ΑΝΩΤΑΤΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΣΕΡΡΩΝ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΕΠΙΚΟΙΝΩΝΙΩΝ

Προγραμματισμός ΙΙ

Δρ. Πάρις Αστ. Μαστοροκώστας
Αναπληρωτής Καθηγητής

Τομέας Υπολογιστικών Τεχνικών & Συστημάτων

Σέρρες 2007

Περιεχόμενα

Κατάλογος Σχημάτων v

1. Δομές - απαριθμητικοί τύποι δεδομένων 1-1

1.1 Απαριθμητικοί τύποι δεδομένων 1-1

Παράδειγμα 1.1 1-3

1.2 Η λέξη κλειδί typedef 1-3

1.3 Ορισμός δομής - δήλωση μεταβλητών 1-4

Παράδειγμα 1.2 1-7

1.4 Απόδοση αρχικών τιμών στις δομές 1-9

1.5 Αναφορά στα μέλη δομής 1-10

1.6 Ένθεση δομών 1-10

Παράδειγμα 1.3 1-11

Παράδειγμα 1.4 1-14

2. Συναρτήσεις 2-1

2.1 Οι έννοιες του αρθρωτού σχεδιασμού και της συνάρτησης 2-1

2.2 Βασικά στοιχεία συναρτήσεων 2-1

2.2.1 Δήλωση συνάρτησης 2-2

2.2.2 Ορισμός συνάρτησης 2-3

2.2.3 Κλήση συνάρτησης 2-4

Παράδειγμα 2.1 2-5

Παράδειγμα 2.2 2-6

Παράδειγμα 2.3 2-7

2.3 Είδη και εμβέλεια μεταβλητών 2-8

2.3.1 Τοπικές μεταβλητές 2-8

Παράδειγμα 2.4 2-9

2.3.2 Καθολικές μεταβλητές 2-10

Παράδειγμα 2.5 2-10

2.3.3 Εμβέλεια μεταβλητών 2-11
Παράδειγμα 2.6 2-12
2.3.4 Διάρκεια μεταβλητών 2-14
Παράδειγμα 2.7 2-15
Παράδειγμα 2.8 2-15
2.4 Πίνακες ως παράμετροι συναρτήσεων 2-17
Παράδειγμα 2.9 2-18
2.5 Αναδρομικές συναρτήσεις 2-20
Παράδειγμα 2.10 2-23
3. Δείκτες 3-1
3.1 Η έννοια του δείκτη 3-1
3.2 Δήλωση δείκτη 3-1
3.3 Ανάθεση τιμής σε δείκτη 3-3
3.4 Προσπέλαση μεταβλητής με χρήση δείκτη 3-5
Παράδειγμα 3.1 3-6
3.5 Δείκτες και συναρτήσεις 3-11
3.5.1 Μεταβίβαση παραμέτρων 3-11
Παράδειγμα 3.2 3-12
Παράδειγμα 3.3 3-13
Παράδειγμα 3.4 3-14
Παράδειγμα 3.5 3-16
3.5.2 Συναρτήσεις με τύπο επιστροφής δείκτη 3-18
Παράδειγμα 3.6 3-18
3.6 Δείκτες και δομές 3-19
3.6.1 Δείκτες εντός δομών 3-20
Παράδειγμα 3.7 3-21
3.7 Δείκτες και η λέξη κλειδί typedef 3-22
3.8 Ορίσματα της γραμμής διαταγής 3-23
Παράδειγμα 3.8 3-24

4. Δυναμική διαχείριση μνήμης - δείκτες σε αλφαριθμητικά 4-1

4.1 Η έννοια της δυναμικής διαχείρισης μνήμης 4-1

4.2 Οι συναρτήσεις malloc, calloc και free 4-2

Παράδειγμα 4.1 4-5

4.3 Η συνάρτηση realloc 4-6

Παράδειγμα 4.2 4-6

4.4 Πίνακες δεικτών 4-8

Παράδειγμα 4.3 4-8

4.5 Δείκτες σε δείκτες 4-9

4.5.1 Πολυδιάστατοι πίνακες με δεδομένα αριθμητικών τύπων 4-12

Παράδειγμα 4.4 4-14

4.6 Δείκτες και συναρτήσεις αλφαριθμητικών 4-14

4.6.1 Η συνάρτηση εύρεσης χαρακτήρα σε αλφαριθμητικό 4-14

Παράδειγμα 4.5 4-15

4.6.2 Η συνάρτηση εύρεσης αλφαριθμητικού σε αλφαριθμητικό 4-15

Παράδειγμα 4.6 4-16

Παράδειγμα 4.7 4-17

Κεφάλαιο 5: Αρχεία 5-1

5.1 Γενικά 5-1

5.1.1 Τα κανάλια stdin, stdout, stderr 5-1

5.1.2 Η ενδιάμεση μνήμη - δείκτης αρχείου 5-2

5.1.3 Κατηγορίες αρχείων 5-3

5.2 Άνοιγμα - κλείσιμο αρχείου 5-4

5.3 Ανάγνωση - εγγραφή χαρακτήρων σε αρχεία 5-6

5.3.1 Η συνάρτηση εγγραφής χαρακτήρων putc 5-6

Παράδειγμα 5.1 5-6

5.3.2 Η συνάρτηση ανάγνωσης χαρακτήρων getc 5-8

Παράδειγμα 5.2 5-8

Παράδειγμα 5.3 5-9

5.4 Μορφοποιούμενες συναρτήσεις εισόδου - εξόδου σε αρχεία 5-11
5.4.1 Η συνάρτηση fprintf 5-11
Παράδειγμα 5.4 5-11
5.4.2 Η συνάρτηση fscanf 5-13
Παράδειγμα 5.5 5-13
5.5 Ανάγνωση - εγγραφή σε δυαδικά αρχεία 5-15
5.5.1 Η συνάρτηση fread 5-15
Παράδειγμα 5.6 5-16
5.5.2 Η συνάρτηση fwrite 5-17
Παράδειγμα 5.7 5-18
Παράδειγμα 5.8 5-19
5.5.3 Η συνάρτηση feof 5-20
5.6 Τυχαία προσπέλαση δυαδικού αρχείου 5-21
5.6.1 Η συνάρτηση fseek 5-21
Παράδειγμα 5.9 5-22
5.7 Ανάγνωση - εγγραφή χαρακτήρων με χρήση των fread/fwrite 5-24
5.8 Ανάγνωση - εγγραφή γραμμή ανά γραμμή 5-26
Παράδειγμα 5.10 . . . . . 5.26
Παράδειγμα 5.11 . . . . . 5.27
Βιβλιογραφία B-2

Κατάλογος Σχημάτων

Σχήμα 2-1. *Γενική μορφή προγράμματος στη C* 2-2
Σχήμα 3-1. *Απεικόνιση του τρόπου λειτουργίας των δεικτών* 3-2
Σχήμα 3-2. *Ανάθεση τιμής σε δείκτη με χρήση πίνακα* 3-3
Σχήμα 3-3. *Ανάθεση τιμής σε δείκτη με χρήση άλλων δεικτών ίδιου τύπου* 3-4
Σχήμα 3-4. *Ανάθεση τιμής σε δείκτη με χρήση αριθμητικής δεικτών* 3-4
Σχήμα 3-5. *Ανάθεση τιμής σε δείκτη με χρήση του τελεστή διεύθυνσης* . . . 3-5
Σχήμα 3-6. *Προσπέλαση μεταβλητής με χρήση δείκτη* 3-6
Σχήμα 3-7. 3-7
Σχήμα 4-1. *Χάρτης μνήμης* 4-6

1. Δομές - απαριθμητικοί τύποι δεδομένων

1.1 Απαριθμητικοί τύποι δεδομένων

Η C, πέραν των ενσωματωμένων τύπων δεδομένων (**char**, **int**, **float**, **double**), παρέχει τη δυνατότητα στο χρήστη να δημιουργήσει τους δικούς του τύπους, οι οποίοι είναι ενεργοί αποκλειστικά μέσα στο πρόγραμμα που δημιουργούνται. Στην απλούστερη περίπτωση ονομάζονται **απαριθμητικοί τύποι** δεδομένων (enumerated types) ενώ οι πιο σύνθετες μορφές ονομάζονται **δομές** (structures). Οι δομές θα μελετηθούν στη συνέχεια του κεφαλαίου.

Ο απαριθμητικός τύπος ορίζεται στην αρχή του προγράμματος, με χρήση της λέξης κλειδί ***enum***, ως εξής:

enum <όνομα_τύπου> { πεδίο τιμών };

Το πεδίο τιμών του είναι ένα σύνολο από σταθερές, στις οποίες έχουν δοθεί συμβολικά ονόματα. Η σημασία των απαριθμητικών τύπων είναι διττή: αφενός μεν διευκολύνουν την κατανόηση του προγράμματος, αφετέρου δε επιτρέπουν στο μεταγλωττιστή να ελέγχει τους τύπους ώστε να μην προκαλούνται λογικά λάθη. Για παράδειγμα, εάν χρησιμοποιηθεί μία ακέραια μεταβλητή **days** για να επεξεργασθεί πληροφορία σχετική με τις ημέρες της εβδομάδας, θα μπορούσε να αντιστοιχηθεί το **1** στην Κυριακή, το **2** στη Δευτέρα κ.ο.κ. Ωστόσο, στην περίπτωση που ανατεθεί στην **days** το **9**, ο μεταγλωττιστής δε θα εντοπίσει το λάθος, καθώς υπάρχει λογικό και όχι προγραμματιστικό σφάλμα. Για το λόγο αυτό μπορεί να ορισθεί ένας νέος τύπος δεδομένων με το όνομα **week_days**, με πεδίο τιμών επτά συμβολικά ονόματα, καθένα από τα οποία αντιστοιχεί σε μία ημέρα της εβδομάδας:

enum week_days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

Το πεδίο τιμών του τύπου **week_days** είναι αυστηρά καθορισμένο, γεγονός που σημαίνει ότι μία μεταβλητή τύπου **week_days** μπορεί να λάβει μόνο τις ανωτέρω επτά τιμές. Η δήλωση μεταβλητής τύπου **week_days** έχει την ακόλουθη μορφή:

enum week_days days;

<u>Παρατηρήσεις:</u>

1. Εσωτερικά, ο χειρισμός των τύπων δεδομένων ***enum*** γίνεται σαν να ήταν

ακέραιοι (έτσι μπορούν να εκετελεσθούν αριθμητικές και συγκριτικές πράξεις με αυτούς). Στο πρώτο όνομα δίνεται η τιμή **0** (**Sun** στην προηγούμενη περίπτωση), στο επόμενο η τιμή **1** (**Mon**) κ.ο.κ. Μπορεί η αρίθμηση να ξεκινά από άλλον ακέραιο, όπως φαίνεται ακολούθως:

enum week_days { Sun=30, Mon, Tue, Wed, Thu, Fri, Sat };

οπότε η αρίθμηση ξεκινά από το **30**. Ωστόσο, εάν αντί για **days=Sun** ή **Mon** τεθεί π.χ. **days=5**, ο μεταγλωττιστής θα βγάλει μήνυμα σφάλματος.

Τέλος, μπορούν να τεθούν ακέραιες τιμές σε οποιαδήποτε από τα συμβολικά ονόματα, όπως ακολούθως:

enum week_days { Sun, Mon=30, Tue, Wed=500, Thu=1000, Fri, Sat };

οπότε **Sun=0**, **Mon=30**, **Tue=31**, **Wed=500**, **Thu=1000**, **Fri=1001**, **Sat=1002**.

2. Με τον τύπο ***enum*** μπορούν να ορισθούν οι λογικές τιμές της άλγεβρας Boole **true** και **false** ως εξής:

enum boolean {False, True};

οπότε η **False=0** και η **True=1**. Έτσι μπορούν στη συνέχεια να ορισθούν **boolean** μεταβλητές.

Εναλλακτικά τα παραπάνω επιτελούνται με χρήση της πρότασης ***#define*** του προεπεξεργαστή:

#define False 0

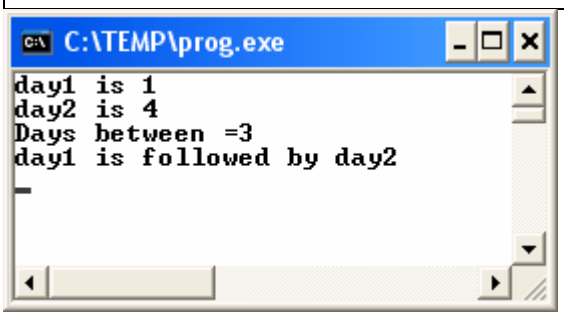
#define True 1

3. Οι μεταβλητές απαριθμητικού τύπου παρουσιάζουν το μειονέκτημα ότι δεν μπορούν να χρησιμοποιηθούν <u>άμεσα</u> στις συναρτήσεις εισόδου–εξόδου (***scanf()***, ***printf()***). Οι συναρτήσεις αυτές τυπώνουν τον ακέραιο στον οποίο αντιστοιχεί το συμβολικό όνομα, όπως φαίνεται στο Παράδειγμα 1.1.

4. Στη C απαιτείται να συμπεριληφθεί στη δήλωση μεταβλητής απαριθμητικού τύπου η λέξη κλειδί ***enum***, π.χ. **enum week_days day1,day2;**. Στη C++ η λέξη ***enum*** δεν είναι απαραίτητη και η δήλωση γίνεται **week_days day1,day2;**, όπως ακριβώς ορίζεται ένας βασικός τύπος δεδομένου, π.χ. **int var1**.

