7. Δυναμική διαχείριση μνήμης

Σύνοψη
Στο κεφάλαιο αυτό ο αναγνώστης εισάγεται στην έννοια της δυναμικής διαχείρισης μνήμης. Αρχικά, δίνονται ο ορισμός, τα χαρακτηριστικά της και συγκρίνεται με τους υπόλοιπους τρόπους διαχείρισης μνήμης. Ακολούθως, παρουσιάζονται οι συναρτήσεις δυναμικής διαχείρισης μνήμης. Στην επόμενη ενότητα μελετώνται οι μονοδιάστατοι και πολυδιάστατοι δυναμικοί πίνακες και περιγράφεται η έννοια του δείκτη σε δείκτες με τη βοήθεια αναλυτικών παραδειγμάτων. Το κεφάλαιο ολοκληρώνεται με ένα εκτενές παράδειγμα διαδικαστικού προγραμματισμού, στο οποίο γίνεται χρήση των εννοιών, των γλωσσικών κατασκευών και των εργαλείων που μελετήθηκαν στο παρόν και τα προηγούμενα κεφάλαια.

Λέξεις κλειδιά
στατική/αυτόματη/δυναμική διαχείριση μνήμης, malloc, calloc, realloc, free, στοίβα, σωρός, δυναμικοί πίνακες, δείκτης σε δείκτες.

Προαπαιτούμενη γνώση
Λεξιλόγιο της γλώσσας C – μεταβλητές – εκφράσεις – τελεστές – έλεγχος ροής προγράμματος – συναρτήσεις – πίνακες – δείκτες

7.1 Η έννοια της δυναμικής διαχείρισης μνήμης

Όταν δηλώνεται μία καθολική μεταβλητή, ο μεταγλωττιστής κατανέμει χώρο μνήμης για τη μεταβλητή και ο χώρος αυτός παραμένει δεσμευμένος καθόλη τη διάρκεια εκτέλεσης του προγράμματος. Αυτός ο τρόπος κατανομής μνήμης ονομάζεται **στατική κατανομή** (static allocation).

Σε ό,τι αφορά τις τοπικές μεταβλητές που δηλώνονται σε μία συνάρτηση, αυτές αποθηκεύονται στην *στοίβα* (stack). Η κλήση της συνάρτησης έχει ως αποτέλεσμα να εκχωρηθεί μνήμη στην τοπική μεταβλητή, η οποία και αποδεσμεύεται με το πέρας της συνάρτησης. Αυτός ο τρόπος κατανομής μνήμης ονομάζεται **αυτόματη κατανομή** (automatic allocation).

Στη γλώσσα C υπάρχουν δομές, όπως η συνδεδεμένη λίστα (linked list), οι οποίες επεκτείνονται δυναμικά κατά τη διάρκεια εκτέλεσης του προγράμματος, χαρακτηριστικό που τις καθιστά ιδιαίτερα χρήσιμες για τις περιπτώσεις που κατά τον χρόνο μεταγλώττισης δεν είναι γνωστό το μέγεθος της μνήμης που θα απαιτηθεί για την αποθήκευση των δεδομένων. Ενδεικτικά, μπορεί να αναφερθεί ότι για ένα πρόγραμμα διαχείρισης ταχυδρομικών διευθύνσεων οι απαιτήσεις μνήμης δεν είναι γνωστές εκ των προτέρων, καθώς κατά τη διάρκεια της εκτέλεσης δημιουργούνται νέες διευθύνσεις και διαγράφονται παλιές. Σε μία τέτοια περίπτωση θα πρέπει να γίνεται **δυναμική διαχείριση** της μνήμης (dynamic allocation): όταν καταχωρούνται νέες ταχυδρομικές διευθύνσεις, θα πρέπει να εκχωρείται μνήμη στο πρόγραμμα, ενώ κατά τη διαγραφή διευθύνσεων η μνήμη που αυτές καταλάμβαναν θα πρέπει να απελευθερώνεται και να αποδίδεται στο σύστημα.

Συνοψίζοντας, στη γλώσσα C η διαθέσιμη μνήμη για την εκτέλεση ενός προγράμματος χωρίζεται συνήθως στα ακόλουθα τέσσερα τμήματα:

1. Στο *τμήμα κώδικα* (code segment), το οποίο χρησιμοποιείται για την αποθήκευση του μεταγλωττισμένου κώδικα του προγράμματος.

2. Στο *τμήμα δεδομένων* (data segment), το οποίο χρησιμοποιείται για την αποθήκευση καθολικών και στατικών μεταβλητών.

3. Στη *στοίβα* (stack segment), η οποία χρησιμοποιείται για την αποθήκευση των δεδομένων των συναρτήσεων (τοπικές μεταβλητές, πληροφορίες που αφορούν στις κλήσεις των συναρτήσεων, όπως η διεύθυνση επιστροφής στον κώδικα από τον οποίο κλήθηκε η συνάρτηση).

4. Στον *σωρό* (heap), τμήμα μνήμης ευρύτερο της στοίβας, που χρησιμοποιείται για τη δυναμική δέσμευση μνήμης.

Η γλώσσα C υποστηρίζει τη δυναμική διαχείριση μνήμης παρέχοντας ένα σύνολο από συναρτήσεις της βασικής βιβλιοθήκης. Οι συνήθεις συναρτήσεις διαχείρισης μνήμης είναι:
(α) Οι **`malloc()`**, **`calloc()`** για τον καθορισμό του μεγέθους της εκχωρούμενης μνήμης κατά την εκτέλεση ενός προγράμματος.

(β) Η **`realloc()`** για την αλλαγή του μεγέθους της εκχωρούμενης μνήμης κατά την εκτέλεση ενός προγράμματος.

(γ) Η **`free()`** για την απελευθέρωση μνήμης.

7.2 Οι συναρτήσεις malloc, calloc και free

Η συνάρτηση **`malloc()`** χρησιμοποιείται για τη δέσμευση μνήμης. Δεσμεύει ένα μπλοκ διαδοχικών θέσεων μνήμης. Ορίζεται στο αρχείο κεφαλίδας **`stdlib.h`** και δηλώνεται ως εξής:

```
void *malloc(int size);
```

Η **`malloc()`** επιστρέφει έναν δείκτη στην αρχή του μπλοκ μνήμης, στο οποίο γίνεται η εκχώρηση. Πρέπει να γίνεται μετατροπή τύπου, έτσι ώστε ο τύπος του δείκτη να είναι ίδιος με τα στοιχεία στα οποία δείχνει. Η παράμετρος **`size`** δίνει τον αριθμό των bytes που θα εκχωρηθούν. Δεν θα πρέπει να εισάγεται συγκεκριμένος αριθμός bytes, αλλά να χρησιμοποιείται η **`sizeof()`**, για να βρεθεί το μέγεθος ενός τύπου. Ο λόγος είναι ότι εάν η **`size`** λάβει συγκεκριμένο αριθμό bytes, δεν είναι ξεκάθαρο πόσοι αριθμοί θα ενταχθούν σ' αυτό το μπλοκ (για την περίπτωση των ακεραίων μπορεί κάθε αριθμός να αντιστοιχεί σε **2** ή σε **4** bytes). Για παράδειγμα, εάν πρέπει να δεσμευτεί μνήμη για **20** ακεραίους η **`malloc()`** συντάσσεται ως εξής:

```
int *pstart;
pstart=(int *)malloc(20*sizeof(int));
```

Στην παραπάνω πρόταση το **`size`** είναι ίσο με **`20*sizeof(int)`** και γίνεται μετατροπή τύπου (**`int *`**), ώστε να ταιριάζει η έξοδος της **`malloc()`** με τον δείκτη **`pstart`**.

Εάν δεν υπάρχει διαθέσιμη μνήμη, η **`malloc()`** επιστρέφει **NULL**, δηλαδή τη διεύθυνση **0**. Το **NULL** είναι έγκυρη διεύθυνση, που εγγυημένα δεν περιέχει ποτέ έγκυρα δεδομένα. Είναι καλή προγραμματιστική πρακτική να γίνεται πάντοτε έλεγχος κατά πόσον η **`malloc()`** επιστρέφει **NULL**, όπως φαίνεται στο πρόγραμμα που ακολουθεί:

```
char *pmessage;
pmessage=(char *)malloc(20*sizeof(char);
if (pmessage==NULL)
{
  printf( "Insufficient memory. Exiting..." );
  return(-1);
 }
```

Εναλλακτικά, μπορεί να χρησιμοποιηθεί η μακροεντολή **`assert()`**, η οποία ορίζεται στο αρχείο κεφαλίδας **`assert.h`** και ελέγχει κατά πόσον ισχύει μία συνθήκη. Σε περίπτωση που δεν ισχύει διακόπτεται το πρόγραμμα. Έτσι, θέτοντας ως συνθήκη ο δείκτης που δείχνει στο μπλοκ μνήμης να μην είναι **NULL**, ο έλεγχος διαθέσιμης μνήμης λαμβάνει την ακόλουθη μορφή:

```
char *pmessage;
pmessage=(char *) malloc(20*sizeof(char);
assert(pmessage!=NULL);
```

Στην περίπτωση που δεν υπάρχει διαθέσιμη μνήμη, το πρόγραμμα σταματά και εμφανίζεται το μήνυμα ***Assertion failed***, καθώς και η γραμμή κώδικα, στην οποία εμφανίστηκε η έλλειψη μνήμης.

Η συνάρτηση **calloc()** χρησιμοποιείται για δέσμευση μνήμης. Ορίζεται στο αρχείο κεφαλίδας **stdlib.h** και δηλώνεται ως εξής:

```
void  *calloc(int n, int size);
```

Η **calloc()** επιστρέφει έναν δείκτη στην αρχή του μπλοκ, στο οποίο γίνεται η εκχώρηση ή το **NULL**, εάν δεν υπάρχει διαθέσιμη μνήμη. Το **NULL** επιστρέφεται και όταν **n=0** ή **size=0**. Τέλος, το δεσμευμένο μπλοκ αρχικοποιείται με την τιμή **0**.

