{
    int mboardXLoc = 0, mboardYLoc = 0;
    bool gameOver = false, cantFlag = false;
    int numDefused = 0;
    int yMax, xMax, yMiddle, xMiddle;
```

```
9       char move;
10      getmaxyx(stdscr, yMax, xMax);
11      yMiddle = yMax / 2;
12      xMiddle = xMax / 2;
13
14      print_board(gameWin, dispboard, COLS, ROWS);
15
16      do
17      {
18          navigate(gameWin, dispboard, &move, &mboardXLoc, &
    mboardYLoc);
19
20          if (move == ENTER || move == OPEN_LOWER || move ==
    OPEN_UPPER) // handle cell opening
21          {
22              transfer(dispboard, mineboard, mboardYLoc,
    mboardXLoc);
23              reveal(gameWin, dispboard, mboardYLoc,
    mboardXLoc, mboardYLoc + 1, 3*mboardXLoc + 2);
24              cantFlag = true;
25              if (dispboard[mboardYLoc][mboardXLoc] == MINE)
    gameOver = true;
26          }
27          else if (move == FLAG_LOWER || move == FLAG_UPPER)
    // handle falgs
28          {
29              if (dispboard[mboardYLoc][mboardXLoc] == FLAG)
    dispboard[mboardYLoc][mboardXLoc] = BLANK; // undo flag
30              else if (dispboard[mboardYLoc][mboardXLoc] !=
    FLAG && dispboard[mboardYLoc][mboardXLoc] != BLANK)
    continue; // dont flag an already opened mine
31              else dispboard[mboardYLoc][mboardXLoc] = FLAG;
    // flag if not flagged already
32              reveal(gameWin, dispboard, mboardYLoc,
    mboardXLoc, mboardYLoc + 1, 3*mboardXLoc + 2);
33          }
34          else if (move == DEFUSE_LOWER || move ==
    DEFUSE_UPPER) // check for defuse
35          {
36              if (dispboard[mboardYLoc][mboardXLoc] == FLAG &&
     mineboard[mboardYLoc][mboardXLoc] == MINE) // is_mine
    replace
37              {
38                  numDefused++;
39                  refresh();
40                  dispboard[mboardYLoc][mboardXLoc] =
    mineboard[mboardYLoc][mboardXLoc] = DEFUSED;
41                  reveal(gameWin, dispboard, mboardYLoc,
    mboardXLoc, mboardYLoc + 1, 3*mboardXLoc + 2);
42              }
```

```
43              else if (dispboard[mboardYLoc][mboardXLoc] ==
    FLAG && mineboard[mboardYLoc][mboardXLoc] != MINE)
    gameOver = true; // handle false defusal
44          }
45          else if (move == PAUSE_AUDIO) pause_audio(); //
    handle audio
46          else if (move == VOLUME_UP || move == VOLUME_DOWN)
    volume(move);
47
48          mvprintw(1, xMiddle-8, "Defused mines: %d/%d",
    numDefused, NMINES);
49
50      } while (((mboardYLoc >= 0 && mboardYLoc < ROWS) && (
    mboardXLoc >= 0 && mboardXLoc < COLS)) &&
51               numDefused < NMINES && !gameOver && move !=
    QUIT);
52
53      if (gameOver == true)
54      {
55          game_over(gameWin, mineboard, yMiddle, xMiddle);
56          getchar();
57          print_board(gameWin, mineboard, COLS, ROWS);
58          filewrite(mineboard, COLS, ROWS, mboardXLoc,
    mboardYLoc, "lost");
59      }
60
61      if (numDefused == NMINES)
62      {
63          game_won(gameWin, yMiddle, xMiddle);
64          getchar();
65          filewrite(mineboard, COLS, ROWS, mboardXLoc,
    mboardYLoc, "won");
66      }
67  }
68
69
70  void transfer(char **dispboard, char **mineboard, int
    mboardYLoc, int mboardXLoc)
71  {
72      dispboard[mboardYLoc][mboardXLoc] = mineboard[mboardYLoc
    ][mboardXLoc];
73  }
74
75
76  void reveal(WINDOW *gameWin, char **dispboard, int
    mboardYLoc, int mboardXLoc, int yLoc, int xLoc)
77  {
78      mvwaddch(gameWin, yLoc, xLoc, dispboard[mboardYLoc][
    mboardXLoc]);
79      wrefresh(gameWin);
```