Παράδειγμα 1.1

```
#include <stdio.h>
// Προσδιορισμός τύπου enum
enum week_days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
void main()
{
    week_days day1,day2;      // Ορισμός μεταβλητών
    day1=Mon;  // Απόδοση τιμής. Δεν περικλείεται σε εισαγωγικά.
    day2=Thu;
    int diff=day2-day1;    // Εκτελείται αριθμητική ακεραίων
    printf( "day1 is %d\n",day1 );
    printf( "day2 is %d\n",day2 );
    printf( "Days between =%d\n",diff );
    if (day1<day2)     // Δύναται να κάνει συγκρίσεις
    printf( "day1 is followed by day2\n" );
}
```



1.2 Η λέξη κλειδί typedef

Η C παρέχει τη δυνατότητα απόδοσης νέων ονομάτων σε τύπους δεδομένων. Ο μηχανισμός απόδοσης ονομάτων βασίζεται στη λέξη κλειδί ***typedef***, βρίσκει ιδιαίτερη χρήση στις δομές και έχει την ακόλουθη σύνταξη:

typedef <τύπος> <όνομα>;

Για παράδειγμα, η δήλωση

typedef float real_number;

καθιστά το όνομα **real_number** συνώνυμο του **float**. Ο τύπος **real_number** μπορεί πλέον να χρησιμοποιηθεί σε δηλώσεις, μετατροπές τύπων κ.λ.π., όπως ακριβώς χρησιμοποιείται ο τύπος **float**, με τη διαφορά ότι ο **real_number** θα είναι ενεργός αποκλειστικά μέσα στο πρόγραμμα που δημιουργείται[1]. Θα πρέπει να σημειωθεί ότι με την ***typedef*** δε δημιουργούνται νέοι τύποι, απλά αλλάζουν οι ετικέτες. Έτσι, η παρακάτω δήλωση

real_number num1, num2;

δηλώνει τις μεταβλητές κινητής υποδιαστολής **num1** και **num2**.

1.3 Ορισμός δομής - δήλωση μεταβλητών

Η δομή αποτελεί έναν συναθροιστικό τύπο δεδομένων, οριζόμενο από το χρήστη, και χρησιμοποιείται είτε για να αναπαρασταθεί μία έννοια που μπορεί να διαθέτει διαφορετικού τύπου επιμέρους ιδιότητες είτε για να ομαδοποιηθούν διαφορετικού τύπου μεταβλητές. Μπορεί να ορισθεί ως ***μία συλλογή μεταβλητών, η οποία αποθηκεύεται και παρουσιάζεται ως μία λογική οντότητα***. Διαφέρει από τους πίνακες καθόσον οι τελευταίοι αποτελούνται από μεταβλητές ίδιου τύπου.

Οι επιμέρους μεταβλητές ονομάζεται **μέλη** ή **πεδία** και μπορούν να είναι:

- Οι βασικοί τύποι δεδομένων (**int**, **char**, **float**, **double**).
- Απαριθμητικοί τύποι, οι οποίοι θα πρέπει να έχουν ορισθεί <u>πριν</u> τον ορισμό της δομής.
- Πίνακες.
- Άλλες δομές, οι οποίες θα πρέπει να έχουν ορισθεί <u>πριν</u> τον ορισμό της δομής.

Ο ορισμός της δομής έχει την ακόλουθη μορφή:

```
struct  <όνομα_δομής>
{
    <τύπος_δεδομένων>  <όνομα_1ου_μέλους>;
    <τύπος_δεδομένων>  <όνομα_2ου_μέλους>;
    ......
    <τύπος_δεδομένων>  <όνομα_τελευταίου_μέλους>;
```

[1] Περισσότερα περί εμβέλειας τύπων και μεταβλητών στο κεφάλαιο των συναρτήσεων.

```
};
```

Εάν δύο ή περισσότερα μέλη έχουν τον ίδιο τύπο, η αναφορά τους γίνεται με την πιο απλοποιημένη μορφή

<τύπος_δεδομένων> <όνομα_1ου_μέλους> <όνομα_2ου_μέλους>;

Για παράδειγμα, μία ταχυδρομική διεύθυνση αποτελεί σύνθετη πληροφορία, καθώς περιλαμβάνει το ονοματεπώνυμο του κατόχου, την οδό και τον αριθμό, την πόλη, τον ταχυδρομικό κώδικα. Ομαδοποιώντας σε έναν τύπο δεδομένου τις τέσσερις παραπάνω συνιστώσες, δημιουργείται η δομή **addrT**[2]**,** η οποία περιέχει τέσσερα μέλη:

```
struct addrT
{
    char name[40];    // πρώτο μέλος
    char street[40];
    char city[30];
    unsigned int zip_code;      // τέταρτο μέλος
};
```

ή εναλλακτικά:

```
struct addrT
{
    char name[40], street[40], city[30];
    unsigned int zip_code;
};
```

Οι μεταβλητές τύπου δομής δηλώνονται όπως και οι απλές μεταβλητές. Η δήλωση

struct addrT address1, address2;

ορίζει οι **address1**, **address2** να είναι μεταβλητές τύπου **addrT**. Η δήλωση των μεταβλητών μπορεί να συνδυασθεί με τον ορισμό της δομής. Η παραπάνω δήλωση θα μπορούσε να γίνει ως εξής:

```
struct addrT
```

```
{
    char name[40];
    char street[40];
    char city[30];
    unsigned int zip_code;
} address1, address2;
```

Με παρόμοιο τρόπο δηλώνεται και ένας πίνακας με στοιχεία δομές. Η δήλωση

struct addrT addresses[100];

δημιουργεί τον πίνακα εκατό στοιχείων **addresses**, κάθε στοιχείο του οποίου είναι μία μεταβλητή τύπου δομής **addrT** και περιέχει τέσσερα μέλη.

Παρατηρήσεις:

1. Η λέξη κλειδί ***struct*** προσδιορίζει την αρχή τού ορισμού μίας δομής και δημιουργεί έναν νέο τύπο. Το όνομα του τύπου είναι **struct addrT**. Το όνομα **addrT** είναι η *ετικέτα* της δομής (structure tag) και χρησιμοποιείται μαζί με τη ***struct*** για να δηλώνονται μεταβλητές τύπου **struct addrT**. Στη C++ η λέξη ***struct*** δεν είναι απαραίτητη στη δήλωση μεταβλητών αλλά μόνο στον ορισμό του τύπου.

2. Κάθε ορισμός δομής πρέπει να τελειώνει με ερωτηματικό (**;**).

3. Ο ορισμός μίας δομής δε δεσμεύει χώρο στη μνήμη για μεταβλητές. Απλά δημιουργεί ένα νέο τύπο.

4. Η ετικέτα μίας δομής είναι μεν προαιρετική αλλά στην πράξη θα πρέπει να χρησιμοποιείται πάντοτε, για να αποφεύγονται δυσχέρειες στον κώδικα. Εάν δεν υπάρχει η ετικέτα, τότε μεταβλητές του συγκεκριμένου τύπου δομής μπορούν να δηλωθούν μόνο κατά τον ορισμό του τύπου και όχι αργότερα. Εάν οριζόταν μία δομή χωρίς τον προσδιοριστή **addrT**:

```
struct
```

[2] Το γράμμα **T**, το οποίο προέρχεται από τη λέξη **type**, συνήθως προστίθεται στο όνομα ενός απαριθμητικού ή μίας δομής για να υποδηλώσει ότι αφορά σε τύπο δεδομένου που δημιουργήθηκε από

```
{
    char name[40], street[40], city[30];
    unsigned int zip_code;
} address1;
```

τότε θα προέκυπτε η μεταβλητή **address1** του συγκεκριμένου τύπου αλλά ο τύπος δε θα μπορούσε να ξαναχρησιμοποιηθεί γιατί δε θα είχε όνομα. Σε περίπτωση που απαιτείτο να δηλωθεί νέα μεταβλητή τέτοιου τύπου, π.χ. **address2**, θα έπρεπε να ορισθεί εκ νέου ο τύπος, δηλαδή

```
struct
{
    char name[40], street[40], city[30];
    unsigned int zip_code;
} address2;
```

5. Ο ορισμός μίας δομής πρέπει να γίνεται στην αρχή του κώδικα, πριν από τη ***main()***, ώστε να μπορεί να χρησιμοποιηθεί τόσο μέσα στη ***main()*** όσο και σε οποιοδήποτε σημείο του προγράμματος (σε συναρτήσεις κ.λ.π.).

6. Οι μόνες λειτουργίες που μπορούν να εκτελεσθούν σε μία δομή είναι:

- Ανάθεση μεταβλητών του ίδιου τύπου. Εάν **address1** και **address2** είναι δύο μεταβλητές τύπου δομής **addrT**, με την ανάθεση **address1=address2** τα μέλη τής **address1** αποκτούν τις τιμές των αντίστοιχων μελών τής **address2**.
- Η διεύθυνση μίας μεταβλητής τύπου δομής μπορεί να ανατεθεί σε ένα δείκτη (pointer). Περισσότερες λεπτομέρειες θα δοθούν στο κεφάλαιο των δεικτών.
- Χρήση του τελεστή ***sizeof***. Η εντολή **sizeof(struct address1)** επιστρέφει τον αριθμό των bytes που απαιτούνται για την αποθήκευση στη μνήμη μίας μεταβλητής τύπου δομής **addrT**.

Παράδειγμα 1.2

Μία επιχείρηση πώλησης αυτοκινήτων διαθέτει τα ακόλουθα αυτοκίνητα:

το χρήστη.

Κατασκευαστής	Τύπος	Τιμή	Διαθέσιμα τεμάχια
Mercedes	*SL500*	*87000*	*4*
Mercedes	*SLK320*	*44500*	*2*
BMW	*M3*	*54000*	*4*
Audi	*A4*	*34000*	*1*

Για την καταχώρηση των προϊόντων της επιχείρησης μπορεί να χρησιμοποιηθεί μία δομή με τέσσερα μέλη:

make: απαριθμητικός τύπος δεδομένων, που αφορά στον κατασκευαστή

model: πίνακας χαρακτήρων για την περιγραφή του μοντέλου

price: αριθμός κινητής υποδιαστολής (float) για την τιμή του κάθε μοντέλου

avail: ακέραιος (integer) για τον αριθμό των διαθέσιμων τεμαχίων κάθε μοντέλου

Το πρώτο μέλος απαιτεί τη δημιουργία του απαριθμητικού τύπου **make**, η οποία θα προηγείται του ορισμού της δομής:

enum carmakeT { Mercedes, BMW, Audi };

Κατά συνέπεια ο κώδικας ορισμού της δομής και δήλωσης ενός πίνακα που θα περιλαμβάνει τα προϊόντα της επιχείρησης είναι ο ακόλουθος:

```
enum carmakeT { Mercedes, BMW, Audi };
struct stockT
{
    carmakeT make;
    char model[15];
    float price;
    int avail;
};
void main()  {
    stockT inventory[40];
    . . . . . .
}
```

1.4 Απόδοση αρχικών τιμών στις δομές

Οι αρχικές τιμές μπορούν να αποδοθούν στις μεταβλητές δομής και τη στιγμή της δήλωσής τους, όπως στην περίπτωση των πινάκων, με λίστες από αρχικές τιμές μέσα σε άγκιστρα. Η απόδοση των τιμών μπορεί να γίνει:

- είτε μαζί με ορισμό και δήλωση:

```
struct addrT
{
    char name[40];
    char street[40];
    char city[30];
    unsigned int zip_code;
}   address1={ "Demis Pappas", "Rodou 23", "Serres", 61124 },
    address2={ "John Doe", "Limnou 32", "Serres", 61124 };
```

- είτε με τη δήλωση:

```
struct addrT address1={ "Demis Pappas", “Rodou 23", "Serres", 61124
},
                address2={ "John Doe", "Limnou 32", "Serres", 61124 };
```

Εάν σε μία δομή υπάρχουν λιγότερες αρχικές τιμές από τον αριθμό των μελών, τα υπόλοιπα μέλη αρχικοποιούνται με το μηδέν (ή με το **NULL** εάν το μέλος είναι δείκτης).

Στην περίπτωση του πίνακα δομής, η απόδοση αρχικής τιμής στη δομή ακολουθεί το γενικό κανόνα αρχικοποίησης. Έτσι, η αρχικοποίηση των τριών πρώτων στοιχείων του πίνακα **addr** γίνεται ως ακολούθως:

```
struct addrT addr[10]={
        { "Demis Pappas", "Rodou 23", "Serres", 61124 },
        { "John Doe", "Limnou 32", "Serres", 61124 },
        { "Eleni", "Skirou 12", "Serres", 61124 }
};
```

1.5 Αναφορά στα μέλη δομής

Η προσπέλαση των μελών μίας δομής γίνεται με χρήση δύο τελεστών: του **τελεστή μέλους δομής** (**.**) ή **τελεστή τελείας** και του **τελεστή δείκτη δομής** (**->**) ή **τελεστή βέλους**. Ο τελεστής βέλους θα μελετηθεί στο κεφάλαιο των δεικτών.

Η αναφορά στα μέλη δομής με χρήση του τελεστή *τελεία* γίνεται ως εξής:

<όνομα_μεταβλητής>.<όνομα_μέλους>

Έτσι η έκφραση **address1.street** αναφέρεται στο μέλος **street** της μεταβλητής **address1**, η οποία είναι τύπου δομής **addrT**.

Σε έναν πίνακα **address2** με στοιχεία δομές τύπου **addrT**, η ανάθεση στο μέλος **city** του δέκατου στοιχείου έχει την ακόλουθη μορφή:

address[9].city = "Serres"

<u>Παρατήρηση:</u> Η έκφραση **address1.city = "Serres"** είναι σωστή και αποδίδει τιμή στο μέλος **city** της μεταβλητής **address1**. Η έκφραση **addrT.city = "Serres"** είναι λανθασμένη γιατί η **addrT είναι τύπος δεδομένων**. Δε θα πρέπει να συγχέεται ο τύπος δεδομένων που ορίσθηκε με τις μεταβλητές τέτοιου τύπου.

1.6 Ένθεση δομών

Μία δομή μπορεί να περιλαμβάνει μέλη τα οποία με τη σειρά τους είναι δομές. Η C δε θέτει περιορισμό στο βαθμό ένθεσης αλλά επιτάσσει κάθε ένθετη δομή να έχει ορισθεί πριν τη χρήση της ως τύπος δεδομένου ενός μέλους ευρύτερης δομής. Για παράδειγμα, ο τύπος δεδομένου **personT**

```
struct personT
{
    char name[16];
    char address[12];
    char tel[10];
    struct dateT birthdate;
    struct dateT hiredate;
};
```

περιλαμβάνει τα μέλη **birthdate** (ημερομηνία γέννησης) και **hiredate** (ημερομηνία

πρόσληψης), τα οποία είναι μεταβλητές τύπου δομής **dateT**. Ο τύπος της **dateT** είναι ο ακόλουθος:

```
struct dateT
{
    int day;
    char month_name[4];
    int year;
};
```

και ο ορισμός του πρέπει να προηγείται του ορισμού του τύπου δεδομένων δομής **personT**.