Ένα παράδειγμα χρήσης της **calloc()** είναι το ακόλουθο, όπου δεσμεύεται μνήμη για αλφαριθμητικό δέκα χαρακτήρων, δηλαδή **n=10** και **size=sizeof(char)**.

```
char *str = NULL;
str=(char *)calloc(10,sizeof(char));
```

Η συνάρτηση **free()** χρησιμοποιείται για την αποδέσμευση μνήμης. Δεσμεύει ένα μπλοκ διαδοχικών θέσεων μνήμης. Ορίζεται στο αρχείο κεφαλίδας **stdlib.h** και δηλώνεται ως εξής:

```
void free (void *);
```

Η **free()** δέχεται ως όρισμα έναν δείκτη, ο οποίος δείχνει στην αρχή του μπλοκ που απελευθερώνεται. Για παράδειγμα, εάν έχει δεσμευτεί μνήμη για **20** ακεραίους, η δέσμευση– αποδέσμευση μνήμης υλοποιούνται ως εξής:

```
int *pstart;
pstart=(int *)malloc(20*sizeof(int));
free(pstart);
```

Η **free()** δεν έχει επιστρεφόμενη τιμή. Απλώς αποδεσμεύει τη μνήμη που είχε εκχωρηθεί από τη **malloc()**. Οι συναρτήσεις **malloc()/calloc()** και **free()** αναγκαιούν η μία την άλλη. Εάν χρησιμοποιηθούν οι **malloc()/calloc()** για εκχώρηση μνήμης, πρέπει οπωσδήποτε να ακολουθήσει η **free()** για την αποδέσμευσή της.

Παρατηρήσεις:

(α) Η πρόταση **free(ptr);** είναι επικίνδυνη, εάν ο **ptr** δεν είναι έγκυρος. Το αποτέλεσμα είναι απρόβλεπτο: μπορεί να μη συμβεί τίποτε είτε να υπάρξει κάποιο σφάλμα είτε ακόμη και να κολλήσει το πρόγραμμα. Ωστόσο, ο δείκτης που χρησιμοποιείται στη **free()** δεν είναι κατ' ανάγκη ο ίδιος δείκτης που χρησιμοποιήθηκε στη **malloc()/calloc()**.Το ακόλουθο τμήμα κώδικα είναι σωστό:

```
char *pmessage, *pmsg, aLetter;
pmessage=(char *)malloc(20*sizeof(char);
. . . . . .
pmsg=pmessage;/* Πλέον και οι δύο δείκτες δείχνουν στην ίδια θέση */
pmessage=&aLetter;     / * Πλέον ο pmessage δείχνει στο aLetter */
free(pmsg);  /* Απελευθερώνεται η δεσμευθείσα μνήμη, στην οποία
                αρχικά έδειχνε ο pmessage */
```

(β) Όταν σε ένα πρόγραμμα γίνουν επαναλαμβανόμενες εκχωρήσεις μνήμης χωρίς τις αντίστοιχες απελευθερώσεις, συμβαίνουν «διαρροές μνήμης» (memory leakages): το πρόγραμμα απαιτεί ολοένα και περισσότερη μνήμη, καθώς εκτελείται, και τελικά είτε θα πρέπει να σταματήσει είτε θα κολλήσει. Για την αποφυγή αυτών των δυσχερειών θα πρέπει τα ζεύγη **malloc()/calloc()–free()** να διατηρούνται στο ίδιο τμήμα κώδικα.

(γ) Δεν θα πρέπει να επιχειρηθεί να απελευθερωθεί η ίδια μνήμη δύο φορές. Το ακόλουθο τμήμα κώδικα είναι εσφαλμένο:

```
char *pmessage, *pmsg, aLetter;
```

```
pmessage=(char *)malloc(20*sizeof(char);
. . . . . .
pmsg=pmessage;/* Πλέον και οι δύο δείκτες δείχνουν στην ίδια θέση */
free(pmsg);
free(pmessage); /* ΛΑΘΟΣ: Το μπλοκ μνήμης έχει ήδη απελευθερωθεί! */
```

7.2.1 Παράδειγμα

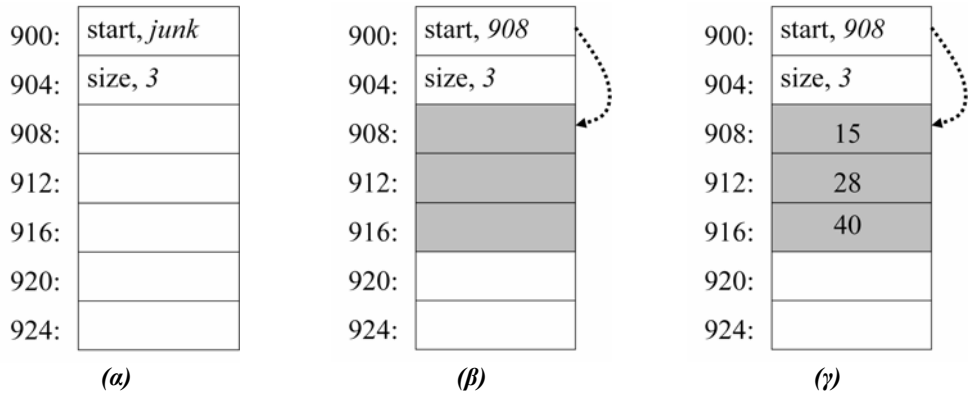
Να περιγραφεί αναλυτικά η λειτουργία του ακόλουθου προγράμματος και να απεικονιστούν οι μεταβολές που συντελούνται στον χάρτη μνήμης:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
  int *pstart, size=3;
  pstart=(int *)malloc(size*sizeof(int));
  *pstart=15;
  *(pstart+1)=28;
  *(pstart+2)=*(pstart+1)+12;
  free(pstart);

  return 0;
}
```

- Δηλώνονται ο δείκτης σε ακέραιο **pstart** και η ακέραια μεταβλητή **size**, η οποία αρχικοποιείται λαμβάνοντας την τιμή **3** (**Σχήμα 7.1.α**).
- Η εκτέλεση της **malloc()** έχει ως αποτέλεσμα την αναζήτηση ενός συνεχούς μπλοκ **3x4=12** bytes και, όταν το μπλοκ βρεθεί, επιστρέφεται ένας δείκτης στην αρχή του μπλοκ μνήμης. Δηλαδή, *επιστρέφεται η διεύθυνση* του πρώτου byte σ' αυτό το μπλοκ. Η διεύθυνση αυτή ανατίθεται στον δείκτη **pstart** (**Σχήμα 7.1.β**).
- Με χρήση αριθμητικής δεικτών και του τελεστή περιεχομένου αποδίδονται οι **15**, **28** και **40** στις διευθύνσεις **908-911**, **912-915** και **916-919**, αντίστοιχα (**Σχήμα 7.1.γ**).
- Η εκτέλεση της **free()** έχει ως αποτέλεσμα την απελευθέρωση της δεσμευθείσας μνήμης και ο χάρτης μνήμης επανέρχεται στην αρχική του μορφή (**Σχήμα 7.1.α**).



Σχήμα 7.1 *Χάρτης μνήμης του παραδείγματος 7.2.1*

7.3 Η συνάρτηση realloc

Η συνάρτηση **realloc()** χρησιμοποιείται για τη διεύρυνση ή συρρίκνωση ενός ήδη δεσμευμένου μπλοκ μνήμης. Ορίζεται στο αρχείο κεφαλίδας **stdlib.h** και δηλώνεται ως εξής:

```
void  *realloc(void *pblock, int size);
```

Η **realloc()** μεταβάλλει το μέγεθος ενός τμήματος μνήμης που είχε προηγουμένως δεσμευθεί και στο οποίο έδειχνε ο δείκτης **pblock**. Το νέο μέγεθος καθορίζεται σε **size** bytes. Εάν **size=0**, το μπλοκ μνήμης απελευθερώνεται και επιστρέφεται το **NULL**. Η **realloc()** διασφαλίζει τα υπάρχοντα περιεχόμενα στη μνήμη και επιστρέφει έναν δείκτη στο νέο τμήμα μνήμης. Ο δείκτης αυτός μπορεί να είναι είτε ίδιος με τον δείκτη **pblock**, εάν διατηρηθεί η ίδια αρχή και για το νέο τμήμα μνήμης, είτε διαφορετικός, στην περίπτωση που το τμήμα μετακινηθεί. Τέλος, εάν ο **pblock** είναι **NULL**, η **realloc()** λειτουργεί όπως η **malloc()**.

Παρατήρηση: Οι συναρτήσεις δέσμευσης μνήμης **malloc()**, **calloc()**, **realloc()** επιστρέφουν έναν δείκτη τύπου **void**, ο οποίος δείχνει στον δεσμευθέντα χώρο μνήμης. Έως τώρα χρησιμοποιήθηκε η μετατροπή τύπου, για να μετατραπεί ο δείκτης αυτός σε δείκτη του τύπου δεδομένων, τα οποία θα αποθηκευτούν στον χώρο αυτόν. Ωστόσο, η μετατροπή τύπου δεν είναι απαραίτητη, καθώς σε μία πρόταση ανάθεσης στη γλώσσα C γίνεται ούτως ή άλλως αυτόματη μετατροπή τύπου. Για παράδειγμα, στο ακόλουθο τμήμα κώδικα:

```
double ptr;
ptr=calloc(10,sizeof(double));
```

με την εκχώρηση της διεύθυνσης του πρώτου byte της δεσμευθείσας μνήμης στον δείκτη **ptr**, τύπου **double**, γίνεται αυτόματη μετατροπή τύπου και ο **ptr** μπορεί να διαχειριστεί τα **8x10=80** bytes της δεσμευθείσας μνήμης.

7.3.1 Παράδειγμα

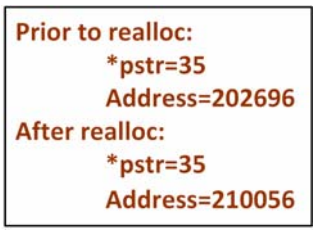
Στον κώδικα που ακολουθεί, αρχικά δεσμεύονται **10** τετράδες bytes για την αποθήκευση ακεραίων και στη συνέχεια χρησιμοποιείται η **realloc()** για τη διεύρυνση του μπλοκ μνήμης στις **20** τετράδες bytes. Ελέγχοντας τις θέσεις μνήμης διαπιστώνεται ότι το νέο μπλοκ μνήμης ανευρέθηκε σε άλλο σημείο της μνήμης σε σχέση με το αρχικό μπλοκ και τα περιεχόμενα του αρχικού μπλοκ διατηρήθηκαν αναλλοίωτα.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *pstr,*pstr2;
  pstr=(int *)malloc(10*sizeof(int));
  *pstr=35;
  *(pstr+1)=-12;
  printf( "Prior to realloc:\n\t*pstr=%d\n\tAddress is %d\n", *pstr,
pstr);
  pstr2=(int *)realloc(pstr, 20*sizeof(int));
  /* Θα μπορούσε να χρησιμοποιηθεί εκ νέου ο pstr, δηλαδή
     pstr=(int *)realloc(pstr, 20*sizeof(int)); */
  printf( "After realloc:\n\t*pstr2=%d\n\tAddress is %d\n", *pstr2,
pstr2);
  free(pstr2);
  free(pstr);

  return 0;
```

```
}
```



