ΕΡΓΑΣΤΗΡΙΟ 8:
Πολυδιάστατοι Πίνακες και Δυναμική Δέσμευση Μνήμης

Στο εργαστήριο αυτό θα μελετήσουμε τον τρόπο με τον οποίο ορίζουμε στην C πολυδιάστατους πίνακες και θα δούμε πώς μπορούμε να δεσμεύουμε δυναμικά μνήμη για να δημιουργούμε πίνακες, όταν δεν γνωρίζουμε κατά τη φάση συγγραφής του προγράμματος τις διαστάσεις των πινάκων που χρειαζόμαστε.

Άσκηση 1: Διδιάστατοι πίνακες

1.1 Κατασκευάστε το πρόγραμμα `twodim.c` που ορίζει στατικά έναν πίνακα `A` διαστάσεων 6 x 10 και αρχικοποιεί το στοιχείο `A[i][j]` που βρίσκεται στη γραμμή `i` και στη στήλη `j`, σύμφωνα με τον τύπο:

$$A[i][j] = i(5-i) + j(9-j)$$

Εκτυπώστε τον πίνακα κατά γραμμές, με τα στοιχεία κάθε γραμμής χωρισμένα με στηλογνώμονα (tab).

1.2 Επεκτείνετε το πρόγραμμα `twodim.c` ώστε να εκτυπώνει τον ανάστροφο του πίνακα, δηλαδή αυτόν που έχει στήλες τις γραμμές του αρχικού και γραμμές τις στήλες του αρχικού.

Ο αρχικός πίνακας:	Ο ανάστροφος πίνακας:
1 4 6 7 8 5 6 7 1 3 1 0 4 7 2	1 5 1 4 6 0 6 7 4 7 1 7 8 3 2

1.3 Επεκτείνετε το πρόγραμμα `twodim.c` ώστε να εκτυπώνει τις γραμμές του αρχικού πίνακα με αντίστροφη σειρά.

Ο αρχικός πίνακας:	Εκτύπωση γραμμών με αντίστροφη σειρά:
1 4 6 7 8 5 6 7 1 3 1 0 4 7 2	8 7 6 4 1 3 1 7 6 5 2 7 4 0 1

1.4 Επεκτείνετε το πρόγραμμα `twodim.c` ώστε να εκτυπώνει όλα τα στοιχεία του πίνακα σε μία σειρά, διασχίζοντάς τον με ένα «φιδοειδή» τρόπο, όπως φαίνεται στο σχήμα:

Ο αρχικός πίνακας:	«Φιδοειδής» εκτύπωση των στοιχείων:
1 4 6 7 8 5 6 7 1 3 1 0 4 7 2	1 4 6 7 8 3 1 7 6 5 1 0 4 7 2

Άσκηση 2: Δυναμική δέσμευση μνήμης για μονοδιάστατο πίνακα

2.1 Κατασκευάστε το πρόγραμμα `array.c` που να διαβάζει από την είσοδο τη διάσταση ενός μονοδιάστατου πίνακα ακεραίων (έστω `N`), να δεσμεύει χώρο `N` θέσεων δυναμικά και έπειτα να τον αρχικοποιεί διαβάζοντας από την είσοδο ακέραιους αριθμούς.

Δυναμική δέσμευση μνήμης για μονοδιάστατο πίνακα.

Συνάρτηση `void *malloc(unsigned int size)`

Χρήση:
```
ΤΔ *p;                        //Δηλωση ενος δεικτη σε στοιχεια τυπου ΤΔ
p = malloc(N * sizeof(ΤΔ)); //Δεσμευση μνημης για Ν στοιχεια τυπου ΤΔ
```

Η `malloc` επιστρέφει `NULL` σε περίπτωση αποτυχίας δέσμευσης της αιτούμενης μνήμης.

Αποδέσμευση μνήμης για δυναμικά δεσμευμένους πίνακες.

Συνάρτηση `void free(void *p)`

Χρήση: `free(p);`
όπου `p` είναι δείκτης σε θέσεις μνήμης που έχουν δεσμευθεί δυναμικά.

Όταν χρησιμοποιούνται οι συναρτήσεις `malloc()` και `free()`, πρέπει να γίνεται συμπερίληψη του αρχείου επικεφαλίδας `stdlib.h`.