```
80 }
81
82
83 void flag_handler()
84 {
85
86 }
87
88
89 bool is_flagged()
90 {
91
92 }
93
94
95 bool is_defused(char **dispboard, char **mineboard, int
       mboardYLoc, int mboardXLoc)
96 {
97     return ((dispboard[mboardYLoc][mboardXLoc] == DEFUSED))
       ? true : false;
98 }
```

5.5 gameplay.h

```
1 #ifndef GAMEPLAY_H
2 #define GAMEPLAY_H
3
4 #include "minesweeper.h"
5
6 void play_minesweeper(struct _win_st*, char **, char **, int
      , int, int);
7 void transfer(char **, char **, int, int);
8 void reveal(struct _win_st*, char **, int, int, int, int);
9 void flag_handler();
10 bool is_flagged();
11 bool is_defused(char **, char **, int, int);
12
13 #endif
```

5.6 navigation.c

```
1 #include "navigation.h"
2
3 void navigate(WINDOW *gameWin, char **mineboard, char *move,
       int *mboardXLoc, int *mboardYLoc)
4 {
5     int yMax, xMax;
6     static int yLoc = 1, xLoc = 2;
7     getmaxyx(gameWin, yMax, xMax);
8
```

```
9       update_curs(gameWin, yLoc, xLoc);
10      *mboardXLoc = (xLoc-2)/3;
11      *mboardYLoc = yLoc-1;
12      mvprintw(1, 1, "Current position: (%d, %d) ", *
        mboardXLoc, *mboardYLoc);
13      refresh();
14      getmv(gameWin, move, &yLoc, &xLoc, yMax, xMax);
15  }
16
17
18  void getmv(WINDOW *gameWin, char *move, int *yLoc, int *xLoc
        , int yMax, int xMax)
19  {
20      *move = wgetch(gameWin);
21      switch (*move) // vim keys support!!
22      {
23          case 'k': case 'K':
24          case 'w': case 'W':
25              mvup(yLoc);
26              break;
27          case 'j': case 'J':
28          case 's': case 'S':
29              mvdown(yLoc, yMax);
30              break;
31          case 'h': case 'H':
32          case 'a': case 'A':
33              mvleft(xLoc);
34              break;
35          case 'l': case 'L':
36          case 'd': case 'D':
37              mvright(xLoc, xMax);
38              break;
39      }
40  }
41
42
43  void mvup(int *yLoc)
44  {
45      (*yLoc)--;
46      if (*yLoc < 1) *yLoc = 1;
47  }
48
49
50  void mvdown(int *yLoc, int yMax)
51  {
52      (*yLoc)++;
53      if (*yLoc > yMax-2) *yLoc = yMax-2;
54  }
55
56
```

```
57 void mvleft(int *xLoc)
58 {
59     *xLoc -= 3;
60     if (*xLoc < 2) *xLoc = 2;
61 }
62
63
64 void mvright(int *xLoc, int xMax)
65 {
66     *xLoc += 3;
67     if (*xLoc > xMax-3) *xLoc = xMax-3;
68 }
69
70
71 void update_curs(WINDOW *gameWin, int yLoc, int xLoc)
72 {
73     wmove(gameWin, yLoc, xLoc);
74 }
```

5.7 navigation.h

```
1 #ifndef NAVIGATION_H
2 #define NAVIGATION_H
3
4 #include "minesweeper.h"
5
6 void navigate(struct _win_st*, char **, char *, int *, int *
      );
7 void getmv(struct _win_st*, char *, int *, int *, int, int);
8 void mvup(int *);
9 void mvdown(int *, int);
10 void mvleft(int *);
11 void mvright(int *, int);
12 void update_curs(struct _win_st*, int, int);
13
14 #endif
```

5.8 settings.c

```
1 #include "settings.h"
2
3 int set_cols(WINDOW *menuWin, int xMax)
4 {
5     int COLS;
6
7     do
8     {
9         mvwprintw(menuWin, 1, 1, "Columns (Min = 5, Max = %d
      ): ", (xMax-2)/3 - 2);
10         wrefresh(menuWin);
```