Εικόνα 7.2 *Η έξοδος του προγράμματος του παραδείγματος 7.3.1*

7.4 Μονοδιάστατοι δυναμικοί πίνακες

Οι μονοδιάστατοι δυναμικοί πίνακες παρουσιάζουν το σημαντικό πλεονέκτημα να δημιουργούνται κατά τον χρόνο εκτέλεσης του προγράμματος, χρησιμοποιώντας δυναμική δέσμευση μνήμης για τη δημιουργία τους. Τους δυναμικούς πίνακες διαχειρίζονται δείκτες. Ένα επιπρόσθετο πλεονέκτημα είναι ότι αποθηκεύονται στον σωρό αντί για τη στοίβα που χρησιμοποιείται στην κλασική δήλωση πινάκων. Για τη δημιουργία τους απαιτείται να ακολουθηθεί η εξής σειρά βημάτων:

1. Δηλώνεται ένας δείκτης που θα διαχειριστεί τον πίνακα.

2. Καλείται η συνάρτηση **`malloc()`** (ή η **`calloc()`**) για να δεσμευθεί η απαραίτητη μνήμη για τα στοιχεία του πίνακα. Επειδή ο κάθε τύπος δεδομένων απαιτεί διαφορετικό αποθηκευτικό χώρο, με την κλήση της **`malloc()`** θα πρέπει να ζητηθεί η δέσμευση τόσων bytes μνήμης όσο και το γινόμενο του πλήθους των στοιχείων του πίνακα επί το μέγεθος του τύπου δεδομένων.

3. Το αποτέλεσμα της **`malloc()`** ανατίθεται στον δείκτη του βήματος **1**.

Οι δυναμικοί πίνακες παρουσιάζουν το πλεονέκτημα ότι, όταν δεν πρόκειται να χρησιμοποιηθούν περαιτέρω από το πρόγραμμα, τότε η μνήμη που αυτοί καταλαμβάνουν μπορεί να αποδεσμευτεί με χρήση της συνάρτησης **`free()`**.

Για παράδειγμα, προκειμένου να δημιουργηθεί ένας δυναμικός πίνακας αριθμών κινητής υποδιαστολής διπλής ακρίβειας **10** θέσεων, δηλώνεται ένας δείκτης σε **`double`**:

```
double *ptr;
```

και στη συνέχεια δεσμεύεται μνήμη, την οποία θα διαχειρίζεται ο δείκτης **`ptr`**:

```
double ptr=malloc(10*sizeof(double));
```

Θα πρέπει να σημειωθεί ότι, ενώ δεν έχει δημιουργηθεί πίνακας **`double`** αριθμών μέσω δήλωσης πίνακα, η σημειογραφία διαχείρισης του πίνακα παραμένει η ίδια με εκείνη του δηλωμένου πίνακα, π.χ. καθώς το **`ptr[1]`** αντιστοιχεί στο δεύτερο στοιχείο του πίνακα.

Η αποδέσμευση του δεσμευμένου χώρου μνήμης γίνεται ως εξής:

```
free(ptr);
```

<u>Παρατηρήσεις:</u>

1. Για τη δημιουργία ενός μονοδιάστατου δυναμικού πίνακα **N** θέσεων απαιτούνται **`N*sizeof(<τύπος δεδομένου>)`** για να αποθηκευτούν τα δεδομένα, χώρος που δεσμεύεται στον σωρό, καθώς και **4** bytes για την αποθήκευση του δείκτη που διαχειρίζεται τα στοιχεία του πίνακα. Δηλαδή ο δυναμικός πίνακας απαιτεί **4** παραπάνω bytes από τον δηλωμένο πίνακα, γεγονός που δεν αποτελεί σημαντικό μειονέκτημα σε σχέση με τα οφέλη του δυναμικού πίνακα που αναφέρθηκαν παραπάνω.

2. Οι πίνακες μεταβλητού μήκους (VLA), που μελετήθηκαν στο Κεφάλαιο 5, είναι μία μορφή δυναμικής κατανομής μνήμης, καθώς το μέγεθός τους καθορίζεται κατά τον χρόνο εκτέλεσης του προγράμματος, ωστόσο δεν μπορεί να τροποποιηθεί το μέγεθός τους ούτε να απελευθερωθεί η μνήμη που καταλαμβάνουν.

7.5 Πίνακας δεικτών για τη διαχείριση αλφαριθμητικών

Ένας πίνακας δεικτών (array of pointers) ορίζεται ως εξής:

```
<τύπος δεδομένων δείκτη>   *<όνομα πίνακα>[μέγεθος];
```

Η πρόταση

```
char *name[3];
```

ορίζει τον πίνακα **name** τριών θέσεων, τα στοιχεία του οποίου είναι δείκτες σε χαρακτήρα. Με αυτόν τον τρόπο τα στοιχεία του πίνακα δείχνουν σε αλφαριθμητικά και ο αριθμοδείκτης (index) του πίνακα επιλέγει ένα αλφαριθμητικό. Η λειτουργία του πίνακα αυτού θα περιγραφεί με τη βοήθεια του ακόλουθου παραδείγματος.

7.5.1 Παράδειγμα

Θεωρούμε το ακόλουθο πρόγραμμα.

```
#include<stdio.h>

int main()
{
  int i;
  char *name[3]={"John","James","Jack" };
  char *tmp;
  printf( "\nAddresses of pointers:\n");
  for (i=0;i<3;i++)
    printf( "&name[%d]=%d  ",i,&name[i] );
  printf( "\n\nAddresses of first character:\n");
  for (i=0;i<3;i++)
    printf( "&name[%d][0]=%d  ",i,name[i] );
  printf( "\n\nContents of strings:\n");
  for (i=0;i<3;i++)
    printf( "name[%d]=%s  ",i,name[i] );
  tmp=name[0]; /* Αντιμετάθεση των name[0] και name[2] */
  name[0]=name[2];
  name[2]=tmp;
  printf( "\n\nContents of strings after swapping:\n");
  for (i=0;i<3;i++)
    printf( "name[%d]=%s  ",i,name[i] );

  return 0;
}
```

- Αρχικά δημιουργείται ο πίνακας δεικτών **name**, για τον οποίο δεσμεύεται μνήμη στη στοίβα. Για το μηχάνημα στο οποίο εκτελέστηκε το πρόγραμμα, τα δεσμευθέντα bytes είναι τα **2686752-2686755**, **2686756-2686759** και **2686760-2686763** για τους δείκτες **name[0]**, **name[1]** και **name[2]**, αντίστοιχα.
- Σε κάθε δείκτη αποδίδεται η διεύθυνση του πρώτου byte ενός αλφαριθμητικού, το οποίο αποθηκεύεται στον σωρό. Ειδικότερα, στον δείκτη **name[0]** αντιστοιχεί το αλφαριθμητικό **"John"**, τεσσάρων χαρακτήρων, το οποίο αποθηκεύεται στις διευθύνσεις **4206592-4206595**, ενώ στο byte **4206596** αποθηκεύεται ο μηδενικός χαρακτήρας τερματισμού του αλφαριθμητικού. Κατ' αντίστοιχο τρόπο, ο **name[1]** δείχνει στο **"James"** και ο **name[2]** στο **"Jack"**.

- Με τη βοήθεια του δείκτη σε χαρακτήρα **temp** ανταλλάσονται οι διευθύνσεις των **name[0]** και **name[2]**. Στο τέλος του προγράμματος ο **name[0]** δείχνει στο **"Jack"** και ο **name[2]** στο **"John"**.
- Όπως και στην περίπτωση των δυναμικών πινάκων, ενώ δεν έχει δημιουργηθεί πίνακας αλφαριθμητικών αλλά πίνακας δεικτών που δείχνουν σε αλφαριθμητικά, η σημειογραφία διαχείρισης των αλφαριθμητικών παραμένει η ίδια με εκείνη των πινάκων αλφαριθμητικών, π.χ. καθώς το **name[1]** αντιστοιχεί στο δεύτερο αλφαριθμητικό και το **name[1][3]** αντιστοιχεί στον τέταρτο χαρακτήρα του δεύτερου αλφαριθμητικού.

```
Addresses of pointers:
&name[0]=2686752  &name[1]=2686756  &name[2]=2686760

Addresses of first character:
&name[0][0]=4206592  &name[1][0]=4206597  &name[2][0]=4206603

Contents of strings:
name[0]=John  name[1]=James  name[2]=Jack

Contents of strings after swapping:
name[0]=Jack  name[1]=James  name[2]=John
```

Εικόνα 7.3 *Η έξοδος του προγράμματος του παραδείγματος 7.5.1*

7.6 Δείκτης σε δείκτες για τη διαχείριση πολυδιάστατων πινάκων δεδομένων

Ο δείκτης σε δείκτη είναι μία μορφή *έμμεσης αναφοράς* σε δεδομένα. Στην περίπτωση ενός κοινού δείκτη, η τιμή του δείκτη είναι η διεύθυνση μίας «κανονικής» μεταβλητής. Στην περίπτωση ενός δείκτη σε δείκτη, το περιεχόμενο του πρώτου δείκτη είναι η διεύθυνση του δεύτερου δείκτη, ο οποίος δείχνει στην κανονική μεταβλητή.

Η έμμεση αναφορά μπορεί να λάβει ένθεση οιουδήποτε βάθους (δείκτης σε δείκτη σε δείκτη κ.λπ.), ωστόσο θα πρέπει να αποφεύγονται οι υπερβολές, γιατί ο κώδικας αφενός μεν θα γίνει δυσανάγνωστος, αφετέρου θα είναι επιρρεπής σε σφάλματα.