Εκτυπώστε τα στοιχεία του πίνακα σε μία γραμμή χωρισμένα με στηλογνώμονα (tab).

2.2 Κατασκευάστε το αρχείο `input.txt` το οποίο να περιέχει `N+1` ακέραιους αριθμούς ως εξής: Ο πρώτος αριθμός είναι το πλήθος των στοιχείων (`N`) και ακολουθούν οι `N` ακέραιοι χωρισμένοι με κενά.

2.3 Εκτελέστε το πρόγραμμα `array` με ανακατεύθυνση εισόδου από το αρχείο `input.txt`.

2.4 Επεκτείνετε το πρόγραμμά σας, ώστε να εκτυπώνεται ο μέσος όρος των στοιχείων του πίνακα.

Άσκηση 3: Δυναμική δέσμευση μνήμης για διδιάστατο πίνακα

3.1 Κατασκευάστε το πρόγραμμα `mines.c` που να διαβάζει από την είσοδο τις διαστάσεις ενός διδιάστατου πίνακα χαρακτήρων (έστω `N` x `M`), να δεσμεύει χώρο `N` x `M` θέσεων δυναμικά και έπειτα να τον αρχικοποιεί διαβάζοντας από την είσοδο χαρακτήρες.

Δυναμική δέσμευση μνήμης για διδιάστατο πίνακα διάστασης `N` x `M`

```
ΤΔ **p;
P = malloc(N * sizeof(ΤΔ *));
for (i = 0 ; i < N ; i++)
   p[i] = malloc(M * sizeof(ΤΔ));
```

Αποδέσμευση μνήμης για δυναμικά δεσμευμένο πίνακα `N` x `M`

```
for (i = 0 ; i < N ; i++)
   free(p[i]);
free(p);
```

3.2 Κατασκευάστε το αρχείο `mines.txt` που αναπαριστά τα δεδομένα ενός ναρκοπεδίου. Συγκεκριμένα, στην πρώτη γραμμή του αρχείου υπάρχουν οι διαστάσεις του ναρκοπεδίου (`N` και `M`) και μετά ακολουθούν γραμμή-γραμμή τα περιεχόμενα των κελιών, που είναι ο χαρακτήρας `'.'` όταν δεν υπάρχει νάρκη και ο χαρακτήρας `'*'` όταν υπάρχει νάρκη.

```
3 4
..*.
.**.
*.*.
```

3.3 Επεκτείνετε το πρόγραμμά σας, ώστε να εκτυπώνονται τα περιεχόμενα του πίνακα που διαβάστηκε. Εκτελέστε το πρόγραμμα σας με ανακατεύθυνση εισόδου από το αρχείο `mines.txt`.

3.4 Επεκτείνετε το πρόγραμμα σας, ώστε να εκτυπώνεται μια τροποποιημένη μορφή του ναρκοπεδίου, στην οποία, στα κελιά που υπάρχει νάρκη να εμφανίζεται πάλι το `'*'`, ενώ στα κελιά που δεν υπάρχει νάρκη να φαίνεται ένας αριθμός που δείχνει σε πόσα γειτονικά κελιά υπάρχει νάρκη. Σαν γειτονικά θεωρούνται όχι μόνο συνεχόμενα οριζόντια ή κατακόρυφα κελιά, αλλά και συνεχόμενα σε διαγώνια κατεύθυνση. Για παράδειγμα, για το ναρκοπέδιο που είδαμε, θα πρέπει να εμφανίζεται η έξοδος:

```
13*2
2**3
*4*2
```

ΠΑΡΑΡΤΗΜΑ: Αποσφαλμάτωση προγραμμάτων (Πράξη 3η)

Στα εργαστήρια 3 και 4 είχαμε αναφερθεί στα συντακτικά και λογικά λάθη που μπορεί να έχουν τα προγράμματά μας. Στον προγραμματισμό υπάρχει και ένα ακόμα σημαντικό είδος σφαλμάτων, τα λάθη διαχείρισης μνήμης. Η ύπαρξή τους στη γλώσσα προγραμματισμού C οφείλεται, σε μεγάλο βαθμό, στην ελευθερία που δίνεται στον προγραμματιστή μέσω των δεικτών, ένα πολύ ισχυρό εργαλείο, που όμως πρέπει να χειριζόμαστε με προσοχή. Στο σημερινό εργαστήριο θα δούμε τι εννοούμε όταν αναφερόμαστε σε σφάλματα διαχείρισης μνήμης, καθώς και πώς μπορούμε να τα ανιχνεύουμε και να τα διορθώνουμε με τη βοήθεια του debugger gdb, που συνήθως είναι εγκατεστημένος σε συστήματα Unix/Linux.