```
11          scanw("%d", &COLS);
12          mvwprintw(menuWin, 1, COLS_CHAR_LENGTH, "%d", COLS);
13          wrefresh(menuWin);
14      } while (COLS < 5 || COLS > (xMax-2)/3 - 2);
15
16      return COLS;
17  }
18
19
20  int set_rows(WINDOW *menuWin, int yMax)
21  {
22      int ROWS;
23
24      do
25      {
26          mvwprintw(menuWin, 2, 1, "Rows (Min = 5, Max = %d):
        ", yMax-14);
27          wrefresh(menuWin);
28          scanw("%d", &ROWS);
29          mvwprintw(menuWin, 2, ROWS_CHAR_LENGTH, "%d", ROWS);
30          wrefresh(menuWin);
31      } while (ROWS < 5 || ROWS > yMax - 14);
32
33      return ROWS;
34  }
35
36
37  int set_nmines(WINDOW *menuWin, int DIMENSIONS)
38  {
39      int NMINES;
40
41      do
42      {
43          mvwprintw(menuWin, 3, 1, "Mines (Max = %d): ",
        DIMENSIONS-15); // -10 so the player has a chance to win
44          wrefresh(menuWin);
45          scanw("%d", &NMINES);
46          mvwprintw(menuWin, 3, NMINES_CHAR_LENGTH, "%d",
        NMINES);
47          wrefresh(menuWin);
48      } while (NMINES < 1 || NMINES > DIMENSIONS-15);
49
50      return NMINES;
51  }
```

5.9 settings.h

```
1  #ifndef SETTINGS_H
2  #define SETTINGS_H
3
```

```
4 #include "minesweeper.h"
5
6 #define COLS_CHAR_LENGTH strlen("Columns (Min = 5, Max = XXX
      ): ") + 1
7 #define ROWS_CHAR_LENGTH strlen("Rows (Min = 5, Max = YYY):
      ") + 1
8 #define NMINES_CHAR_LENGTH strlen("Mines (Max = MMM): ") + 1
9
10 int set_cols(struct _win_st*, int);
11 int set_rows(struct _win_st*, int);
12 int set_nmines(struct _win_st*, int);
13 void init_colors();
14
15 #endif
```

5.10 outputs.c

```
1 #include "outputs.h"
2
3 void print_board(WINDOW *gameWin, char **board, int COLS,
      int ROWS)
4 {
5     int i, j, x, y = 1;
6
7     print_grid(gameWin, ROWS, COLS);
8
9     for (i = 0; i < ROWS; i++)
10     {
11         x = 2;
12         for (j = 0; j < COLS; j++)
13         {
14             wattron(gameWin, A_BOLD);
15             mvwaddch(gameWin, y, x, board[i][j]);
16             x += 3;
17         }
18         y++;
19     }
20
21     wrefresh(gameWin);
22     wattron(gameWin, A_BOLD);
23 }
24
25
26 void print_grid(WINDOW *gameWin, int ROWS, int COLS)
27 {
28     int i, j;
29
30     for (i = 1; i <= ROWS; i++)
31     {
32         wmove(gameWin, i, 1);
```

```
33          for (j = 0; j < COLS; j++)
34              wprintw(gameWin, "[ ]");
35      }
36
37      wrefresh(gameWin);
38  }
39
40
41  void filewrite(char **mineboard, int COLS, int ROWS, int
        hitRow, int hitCol, const char *status)
42  {
43      int i, j;
44      FILE *mnsOut = fopen("txt/mnsout.txt", "w");
45
46      if (mnsOut == NULL)
47      {
48          mvprintw(1, 1, "Error opening file, exiting...");
49          exit(EXIT_FAILURE);
50      }
51      else
52      {
53          strcmp(status, "won")
54              ? fprintf(mnsOut, "Mine hit at position (%d, %d)
        \n\n", hitRow+1, hitCol+1)
55              : fprintf(mnsOut, "Last mine defused at position
         (%d, %d)\n\n", hitRow+1, hitCol+1);
56          fprintf(mnsOut, "Board overview\n\n");
57
58          for (i = 0; i < ROWS; i++)
59          {
60              for (j = 0; j < COLS; j++)
61                  fprintf(mnsOut, "%c ", mineboard[i][j]);
62              fprintf(mnsOut, "\n");
63          }
64
65          mvprintw(1, 1, "Session written to file       ");
66          refresh();
67          getchar();
68      }
69
70      fclose(mnsOut);
71  }
72
73
74  void game_won(WINDOW *gameWin, int yMiddle, int xMiddle)
75  {
76      wclear(gameWin);
77      wrefresh(gameWin);
78      wattron(stdscr, A_BOLD);
79      mvwprintw(stdscr, yMiddle-2, xMiddle-11, "You defused
```