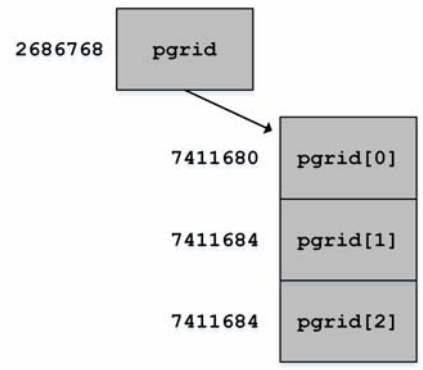
Ο τρόπος λειτουργίας του δείκτη σε δείκτες παρουσιάζεται με τη βοήθεια του ακόλουθου κώδικα:

```
#include <stdio.h>

int main()
{
   int i,rowNumber=3,columnNumber=5;
1  float **matrixPointer; /* δείκτης σε δείκτες σε float
                             (pointer-to-(pointers-to-float) */
2  matrixPointer=(float **)malloc(rowNumber*sizeof(float *));
3  for(i=0;i<rowNumber;i++)
   {
      matrixPointer[i]=(float *)malloc(columnNumber*sizeof(float));
   }
   . . . . . .
4  for(i=0;i<rowNumber;i++)
   {
      free(matrixPointer[i]);
   }
5  free(matrixPointer);
```

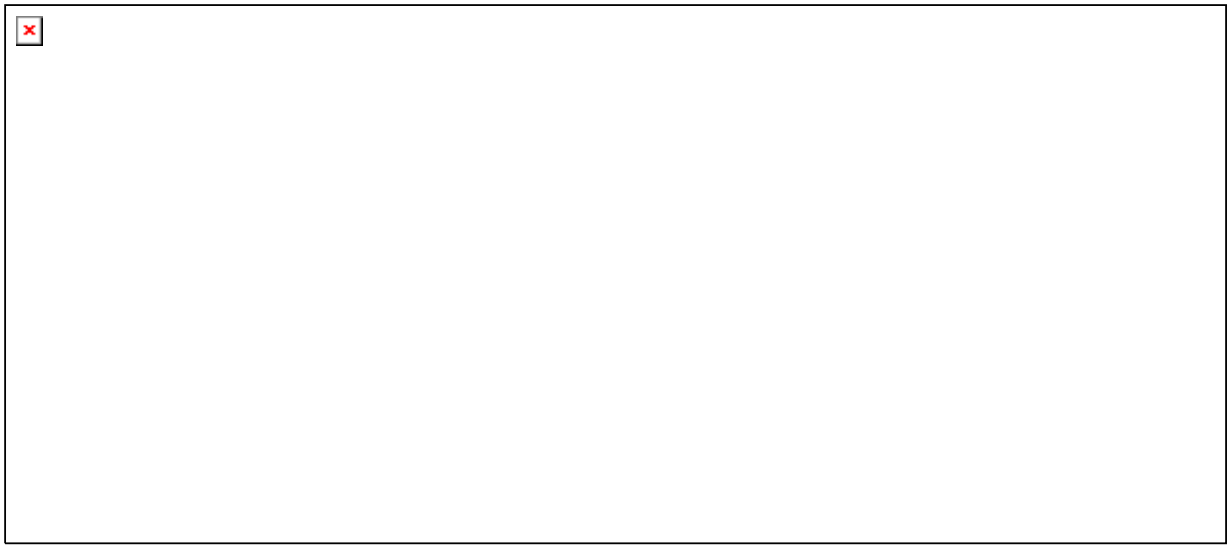
```
    return 0;
}
```

- **Γραμμή 1:** Δηλώνεται ένας διπλός δείκτης, ο οποίος δείχνει σε μία λίστα δεικτών σε `float`.
- **Γραμμή 2:** Δεσμεύεται ένα μπλοκ μνήμης, επαρκές για `rowNumber` δείκτες σε `float`. Στο **Σχήμα 7.2.α** απεικονίζεται ο χάρτης μνήμης:



Εικόνα 7.2.α *Η χάρτης μνήμης μετά την εκτέλεση της γραμμής 2*

- **Γραμμή 3:** Για κάθε δείκτη σε `float`, `pgrid[i]` δεσμεύεται μνήμη για `columnNumber` σε αριθμό δεδομένα τύπου `float`.



Εικόνα 7.2.β *Η χάρτης μνήμης μετά την εκτέλεση της γραμμής 3*

- **Γραμμές 4-5:** Για την απελευθέρωση της μνήμης αντιστρέφεται η διαδικασία: αρχικά απελευθερώνεται κάθε μπλοκ από αριθμούς κινητής υποδιαστολής και στη συνέχεια κάθε μπλοκ από δείκτες σε float.

Κατ' αντιστοιχία με τους μονοδιάστατους δυναμικούς πίνακες, μακροσκοπικά η προσπέλαση πολυδιάστατων δυναμικών πινάκων με χρήση δεικτών είναι απολύτως ίδια με την προσπέλαση των κλασικών πινάκων, π.χ. το στοιχείο `matrixPointer[1][2]` αντιστοιχεί στη δεύτερη γραμμή και τρίτη στήλη του πίνακα. Η διαφορά έγκειται στο ότι στους πίνακες με δείκτες υπάρχουν `rowNumber` ομάδες από `columnNumber` στοιχεία και οι ομάδες δεν καταλαμβάνουν κατ' ανάγκη διαδοχικές θέσεις μνήμης. Όταν ζητείται το στοιχείο `matrixPointer[1][2]`, ουσιαστικά ζητείται να προσπελαστεί ο δεύτερος δείκτης της λίστας των δεικτών και να ληφθεί η τρίτη τιμή των δεδομένων τύπου `float`, στα οποίους δείχνει ο συγκεκριμένος δείκτης.

Τα πλεονεκτήματα των μονοδιάστατων δυναμικών πινάκων έναντι των κλασικών δηλωμένων πινάκων όχι μόνο ισχύουν στην περίπτωση των πολυδιάστατων πινάκων, αλλά ενισχύονται, καθώς όσο μεγαλώνουν οι διαστάσεις και ο αριθμός των στοιχείων ενός πίνακα, τόσο αυξάνει η πιθανότητα να μη βρεθούν οι απαιτούμενες διαδοχικές θέσεις μνήμης στην στοίβα για την αποθήκευση των στοιχείων του πίνακα. Κατά συνέπεια, η αποθήκευση των δεδομένων στον σωρό αποτελεί άμεση λύση σ' αυτό το πρόβλημα. Βέβαια, η χρήση δεικτών σε δείκτες προϋποθέτει τη δημιουργία των πινάκων δεικτών, δηλαδή επιπρόσθετη μνήμη. Ωστόσο, το ποσοστό αύξησης της μνήμης είναι πολύ μικρό για πραγματικές περιπτώσεις χρήσης. Για παράδειγμα, ενας πίνακας στον οποίο αποθηκεύονται **4** στοιχεία για κάθε Έλληνα πολίτη σε μορφή ακεραίων (ΑΦΜ, ημέρα/μήνας/έτος γέννησης), απαιτεί **4x11.000.000x4=176.000.000** διαδοχικά bytes. Ένας τέτοιος πίνακας **4x11.000.000** δεν μπορεί να αποθηκευτεί στη στοίβα. Εάν χρησιμοποιείτο ένας δείκτης σε πίνακα **4** δεικτών, όπως παρουσιάζεται στο παρακάτω πρόγραμμα, η υλοποίηση του πίνακα είναι εφικτή.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main()
{
  int **pmatr,i;

  printf( "address of pmatr=%d\n",&pmatr );
  pmatr=(int **)malloc(4*sizeof(int *));
  for (i=0;i<4;i++)
  {
    pmatr[i]=(int *)malloc(11000000*sizeof(int));
    assert(pmatr[i]!=NULL);
    printf( "address of pmatr[%d]=%d\n",i,&pmatr[i] );
    printf( "   address of pmatr[%d][%d]=%d\n",i,0,&pmatr[i][0] );
    printf(          "                                    address          of
pmatr[%d][%d]=%d\n",i,10999999,&pmatr[i][10999999] );
  }
  for (i=3;i>=0;i--)   free(pmatr[i]);
  free(pmatr);
  return 0;
}
```

```
address of pmatr=2686780

address of pmatr[0]=7280632
    address of pmatr[0][0]=7340064
    address of pmatr[0][10999999]=51340060

address of pmatr[1]=7280636
    address of pmatr[1][0]=51380256
    address of pmatr[1][10999999]=95380252

address of pmatr[2]=7280640
    address of pmatr[1][0]=95420448
    address of pmatr[1][10999999]=139420444

address of pmatr[3]=7280644
    address of pmatr[1][0]=139460640
    address of pmatr[1][10999999]=183460636
```

Εικόνα 7.4 *Η έξοδος του προγράμματος του παραδείγματος 7.6*

Η επιβάρυνση στη μνήμη είναι μόνο **20** bytes (**4** bytes για τον διπλό δείκτη **pmatr** και **16** bytes για τον πίνακα των απλών δεικτών **pmatr[0]**,...,**pmatr[3]**). Σε σχέση με τη συνολική μνήμη που απαιτείται για τα δεδομένα (**176.000.000** bytes), η επιβάρυνση ανέρχεται στο **0.0000136%**.

7.6.1 Παράδειγμα

Η δέσμευση και η αποδέσμευση μνήμης τρισδιάστατου δυναμικού πίνακα, διαστάσεων **nxmxk** (π.χ. **3x2x250**), υλοποιείται ως εξής:

```
/* Δέσμευση μνήμης */
float ***parr;
parr=(float ***)malloc(3*sizeof(float**));
assert(parr!=NULL);

for (i=0;i<3;i++)
{
  parr[i]=(float **)malloc(2*sizeof(float*));
  assert(parr[i]!=NULL);

  for (j=0; j<2; j++)
  {
    parr[i][j]=(float *)malloc(250*sizeof(float));
    assert(parr[i][j]!=NULL);
  }
}
/* Αποδέσμευση μνήμης */
for (i=2;i>=0;i--)
  for (j=1;j>=0;j--)
    free(parr[i][j]);
for (i=2;i>=0;i--)
  free(parr[i]);
free(parr);
```

7.7 Συναρτήσεις οριζόμενες από τον χρήστη για τη δέσμευση/αποδέσμευση μνήμης

Στην περίπτωση που σε ένα πρόγραμμα γίνεται επανειλημμένα δέσμευση και αποδέσμευση μνήμης, μπορούν να οριστούν συναρτήσεις που θα δεσμεύουν και θα αποδεσμεύουν μνήμη για πίνακες **float**, **int** κ.λπ.

Οι συναρτήσεις δέσμευσης μνήμης θα έχουν παραμέτρους τα μεγέθη της μνήμης που ζητείται να δεσμευτεί και θα επιστρέφουν τον δείκτη που θα διαχειρίζεται τη μνήμη. Οι συναρτήσεις αποδέσμευσης μνήμης θα έχουν παραμέτρους τον δείκτη που διαχειρίστηκε τη μνήμη και το μέγεθος αυτής. Δεν θα έχουν επιστρεφόμενη τιμή.

7.7.1 Παράδειγμα

Στο πρόγραμμα που ακολουθεί, χρησιμοποιούνται οι συναρτήσεις **allocate_2()** και **free_2()** για τη δέσμευση/αποδέσμευση μνήμης για ένα δισδιάστατο πίνακα αριθμών κινητής υποδιαστολής απλής ακριβείας.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
```

```
float **allocate_2(int size1, int size2);
void free_2(float **parr, int size1);

int main()
{
  int i;
  float **ps;
  ps=allocate_2(3,250); /* Πίνακας 3x250 */

  printf( "\n\nps=%d\t\taddr(ps)=%d\n",ps,&ps );
  printf( "ps[0]=%d\t\taddr(ps[0])=%d\n",ps[0],&ps[0] );
  printf( "ps[1]=%d\t\taddr(ps[1])=%d\n",ps[1],&ps[1] );
  printf( "ps[2]=%d\t\taddr(ps[2])=%d\n",ps[2],&ps[2] );
  ps[0][0]=400; /* Για να ελεγχθεί η απελευθέρωση μνήμης */
  printf( "\n\n(Prior to free)  ps[0][0]=%f\n",ps[0][0] );

  free_2(ps,3);
  printf( "\n\n(Afterwards)  ps[0][0]=%f\n",ps[0][0] );

  return 0;
}

float **allocate_2(int size1, int size2)
{
  int i;
  float **parr;
  parr=(float **)malloc(size1*sizeof(float *));
  assert(parr!=NULL);
  for (i=0;i<size1;i++)
  {
    parr[i]=(float *)malloc(size2*sizeof(float));
    assert(parr[i]!=NULL);
  }
  printf( "\n\nparr=%d\t\taddr(parr)=%d\n",parr,&parr );
  printf( "parr[0]=%d\t\taddr(parr[0])=%d\n",parr[0],&parr[0] );
  printf("parr[1]=%d\t\taddr(parr[1])=%d\n",parr[1],&parr[1]);
  printf("parr[2]=%d\t\taddr(parr[2])=%d\n",parr[2],&parr[2]);

  return(parr);
}

void free_2(float **parr, int size1)
{
  int i;
  for (i=(size1-1);i>=0;i--) free(parr[i]);
  free(parr);
}
```

Εικόνα 7.5 *Η έξοδος του προγράμματος του παραδείγματος 7.7.1*

Η πρόταση `ps=allocate_2(3,250);` καλεί τη συνάρτηση `allocate_2()`, για να δεσμευτεί μνήμη για έναν πίνακα **3x250** αριθμών κινητής υποδιαστολής απλής ακριβείας. Η `allocate_2(3,250)` δεσμεύει τη μνήμη με τον τρόπο που περιγράφηκε στην ενότητα 7.6 και επιστρέφει τη διεύθυνση ενός τοπικού διπλού δείκτη `parr`, ο οποίος διαχειρίζεται τον δεσμευμένο χώρο. Η διεύθυνση αυτή αποθηκεύεται στον διπλό δείκτη `ps`, ο οποίος πλέον θα διαχειρίζεται τη δεσμευθείσα μνήμη μέσα στη συνάρτηση `main()`.

Από τα αποτελέσματα της **Εικόνας 7.5** προκύπτει ότι όντως ο `ps` επιτελεί το έργο της διαχείρισης της μνήμης, καθώς το σχήμα εκχώρησης μνήμης που υλοποιήθηκε μέσα στη συνάρτηση έχει παραμείνει αυτούσιο και μετά το πέρας του, με τον `ps` να έχει αναλάβει πλέον τα καθήκοντα του `parr`: ενώ οι `parr` και `ps` προφανώς αποθηκεύτηκαν σε διαφορετικές διευθύνσεις, δείχνουν στην ίδια διεύθυνση (**5249008**) από την οποία ξεκινά ο πίνακας των τριών απλών δεικτών σε `float`. Επιπλέον, τα `parr[i]` και `ps[i]` (**i=0,1,2**) έχουν τις ίδιες διευθύνσεις και δείχνουν στα ίδια μπλοκ μνήμης, μεγέθους **4x250=1000** bytes.

Σε ό,τι αφορά την αποδέσμευση της μνήμης, καλείται η συνάρτηση `free_2()` με ορίσματα τη διεύθυνση από την οποία ξεκινά ο πίνακας των τριών απλών δεικτών σε `float` και την πρώτη διάσταση του πίνακα. Μέσα στη `free_2()` γίνεται η αποδέσμευση της μνήμης, με τον τρόπο που περιγράφηκε στην ενότητα 7.6. Για να δειχθεί έμπρακτα ότι η μνήμη έχει αποδεσμευτεί, εμφανίζεται στην οθόνη το περιεχόμενο του στοιχείου `ps[0][0]`, το οποίο έχει λάβει την τιμή **400**, πριν και μετά την αποδέσμευση. Μετά την αποδέσμευση παρατηρείται ότι το στοιχείο `ps[0][0]` δεν έχει διατηρήσει την τιμή του, αλλά έχει μηδενιστεί, γεγονός που υποδηλώνει ότι ο χώρος μνήμης που είχε δεσμευτεί για τον πίνακα δεν είναι πλέον διαχειρίσιμος από τον δείκτη `ps`.

7.7.2 Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

1. Θα δεσμεύει δυναμικά μνήμη για πίνακες αλφαριθμητικών (δισδιάστατους πίνακες χαρακτήρων). Για τον σκοπό αυτό θα αναπτυχθεί η συνάρτηση `char **alloc_2(int size 1, int size2)`.

2. Με χρήση της ανωτέρω συνάρτησης θα δεσμευτεί μνήμη 5 φορές:
- **2x41** χαρακτήρες, την οποία θα διαχειρίζεται ο δείκτης `char **all`.
- **2x16** χαρακτήρες, την οποία θα διαχειρίζεται ο δείκτης `char **nm`.
- **2x16** χαρακτήρες, την οποία θα διαχειρίζεται ο δείκτης `char **nm_new`.
- **2x26** χαρακτήρες, την οποία θα διαχειρίζεται ο δείκτης `char **sr`.
- **2x26** χαρακτήρες, την οποία θα διαχειρίζεται ο δείκτης `char **sr_new`.

3. Θα δέχεται από το πληκτρολόγιο τα ονοματεπώνυμα δύο φοιτητών, έως **40** χαρακτήρων το καθένα και δοσμένα με κεφαλαία λατινικά γράμματα, και θα τα αποθηκεύει στη μνήμη μέσω του `all`.

4. Θα διαχωρίζει τα μικρά ονόματα από τα επώνυμα και θα τα αποθηκεύει σε δύο ξεχωριστούς δισδιάστατους πίνακες χαρακτήρων. Τον πίνακα για τα μικρά ονόματα θα τον διαχειρίζεται ο **`**nm`** και τον πίνακα για τα επώνυμα ο **`**sr`**. Για τον διαχωρισμό θα αναπτυχθεί η συνάρτηση **`void separate(char **pall, char **pnm, char **psr)`**, η οποία θα δέχεται ως ορίσματα τους **`all`**, **`nm`**, **`sr`**, θα επιτελεί τον ζητούμενο διαχωρισμό και μέσω της κλήσης κατ' αναφορά (ο **`pnm`** αντιστοιχίζεται στον **`nm`** και ο **`psr`** αντιστοιχίζεται στον **`sr`**) στους **`nm`** και **`sr`** θα αποδίδονται τα μικρά ονόματα και τα επώνυμα, αντίστοιχα.

5. Θα ταξινομεί αλφαβητικά τα επώνυμα και θα αναδιατάσσει τόσο τα μικρά ονόματα όσο και τα επώνυμα σε δύο νέους πίνακες, τους οποίους θα διαχειρίζονται οι **`**nm_new`** και **`**sr_new`**. Θα πρέπει να ληφθεί μέριμνα, ώστε, εάν το ένα επώνυμο είναι υποσύνολο του άλλου (π.χ. **XATZISAVVAS** και **XATZIS**), να ταξινομείται ως πρώτο το συντομότερο εξ αυτών.

Για την κατάταξη θα αναπτυχθεί η συνάρτηση **`void sort(char **pnm, char **psr, char **pnm_new, char **psr_new)`**, η οποία θα δέχεται ως ορίσματα τους **`nm`**, **`sr`**, **`nm_new`**, **`sr_new`**, θα επιτελεί τη ζητούμενη κατάταξη κατ' αλφαβητική σειρά, και μέσω της κλήσης κατ' αναφορά (ο **`pnm_new`** αντιστοιχίζεται στον **`nm_new`** και ο **`psr_new`** αντιστοιχίζεται στον **`sr_new`**) οι **`nm_new`** και **`sr_new`** θα λαμβάνουν τα μικρά ονόματα και τα επώνυμα, αντίστοιχα, μετά την κατάταξη.

6. Θα εμφανίζει τους πίνακες **`nm`**, **`sr`**, **`nm_new`**, **`sr_new`** στην οθόνη.

7. Για την απελευθέρωση της δεσμευθείσας μνήμης θα αναπτυχθεί η συνάρτηση **`void free_2(char **deiktis, int size1)`**, η οποία θα δέχεται το όνομα ενός διπλού δείκτη σε χαρακτήρα και την πρώτη διάσταση του πίνακα, στον οποίο αυτός δείχνει, και με χρήση της συνάρτησης **`free()`** θα απελευθερώνει την αντίστοιχη μνήμη. Για παράδειγμα, η κλήση της συνάρτησης **`free_2(all,2);`** απελευθερώνει τη μνήμη που δεσμεύτηκε με τον δείκτη **`all`**. Αντίστοιχες κλήσεις της **`free_2()`** θα οδηγήσουν στην αποδέσμευση της μνήμης που δεσμεύτηκε στο βήμα (2) με τους διπλούς δείκτες **`nm`**, **`sr`**, **`nm_new`**, **`sr_new`**.