Θεωρώντας τη μεταβλητή **bemp**, η οποία είναι τύπου **personT**, οι αναφορές στο έτος γέννησης και στον τρίτο χαρακτήρα του αλφαριθμητικού που αντιστοιχεί στο μήνα πρόσληψης έχουν τη μορφή

bemp.birthdate.year

και

bemp.hiredate.month_name[2]

αντίστοιχα.

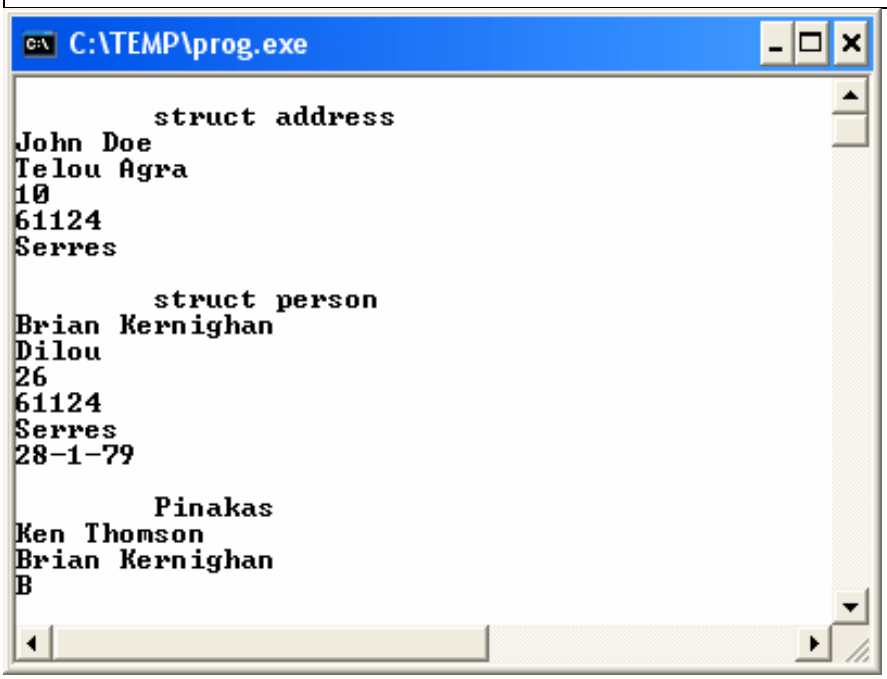
Παράδειγμα 1.3

Στο πρόγραμμα που ακολουθεί συνοψίζονται τα στοιχεία που αφορούν στις δομές, με ιδιαίτερη έμφαση στην ένθεση δομών και στους πίνακες δομών. Να γίνει σχολιασμός του κώδικα.

```
#include <stdio.h>
struct addressT
{
    char name[40];
    char street[15];
    int number;
    int zip_code;
    char city[15];
```

```
};    // τέλος του τύπου struct addressT
struct dayT
{
    int date;
    int month;
    int year;
};    // τέλος του τύπου struct dayT
struct personT
{
    struct addressT addr;
    struct dayT birthday;
};    // τέλος του τύπου struct personT
void main()
{
    struct addressT addr1={"John Doe","Telou
Agra",10,61124,"Serres"};
    struct addressT addr[10]={
        {"Ken Thomson","Rodou",23,61124,"Serres"},
        {"Brian Kernighan","Dilou",26,61124,"Serres"},
    };
    struct personT p={
        {"Brian Kernighan","Dilou",26,61124,"Serres"},
        {28,1,79},
    };
    printf( "\n\tstruct address\n");
    printf( "%s\n%s\n%d\n%d\n%s\n",addr1.name, addr1.street,
            addr1.number, addr1.zip_code, addr1.city );
    printf( "\n\tstruct person\n");
    printf( "%s\n%s\n%d\n%d\n%s\n",p.addr.name, p.addr.street,
            p.addr.number, p.addr.zip_code, p.addr.city );
    printf( "%d-%d-%d\n",p.birthday.date, p.birthday.month,
            p.birthday.year );
    printf( "\n\tPinakas\n" );
```

```
        printf( "%s\n%s\n",addr[0].name,addr[1].name );
        printf( "%c\n",addr[1].name[0] );
    }   // τέλος της main
```



Αρχικά ορίζεται ο τύπος δεδομένων **struct addressT**. Η νέα δομή περιλαμβάνει τα μέλη **name**, **street**, **number**, **zip** και **city**.

Ακολουθεί ο ορισμός του τύπου **dayT** με μέλη **date**, **month**, **year** και ο ορισμός του τύπου **personT** με μέλη **addr** και **birthday**, τα οποία είναι και αυτά δομές τύπων **addressT** και **dayT**, αντίστοιχα.

Στην αρχή της ***main()*** δηλώνονται:

- Η μεταβλητή τύπου **addressT** με όνομα **addr1**, η οποία αρχικοποιείται.
- Ένας πίνακας με όνομα **addr**, ο οποίος έχει 10 στοιχεία τύπου **address**. Αρχικοποιούνται τα δύο πρώτα στοιχεία του πίνακα.
- Η μεταβλητή **p** ως τύπου **person**. Θα πρέπει να προσεχθεί πως οι αρχικοποιήσεις κάθε μέλους της δομής περικλείονται σε άγκιστρα.

Στη συνέχεια της main() παρουσιάζονται εντολές εκτύπωσης. Στην εντολή

printf("%d-%d-%d\n",p.birthday.date,p.birthday.month,p.birthday.year);

θα πρέπει να προσεχθεί η αναφορά στα μέλη ένθετων δομών. Χρησιμοποιείται ο τελεστής **τελεία** (**.**) χωρίς περιορισμό στο βάθος έκθεσης.

Με την εντολή

printf("%s\n%s\n",addr[0].name,addr[1].name);

εκτυπώνεται το μέλος **name** του πρώτου και του δεύτερου στοιχείου του πίνακα **addr** ενώ με την εντολή

printf("%c\n",addr[1].name[0]);

εκτυπώνεται ο πρώτος χαρακτήρας του **name** του δεύτερου στοιχείου του πίνακα **addr**.

Παράδειγμα 1.4

Να δημιουργηθεί ο πίνακας **directory[40]**, ο οποίος θα αντιστοιχεί σε προσωπική ατζέντα. Κάθε στοιχείο του **directory** θα αποτελεί μεταβλητή τύπου δομής **personT**, η οποία θα έχει μέλη:

α) Δομή **idT** με: *i)* το ονοματεπώνυμο και *ii)* δομή **addressT** με τη διεύθυνση του καταγεγραμμένου.

β) Δομή **teleT** με τα τηλέφωνα (σταθερό, κινητό) και fax του καταγεγραμμένου.

γ) Δομή **emT** με το προσωπικό email και αυτό της εργασίας του καταγεγραμμένου.

Να γραφεί η main, μέσα στην οποία θα γίνεται εγγραφή και εκτύπωση δύο στοιχείων του **directory**.

<u>Παρατηρήσεις:</u>

1) Σύμφωνα με την υπόθεση ο τύπος **personT** περιλαμβάνει ως μέλη μεταβλητές που είναι αποκλειστικά τύπου δομής. Επιπρόσθετα ο τύπος **idT** περιλαμβάνει ένα μέλος που είναι τύπου δομής (**addressT**). Κατά συνέπεια, η δήλωση των τύπων δεδομένων θα πρέπει να γίνει με την ακόλουθη σειρά:

```
struct addressT
{
    . . .
};

struct idT      Ή      struct teleT      Ή      struct emT
{                      {                        {
    . . .                  . . .                    . . .
```

```
};                    };                    };

struct personT
{
    . . .
};
```

2) Εφόσον δεν προσδιορίζονται επακριβώς τα περιεχόμενα του τύπου **addressT**, αυτά επιλέγονται κατά το δοκούν:

```
struct addressT
{
    char street_name[ ];
    int street_number;
    char city[ ];
    int zip_code;
};
```

3) Είναι προτιμητέο οι τηλεφωνικοί αριθμοί να δηλώνονται ως αλφαριθμητικά γιατί αποτελούν 10-ψήφιους ή 14-ψήφιους ακέραιους. Κατά συνέπεια ο τύπος **teleT** μπορεί να έχει τη μορφή:

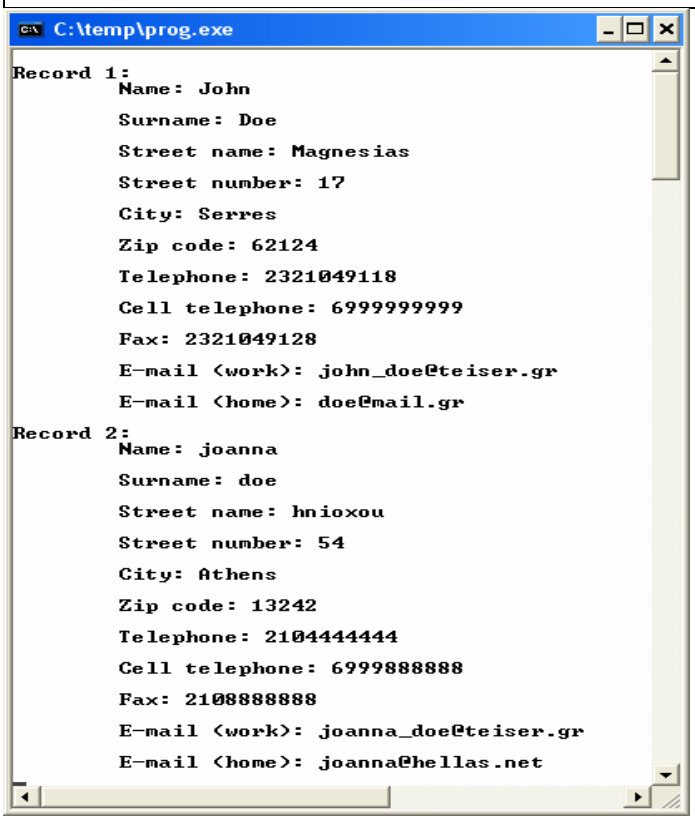
```
struct teleT
{
    char wr_no[ ];     // σταθερό τηλέφωνο
    char cell_no[ ];   // κινητό τηλέφωνο
    char fax_no[ ];
} ;
```

Συνολικά ο κώδικας είναι ο ακόλουθος:

```
#include <stdio.h>
#include <conio.h>
struct addressT  {
    char street_name[ ];
    int street_number;
```

```
        char city[ ];
        int zip_code;
    };
    struct idT  {
        char name[ ];
        char surname[ ];
        addressT addr;
    };
    struct teleT  {
        char wr_no[ ];
        char cell_no[ ];
        char fax_no[ ];
    } ;
    struct emT  {
        char em_work[ ];
        char em_home[ ];
    } ;
    struct personT  {
        idT ident;
        teleT tel;
        emT email;
    };
    void main()
    {
        personT directory[40];
        int i;
        for (i=0;i<=1;i++)  {
            printf( "\nRecord %d:",i+1 );
            printf( "\n\tName: ");    scanf( "%s",directory[i].ident.name );
            printf( "\n\tSurname: ");   scanf( "%s",directory[i].ident.surname
            );
            printf( "\n\tStreet name: ");
            scanf( "%s",directory[i].ident.addr.street_name );
```

```
            printf( "\n\tStreet number: ");
            scanf( "%d",&directory[i].ident.addr.street_number );
            printf( "\n\tCity: ");    scanf( "%s",directory[i].ident.addr.city );
            printf( "\n\tZip code: ");
            scanf( "%d",&directory[i].ident.addr.zip_code );
            printf( "\n\tTelephone: ");    scanf( "%s",directory[i].tel.wr_no );
            printf( "\n\tCell telephone: ");  scanf( "%s",directory[i].tel.cell_no
        );
            printf( "\n\tFax: ");  scanf( "%s",directory[i].tel.fax_no );
            printf( "\n\tE-mail (work): ");
            scanf( "%s",directory[i].email.em_work );
            printf( "\n\tE-mail (home): ");
            scanf( "%s",directory[i].email.em_home );
        }  // τέλος της for
    }  // τέλος της main
```



2. Συναρτήσεις

2.1 Οι έννοιες του αρθρωτού σχεδιασμού και της συνάρτησης

Ο δομημένος προγραμματισμός στηρίζεται στην έννοια του *αρθρωτού σχεδιασμού* (modular design), δηλαδή στο μερισμό σύνθετων προβλημάτων σε επιμέρους μικρά και απλούστερα τμήματα. Κατ' αντιστοιχία, ένα μεγάλο πρόγραμμα μπορεί να τεμαχισθεί σε μικρότερες γλωσσικές κατασκευές. Στη C αυτές οι κατασκευές ονομάζονται **συναρτήσεις** (functions) και αποτελούν αυτόνομες, επώνυμες μονάδες κώδικα, σχεδιασμένες να επιτελούν συγκεκριμένο έργο. Μπορούν να κληθούν επανειλημμένα σε ένα πρόγραμμα, δεχόμενες κάθε φορά διαφορετικές τιμές στις εισόδους τους.