Σφάλματα διαχείρισης μνήμης

Ένα σφάλμα διαχείρισης μνήμης συνήθως το συνδέουμε στο μυαλό μας με την εμφάνιση του μηνύματος "`Segmentation Fault`" (ή ενός παραθύρου για το κλείσιμο του προγράμματος αν το τρέχουμε μέσα από το Dev C++). Γενικά, τα σφάλματα διαχείρισης μνήμης έχουν να κάνουν με το λανθασμένο χειρισμό κάποιας περιοχής της μνήμης, που ενώ νομίζουμε ότι περιέχει κάτι (π.χ. έναν πίνακα) και προσπελαύνουμε αυτές τις θέσεις μνήμης σαν αυτό το κάτι να ήταν εκεί, τελικά δεν υπάρχει αυτό που νομίζαμε.

Το πιο συνηθισμένο σφάλμα διαχείρισης μνήμης είναι να προσπελάσουμε το περιεχόμενο ενός δείκτη, χωρίς αυτός να δείχνει σε δεσμευμένες θέσεις μνήμης ή να προσπελάσουμε θέσεις μνήμης πέρα απ' αυτές που έχουμε δεσμεύσει. Ας δούμε κάποια σχετικά παραδείγματα.

```
#include <stdio.h>

int main(void) {
    int *p;
    *p = 10;
    return 0;
}
```

Στο προηγούμενο παράδειγμα, προσπαθούμε να βάλουμε στις θέσεις μνήμης που δείχνει ο `p` τον αριθμό 10. Όμως, δεν έχουμε δεσμεύσει χώρο, στον οποίο θα δείχνει το `p`, ικανό να χωρέσει έναν ακέραιο. Άρα, το προηγούμενο παράδειγμα, κατά πάσα πιθανότητα, θα κάνει segmentation fault.

```
#include <stdio.h>

int main(void) {
    int array[10];
    array[10] = 5;
    return 0;
}
```

Εδώ, πηγαίνουμε και προσπελαύνουμε την ενδέκατη θέση του πίνακα `array`, ενώ ο πίνακας έχει μόνο 10 θέσεις (θυμηθείτε ότι η πρώτη θέση ενός πίνακα είναι η 0). Αν και αυτό είναι ένα προφανές λάθος διαχείρισης μνήμης, προσπαθήστε να τρέξετε το πρόγραμμα. Η εκτέλεσή του έγινε ομαλά; Γιατί πιστεύετε ότι συνέβη αυτό;

Ενώ είναι εφικτό η ανίχνευση των σφαλμάτων διαχείρισης μνήμης να γίνει με εξαντλητική ιχνηλάτηση του κώδικα και χρήση `printf`, ο τρόπος αυτός είναι πολύ αναποτελεσματικός και κουραστικός. Γι' αυτό το λόγο, αλλά και για την ευκολότερη ανίχνευση και των λογικών λαθών στα οποία αναφερθήκαμε στο εργαστήριο 4, θα δούμε στη συνέχεια πώς δουλεύει ένα εργαλείο

αποσφαλμάτωσης (debugging), ο gdb Το εργαλείο αυτό θα κάνει την ανίχνευση και τη διόρθωση των λογικών λαθών και των σφαλμάτων διαχείρισης μνήμης πολύ πιο εύκολη, ενώ δεν θα απαιτεί να πειράξουμε τον κώδικα για την εισαγωγή `printf`.