Δίνεται ότι:

- Τα ονοματεπώνυμα δόθηκαν σωστά, με κεφαλαία λατινικά γράμματα, και δεν απαιτείται έλεγχος γι' αυτό.
- Όταν ο χρήστης πληκτρολογεί ένα ονοματεπώνυμο, διαχωρίζει το όνομα από το επώνυμο με απλό κενό.
- Κάθε μικρό όνομα και επώνυμο είναι απλό, δεν περιέχει διπλά ονόματα, τίτλους ευγενείας ή άλλου είδους προσφωνήσεις.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

char **alloc_2_char(int size1, int size2);
void free_2_char(char **deikt, int size1);
void separate(char **pall, char **pnm, char **psr);
void sort(char **pnm, char **psr, char **pnm_new, char **psr_new);

int main()
{
  char **all,**nm,**nm_new,**sr,**sr_new;
  int i,j;

  all=alloc_2_char(2,41);
  nm=alloc_2_char(2,16);
  nm_new=alloc_2_char(2,16);
  sr=alloc_2_char(2,26);
  sr_new=alloc_2_char(2,26);

  printf( "Give first name:  " );
  gets(all[0]);
  printf( "\nGive second name:  " );
```

```
  gets(all[1]);

  separate(all,nm,sr);
  for (i=0;i<2;i++)
     printf( "\nnm[%d]=%s\tsr[%d]=%s",i,nm[i],i,sr[i] );
  sort(nm,sr,nm_new,sr_new);
  for (i=0;i<2;i++)
  printf("\nnm_new[%d]=%s\tsr_new[%d]=%s",i,nm_new[i],i,sr_new[i] );

  free_2_char(sr_new,2);
  free_2_char(sr,2);
  free_2_char(nm_new,2);
  free_2_char(nm,2);
  free_2_char(all,2);

  return 0;
}

char **alloc_2_char(int size1, int size2)
{
  int i;
  char **deikt;
  deikt=(char **)malloc(size1*sizeof(char *));
  assert(deikt!=NULL);
  for (i=0;i<size1;i++)
  {
    deikt[i]=(char *)malloc(size2*sizeof(char));
    assert(deikt[i]!=NULL);
  }
  return(deikt);
}

void free_2_char(char **deikt, int size1)
{
  int i;
  for (i=(size1-1);i>=0;i--) free(deikt[i]);
  free(deikt);
}

void separate(char **pall, char **pnm, char **psr)
{
  int i,j;

  for (i=0;i<2;i++)
  {
    j=0;
    while (pall[i][j]!=' ')
    {
      pnm[i][j]=pall[i][j];
      j++;
    }
    pnm[i][j]='\0';

    j=0;
    do
```

```
    {
      psr[i][j]=pall[i][strlen(pnm[i])+j];
      j++;
    } while (pall[i][j]!='\0');
  }
}

void sort(char **pnm, char **psr, char **pnm_new, char **psr_new)
{
  int i,j,mikos,count=0;
  mikos=(strlen(psr[0])<strlen(psr[1])) ? strlen(psr[0]) :
     strlen(psr[1]);
  i=0;
  do
  {
    if (psr[0][i]<psr[1][i]) count++;
    i++;
  } while ((i<mikos) && (!count));
  if ((count) || (strlen(psr[0])>strlen(psr[1])))
  {
    strcpy(psr_new[0],psr[1]);
    strcpy(psr_new[1],psr[0]);
    strcpy(pnm_new[0],pnm[1]);
    strcpy(pnm_new[1],pnm[0]);
  }
  else for (i=0;i<2;i++)
  {
    strcpy(psr_new[i],psr[i]);
    strcpy(pnm_new[i],pnm[i]);
  }
}
```

```
Give first name:        JOAN JAMESON

Give second name:       ANDREW DOW

nm[0]= JOAN                 sr[0]= JAMESON
nm[1]= ANDREW               sr[1]= DOW
nm_new[0]= ANDREW           sr[0]= DOW
nm_new[1]= JOAN                  sr_new[1]= JAMESON
```

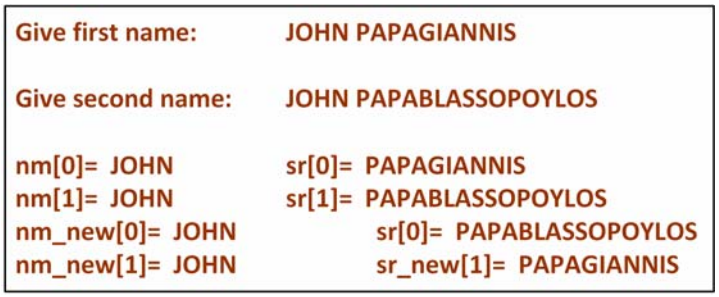
(α)

```
Give first name:        JOHN HATZISAVVAS

Give second name:       JOHN HATZIS

nm[0]= JOHN                 sr[0]= HATZISAVVAS
nm[1]= JOHN                 sr[1]= HATZIS
nm_new[0]= JOHN             sr[0]= HATZIS
nm_new[1]= JOHN             sr_new[1]= HATZISAVVAS
```

(β)



(γ)

Εικόνα 7.6 *Τρία στιγμιότυπα της εξόδου του προγράμματος του παραδείγματος 7.7.2*

7.8. Παράδειγμα ανάπτυξης προγράμματος

Στην παρούσα ενότητα παρουσιάζεται ένα παράδειγμα ανάπτυξης προγράμματος βάσει των αρχών του διαδικαστικού προγραματισμού. Συγκεκριμένα, θα καταστρωθεί ένα πρόγραμμα επεξεργασίας τετραγωνικών πινάκων, το οποίο θα επιτελεί τα ακόλουθα:

1. Θα δέχεται από το πληκτρολόγιο τη διάσταση **n** ενός τετραγωνικού πίνακα **A** διαστάσεων **nxn**, ο οποίος θα περιέχει πραγματικούς αριθμούς. Η διάσταση του πίνακα πρέπει να είναι μεγαλύτερη ή ίση του **3**.

2. Θα λαμβάνει από το πληκτρολόγιο τις τιμές των στοιχείων του πίνακα και θα εμφανίζει τα περιεχόμενα του πίνακα στην οθόνη.

3. Θα βρίσκει σε κάθε γραμμή το στοιχείο με τη μέγιστη απόλυτη τιμή.

4. Για κάθε γραμμή του πίνακα θα απεικονίζει στην οθόνη την απόλυτη τιμή του μέγιστου στοιχείου και τη στήλη, στην οποία βρίσκεται το στοιχείο αυτό.

5. Θα υπολογίζει το άθροισμα των στοιχείων της κύριας διαγωνίου (ίχνος του πίνακα).

6. Θα υπολογίζει τον πίνακα που θα προκύψει ,εάν αντιμετατεθούν η δεύτερη με την τρίτη στήλη του πίνακα και, στη συνέχεια, η πρώτη με την τρίτη γραμμή. Ο προκύπτων πίνακας θα εμφανίζεται στην οθόνη.

Με βάση τις προδιαγραφές, οι λειτουργίες του προγράμματος είναι οι ακόλουθες:

i. ανάγνωση του μεγέθους του πίνακα,

ii. δημιουργία πίνακα,

iii. ανάγνωση των στοιχείων του πίνακα,

iv. εκτύπωση των στοιχείων του πίνακα,

v. για κάθε γραμμή του πίνακα, εξαγωγή του στοιχείου με τη μέγιστη απόλυτη τιμή και εμφάνιση αυτού και της θέσης του στην οθόνη,

vi. υπολογισμός του ίχνους του πίνακα,

vii. αντιμετάθεση δύο στηλών του πίνακα,

viii. αντιμετάθεση δύο γραμμών του πίνακα.

Λαμβάνοντας υπόψη τις προδιαγραφές και τις λειτουργίες του προγράμματος, η κατάστρωσή του ακολουθεί τα εξής βήματα:

- Εφόσον στις προδιαγραφές αναφέρεται ότι ο πίνακας θα περιέχει πραγματικούς αριθμούς, αυτοί θα προσεγγιστούν με αριθμούς κινητής υποδιαστολής απλής ακριβείας, `float`.
- Είσοδοι του προγράμματος είναι η διάσταση του πίνακα και τα στοιχεία του πίνακα.
- Έξοδοι του προγράμματος (στην οθόνη) είναι τα στοιχεία του πίνακα, το ίχνος του πίνακα και τα στοιχεία με τη μέγιστη τιμή για κάθε γραμμή του πίνακα.
- Για την υλοποίηση της λειτουργίας **(i)** δημιουργείται μία συνάρτηση `int getSize(void)`, η οποία θα δέχεται από το πληκτρολόγιο έναν ακέραιο αριθμό που θα εκφράζει τη διάσταση του τετραγωνικού πίνακα.

Στη συνάρτηση θα υπάρχει κατάλληλη δομή επανάληψης που θα διασφαλίζει ότι η διάσταση του πίνακα θα είναι μεγαλύτερη ή ίση του **3**. Η δοθείσα διάσταση θα επιστρέφει στη **main()**:

```
int getSize(void)
{
  int size;
  do
  {
    printf(" Give the size of the array (>=3):  " );
    scanf("%d",&size);
  } while (size<3);

  return size;
}
```

- Εφόσον η διάσταση του τετραγωνικού πίνακα δίνεται κατά τον χρόνο εκτέλεσης του προγράμματος, για την υλοποίηση της λειτουργίας **(ii)** θα χρησιμοποιηθεί δυναμικός πίνακας δύο διαστάσεων, με το όνομα **pArr**. Προς τούτο, θα ενταχθούν στο πρόγραμμα οι συναρτήσεις **allocate_2()** και **free_2()** του παραδείγματος 7.7.1.
- Για την υλοποίηση της λειτουργίας **(iii)** δημιουργείται μία συνάρτηση **void getData(float **ptr, int size)**, η οποία θα καλείται από τη **main()** με την πρόταση

```
getData(pArr, size);
```

και θα διαβάζει από το πληκτρολόγιο τιμές για τον πίνακα **pArr** μέσω του τοπικού διπλού δείκτη **ptr**:

```
void getData(float **ptr, int size)
{
  int i,j;
  for (i=0;i<size;i++)
     for (j=0;j<size;j++)
     {
       printf("\nA[%d][%d]:  ",i+1,j+1);
       scanf("%f",&ptr[i][j]);
     }
}
```

- Εφόσον η λειτουργία **(iv)** απαιτείται σε περισσότερα του ενός σημεία του προγράμματος, θα υλοποιηθεί μέσω της συνάρτησης **void printData(float **ptr, int size)**:

```
void printData(float **ptr, int size)
{
  int i,j;
  for (i=0;i<size;i++)
  {
     printf("\n");
     for (j=0;j<size;j++)  printf("\t%10.4f",ptr[i][j]);
  }
}
```

- Για να υλοποιηθούν η λειτουργία **(v)** και η προδιαγραφή **4**, θα πρέπει να δημιουργηθεί επαναληπτική πρόταση **for**, που θα επαναλαμβάνεται για όλες τις γραμμές του πίνακα. Σε κάθε επανάληψη **i** θα αναζητείται το στοιχείο αντίστοιχης γραμμής του πίνακα με τη μέγιστη απόλυτη τιμή, το οποίο και θα εμφανίζεται στην οθόνη, μαζί με τη στήλη στην οποία βρίσκεται. Προς τούτο θα καλείται η συνάρτηση

```
void getMax(float **pArray, int size, int i);
```

η οποία θα δέχεται τον αριθμό της γραμμής **i** του πίνακα, στην οποία θα γίνει η αναζήτηση του μέγιστου στοιχείου. Η εκτύπωση των ζητουμένων θα γίνεται μέσα από τη συνάρτηση. Η συνάρτηση **fabs()**, που παρέχει την απόλυτη τιμή **float** αριθμών, βρίσκεται στο αρχείο κεφαλίδας **math.h**.

```
void getMax(float **pArray, int size, int i)
{
  int j,col;
  float maxim;
  maxim=fabs(pArray[i][0]);
  col=0;
  for (j=1;j<size;j++)
     if (fabs(pArray[i][j])>maxim)
     {
        maxim=fabs(pArray[i][j]);
        col=j;
     }
  printf( "\nLine %d: column %d, size=%f",i+1,col+1,fabs(pArray[i]
[col]) );
}
```

- Το ίχνος του πίνακα (λειτουργία **vi**) είναι το άθροισμα των στοιχείων της κύριας διαγωνίου. Θα υλοποιηθεί με τη συνάρτηση

```
float trace(float **pArray, int size);
```

η οποία επιστρέφει το ζητούμενο ίχνος. Ο κώδικάς της είναι ο ακόλουθος:

```
float trace(float **pArray, int size)
{
  int i;
  float trc=0.0;
  for (i=0;i<size;i++)
     trc=trc+pArray[i][i];

  return trc;
}
```

- Για την υλοποίηση της λειτουργίας (**vii**) δημιουργείται η συνάρτηση

```
void permuteColumns(float **pArray, int size, int column1, int col-
umn2);
```

Οι αριθμοί των δύο στηλών, που θα αντιμετατεθούν, αποθηκεύονται στις μεταβλητές **column1** και **column2**. Ο κώδικας της συνάρτησης δίνεται ακολούθως:

```
void permuteColumns(float **pArray, int size, int column1, int col-
umn2)
{
  int j;
  float temp;
  for (j=0;j<size;j++)
  {
    temp=pArray[j][column1];
    pArray[j][column1]=pArray[j][column2];
    pArray[j][column2]=temp;
  }
}
```

- Κατ' αντιστοιχία με τη συνάρτηση **permuteColumns()**, η λειτουργία της αντιμετάθεσης γραμμών γίνεται με τη συνάρτηση

```
void permuteLines(float **pArray, int size, int line1, int line2);
```

η οποία προκύπτει από τη **permuteColumns()** με αντικατάσταση των γραμμών με τις στήλες:

```
void permuteLines(float **pArray, int size, int line1, int line2)
{
  int j;
  float temp;
  for (j=0;j<size;j++)
  {
    temp=pArray[line1][j];
    pArray[line1][j]=pArray[line2][j];
    pArray[line2][j]=temp;
  }
}
```

Ενοποιώντας τα ανωτέρω βήματα, το συνολικό πρόγραμμα επεξεργασίας τετραγωνικών πινάκων λαμβάνει την ακόλουθη μορφή:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>

int getSize(void);
float **allocate_2(int size1, int size2);
void free_2(float **parr, int size1);
void getData(float **ptr, int size);
void printData(float **ptr, int size);
void getMax(float **pArray, int size, int i);
float trace(float **pArray, int size);
void permuteColumns(float **pArray, int size, int column1, int col-
umn2);
void permuteLines(float **pArray, int size, int line1, int line2);

int main()
{
  int i,j,col,size;
  float **pArr;

  size=getSize();
  pArr=allocate_2(size,size);

  getData(pArr,size);
  printf("\n\nInitial array A:");
  printData(pArr,size);

  for (i=0;i<size;i++)
     getMax(pArr,size,i);
  printf("\n\nTrace(A)=%f\n",trace(pArr,size));
  permuteColumns(pArr,size,1,2);
  permuteLines(pArr,size,0,2);

  printf("\n\nFinal array:");
  printData(pArr,size);
```

```
  free_2(pArr,size);

  return 0;
}
/*-----------------------------------*/
int getSize(void)
{
  int size;
  do
  {
    printf(" Give the size of the array (>=3):  " );
    scanf("%d",&size);
  } while (size<3);

  return size;
}
/*-----------------------------------*/
float **allocate_2(int size1, int size2)
{
  int i;
  float **parr;
  pdeikt=(float **)malloc(size1*sizeof(float *));
  assert(parr!=NULL);
  for (i=0;i<size1;i++)
  {
    parr[i]=(float *)malloc(size2*sizeof(float));
    assert(parr[i]!=NULL);
  }

  return(parr);
}
/*-----------------------------------*/
void free_2(float **parr, int size1)
{
  int i;
  for (i=(size1-1);i>=0;i--) free(parr[i]);
  free(parr);
}
/*-----------------------------------*/
void getData(float **ptr, int size)
{
  int i,j;
  for (i=0;i<size;i++)
     for (j=0;j<size;j++)
     {
        printf("\nA[%d][%d]:  ",i+1,j+1);
        scanf("%f",&ptr[i][j]);
     }
}
/*-----------------------------------*/
void printData(float **ptr, int size)
{
  int i,j;
  for (i=0;i<size;i++)
```

```
        {
           printf("\n");
           for (j=0;j<size;j++)  printf("\t%10.4f",ptr[i][j]);
        }
      }
      /*-------------------------------------*/
      void getMax(float **pArray, int size, int i)
      {
        int j,col;
        float maxim;
        maxim=fabs(pArray[i][0]);
        col=0;
        for (j=1;j<size;j++)
           if (fabs(pArray[i][j])>maxim)
           {
              maxim=fabs(pArray[i][j]);
              col=j;
           }
        printf( "\nLine %d: column %d, size=%f",i+1,col+1,fabs(pArray[i]
[col]) );
      }
      /*-------------------------------------*/
      float trace(float **pArray, int size)
      {
        int i;
        float trc=0.0;
        for (i=0;i<size;i++)
           trc=trc+pArray[i][i];

        return trc;
      }
      /*-------------------------------------*/
      void permuteColumns(float **pArray, int size, int column1, int col-
umn2)
      {
        int j;
        float temp;
        for (j=0;j<size;j++)
        {
          temp=pArray[j][column1];
          pArray[j][column1]=pArray[j][column2];
          pArray[j][column2]=temp;
        }
      }
      /*-------------------------------------*/
      void permuteLines(float **pArray, int size, int line1, int line2)
      {
        int j;
        float temp;
        for (j=0;j<size;j++)
        {
          temp=pArray[line1][j];
          pArray[line1][j]=pArray[line2][j];
          pArray[line2][j]=temp;
        }
```

```
}
```

```
Give the size of the array:     3

A[1][1]:        23.1

A[1][2]:        -32

A[1][3]:        5.7

A[2][1]:        1.15

A[2][2]:        -0.02

A[2][3]:        0.17

A[3][1]:        3.41

A[3][2]:        4.2

A[3][3]:        -1

Line 1:  column:        2,      size=32.000000
Line 2:  column:        1,      size=1.150000
Line 3:  column:        2,      size=4.200000

Trace(A)=22.080000


Initial array:
23.1000-32.00005.70001.1500-0.02000.17003.41004.2000-1.0000
Final array:
3.4100-1.00004.20001.15000.17000.020023.10005.7000-32.0000
```

Εικόνα 7.4 *Η έξοδος του προγράμματος*

Συμπέρασμα: Με βάση τον παραπάνω σχεδιασμό, το πρόβλημα έχει δομηθεί σε μία σειρά διαδικασιών, καθεμία εκ των οποίων υλοποιείται μέσω μίας συνάρτησης. Κατ' αυτόν τον τρόπο η συνάρτηση `main()` διαδραματίζει επιτελικό ρόλο, κατανέμοντας το έργο στις συναρτήσεις. Επιπρόσθετα, αφενός μεν επιτεύχθηκε επαναχρησιμοποίηση κώδικα (π.χ. οι συναρτήσεις `allocate_2()` και `free_2()`), αφετέρου δημιουργήθηκαν συναρτήσεις ανάγνωσης/εκτύπωσης τετραγωνικού πίνακα ή υπολογισμού του ίχνους του πίνακα, οι οποίες μπορούν να ενταχθούν αυτούσιες σε πλήθος προγραμμάτων.

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Ο αναγνώστης καλείται να επιλέξει μία από τις τέσσερις απαντήσεις.

(1) Ποια από τις ακόλουθες προτάσεις είναι λανθασμένη;

(α) Η συνάρτηση **`calloc()`** χρησιμοποιείται για την αλλαγή του μεγέθους της εκχωρούμενης μνήμης κατά την εκτέλεση ενός προγράμματος.
(β) Η συνάρτηση **`realloc()`** χρησιμοποιείται για την αλλαγή του μεγέθους της εκχωρούμενης μνήμης κατά την εκτέλεση ενός προγράμματος.
(γ) Η συνάρτηση **`free()`** χρησιμοποιείται για την απελευθέρωση μνήμης κατά την εκτέλεση ενός προγράμματος.
(δ) Η συνάρτηση **`malloc()`** χρησιμοποιείται για τον καθορισμό του μεγέθους της εκχωρούμενης μνήμης κατά την εκτέλεση ενός προγράμματος.