Έως τώρα έχει μελετηθεί και χρησιμοποιηθεί μία σειρά συναρτήσεων όπως η ***main()***, οι ***printf()*** και ***scanf()***, οι συναρτήσεις χειρισμού αλφαριθμητικών κ.λ.π. Με βάση τις συναρτήσεις αυτές μπορούν να εξαχθούν τα ακόλουθα βασικά χαρακτηριστικά:

- Μία συνάρτηση εκτελεί ένα σαφώς καθορισμένο έργο (π.χ. η ***printf()*** παρέχει μορφοποιημένη έξοδο).
- Μπορεί να χρησιμοποιηθεί από άλλα προγράμματα.
- Είναι ένα «μαύρο κουτί», ένα μικροπρόγραμμα το οποίο έχει:
  - *ένα όνομα*, για να λειτουργήσει μία συνάρτηση πρέπει να κληθεί κατ' όνομα
  - *ένα σώμα*, ένα σύνολο προτάσεων και μεταβλητών
  - (προαιρετικά) *εισόδους*, μία λίστα ορισμάτων
  - (προαιρετικά) *μία έξοδο*, η οποία με το τέλος της συνάρτησης επιστρέφει μία τιμή στο σημείο του προγράμματος από το οποίο εκλήθη η συνάρτηση

2.2 Βασικά στοιχεία συναρτήσεων

Κάθε πρόγραμμα αποτελείται από μία ή περισσότερες συναρτήσεις, συμπεριλαμβανομένης πάντοτε της ***main()***, από την οποία αρχίζει η εκτέλεση του προγράμματος. Το σχήμα 2-1 δίνει τη μορφή ενός δομημένου προγράμματος στη C:

```
εντολές προεπεξεργαστή (#include, #define,…)
δηλώσεις συναρτήσεων
δηλώσεις μεταβλητών (εφόσον είναι απαραίτητες)
void main()
{
        δηλώσεις μεταβλητών
        προτάσεις
 }
 func1()
 {
        ……
 }
 func2()
 {
        ……
 }
```

Σχήμα 2-1. *Γενική μορφή προγράμματος στη C*

Μία συνάρτηση περιλαμβάνει τρεις φάσεις:

- τη δήλωση (προαιρετικά)
- τον ορισμό
- την κλήση ή τις κλήσεις

2.2.1 Δήλωση συνάρτησης

Στη δήλωση μίας συνάρτησης παρουσιάζεται το «πρότυπο» ή «πρωτότυπο» συνάρτησης, το οποίο αποτελείται από τρία τμήματα, όπου ορίζονται:

- **Το όνομα**: θα πρέπει να είναι ενδεικτικό της λειτουργίας της.
- **Οι είσοδοι** (εφόσον υπάρχουν): μία *λίστα τυπικών ορισμάτων* ή *παραμέτρων* (formal parameters), αποτελούμενη από ονόματα μεταβλητών εισόδου και τύπων δεδομένων.

- **Ο τύπος της εξόδου**: τύπος δεδομένων της επιστρεφόμενης τιμής, εφόσον επιστρέφεται τιμή (προκαθορισμένος τύπος: **int**).

<τύπος δεδομένων επιστροφής> <όνομα συναρτησης>(***λίστα ορισμάτων*** **);**

Για παράδειγμα, η πρόταση

int maximum_two_integers(int first_integer, int second_integer);

αποτελεί τη δήλωση μίας συνάρτησης ονόματι **maximum_two_integers**, η οποία δέχεται δύο εισόδους, τις ακέραιες μεταβλητές **first_integer** και **second_integer**, και επιστρέφει ακέραια έξοδο (ο τύπος **int** πριν από το όνομά της).

Η δήλωση των συναρτήσεων γίνεται πριν από τη ***main()***, συνήθως μετά τις εντολές προεπεξεργαστή (**#include, #define**).

2.2.2 Ορισμός συνάρτησης

Ο ορισμός μίας συνάρτησης περιλαμβάνει το πρότυπο συνάρτησης χωρίς το καταληκτικό ερωτηματικό, ακολουθούμενο από το σώμα της συνάρτησης, το οποίο αναπτύσσεται μέσα σε άγκιστρα:

```
<τύπος δεδομένων επιστροφής>  <όνομα συναρτησης>( λίστα ορισμάτων )
{
    πρόταση;
    ......
    πρόταση επιστροφής; // εφόσον η συνάρτηση επιστρέφει τιμή
}
```

Για παράδειγμα, ο ορισμός της συνάρτησης ***maximum_two_integers()*** είναι ο ακόλουθος:

```
int maximum_two_integers( int first_integer, int second_integer )
{
    if (first_integer > second_integer) return(first_integer);
    else return(second_integer);
```

```
}
```

Παρατηρήσεις:

1. Εάν η συνάρτηση έχει έξοδο, αυτή θα πρέπει να επιστρέφεται με χρήση της εντολής ***return*** στο τέλος του σώματος της συνάρτησης. Εάν δεν έχει έξοδο, ο **<τύπος δεδομένων επιστροφής>** αντικαθίσταται από τη λέξη ***void***, π.χ.

```
void print_max_two_integers( int first_integer, int second_integer )
{
    if (first_integer > second_integer) printf( "max=%d\n",first_integer );
    else printf( "max=%d\n",first_integer );
}
```

2. Οι συναρτήσεις μπορούν να έχουν τις δικές τους εσωτερικές μεταβλητές, όπως ακριβώς έχει η ***main()***.

2.2.3 Κλήση συνάρτησης

Μία συνάρτηση ενεργοποιείται όταν κληθεί. Εάν η συνάρτηση δεν επιστρέφει τιμή, η κλήση της γίνεται από ένα σημείο του προγράμματος ως εξής:

<όνομα συνάρτησης> (πρώτο όρισμα, ..., τελευταίο όρισμα);

Οι τιμές **πρώτο όρισμα**, ..., **τελευταίο όρισμα** καλούνται *πραγματικά ορίσματα* ή *πραγματικές παράμετροι* (actual parameters). Τα πραγματικά ορίσματα χωρίζονται με κόμμα και περικλείονται σε παρενθέσεις, αντιστοιχούν δε ένα προς ένα στη λίστα τυπικών ορισμάτων. Ακόμη κι όταν δεν υπάρχουν ορίσματα οι παρενθέσεις είναι υποχρεωτικές, καθώς δηλώνουν στο μεταγλωττιστή ότι το όνομα αντιπροσωπεύει συνάρτηση και όχι μεταβλητή. Τα πραγματικά ορίσματα μπορούν να είναι σταθερές, τιμές μεταβλητών ή ακόμη και τιμές εκφράσεων αλλά σε κάθε περίπτωση πρέπει να είναι ίδιου τύπου με τα τυπικά ορίσματα. Για παράδειγμα, η κλήση της συνάρτησης ***print_max_two_integers()*** μπορεί να λάβει την ακόλουθη μορφή:

```
x=10;
print_max_two_integers( x, 32 );
<επόμενη πρόταση>;
```

Όταν κληθεί η συνάρτηση, το τυπικό όρισμα **first_integer** αντιστοιχεί στο πραγματικό όρισμα **x**, οπότε **first_integer=x=10**, και το τυπικό όρισμα **second_integer** λαμβάνει την τιμή **32**. Ο έλεγχος του προγράμματος περνάει στις επόμενες προτάσεις της συνάρτησης. Μετά την εκτέλεση και της τελευταίας πρότασης η συνάρτηση τερματίζει και ο έλεγχος μεταφέρεται στο κυρίως πρόγραμμα, στην **<επόμενη πρόταση>;**.

Όταν η συνάρτηση έχει έξοδο υπάρχει μία μικρή διαφοροποίηση · για παράδειγμα η κλήση της ***maximum_two_integers()*** μπορεί να λάβει την ακόλουθη μορφή:

x=10;

y=maximum_two_integers(x, 32);

<επόμενη πρόταση>;

Η διαφορά έγκειται στο ότι στο τέλος της συνάρτησης η επιστρεφόμενη τιμή θα πρέπει να αποδοθεί σε μία μεταβλητή τού κυρίως προγράμματος, στην παρούσα περίπτωση στην **y**. Κατά συνέπεια, αφενός μεν η **y** θα πρέπει να είναι ίδιου τύπου με την επιστρεφόμενη τιμή (integer στο συγκεκριμένο παράδειγμα), αφετέρου δε μετά το τέλος της συνάρτησης ο έλεγχος του προγράμματος περνά στην πρόταση ανάθεσης **y=maximum_two_integers(x, 32);** και ακολούθως στην **<επόμενη πρόταση>;**.

Προσοχή: Θα πρέπει να σημειωθεί ότι μπορεί να παραληφθεί η δήλωση της συνάρτησης εάν η συνάρτηση παρουσιάζεται μέσα στο πρόγραμμα πριν από την πρώτη κλήσης της, όπως φαίνεται στο παράδειγμα 2.1. Ωστόσο αυτή είναι μία ριψοκίνδυνη τακτική και θα πρέπει να αποφεύγεται.

Παράδειγμα 2.1

Στο πρόγραμμα που ακολουθεί παραλήφθηκε η δήλωση της ***square()*** γιατί αυτή ορίζεται προτού κληθεί. Εάν όμως η ***square()*** οριζόταν κάτω από τη ***main()*** έπρεπε να δηλωθεί.

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

```
float square(float y)
{
    return(y*y);
}
void main()
{
    float x=15.2;
    printf( "x^2=%f\n",square(x) );
}
```

Παράδειγμα 2.2

Να γραφεί πρόγραμμα μετατροπής θερμοκρασιών από την κλίμακα Farenheit στην κλίμακα Celcius, με βάση την εξίσωση μετατροπής **Celcius = (Farenheit-32)*5/9**.

Το πρόγραμμα θα δέχεται μία θερμοκρασία στην κλίμακα Farenheit, θα τη μετατρέπει στην κλίμακα Celcius και θα την εκτυπώνει:

```
#include<stdio.h>
void main ()
{
    float degF,degC, ratio;
    printf( "Enter degrees F: " );
    scanf( "%f",&degF );
    // Κώδικας μετατροπής:
    ratio = (float) 5/9;
    degC = (degF-32)*ratio;
    printf( "%f degrees F are %f degrees C\n",degF,degC );
}
```

Σε περίπωση που η μετατροπή χρειαζόταν σε πολλά σημεία ενός προγράμματος, δε θα έπρεπε να επαναλαμβάνεται ο κώδικας της μετατροπής των θερμοκρασιακών κλιμάκων. Για το λόγο αυτό πρέπει ο κώδικας της μετατροπής να ενταχθεί σε μία ανεξάρτητη οντότητα του προγράμματος, η οποία θα δέχεται ως είσοδο τη θερμοκρασία στη μία κλίμακα και θα αποδίδει στην έξοδό της τη θερμοκρασία στην άλλη κλίμακα. Ακολούθως παρατίθεται το πρόγραμμα με χρήση συναρτήσεων:

```
#include<stdio.h>
float F_to_C (float far);
void main()
{
    float degF,degC;
    printf( "Enter degrees F: " );
    scanf( "%f",&degF );
    degC = F_to_C (degF);
    printf( "%f degrees F are %f degrees C\n",degF,degC );
}  // τέλος της main
float F_to_C (float far)
{
    float ratio = 5.0 / 9.0;
    return((far-32)*ratio);
} // τέλος της συνάρτησης
```

Παράδειγμα 2.3

Να γραφεί πρόγραμμα, στο οποίο να καλούνται οι συναρτήσεις, τα πρωτότυπα των οποίων δίνονται ως ακολούθως:

```
int max(int a, int b);
double power(double x, int n);
```

Πριν από κάθε κλήση συνάρτησης πρέπει να υπάρχει στον πηγαίο κώδικα το πρωτότυπό της, έτσι ώστε ο μεταγλωττιστής να ελέγχει εάν κάθε πρόταση κλήσης είναι σύμφωνη ως προς τον αριθμό και τον τύπο των ορισμάτων, καθώς και τον τύπο της επιστρεφόμενης τιμής. Το κώδικα που υλοποιεί την υπόθεση είναι το ακόλουθο:

```
#include <…h>
#define ......
int max(int a, int b);
double power(double x, int n);
int num=5;
void main()
{
    printf( "%d\n",max(12/2,num+3,2*num );
    printf( "%f\n",power(num,3) );
}
```

2.3 Είδη και εμβέλεια μεταβλητών

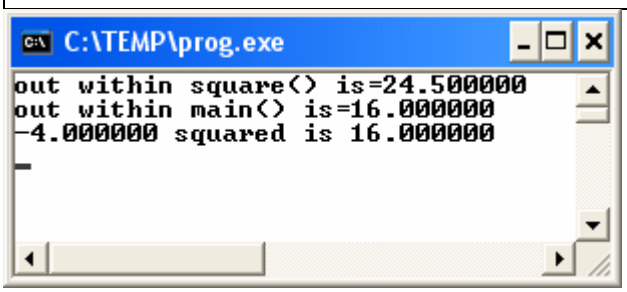
2.3.1 Τοπικές μεταβλητές

Στο παράδειγμα 2.2 η συνάρτηση ***F_to_C()*** δημιουργεί στον κορμό της τη μεταβλητή **ratio**. Η μεταβλητή αυτή είναι ενεργή μόνο μέσα στο τμήμα κώδικα στο οποίο ορίζεται (σώμα της συνάρτησης) και παύει να υφίσταται μετά το πέρας της συνάρτησης. Μεταβλητές τέτοιου είδους καλούνται **τοπικές μεταβλητές** (local variables). Δύο συναρτήσεις μπορούν να έχουν τοπικές μεταβλητές με το ίδιο όνομα χωρίς να παρουσιάζεται πρόβλημα, καθώς καθεμιά ισχύει μέσα στη συνάρτηση που δηλώνεται. Κατ' αντιστοιχία,οι μεταβλητές που δηλώνονται μέσα στη ***main()*** δεν επηρεάζουν τις μεταβλητές των υπόλοιπων συναρτήσεων του προγράμματος.

Παράδειγμα 2.4

Στο πρόγραμμα που ακολουθεί αναδεικνύεται η συμπεριφορά των τοπικών μεταβλητών. Δημιουργούνται δύο μεταβλητές με το ίδιο όνομα **out**, η πρώτη μέσα στη ***main()*** και η δεύτερη μέσα στη συνάρτηση ***square()***. Όπως προκύπτει από τα αποτελέσματα, οι δύο μεταβλητές ενεργούν ανεξάρτητα η μία από την άλλη.

```
#include<stdio.h>
float square (float x);
void main()
{
    float in,out;
    in=-4.0;
    out=square(in);
    printf( "out within main() is=%f\n",out );
    printf( "%f squared is %f\n",in,out );
}
float  square (float x)
{
    float out;
    out=24.5;
    printf( "out within square() is=%f\n",out );
    return(x*x);
}
```



2.3.2 Καθολικές μεταβλητές

Σε αντιδιαστολή με τις τοπικές μεταβλητές, οι οποίες είναι ενεργές μόνο μέσα στο σώμα της συνάρτησης που δηλώνονται, υπάρχει μία κατηγορία μεταβλητών που είναι ενεργές σε όλα τα τμήματα ενός προγράμματος. Οι μεταβλητές αυτές καλούνται **καθολικές μεταβλητές** (global variables) και δηλώνονται πριν από τη ***main()***. Όταν μεταβάλλεται η τιμή μίας καθολικής μεταβλητής σε οποιοδήποτε σημείο του προγράμματος, η νέα τιμή μεταφέρεται σε όλο το υπόλοιπο πρόγραμμα.