O debugger gdb

Ο debugger gdb είναι εγκατεστημένος στα συστήματα Unix και Linux της σχολής. Για να τον χρησιμοποιήσουμε, δίνουμε σαν παράμετρο το `-g3` κατά τη μεταγλώττιση του προγράμματος, π.χ.

```
gcc -g3 -o my_prog my_prog.c
```

Γράφοντας **`gdb ./my_prog`** ξεκινά η εκτέλεση του debugger, οπότε και εμφανίζεται μία γραμμή εντολών. Οι επιλογές που έχουμε στη διάθεσή μας είναι οι εξής (στις παρενθέσεις αναφέρονται τα συντετμημένα ονόματα των εντολών):

`break όνομα_συνάρτησης` (**`b`**)
Με αυτή την εντολή, η εκτέλεση του προγράμματος θα ανασταλεί όταν γίνει η πρώτη εκτέλεση της συνάρτησης που καθορίσαμε. Αν γράψουμε π.χ. `b main`, τότε η εκτέλεση του προγράμματος θα ανασταλεί αμέσως μόλις αυτό ξεκινήσει.

`break γραμμή` (**`b`**)
Με αυτή την εντολή, θέτουμε ένα σημείο διακοπής (breakpoint) σε συγκεκριμένη γραμμή, οπότε η εκτέλεση του προγράμματος θα ανασταλεί μόλις ο έλεγχος φτάσει στη γραμμή που δώσαμε. Τα breakpoints δεν μπορούν να μπουν σε γραμμές που είναι κενές ή έχουν μόνο σχόλια. Η εισαγωγή τουλάχιστον ενός breakpoint είναι απαραίτητη, γατί αλλιώς δεν θα μπορέσουμε να εκτελέσουμε βηματικά τον κώδικά μας.

`run` $\textbf{όρισμα}_1$ $\textbf{όρισμα}_2$ **`...`** $\textbf{όρισμα}_n$ (**`r`**)
Αφότου έχουμε βάλει τουλάχιστον ένα breakpoint, δίνουμε αυτή την εντολή για να ξεκινήσει η εκτέλεση του προγράμματος (μέχρι να φτάσει στο πρώτο breakpoint). Αν το πρόγραμμά μας παίρνει ορίσματα από τη γραμμή εντολής, τα δίνουμε εδώ, αλλιώς γράφουμε απλά `r`.

`step` (**`s`**)
Όταν η εκτέλεση του προγράμματος έχει ανασταλεί, μπορούμε να συνεχίσουμε την εκτέλεση βηματικά, δηλαδή να εκτελείται μόνο μία γραμμή κώδικα κάθε φορά. Με την εντολή `s` εκτελείται η τρέχουσα γραμμή κώδικα, ενώ αν αυτή είναι η κλήση κάποια συνάρτησης, ο έλεγχος μεταφέρεται εντός της.

`next` (**`n`**)
Το ίδιο με την `s`, μόνο που αν συναντήσει συνάρτηση την εκτελεί ολόκληρη χωρίς να μπει μέσα, οπότε ο έλεγχος μεταφέρεται στη γραμμή κώδικα μετά την κλήση της συνάρτησης.

`finish` (**`f`**)
Με αυτή την εντολή εκτελείται μέχρι τέλους η τρέχουσα συνάρτηση και η βηματική εκτέλεση συνεχίζει μέσα στην συνάρτηση που την κάλεσε.

`print παράσταση` (**`p`**)
Με αυτή την εντολή, εμφανίζεται η τιμή της παράστασης που δίνουμε, με βάση τις τρέχουσες τιμές των μεταβλητών. Η παράσταση μπορεί να είναι κάτι περίπλοκο, όπως `array[x]+y`, ή απλά μια μεταβλητή π.χ. `y` ή `array[2]`.

`continue` (**`c`**)
Η εκτέλεση του κώδικα συνεχίζεται κανονικά χωρίς βηματική εκτέλεση.

backtrace (**bt**)
Μας εμφανίζει τη στοίβα των συναρτήσεων. Με αυτή την εντολή, μπορούμε να δούμε ποια συνάρτηση κάλεσε την τρέχουσα συνάρτηση, ποια κάλεσε αυτή κλπ.

quit (**q**)
Σταματά η εκτέλεση του gdb και επιστρέφουμε στη γραμμή εντολής.