(2) Ποια από τις ακόλουθες ιδιότητες της συνάρτησης **`malloc()`** είναι λανθασμένη;
(α) Επιστρέφει έναν δείκτη στην αρχή του μπλοκ μνήμης, στο οποίο γίνεται η εκχώρηση.
(β) Δεσμεύει ένα μπλοκ διαδοχικών θέσεων μνήμης.
(γ) Ορίζεται στο αρχείο κεφαλίδας **`limits.h`**.
(δ) Εάν δεν υπάρχει διαθέσιμη μνήμη, επιστρέφει **NULL**, δηλαδή τη διεύθυνση **0**.

(3) Ποια από τις ακόλουθες ιδιότητες της συνάρτησης **`realloc()`** είναι λανθασμένη;
(α) Χρησιμοποιείται για τη διεύρυνση ή συρρίκνωση ενός ήδη δεσμευμένου μπλοκ μνήμης.
(β) Δηλώνεται ως **`void *realloc(void *block, int size);`**.
(γ) Επιστρέφει έναν δείκτη στο νέο τμήμα μνήμης που δεσμεύεται.
(δ) Δεν διασφαλίζει τα υπάρχοντα περιεχόμενα στη μνήμη.

(4) Ποιο από τα συμπεράσματα, που αφορούν στη λειτουργία του ακόλουθου προγράμματος, είναι λανθασμένο;

```
#include <stdio.h>
#include <alloc.h>
int  main()
{
  int *pstr,*pstr2;
  pstr=(int *)malloc(10*sizeof(int));
  *pstr=35;
  *(pstr+1)=-12;
  pstr2=(int *)realloc(pstr, 20*sizeof(int));
  free(pstr2);
  free(pstr);
  return 0;
}
```

(α) Οι **`pstr`**, **`pstr2`** είναι δείκτες που δείχνουν σε ακεραίους.
(β) Η **`malloc()`** δεσμεύει ένα μπλοκ μνήμης για **10** ακεραίους.
(γ) Ο **`pstr`** αποθηκεύεται στη διεύθυνση **35**.
(δ) Η **`realloc()`** χρησιμοποιείται για τη διεύρυνση του μπλοκ μνήμης στις **20** τετράδες bytes.

(5) Ποιο από τα συμπεράσματα, που αφορούν στη λειτουργία του ακόλουθου προγράμματος, είναι λανθασμένο;

```
#include<stdio.h>
int  main()
{
  int i;
  char *name[4]={"John","Jacob","Jack","James"};
  char *tmp;
  tmp=name[0];
  name[0]=name[2];
  name[2]=tmp;
```

```
    return 0;
}
```

(α) Δημιουργείται ο πίνακας δεικτών `name` με στοιχεία τους δείκτες σε χαρακτήρα, `name[0]`, `name[1]`, `name[2]` , `name[3]`.
(β) Σε κάθε δείκτη αποδίδεται η διεύθυνση του πρώτου byte ενός αλφαριθμητικού, τα οποία αποθηκεύονται σε διαφορετικό τμήμα της μνήμης.
(γ) Με τη βοήθεια του δείκτη χαρακτήρα `temp` ανταλλάσονται οι διευθύνσεις των `name[0]` και `name[2]`.
(δ) Στο τέλος του προγράμματος ο `name[0]` δείχνει στο `"James"`.

Ασκήσεις

Άσκηση 1

Να γραφεί πρόγραμμα, το οποίο θα δημιουργεί δυναμικό πίνακα ακέραιων αριθμών διαστάσεων 4x3x6x2, θα διαβάζει από το πληκτρολόγιο τις τιμές των στοιχείων του και θα αθροίζει τις μη μηδενικές εξ αυτών. Πριν το πέρας του προγράμματος θα γίνεται αποδέσμευση της μνήμης.

Άσκηση 2

Να γραφεί πρόγραμμα, στο οποίο αρχικά θα δεσμεύεται ο απαραίτητος χώρος για την καταχώρηση 50 ακέραιων αριθμών. Ακολούθως, θα δέχεται αριθμούς από το πληκτρολόγιο, οι οποίοι θα καταχωρούνται στις δεσμευθείσες θέσεις. Η διαδικασία θα σταματά είτε όταν πληκτρολογηθεί το 0 είτε όταν γεμίσει η δεσμευθείσα μνήμη. Οι αριθμοί που πληκτρολογήθηκαν, θα εμφανίζονται στην οθόνη διαχωριζόμενοι με κόμμα και το πρόγραμμα θα ολοκληρώνεται με απελευθέρωση της δεσμευθείσας μνήμης.

Άσκηση 3

Να γραφεί πρόγραμμα, στο οποίο θα δεσμεύεται μνήμη για δύο δισδιάστατους πίνακες ακεραίων *B1* και *B2* διαστάσεων 3x3. Η δέσμευση της μνήμης θα γίνεται δυναμικά κατά τον χρόνο εκτέλεσης του προγράμματος, με κλήση κατάλληλης συνάρτησης `int **allocB(int size1, int size2)`. Η δεσμευθείσα μνήμη θα αποδεσμεύεται στο τέλος της εκτέλεσης του προγράμματος, με κλήση κατάλληλης συνάρτησης `void frB(int **deikt, int size1)`.
Θα καλείται η συνάρτηση `void getB(int **pB1, int **pB2)`, η οποία θα αποδίδει από το πληκτρολόγιο τιμές στους *B1* και *B2*.

Άσκηση 4

Στο πρόγραμμα της Άσκησης 3 να προστεθεί η ακόλουθη λειτουργία:
Θα καλείται η συνάρτηση `void substi(int **pB1, int **pB2)`, η οποία θα αντιγράφει την απόλυτη τιμή κάθε στοιχείου του *B1* στην αντίστοιχη θέση του *B2*. Ακολούθως, μέσα από τη `main()` θα εμφανίζονται στην οθόνη τα στοιχεία των πινάκων *B1* και *B2*.

Άσκηση 5

Να τροποποιηθεί το πρόγραμμα της Άσκησης 3 ώστε να δημιουργούνται τρεις πίνακες με δεδομένα αριθμούς κινητής υποδιαστολής διπλής ακριβείας και να προστεθεί η ακόλουθη λειτουργία:
Θα καλείται η συνάρτηση `void sumRev(double **pB1, double **pB2, double **pB3)`, η οποία για κάθε στοιχείο των *B1* και *B2* θα υπολογίζει το αντίστροφο του αθροίσματός τους, το οποίο και θα αναθέτει στο αντίστοιχο στοιχείο του *B3* (δηλαδή με χρήση μαθηματικού φορμαλισμού, $B3_{ij} = \frac{1}{B1_{ij} + B2_{ij}}$). Ακολούθως, μέσα από τη `main()` θα εμφανίζονται στην οθόνη τα στοιχεία των τριών πινάκων.

Άσκηση 6

Να γραφεί πρόγραμμα το οποίο θα επιτελεί τα ακόλουθα:

(α) **Θα ζητά από τον χρήστη τις διαστάσεις δύο δισδιάστατων πινάκων, με μέγιστη διάσταση μικρότερη ή ίση του 10.**
(β) **Θα δημιουργεί τους δύο ανωτέρω δυναμικούς πίνακες και θα λαμβάνει τις αριθμητικές τιμές των θέσεών τους από το πληκτρολόγιο. Θα γίνονται δεκτοί μόνο θετικοί πραγματικοί αριθμοί.**
(γ) **Θα καλείται η συνάρτηση `void **addArrays(float **pArr1, **pArr2, int size1, int size2)`, η οποία θα λαμβάνει με κλήση κατ' αναφορά τους δύο πίνακες, καθώς και τις διαστάσεις τους και θα επιστρέφει στη `main()` το άθροισμα των δύο πινάκων.**
(δ) **Οι δύο πίνακες και το άθροισμά τους θα εμφανίζονται στην οθόνη.**

Άσκηση 7
Να γραφεί πρόγραμμα το οποίο θα επιτελεί τα ακόλουθα:
(α) **Θα ζητά από τον χρήστη τις διαστάσεις k, n, m δύο δισδιάστατων πινάκων ακεραίων με διαστάσεις kxn και nxm.**
(β) **Θα δημιουργεί τους δύο ανωτέρω δυναμικούς πίνακες.**
(γ) Θα λαμβάνει τις αριθμητικές τιμές των θέσεών τους από το πληκτρολόγιο με χρήση της συνάρτησης `void getData(int **pArr, int size1, int size2)`, όπου ο `pArr` είναι διπλός δείκτης για τη διαχείριση πίνακα, `size1` και `size2` είναι οι διαστάσεις του εκάστοτε πίνακα.
(γ) **Θα καλείται η συνάρτηση `void **multiplyArrays(float **pArr1, **pArr2, int size1, int size2, int size3)`, η οποία θα λαμβάνει με κλήση κατ' αναφορά τους δύο πίνακες, καθώς και τις διαστάσεις τους και θα επιστρέφει στη `main()` το γινόμενο των δύο πινάκων.**
(δ) **Οι δύο πίνακες και το άθροισμά τους θα εμφανίζονται στην οθόνη.**

Βιβλιογραφία κεφαλαίου

Καράκος, Αλ. (2010), *Αλγοριθμική Επίλυση Ασκήσεων με τη Γλώσσα Προγραμματισμού C*.

Καρολίδης, Δ. (2013), *Μαθαίνετε Εύκολα C*, αυτοέκδοση.

Τσελίκης, Γ. & Τσελίκας, Ν. (2012), *C από τη Θεωρία στην Εφαρμογή*, 2η έκδοση.

Χατζηγιαννάκης, Ν. (2012), *Η Γλώσσα C σε Βάθος*, 4η Έκδοση, Εκδόσεις Κλειδάριθμος.

Reese, R. (2013), *Understanding and Using C Pointers*, O'Reilly.

Roberts, E. (2008), *Η Τέχνη και Επιστήμη της C*, Εκδόσεις Κλειδάριθμος.

Topo, N. & Dewan, H. (2013), *Pointers in C – A Hands on Approach*, Apress.