Εν γένει οι καθολικές μεταβλητές είναι ένα ριψοκίνδυνο προγραμματιστικό εργαλείο, καθώς αποτρέπουν τον ξεκάθαρο μερισμό του προβλήματος σε ανεξάρτητα τμήματα. Επιπρόσθετα, μία καθολική μεταβλητή δεσμεύει μνήμη καθόλη τη διάρκεια του προγράμματος, ενώ για μία τοπική μεταβλητή ο χώρος στη μνήμη δεσμεύεται μόλις ο έλεγχος περάσει στη συνάρτηση, αποδεσμεύεται δε με το τέλος αυτής, οπότε και η μεταβλητή δεν έχει πλέον νόημα.

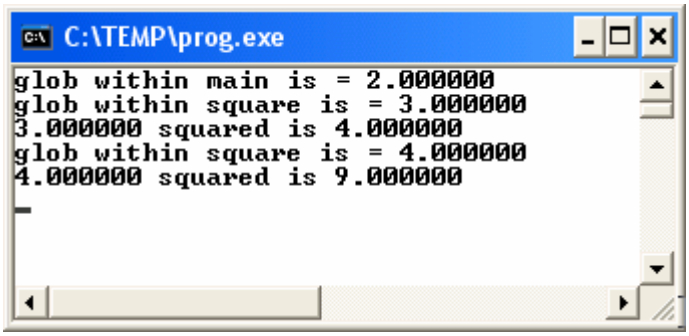
Παράδειγμα 2.5

Στο πρόγραμμα που ακολουθεί αναδεικνύεται η συμπεριφορά των καθολικών μεταβλητών και τα προβλήματα που πιθανόν να ανακύψουν από εσφαλμένη χρήση τους.

```
#include <stdio.h>
float glob;     // καθολική μεταβλητή
float square(float x);
void main()
{
    float in;
    glob=2.0;
    printf( "glob within main is = %f\n",glob );
    in=square(glob);
    printf( "%f squared is %f\n",glob,in );
    in=square(glob);
    printf( "%f squared is %f\n",glob,in );
```

```
}
float square(float x)
{
    glob=glob+1.0;
    printf( "glob within square is = %f\n",glob );
    return (x*x);
}
```

Δηλώνεται μία καθολική μεταβλητή **glob**, η οποία λαμβάνει μέσα στη ***main()*** την τιμή **2**. Ακολούθως καλείται η ***square()*** με όρισμα την τιμή της **glob**, έτσι ώστε να προκύψει το τετράγωνο της **glob**. Η τιμή της **glob** περνά στην τυπική παράμετρο **x**, η οποία και υψώνεται στο τετράγωνο. Ωστόσο, μέσα στη συνάρτηση ***square()*** μεταβάλλεται η τιμή της **glob** κατά μία μονάδα. Κατά συνέπεια η έξοδος της ***square()*** είναι το τετράγωνο του **2** αλλά επιστρέφοντας στη ***main()*** η **glob** έχει λάβει την τιμή **3**, οδηγώντας την ***printf()*** σε εσφαλμένη έξοδο όπως φαίνεται στα αποτελέσματα. Στην επόμενη κλήση της, η ***square()*** δέχεται ως όρισμα το **3** και επιστρέφει το **9**, έχοντας όμως μεταβάλλει τη **glob**.



2.3.3 Εμβέλεια μεταβλητών

Στις §2.3.1 και 2.3.2 παρουσιάσθηκαν δύο είδη μεταβλητών με διαφορετικό εύρος λειτουργίας. Το τμήμα του πηγαίου κώδικα στο οποίο μία μεταβλητή είναι ενεργή προσδιορίζεται με τους **κανόνες εμβέλειας** (scope rules). Διακρίνονται τέσσερις τύποι εμβέλειας:

- **Εμβέλεια προγράμματος**: μεταβλητές αυτής της εμβέλειας είναι οι καθολικές. Είναι ορατές από όλες τις συναρτήσεις που απαρτίζουν το πρόγραμμα, έστω κι αν βρίσκονται σε διαφορετικά αρχεία πηγαίου κώδικα.
- **Εμβέλεια αρχείου**: μεταβλητές αυτής της εμβέλειας είναι ορατές μόνο στο αρχείο που δηλώνονται και μάλιστα από το σημείο της δήλωσής τους και κάτω. Μεταβλητή που δηλώνεται έξω από το μπλοκ με τη λέξη κλειδί **static** πριν από τον τύπο, έχει εμβέλεια αρχείου, π.χ. **static int velocity**.
- **Εμβέλεια συνάρτησης**: Προσδιορίζει την ορατότητα του ονόματος από την αρχή της συνάρτησης έως το τέλος της. Εμβέλεια συνάρτησης έχουν μόνο οι ***goto*** ετικέτες.
- **Εμβέλεια μπλοκ**: Προσδιορίζει την ορατότητα από το σημείο δήλωσης έως το τέλος του μπλοκ στο οποίο δηλώνεται. Μπλοκ είναι ένα σύνολο από προτάσεις, οι οποίες περικλείονται σε άγκιστρα. Μπλοκ είναι η σύνθετη πρόταση αλλά και το σώμα συνάρτησης. Εμβέλεια μπλοκ έχουν και τα τυπικά ορίσματα των συναρτήσεων.

Η C επιτρέπει τη χρήση ενός ονόματος για την αναφορά σε διαφορετικά αντικείμενα, με την προϋπόθεση ότι αυτά έχουν διαφορετική εμβέλεια ώστε να αποφεύγεται η **σύγκρουση ονομάτων** (name conflict). Εάν οι περιοχές εμβέλειας έχουν επικάλυψη, τότε <u>το όνομα με τη μικρότερη εμβέλεια αποκρύπτει το όνομα με τη μεγαλύτερη</u>.

Παράδειγμα 2.6

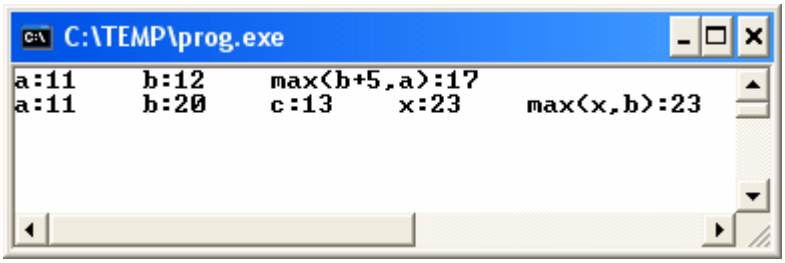
Να προσδιορισθεί η εμβέλεια των μεταβλητών στον ακόλουθο πηγαίο κώδικα:

```
1   #include <stdio.h>
2   int max(int a, int b);
3   static void func(int x);
4   int a; static int b;
5   void main()  {
6         b=12; a=b--;
7         printf( "a:%d\tb:%d\tmax(b+5,a):%d\n",a,b,max(b+5,a) );
```

```
8        func(a+b);
9    }   // τέλος της main
10   int c=13;
11   int max(int a, int b){
12       return(a>b?a:b);
13   }   // τέλος της max
14   static void func(int x){
15       int b=20;
16       printf( "a:%d\tb:%d\tc:%d\tx:%d\tmax(x,b):%d\n",
17       a,b,c,x,max(x,b) );
18   }   // τέλος της func
```

- 4 **int a; static int b;** Η **a** είναι καθολική μεταβλητή με εμβέλεια προγράμματος. Η **b** έχει εμβέλεια αρχείου, όπως προσδιορίζει η λέξη **static**.
- 10 **int c=13;** Έχει εμβέλεια προγράμματος αλλά είναι ενεργή από το σημείο δήλωσής της και κάτω (γραμμή 10).
- 11 **int max(int a, int b){** Οι **a** και **b** έχουν εμβέλεια μπλοκ και αποκρύπτουν για το σώμα της ***max()*** τις καθολικές μεταβλητές **a** και **b**.
- 6 **b=12; a=b--;** Αποδίδονται οι τιμές **11** και **12** στις **a** και **b**, αντίστοιχα.
- 7 **printf("a:%d\tb:%d\tmax(b+5,a):%d\n",a,b,max(b+5,a));**

  Καλείται η συνάρτηση ***max()*** και αυτή δίνει στα τυπικά ορίσματα **a** και **b** τις τιμές **11** και **12**, αντίστοιχα. Η ***max()*** επιστρέφει το **17**, το οποίο τυπώνει η ***printf()***.
- 8 **func(a+b);** Καλείται η συνάρτηση ***func()*** και αυτή δίνει στο τυπικό όρισμα **x** την τιμή **11+12=23**. Το όρισμα **x** έχει εμβέλεια μπλοκ. Η τοπική μεταβλητή **b=20**, που δηλώνεται στη γραμμή 15, αποκρύπτει από το σώμα της ***func()*** την καθολική μεταβλητή **b**. Δε συμβαίνει όμως το ίδιο και για την καθολική μεταβλητή **a**, η οποία είναι ορατή από το σώμα της ***func()***.



2.3.4 Διάρκεια μεταβλητών

Η διάρκεια ορίζει το χρόνο κατά τον οποίο το όνομα της μεταβλητής είναι συνδεδεμένο με τη θέση μνήμης που περιέχει την τιμή της μεταβλητής. Ορίζονται ως **χρόνοι δέσμευσης** και **αποδέσμευσης** οι χρόνοι που το όνομα συνδέεται με τη μνήμη και αποσυνδέεται από αυτή, αντίστοιχα.

Για τις καθολικές μεταβλητές δεσμεύεται χώρος με την έναρξη εκτέλεσης του προγράμματος και η μεταβλητή συσχετίζεται με την ίδια θέση μνήμης έως το τέλος του προγράμματος. Είναι **πλήρους διάρκειας**.

Αντίθετα, οι τοπικές μεταβλητές είναι **περιορισμένης διάρκειας**. Η ανάθεση της μνήμης σε τοπική μεταβλητή γίνεται με τη είσοδο στο χώρο εμβέλειάς της και η αποδέσμευσή της με την έξοδο από αυτόν. Δηλαδή η τοπική μεταβλητή δε διατηρεί την τιμή της από τη μία κλήση της συνάρτησης στην επόμενη.

Εάν προστεθεί στη δήλωση μίας τοπικής μεταβλητής η λέξη **static**, διατηρεί την τιμή της και καθίσταται πλήρους διάρκειας.

Στη συνάρτηση

```
func(int x);
{
    int temp;
    static int num;
    ......
}
```

η μεταβλητή **num** είναι τοπική αλλά έχει διάρκεια προγράμματος, σε αντίθεση με την **temp**, η οποία έχει διάρκεια συνάρτησης.

Παρατήρηση: Θα πρέπει να δοθεί προσοχή στην αρχικοποίηση των τοπικών μεταβλητών. Μία τοπική μεταβλητή περιορισμένης διάρκειας αρχικοποιείται, εφόσον βέβαια κάτι τέτοιο έχει ορισθεί να γίνεται, με κάθε είσοδο στο μπλοκ που αυτή ορίζεται. Αντίθετα, μία τοπική μεταβλητή πλήρους διάρκειας αρχικοποιείται μόνο με την ενεργοποίηση του προγράμματος.

Παράδειγμα 2.7

α) Να περιγραφεί η επίδραση της λέξης κλειδί **static** στις δύο δηλώσεις του ακόλουθου πηγαίου κώδικα.

β) Πότε αρχικοποιείται η **count** και πότε η **num**;

```
static int num;
void func(void)
{
    static int count=0;
    int num=100;
    . . . . . . .
}
```

α) Η **static** στη δήλωση της καθολικής μεταβλητής **num** περιορίζει την ορατότητά της μόνο στο αρχείο που δηλώνεται. Αντίθετα η **static** στη δήλωση της τοπικής μεταβλητής **count** ορίζει γι' αυτήν διάρκεια προγράμματος.

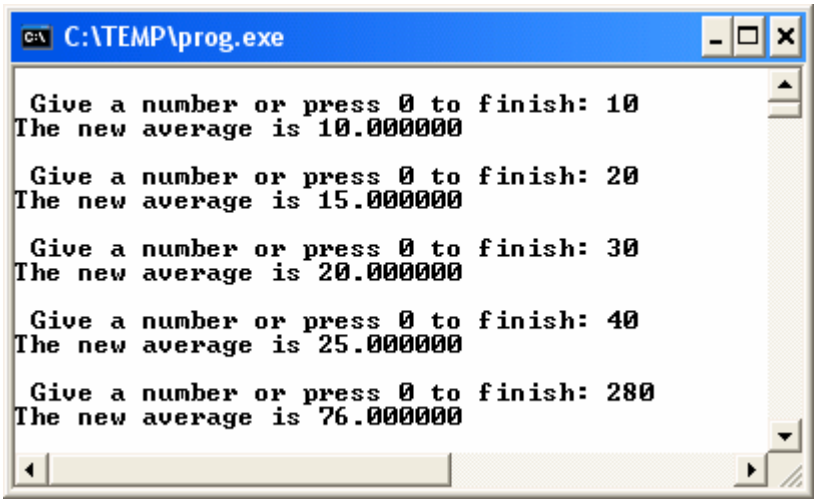
β) Η **count** ως τοπική μεταβλητή πλήρους διάρκειας αρχικοποιείται μία φορά με την είσοδο στο πρόγραμμα. Αντίθετα η **num** ως τοπική μεταβλητή περιορισμένης διάρκειας αρχικοποιείται σε κάθε ενεργοποίηση της συνάρτησης **func**.