```
    all the mines!");
80      mvwprintw(stdscr, yMiddle-1, xMiddle-3, "You won :)");
81      mvwprintw(stdscr, yMiddle, xMiddle-11, "Press any key to
         continue");
82      refresh();
83      wattroff(stdscr, A_BOLD);
84  }
85
86
87  void game_over(WINDOW *gameWin, char **mineboard, int
        yMiddle, int xMiddle)
88  {
89      wclear(gameWin);
90      wrefresh(gameWin);
91      wattron(stdscr, A_BOLD);
92      mvwprintw(stdscr, yMiddle-2, xMiddle-24, "You hit a mine
        ! (or tried to defuse the wrong cell)");
93      mvwprintw(stdscr, yMiddle-1, xMiddle-4, "Game over :(");
94      mvwprintw(stdscr, yMiddle, xMiddle-11, "Press any key to
         continue");
95      refresh();
96      wattroff(stdscr, A_BOLD);
97  }
```

5.11 outputs.h

```
1  #ifndef OUTPUTS_H
2  #define OUTPUTS_H
3
4  #include "minesweeper.h"
5
6  void print_grid(struct _win_st*, int, int);
7  void print_board(struct _win_st*, char **, int, int);
8  void game_won(struct _win_st*, int, int);
9  void game_over(struct _win_st*, char **, int, int);
10 void filewrite(char **, int, int, int, int, const char *);
11
12 #endif
```

5.12 wins.c

```
1  #include "wins.h"
2
3  void main_win()
4  {
5      initscr();
6      noecho();
7      cbreak();
8
9      WINDOW *mainWin = newwin(0, 0, 0, 0);
```

```
10      wattron(mainWin, A_BOLD);
11      box(mainWin, 0, 0);
12      refresh();
13      wrefresh(mainWin);
14      wattroff(mainWin, A_BOLD);
15  }
16
17
18  WINDOW *menu_win(int *yMax, int *xMax)
19  {
20      int numSettings = 3;
21      getmaxyx(stdscr, *yMax, *xMax);
22      WINDOW *menuWin = newwin(numSettings+2, *xMax-8, *yMax-8
        , 4);
23      wattron(menuWin, A_BOLD);
24      box(menuWin, 0, 0);
25      wrefresh(menuWin);
26      wattroff(menuWin, A_BOLD);
27      return menuWin;
28  }
29
30
31  WINDOW *game_win(int COLS, int ROWS, int NMINES)
32  {
33      int winRows = ROWS + 2;
34      int winCols = COLS*3 + 2;
35      WINDOW *gameWin = newwin(winRows, winCols, 2, 4);
36      wattron(gameWin, A_BOLD);
37      box(gameWin, 0, 0);
38      wrefresh(gameWin);
39      wattroff(gameWin, A_BOLD);
40      return gameWin;
41  }
42
43
44  void options_menu()
45  {
46      int yMax = getmaxy(stdscr);
47      mvprintw(yMax-3, 5, "q Quit           w/k Move up      s/j
         Move down       a/h Move Left       d/l Move Right
        [ENTER]/o Open cell");
48      mvprintw(yMax-2, 5, "f Flag cell      g Defuse (if
        flagged only)          p Pause music       + Volume up
              - Volume down");
49      refresh();
50  }
```

5.13 wins.h

```
1  #ifndef WINS_H
```

```
#define WINS_H

#include "minesweeper.h"

void main_win();
WINDOW *menu_win(int *, int *);
WINDOW *game_win(int, int, int);
void options_menu();
void options_win();

#endif
```

5.14 audio.c

```
#include "audio.h"

void *play_audio(void *threadID)
{
    int tID = (long)threadID;
    Mix_Music *music = NULL;
    SDL_Init(SDL_INIT_AUDIO);
    Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 1, 4096);
    music = Mix_LoadMUS(AUDIO_PATH);