Πέρα από τις προφανείς ευκολίες που παρέχει ένας debugger, αν χειριστούμε σωστά τις προηγούμενες εντολές, έχει και τη δυνατότητα να μας δώσει υπερπολύτιμες πληροφορίες, όταν το πρόγραμμά μας κάνει segmentation fault. Ας το δούμε αυτό με ένα παράδειγμα.

```
#include <stdio.h>

int main(void) {
  int p[] = {10, -1, 8, 6, 9, 13, 0, -9, 6};
  int count = 0, sum = 0, i = 0;
  do {
      if (p[i] > 0) {
          count++;
          sum += p[count];
      }
      i++;
  } while (1);
  printf("There are %d positive numbers with sum %d\n", count, sum);
  return 0;
}
```

Το προηγούμενο πρόγραμμα προσπαθεί να μετρήσει πόσοι αριθμοί στον πίνακα `p` είναι θετικοί και να υπολογίσει το άθροισμά τους (παρατηρήστε ότι το μέγεθος του πίνακα δεν είναι καθορισμένο με άμεσο τρόπο).

Τρέχοντας αυτό το πρόγραμμα, αργά η γρήγορα θα μας δώσει segmentation fault. Ας τρέξουμε λοιπόν τον gdb για να μας βοηθήσει.

Αρχικά, μεταγλωττίζουμε το πρόγραμμα γράφοντας

gcc -g3 -o my_prog my_prog.c

όπου `my_prog.c` το όνομα που έχουμε δώσει στο αρχείο με τον πηγαίο κώδικα.

Γράφουμε **gdb ./my_prog** για να αρχίσει η εκτέλεση του gdb.

Τώρα δεν έχουμε παρά να δώσουμε `r` και να αφήσουμε τον gdb να τρέξει το πρόγραμμα. Κάποια στιγμή θα μας δώσει ένα μήνυμα που θα μοιάζει με αυτό

```
Program received signal SIGSEGV, Segmentation fault.
0x080483ac in main () at my_prog.c:7
7             if (p[i] > 0) {
```

το οποίο μας λέει όχι μόνο ότι υπήρξε segmentation fault, αλλά και σε ποια γραμμή ποιας συνάρτησης (`in main () at my_prog.c:7`) εμφανίστηκε, ενώ τυπώνει και το περιεχόμενο αυτής της γραμμής `if (p[i] > 0)`.

Οπότε, ο gdb μας βοήθησε να ανιχνεύσουμε πού συμβαίνει αυτό το λάθος διαχείρισης μνήμης. Ας δούμε πώς μπορεί να μας βοηθήσει να το διορθώσουμε.

Παρατηρήστε ότι η εκτέλεση του προγράμματος δεν έχει σταματήσει. Ο gdb μπορεί ακόμα να δεχτεί εντολές. Αυτό σημαίνει ότι μπορούμε να τον ρωτήσουμε για τις τιμές οποιασδήποτε μεταβλητής θέλουμε.

Αφού λοιπόν, το πρόβλημα έγινε στη γραμμή `if (p[i] > 0)`, ας επικεντρωθούμε σε όσες μεταβλητές περιλαμβάνει.

```
p p[i]
Cannot access memory at address 0xbf8de000
```

Άρα το `p[i]` αναφέρεται σε κάποια περιοχή της μνήμης στην οποία δεν έχουμε πρόσβαση, άρα σωστά το πρόγραμμά μας έκανε segmentation fault. Συνεπώς, αυτό σημαίνει ότι έχουμε βγει έξω από τα όρια του πίνακα `p`. Ας δούμε πού έχουμε φτάσει

```
p i
$3 = 1279
```

Το `$3` δεν πρέπει να σας απασχολεί. Αυτό που μας ενδιαφέρει είναι η αριθμητική τιμή που εμφανίζεται, το 1279. Άρα το `i` έχει φτάσει στο 1279! Είναι προφανές ότι κάπου μέσα στο πρόγραμμά μας έχουμε ξεχάσει να κάνουμε έλεγχο για να μην ξεπερνάμε τα όρια του πίνακα `p`. Όντως, πουθενά στον κώδικά μας δεν ελέγχουμε αν είμαστε μέσα στα όρια του πίνακα. Δεδομένου ότι ο πίνακας `p` είναι απροσδιόριστου μεγέθους, πώς θα αντιμετωπίζατε αυτό το πρόβλημα; Αυτό ήταν αρκετό για να λειτουργήσει σωστά το πρόγραμμα. Αν φταίει και κάτι άλλο, προσπαθήστε να το βρείτε με τη χρήση του gdb.