Παράδειγμα 2.8

Ο κώδικας που ακολουθεί αποτελεί παράδειγμα χρήσης στατικών μεταβλητών. Στη συνάρτηση ***get_average()*** οι μεταβλητές **static float total** και **static int count**

εκτελούνται μόνο την πρώτη φορά. Τις επόμενες διατηρούν το αποτέλεσμα της προηγούμενης κλήσης και σε αυτό προστίθενται στη μεν **total** το **newdata**, στη δε **count** η μονάδα.

```
#include <stdio.h>
float get_average(float newdata);
void main()
{
    float data=1.0;
    float average;
    while (data!=0)
    {
        printf( "\n Give a number or press 0 to finish: " );
        scanf( "%f",&data );
        average=get_average(data);
        printf( "The new average is %f\n",average );
    }
}   // τέλος της main
float get_average(float newdata)
{
    static float total=0.0;
    static int count=0;
    count++;
    total=total+newdata;
    return(total/count);
}   // τέλος της get_average
```



2.4 Πίνακες ως παράμετροι συναρτήσεων

Εάν κατά την κλήση μίας συνάρτησης η πραγματική παράμετρος είναι όνομα πίνακα (πχ. **t**), δεν αποστέλλει στη συνάρτηση ολόκληρο τον πίνακα αλλά τη διεύθυνση του πρώτου byte του πίνακα. Η παράμετρος στη δήλωση της συνάρτησης είναι ένα όνομα τοπικού πίνακα (π.χ. **number**). Κρατά ένα αντίγραφο της ίδιας διεύθυνσης αλλά χρησιμοποιεί διαφορετικό όνομα. Επιπρόσθετα, στη δήλωση δεν αναφέρεται το ακριβές μέγεθός του αλλά μόνο το γεγονός ότι είναι πίνακας καθώς και ο τύπος στοιχείων του. Επομένως ουσιαστικά γνωστοποιείται στο μεταγλωττιστή η διεύθυνση του πρώτου byte και η απόσταση (αριθμός bytes) μεταξύ των στοιχείων:

```
#include <stdio.h>
display(int num[ ]);
void main()
{
    int t[10],i;
    for (i=0; i<=9; i++)    t[i]=i;
    display(t);
}
display(int num[ ])
{
```

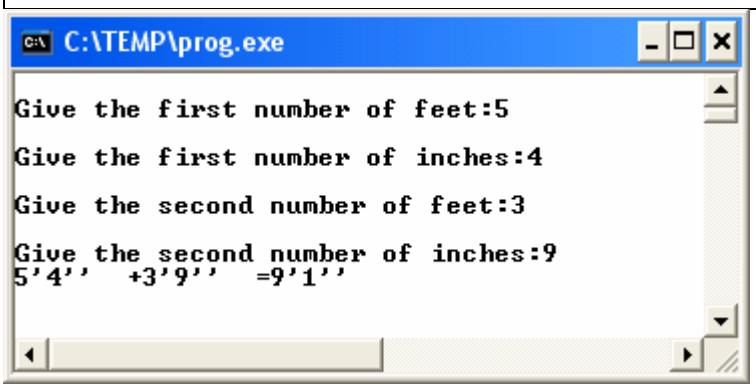
```
        int i;
        for (i=0; i<=9; i++)
        printf( "%d ",num[i] );
    }
```

Παράδειγμα 2.9

Στον κώδικα που ακολουθεί παρουσιάζονται συναρτήσεις που δέχονται και επιστρέφουν δομές. Ο κώδικας υλοποιεί έναν αθροιστή μεγεθών που είναι εκφρασμένα σε πόδια και ίντσες.

```
/* Αν ο αριθμός των ιντσών υπερβαίνει το 12 συμπληρώνεται ένα πόδι,
επομένως πρέπει να προστεθεί η μονάδα στη μεταβλητή feet και να
αφαιρεθούν 12 ίντσες από τη μεταβλητή inches. */
#include <stdio.h>
struct Distance
{
    int feet;
    int inches;
};
Distance addeng1(Distance dd1, Distance dd2);   // δήλωση addeng1
void eng1disp(Distance dd);     // δήλωση eng1disp
void main()
{
    Distance d1,d2,d3;
    printf( "\n1st number of feet:" );              scanf("%d",&d1.feet);
    printf( "\n1st number of inches:" );            scanf("%d",&d1.inches );
    printf( "\n2nd second number of feet:" );       scanf("%d",&d2.feet);
    printf( "\n2nd second number of inches:" ); scanf("%d",&d2.inches);
    d3=addeng1(d1,d2);
    eng1disp(d1);
```

```
        printf( " +" );
        eng1disp(d2);
        printf( " =" );
        eng1disp(d3);
        printf( "\n" );
    }   // τέλος της main
    void eng1disp(Distance dd)
    {
        printf( "%d'%d'' ",dd.feet,dd.inches );
    }   // τέλος της eng1disp
    Distance addeng1(Distance dd1, Distance dd2)
    {
        Distance dd3;
        dd3.inches=dd1.inches+dd2.inches;
        dd3.feet=0;
        if (dd3.inches>=12)
        {
            dd3.inches=dd3.inches-12;
            dd3.feet++;
        }
        dd3.feet=dd3.feet+dd1.feet+dd2.feet;
        return(dd3);
    }   // τέλος της addeng1
```



2.5 Αναδομικές συναρτήσεις

Μία συνάρτηση ονομάζεται αναδρομική όταν μία εντολή του σώματος της συνάρτησης καλεί τον ίδιο της τον εαυτό. Η αναδρομή είναι μία διαδικασία με την οποία ορίζεται κάτι μέσω του ίδιου του οριζόμενου.

Έστω ότι μία αναδρομική συνάρτηση καλείται να λύσει ένα πρόβλημα. Μία τέτοια συνάρτηση δύναται να λύσει μόνο την απλούστερη περίπτωση, τη λεγόμενη **βάση της αναδρομής** (base case). Εάν η περίπτωση είναι πολύπλοκη, το πρόβλημα μερίζεται σε ένα ή περισσότερα υποπροβλήματα, τα οποία μοιάζουν με το αρχικό πρόβλημα αλλά αποτελούν μικρότερες εκδοχές του. Η αναδρομική συνάρτηση καλεί τον εαυτό της για την επίλυση των υποπροβλημάτων. Αυτή είναι μία **αναδρομική κλήση** ή **αναδρομικό βήμα** (recursion step). Η διαδικασία συνεχίζεται έως ότου ο μερισμός σε υποπροβλήματα οδηγήσει στη βάση της αναδρομής, η οποία επιλύεται άμεσα. Κατόπιν ακολουθείται η αντίστροφη διαδικασία, επιλύοντας αρχικά τα μικρότερα υποπροβλήματα και προχωρώντας προς τα μεγαλύτερα. Η όλη διαδικασία θα αποσαφηνισθεί με τη βοήθεια του ακόλουθου παραδείγματος:

Παράδειγμα: Να ορισθεί συνάρτηση που υπολογίζει το άθροισμα των αριθμών από **1** έως **n**.

// Χωρίς αναδρομή	// Με αναδρομή

int sum(int n)
{
  int i, total=0;
  for (i=0; i<=n; i++)
  total+=i;
  return(total);
}
	int sum(int n)
	{
	  if (n<=1) return(n);
	  else return(sum(n-1)+n);
	}

Επεξηγήσεις της αναδρομικής εκδοχής:

```
if (n<=1) return(n);
else return(sum(n-1)+n);
```

Εάν το **n** είναι ίσο με **1**, τότε το άθροισμα ταυτίζεται με το **n** (βάση της αναδομής). Στη γενική περίπτωση, ο υπολογισμός του αθροίσματος **n** μπορεί να θεωρηθεί ως υπολογισμός του αθροίσματος των αριθμών από το **1** έως το **n-1** συν το **n**. Αντίστοιχα, ο υπολογισμός του αθροίσματος **n-1** μπορεί να θεωρηθεί ως υπολογισμός του αθροίσματος των αριθμών από το **1** έως το **n-2** συν το **n-1**. Ακολουθώντας την παραπάνω διαδικασία, μπορούμε να ορίσουμε τα εξής:

1+…+n = (1+…+(n-1)) + n

(1+…+(n-1)) = (1+…+(n-2)) + (n-1)

(1+…+(n-2)) = (1+…+(n-3)) + (n-2)

(1+…+(n-3)) = (1+…+(n-4)) + (n-3)

κ.ο.κ.

Από τα παραπάνω προκύπτει ότι κάθε σχέση είναι ίδια με την προηγούμενη, με απλή αλλαγή των ορισμάτων.

Την πρώτη φορά καλείται η **sum**() με όρισμα **n**. Με την πρόταση **return(sum(n-1)+n)** η συνάρτηση **sum**() καλεί τον ίδιο της τον εαυτό με διαφορετικό όμως όρισμα **(n-1)**. Η ενεργοποίηση αυτή θα προκαλέσει με τη σειρά της νέα ενεργοποίηση και αυτό θα συνεχισθεί έως ότου προκληθεί διακοπή. Η διακοπή είναι αποκλειστική ευθύνη του προγραμματιστή. Στο συγκεκριμένο παράδειγμα η διακοπή προκαλείται με την πρόταση **if (n<=1) return(n)**, που σημαίνει ότι όταν το **n** φθάσει να γίνει **1** υπάρχει πλέον αποτέλεσμα. Έτσι οι διαδοχικές κλήσεις για **n=4** είναι:

sum(4) καλεί τη sum(3)

sum(3) καλεί τη sum(2)

sum(2) καλεί τη sum(1)

η sum(1) δίνει αποτέλεσμα 1 και το επιστρέφει στη sum(2)

η sum(2) δίνει αποτέλεσμα 1+2=3 και το επιστρέφει στη sum(3)

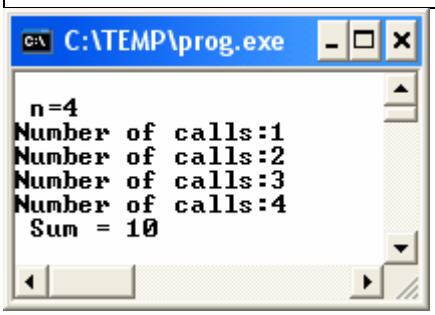
η sum(3) δίνει αποτέλεσμα 3+3=6 και το επιστρέφει στη sum(4)

η sum(4) δίνει αποτέλεσμα 6+4=10, το οποίο είναι και το τελικό

Το πλήρες πρόγραμμα έχει την ακόλουθη μορφή:

```
#include <stdio.h>
# include <conio.h>
int sum(int n);
int number_of_calls=0;
```

```
void main()
{
    int n=4;
    printf( "\n n=%d",n );
    printf( "\n Sum = %d", sum(n) );
    printf( "\n   Press any key to finish" ); getch();
}   // Τέλος της main
int sum(int n)
{
    if (n<=1)
    {
        number_of_calls++;
        printf( "\nNumber of calls:%d",number_of_calls );
        return(n);
    }
    else
    {
        number_of_calls++;
        printf( "\nNumber of calls:%d",number_of_calls );
        return(sum(n-1)+n);
    }
}   // τέλος της sum
```



Πλεονεκτήματα των αναδρομικών συναρτήσεων:

- Το βασικότερο πλεονέκτημα των αναδρομικών συναρτήσεων είναι ότι μπορούν να χρησιμοποιηθούν για να δημιουργηθούν καθαρότερες και απλούστερες εκδοχές πολλών αλγορίθμων.
- Δημιουργείται συμπαγέστερος κώδικας και είναι ιδιαίτερα χρήσιμες σε αναδρομικώς οριζόμενα δεδομένα όπως οι λίστες και τα δένδρα.

Μειονεκτήματα των αναδρομικών συναρτήσεων:

- Οι περισσότερες αναδρομικές συναρτήσεις δεν εξοικονομούν σημαντικό μέγεθος κώδικα ή μνήμης για τις μεταβλητές.
- Οι αναδρομικές εκδοχές των περισσότερων συναρτήσεων μπορεί να εκτελούνται κάπως πιο αργά από τα επαναληπτικά τους ισοδύναμα εξαιτίας των πρόσθετων κλήσεων σε συναρτήσεις. Η πιθανή μείωση όμως δεν είναι αξιοσημείωτη.
- Υπάρχει μικρή πιθανότητα οι πολλές αναδρομικές κλήσεις μίας συνάρτησης να προκαλέσουν υπερχείλιση της στοίβας (stack overflow), επειδή ο χώρος αποθήκευσης των παραμέτρων και των τοπικών μεταβλητών της συνάρτησης είναι στη στοίβα και κάθε νέα κλήση παράγει ένα νέο αντίγραφο αυτών των μεταβλητών. Ωστόσο, εφόσον διατηρείται ο έλεγχος της αναδρομικής συνάρτησης και υπάρχει συνθήκη διακοπής, το ζήτημα είναι ήσσονος σημασίας.

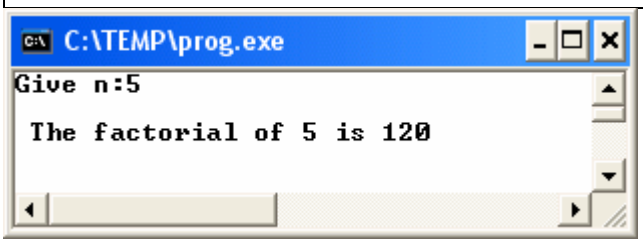
Παράδειγμα 2.10

Να καταστρωθεί πρόγραμμα, με χρήση αναδρομικής συνάρτησης, το οποίο δέχεται ως είσοδο από το πληκτρολογίο έναν ακέραιο αριθμό **n** και επιστρέφει στην οθόνη το παραγοντικό του (**n! = 1x2x…xn**).

Το πρόβλημα είναι απολύτως αντίστοιχο με εκείνο του προηγούμενου παραδείγματος. Οι μόνες διαφορές είναι ότι πρέπει να διαβάζεται ο **n** και ότι πολλαπλασιάζονται ακέραιοι και δεν αθροίζονται. Κατά συνέπεια, μία πιθανή λύση είναι η ακόλουθη:

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

```
#include <conio.h>
int fact(int n);
void main()
{
    int n;
    printf( "Give n:" );
    scanf( "%d",&n );
    printf( "\n The factorial of %d is %d",n,fact(n) );
    printf( "\n      Press any key to finish" );
    getch();
}   // τέλος της main
int fact(int n)
{
    if (n<=0)
    {
        printf( "\nERROR! n should be a positive integer\n");
        return(0);    // σε περίπτωση σφάλματος επιστροφή με το 0
    }
    else if (n<=1) return(n);
    else return(fact(n-1)*n);
}   // τέλος της fact
```



3. Δείκτες

3.1 Η έννοια του δείκτη

Κάθε μεταβλητή σχετίζεται με μία θέση στην κύρια μνήμη του υπολογιστή, η οποία χαρακτηρίζεται από τη διεύθυνσή της. Στη γλώσσα μηχανής μπορεί να γίνει άμεση χρήση αυτής της διεύθυνσης για να αποθηκευθούν ή να ανακληθούν δεδομένα. Αντίθετα, στις γλώσσες προγραμματισμού υψηλού επιπέδου οι διευθύνσεις δεν είναι άμεσα ορατές από τον προγραμματιστή καθώς καλύπτονται από το μανδύα των συμβολικών ονομάτων, τα οποία το σύστημα αντιστοιχεί στις πραγματικές διευθύνσεις.