    Mix_PlayMusic(music, -1);
    while (Mix_PlayingMusic()) ;
    Mix_FreeMusic(music);
    Mix_CloseAudio();
}


void volume(char option)
{
    static int volume = MIX_MAX_VOLUME;

    switch (option)
    {
        case '+':
            if (volume == MIX_MAX_VOLUME) break;
            else Mix_VolumeMusic(volume += 10);
            break;
        case '-':
            if (volume == 0) break;
            else Mix_VolumeMusic(volume -= 10);
            break;
    }
}


void pause_audio()
```

```
37 {
38     if(Mix_PausedMusic() == 1) Mix_ResumeMusic();
39     else Mix_PauseMusic();
40 }
```

5.15 audio.h

```
1  #ifndef AUDIO_H
2  #define AUDIO_H
3
4  #include <SDL2/SDL.h>
5  #include <SDL2/SDL_mixer.h>
6  #include "minesweeper.h"
7
8  #define AUDIO_PATH "audio/detective-8bit.wav"
9
10 void *play_audio(void *);
11 void volume(char);
12 void pause_audio();
13
14 #endif
```

5.16 Περιγραφή υλοποιήσης

Ο ναρκαλιευτής αυτός χρησιμοποιεί ώς βασική βιβλιοθήκη την ncurses και είναι δομημένος ως εξής: Από το main.c καλούνται αρχικά οι συναρτήσεις δημιουργίας των παραθύρων που θα εμφανιστούν στην οθόνη και στην συνέχεια καλούνται οι συναρτήσεις δημιουργίας των πινάκων $M \times N$, για το ναρκοπέδιο και για τον πίνακα που έχει "κρυμμένα" τα κελιά αντίστοιχα. Τέλος από την main καλείται η συνάρτηση που θα ξεκινήσει το παιχνίδι.

Οι συναρτήσεις για τον ορισμό στηλών, γραμμών, και αριθμό των ναρκών βρίσκονται στο settings.c.

Στο minesweeper.c εκτελούνται όλες οι συναρτήσεις δημιουργίας πινάκων, τοποθέτησεις ναρκών, μέτρημα των βομβών στα γειτονικά κελιά, καθώς και γέμισμα των κενών θέσεων τους.

Έπειτα, στο gameplay.c εκτελείται το παιχνίδι - αρχικά τυπώνεται ο πίνακας και το περίγραμμα που υπάρχει ανάμεσα σε κάθε κελί ώστε να είναι πιο εμφανίσιμο και πιο εύχρηστο το παιχνίδι. Προκειμένου τα κελιά να τοποθετηθούν στις κατάλληλες θέσεις στον πίνακα, δηλαδή να είναι ανάμεσα στα [ ], τα στοιχεία των πινάκων τοποθετούνται κάθε φορά με απόσταση δύο χαρακτήρων στον κάθετο άξονα και τριών χαρακτήρων στον οριζόντιο, το ένα από το άλλο. Με αυτά τα δύο νούμερα προκύπτουν και τέσσερις τύποι, οι οποίοι βοηθάνε στην σωστή προσπέλαση των στοιχείων των πινάκων κατά την διάρκεια του παιχνιδιού, και στον υπολογισμό των διαστάσεων του παραθύρου που εμφανίζεται το πεδίο. Οι τύποι είναι οι εξής

$$x_{win} = 3cols + 2 \tag{1}$$

$$x_{board} = \frac{(x_{win} - 2)}{3} \tag{2}$$

$$y_{win} = rows + 2 \tag{3}$$

$$y_{board} = y_{win} - 2 \tag{4}$$

Οι τύποι (2) και (4) μετατρέπουν την θέση του κέρσορα στην οθόνη σε θέσεις πίνακα. Αφού τυπωθεί στην οθόνη ο πίνακας με κρυμμένα τα στοιχεία του, *ο οποίος είναι στην ουσία ένας $M \times N$ πίνακας γεμισμένος με κενά αρχικά*, ξεκινάει το βασικό loop του παιχνιδιού, στο οποίο ο χρήστης μετακινείται από κελί σε κελί, επιλέγει την κίνηση που θέλει να κάνει πάνω σε κάθε κελί, και είτε χάνει είτε νικάει. Προκειμένου να λειτουργήσει κάτι τέτοιο, μέσα στο loop γίνονται οι εξής λειτουργίες: Αρχικά ο κέρσορας μετακίνεται κάθε φορά που και ο χρήστης μετακινείται ώστε να μπορεί να δει σε ποιο κελί βρίσκεται κάθε στιγμή, και στην συνέχεια, μπορούν να εκτελεστούν σε κάθε κελί οι εξής λειτουργίες:

- Άνοιγμα κελιού
- Flag ένα κελί
- Εξουδετέρωση νάρκης (μόνο αν το κελί είναι ήδη flagged)

Σε περίπτωση όμως που ο χρήστης κάνει flag ένα κελί που δεν περιέχει νάρκη, και προσπαθήσει να βγάλει την νάρκη, θα χάσει. Οπότε το παιχνίδι νικιέται *μόνο* όταν ο χρήστης βγάλει όλες τις νάρκες από το πεδίο, και όχι όταν ανοίξει όλα τα κελιά που δεν έχουνε νάρκες, όπως συμβαίνει στον ναρκαλιευτή των Windows.

Συνοπτικά, τα υπόλοιπα αρχεία χειρίζονται ορισμένες λειτουργίες του παιχνιδιού, όπως την κίνηση από κελί σε κελί, τον χειρισμό των εμφανίσεων διαφόρων μηνυμάτων στην οθόνη, και τον ήχο. Συγκεκριμένα για τον ήχο χρησιμοποίησα την βιβλιοθήκη SDL2 σε συνδυασμό με την βιβλιοθήκη PThread ώστε να μπορούν να εκτελούνται *"ταυτόχρονα"* και οι συναρτήσεις χειρισμού ήχου, και το υπόλοιπο παιχνίδι.

6 Διευκρινήσεις

Αρχικά, λόγω του ότι το πρόγραμμα περιέχει πολλές μεταβλητές και συναρτήσεις θεώρησα καλύτερο να εστιάσω στην λειτουργία του προγράμματος γενικότερα και όχι τόσο στο τι συμβολίζει η κάθε μεταβλητή/συνάρτηση, το οποίο θα μπορούσε να γίνει αρκετά κουραστικό. Τα αρχεία και οι συναρτήσεις είναι χωρισμένα έτσι ώστε να εξηγούν από το όνομα τους μόνο ακριβώς ποιες λειτουργίες εκτελούνται μέσα σε αυτά ώστε να είναι πιο εύκολη η μετέπειτα κατανόηση, συντήρηση ή βελτίωση του προγράμματος. Βέβαια, κατανοώ ότι αυτή η προσέγγιση μπορεί να θεωρηθεί πολύ συνοπτική ή και πρόχειρη.

Επίσης, η συγγραφή αυτή τη φορά έγινε στο LaTeX, λόγω του ότι κάνει αυτόματη στοίχηση, χρωματισμό στους κώδικες, και γενικώς πολύ πιο πρακτικό τον χειρισμό μεγάλων εγγράφων, αλλά δεν κατάφερα δυστυχώς να βρω τρόπο να αλλάξω την ελληνική γραμματοσειρά σε κάποια πιο εύκολη στο διάβασμα.

Τέλος, όσο αφορά τον ναρκαλιευτή, θα ήθελα να σημειώσω οτι έχει πολύ περιθώριο για βελτιώση (και λίγο καθαρισμό στον κώδικα), και πολλές λειτουργίες, όπως ο ήχος και το multithreading, τις έβαλα περισσότερο πειραματικά, οπότε

είναι αρκετά πιθανό να περιέχουν τυχόν κακές πρακτικές ή και λάθη, αλλά θεώρησα ότι ήταν μία καλή ευκαιρία για πειραματισμό. Ένα πρόβλημα το οποίο δεν κατάφερα να λύσω ήταν το να κάνει αυτόματο resize το παράθυρο σε περίπτωση που αλλάξει το μέγεθος του terminal. Μία λειτουργία που θα βάλω στο μέλλον επίσης είναι να δίνεται η επιλογή στον χρήστη να ξαναπαίξει αν θέλει, πιθανώς πατώντας το κουμπί r (restart), καθώς και να βάλω χρώματα στους αριθμούς, ώστε να είναι πιο ευδιάκριτοι και να είναι πιο ξεκούραστο στο μάτι το πρόγραμμα.

7 Εργαλεία

- Editors: Visual Studio Code, NVim
- Compiler: gcc
- Βιβλιοθήκες: SDL2, PThread, ncurses
- Shell: zsh
- OS: Arch Linux
- Συγγραφή: LaTeX