Η γλώσσα C, θέλοντας να δώσει στον προγραμματιστή τη δυνατότητα συγγραφής αποδοτικού κώδικα, υποστηρίζει την άμεση διαχείριση των περιεχομένων της μνήμης, εισάγοντας την έννοια του **δείκτη** (pointer). ***Ο δείκτης αποτελεί μία μεταβλητή που περιέχει μία διεύθυνση μνήμης***. Οι δείκτες είναι ένα ισχυρό προγραμματιστικό εργαλείο, που εφαρμόζεται:

- στη δυναμική εκχώρηση μνήμης
- στη διαχείριση σύνθετων δομών δεδομένων
- στην αλλαγή τιμών που έχουν εκχωρηθεί ως ορίσματα σε συναρτήσεις
- για τον αποτελεσματικότερο χειρισμό πινάκων

Ωστόσο, καθώς η χρήση των δεικτών οδηγεί σε επεμβάσεις στη μνήμη και πολλές φορές σε προγραμματιστικές ακροβασίες, συχνά αποτελεί αιτία δύσκολων στον εντοπισμό σφαλμάτων, γι' αυτό και πρέπει να γίνεται με ιδιαίτερη προσοχή.

3.2 Δήλωση δείκτη

Η δήλωση μίας μεταβλητής δείκτη ακολουθεί τον εξής φορμαλισμό:

<τύπος_δεδομένων> * <όνομα_δείκτη>;

π.χ.

int *pnum;

- Όταν ένας δείκτης έχει αποθηκευμένη μία διεύθυνση έχει επικρατήσει να λέγεται ότι *ο δείκτης «δείχνει» στη διεύθυνση*. Στη δήλωση δείκτη ο **<τύπος_δεδομένων>** αφορά στο είδος των δεδομένων που αποθηκεύονται στη διεύθυνση που «δείχνει» ο

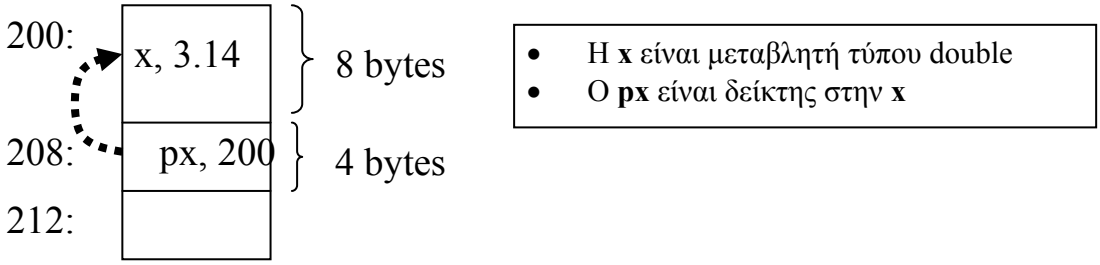
δείκτης. Ο τύπος της κανονικής μεταβλητής πρέπει να δηλώνεται γιατί μία μεταβλητή δεσμεύει συγκεκριμένη μνήμη (8 bytes για float, 4 bytes για int κ.ο.κ.). Εφόσον ο δείκτης χρησιμοποιείται για να γίνεται έμμεση αναφορά στην τιμή της μεταβλητής, πρέπει να είναι γνωστό πόση μνήμη καταλαμβάνει αυτή η τιμή.

- Τον τύπο δεδομένων ακολουθεί ο αστερίσκος, ο οποίος είναι απαραίτητος γιατί προσδιορίζει ότι δηλώνεται μία μεταβλητή δείκτη κι όχι μία «κανονική» μεταβλητή. Ο αστερίσκος συνδέεται με το όνομα και όχι με τον τύπο δεδομένων. Θα πρέπει να σημειωθεί ότι αν και η μεταβλητή με δείκτη περιέχει μία διεύθυνση (δηλαδή έναν ακέραιο αριθμό), δεν είναι ίδια με μία κανονική ακέραια μεταβλητή. Μέσω του αστερίσκου ο μεταγλωττιστής γνωρίζει ότι η τιμή της μεταβλητής με δείκτη είναι μία συγκεκριμένη διεύθυνση μνήμης, σε αντιδιαστολή με την «κανονική» ακέραια τιμή.

- Τη δήλωση δείκτη κλείνει το όνομα του δείκτη. Επιλέγεται με βάση τις ίδιες συμβάσεις που ισχύουν στις κανονικές μεταβλητές. Ωστόσο, συνήθως ο αρχικός χαρακτήρας του ονόματος δείκτη είναι το **p**, έτσι ώστε το πρόγραμμα να καθίσταται περισσότερο ευανάγνωστο, καθώς με τον πρώτο χαρακτήρα φαίνεται εάν μία μεταβλητή είναι δείκτης ή όχι. Εναλλακτικά, μπορεί να προστεθεί η κατάληξη **_ptr**. Ενδεικτικές είναι οι ακόλουθες δηλώσεις:

```
int *pcount, *count_ptr;   /* δείκτες σε ακέραιες μεταβλητές */
char *pword, *word_ptr;  /* δείκτες σε μεταβλητές χαρακτήρα */
```

Καθώς το περιεχόμενο ενός δείκτη είναι μία διεύθυνση, δηλαδή ένας ακέραιος αριθμός, ένας δείκτης θα καταλαμβάνει όσα bytes αντιστοιχούν σε ακέραιο (π.χ. 4 bytes), ανεξάρτητα από τον τύπο της κανονικής μεταβλητής στην οποία δείχνει. Στο σχήμα 3-1 απεικονίζεται ο τρόπος λειτουργίας των δεικτών, με τη **x** να είναι μία κανονική μεταβλητή και τον **px** να είναι δείκτης που δείχνει στη **x**.



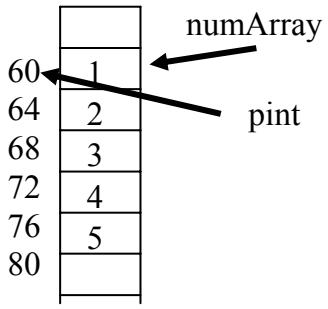
Σχήμα 3-1. *Απεικόνιση του τρόπου λειτουργίας των δεικτών*

3.3 Ανάθεση τιμής σε δείκτη

Η ανάθεση τιμής σε δείκτη μπορεί να γίνει με έναν από τους ακόλουθους τέσσερις τρόπους:

➢ **Με χρήση πινάκων**, δεδομένου ότι το όνομα ενός πίνακα αντιστοιχεί στη διεύθυνση του πρώτου στοιχείου του. Έτσι, με τον ακόλουθο κώδικα αποδίδεται στο δείκτη ακεραίων **pint** η τιμή **60**, δηλαδή η διεύθυνση του πρώτου στοιχείου του πίνακα **numArray**. Στο σχήμα 3-2 απεικονίζεται η διαδικασία ανάθεσης τιμής στο δείκτη **pint**.

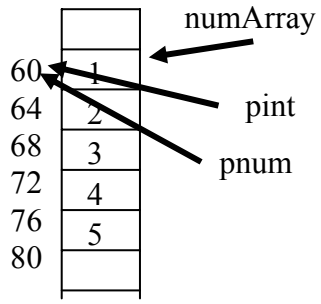
```
int  numArray[5] = {1,2,3,4,5};
int *pint;
pint = numArray;
```



Σχήμα 3-2. *Ανάθεση τιμής σε δείκτη με χρήση πίνακα*

➢ **Με χρήση άλλων δεικτών ίδιου τύπου**. Στον κώδικα που ακολουθεί ο δείκτης **pint** δείχνει ήδη σε κάποια διεύθυνση. Με την έκφραση **pnum = pint;** το περιεχόμενο του **pint** αντιγράφεται στον **pnum** κι έτσι ο τελευταίος δείχνει στην ίδια διεύθυνση, όπως φαίνεται στο σχήμα 3-3.

```
int  numArray[5] = {1,2,3,4,5};
int *pint, *pnum;
pint = numArray;
pnum = pint;
```



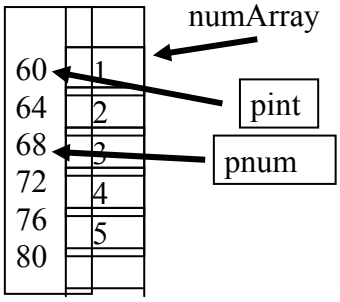
Σχήμα 3-3. *Ανάθεση τιμής σε δείκτη με χρήση άλλων δεικτών ίδιου τύπου*

➢ **Με χρήση αριθμητικής δεικτών**. Οι δείκτες υποστηρίζουν εκφράσεις της μορφής

pnum=pint+y; ή **pnum=pint-y;**

όπου οι **pnum**, **pint** είναι δείκτες ίδιου τύπου και ο **y** είναι ακέραιος. Οι παραπάνω είναι οι μόνες πράξεις που μπορούν να γίνουν με δείκτες. Η διαφοροποίηση της πρώτης έκφρασης έγγειται στο ότι στον **pnum** <u>δε θα ανατεθεί το περιεχόμενο του **pint** αυξημένο κατά **y** μονάδες αλλά αυξημένο κατά **y** επί τον αριθμό των bytes που καταλαμβάνει ο τύπος δεδομένων του δείκτη</u>. Δηλαδή, εάν οι **pnum**, **pint** είναι δείκτες ακεραίων και το **y=2**, ο **pnum** θα δείχνει **2x4=8** bytes κάτω από τη διεύθυνση που δείχνει ο **pint**, όπως φαίνεται στο σχήμα 3-4.

```
int  numArray[5] = {1,2,3,4,5};
int *pint, *pnum;
pint = numArray;
pnum = pint+2;
```



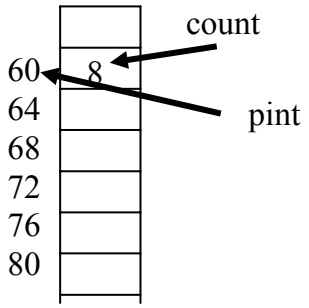
Σχήμα 3-4. *Ανάθεση τιμής σε δείκτη με χρήση αριθμητικής δεικτών*

➢ **Με χρήση του *τελεστή διεύθυνσης* (&) (address-of operator)**. Ο τελεστής διεύθυνσης **&**, ο οποίος πρωτοαπαντήθηκε στη συνάρτηση εισόδου ***scanf()***, εισάγεται μπροστά από μία μεταβλητή εκφράζοντας τη διεύθυνσή της. Ο συμβολισμός

&count

ερμηνεύεται «στη διεύθυνση της **count**». Έτσι, με τον ακόλουθο κώδικα ανατίθεται στο δείκτη **pnum** η διεύθυνση στην οποία βρίσκεται αποθηκευμένη η ακέραια μεταβλητή **count**. Η διαδικασία απεικονίζεται παραστατικά στο σχήμα 3-5.

```
int *pnum;
int count;
pnum = &count;
```



Σχήμα 3-5. *Ανάθεση τιμής σε δείκτη με χρήση του τελεστή διεύθυνσης*

<u>Παρατήρηση:</u> Θα μπορούσε να γίνει άμεση ανάθεση μίας διεύθυνσης, π.χ. **pnum=1000;** Ωστόσο, μία τέτοια επιλογή είναι πολύ επικίνδυνη και πρέπει να αποφεύγεται. Τέτοιου είδους αναθέσεις χρησιμοποιούνται συνήθως μόνο για άμεση πρόσβαση στο υλικό (hardware).

3.4 Προσπέλαση μεταβλητής με χρήση δείκτη

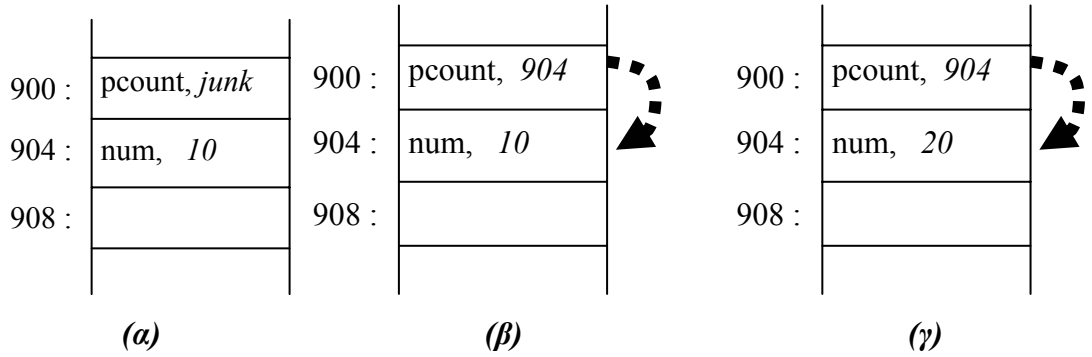
Η προσπέλαση μίας μεταβλητής με χρήση δείκτη και η μεταβολή της τιμής της γίνεται με χρήση του **τελεστή περιεχομένου** (*) (dereferencing operator) ή **τελεστή έμμεσης αναφοράς**, η λειτουργία του οποίου περιγράφεται με τη βοήθεια του ακόλουθου κώδικα:

```
int *pcount, num;
num=10;
pcount=&num;
*pcount = 20;
```

Στον παραπάνω κώδικα αρχικά ορίζεται μία ακέραια μεταβλητή **num**, η οποία λαμβάνει την τιμή **10**, και ένας δείκτης σε ακέραιο **pint**, ο οποίος αρχικά δεν έχει τιμή. Στο σχήμα 3-6α παρουσιάζεται ο χάρτης μνήμης μετά το τέλος της δεύτερης

γραμμής. Ως ***junk*** (απορρίματα) συμβολίζεται το περιεχόμενο μίας θέσης μνήμης όταν αυτή δεν έχει ορισθεί στο τρέχον πρόγραμμα.

Ακολούθως ανατίθεται στο δείκτη **pcount** η διεύθυνση της **num** (σχήμα 3-6β). Στην τελευταία γραμμή χρησιμοποιείται ο τελεστής περιεχομένου μπροστά από το δείκτη. Ο συμβολισμός ***pcount** ερμηνεύεται «στη διεύθυνση που δείχνει ο **pcount**». Έτσι η τελευταία γραμμή κώδικα ερμηνεύεται «να τοποθετηθεί το **20** στη διεύθυνση που δείχνει ο **pcount**», δηλαδή να μεταβληθεί έμμεσα η τιμή της **num** από **10** σε **20** (σχήμα 3-6γ).



Σχήμα 3-6. *Προσπέλαση μεταβλητής με χρήση δείκτη*

Παράδειγμα 3.1

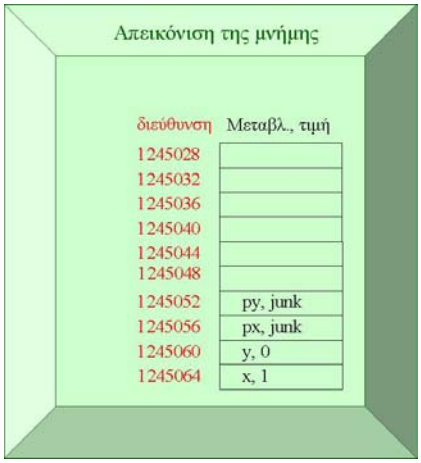
Να περιγραφεί αναλυτικά η λειτουργία κάθε γραμμής κώδικα και να απεικονισθούν τα περιεχόμενα των θέσεων μνήμης που καταλαμβάνουν οι μεταβλητές.

```
1:    int *px, *py, x=1, y=0;
2:    int a[5]={2,4,5,6,7};
3:    int i;
4:    px = &x;
5:    py = a;
6:    for (i=0; i<5; i++)    *(py+i) = 2*i;
7:    y = *px;
8:    py = &y;
9:    px = a+3;
```

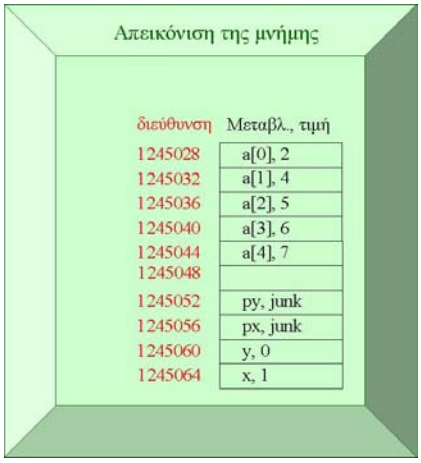
```
10:     *px = 21;
11:     *py = *px+9;
12:     x = *(&y);
```

➢ *Γραμμή 1*: Δήλωση δύο δεικτών σε ακέραιο (**px**, **py**), ακολουθούμενη από δήλωση και αρχικοποίηση δύο ακέραιων μεταβλητών (**x**, **y**) (σχήμα 3-7α).

➢ *Γραμμή 2*: Δήλωση πίνακα ακεραίων πέντε στοιχείων (**a**) και αρχικοποίηση αυτού (σχήμα 3-7β).



Σχήμα 3-7α



Σχήμα 3-7β

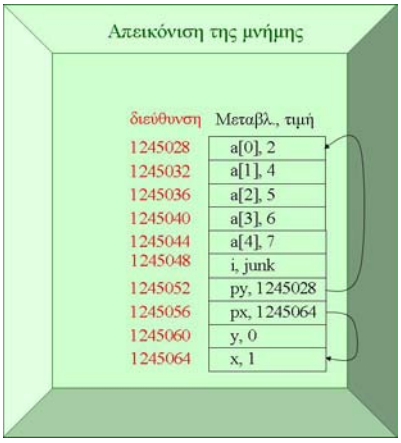
➢ *Γραμμή 3*: Δήλωση ακέραιας μεταβλητής (**i**) (σχήμα 3-7γ).

➢ *Γραμμή 4*: Ανάθεση της διεύθυνσης της μεταβλητής **x** στο δείκτη **px**, δηλαδή το περιεχόμενο του **px** είναι η διεύθυνση - το πρώτο byte – του **x** (σχήμα 3-7δ).

Σχήμα 3-7γ

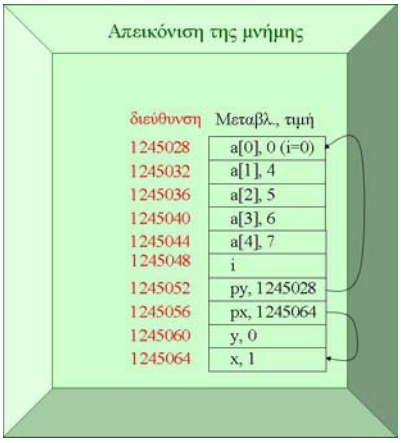
Σχήμα 3-7δ

➢ *Γραμμή 5*: Ανάθεση της διεύθυνσης του πρώτου στοιχείου του πίνακα **a** στο δείκτη **py**, δηλαδή το περιεχόμενο του **py** είναι η διεύθυνση του **a[0]** (σχήμα 3-7ε).

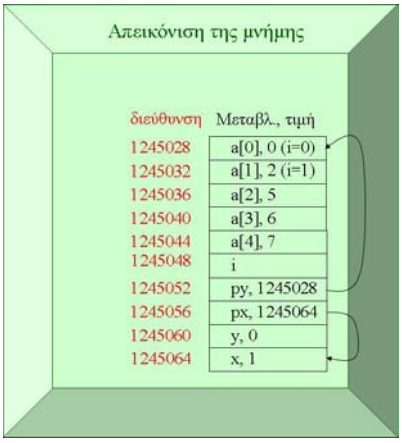


Σχήμα 3-7ε

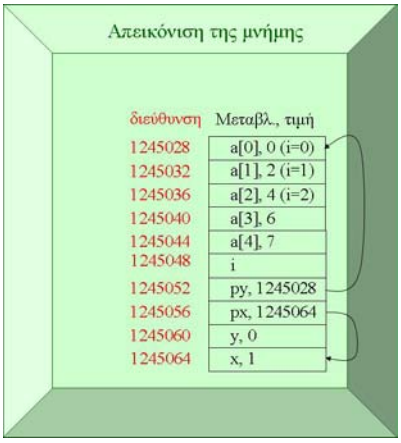
➢ *Γραμμή 6*: Επαναληπτική έκφραση, σε κάθε επανάληψη της οποίας λαμβάνουν χώρα τα ακόλουθα: Η τιμή **2*i** τοποθετείται σε θέση μνήμης με διεύθυνση **2*i** bytes μετά τη διεύθυνση που δείχνει ο **py** (σχήματα 3-7στ - 3.7ι). Για παράδειγμα, εάν **i=3**, η τιμή **6** αποθηκεύεται στη θέση μνήμης **py+i=1245028+3*4=1245028+12=1245040**, θέση που καταλαμβάνει το στοιχείο **a[3]**. Με αυτόν τον τρόπο αποδίδεται στο **a[3]** η τιμή **6**.



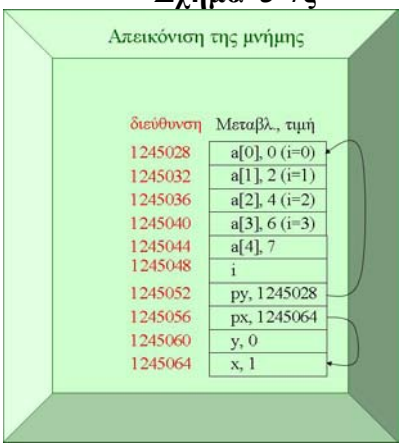
Σχήμα 3-7στ



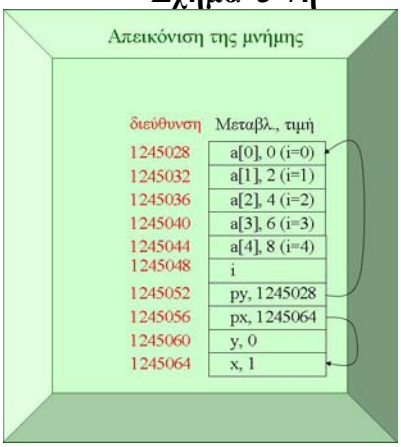
Σχήμα 3-7ζ



Σχήμα 3-7η



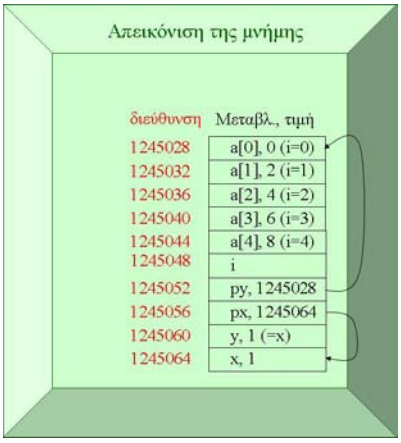
Σχήμα 3-7θ



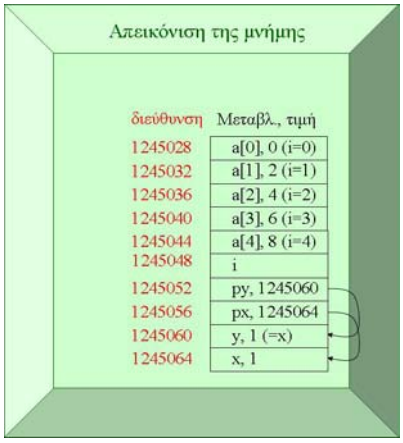
Σχήμα 3-7ι

➢ *Γραμμή 7:* Το περιεχόμενο της θέσης στην οποία δείχνει ο **px**, δηλαδή το **1**, αποδίδεται στο **y** (σχήμα 3-7ια).

➢ *Γραμμή 8:* Ανάθεση της διεύθυνσης της μεταβλητής **y** στο δείκτη **py** (σχήμα 3-7ιβ).

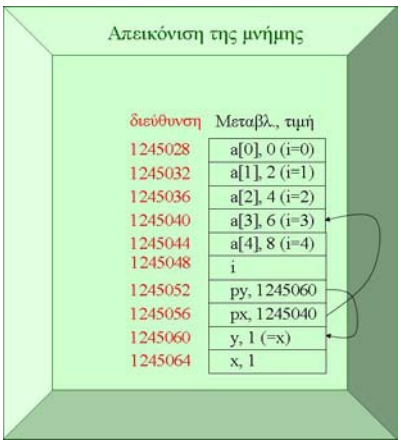


Σχήμα 3-7ια



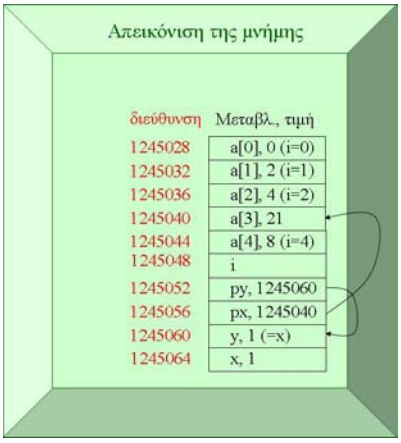
Σχήμα 3-7ιβ

➢ *Γραμμή 9*: Ο δείκτης **px** δείχνει 12 bytes (3*4) μετά τη διεύθυνση του **a[0]**, δηλαδή στη διεύθυνση του πρώτου byte του **a[3]** (σχήμα 3-7ιγ).



Σχήμα 3-7ιγ

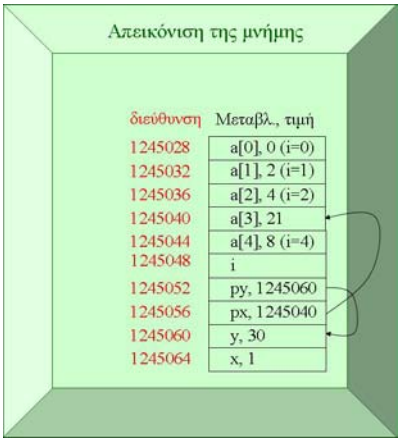
➢ *Γραμμή 10*: Το περιεχόμενο της διεύθυνσης στην οποία δείχνει ο **px** γίνεται **21** (σχήμα 3-7ιδ).



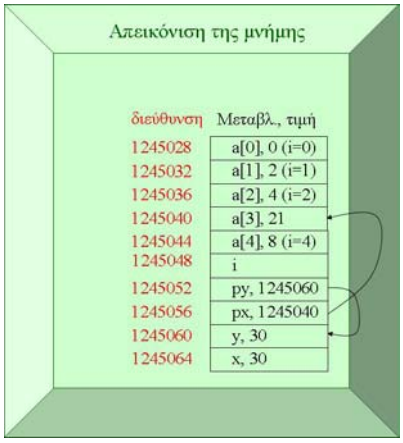
Σχήμα 3-7ιδ

➢ *Γραμμή 11*: Το περιεχόμενο της διεύθυνσης στην οποία δείχνει ο **py** γίνεται ίσο με το περιεχόμενο της διεύθυνσης στην οποία δείχνει ο **px**, αυξημένο κατά **9**, δηλαδή ισούται με **30** (σχήμα 3-7ιε).

➢ *Γραμμή 12*: Η μεταβλητή **x** γίνεται ίση με τη μεταβλητή **y** (σχήμα 3-7ιστ).



Σχήμα 3-7ιε



Σχήμα 3-7ιστ

3.5 Δείκτες και συναρτήσεις

3.5.1 Μεταβίβαση παραμέτρων

Στο προηγούμενο κεφάλαιο αναφέρθηκε ο τρόπος κλήσης μίας συνάρτησης και η μεταβίβαση των πραγματικών ορισμάτων στις παραμέτρους της καλούμενης συνάρτησης, σύμφωνα με τον οποίο κατά την κλήση μίας συνάρτησης οι παράμετροι αποτελούν αντίγραφο των πραγματικών ορισμάτων, καταλαμβάνοντας διαφορετικές θέσεις μνήμης. Κατά συνέπεια, η όποια επεξεργασία υφίστανται οι τυπικές παράμετροι δεν επηρεάζει τις τιμές των πραγματικών ορισμάτων. Ο παραπάνω τρόπος κλήσης ονομάζεται **κλήση κατ' αξία** (call by value).

Ωστόσο υπάρχουν περιπτώσεις, όπως θα φανεί στο παράδειγμα 3.3, κατά τις οποίες ο συγκεκριμένος τρόπος κλήσης μίας συνάρτησης είναι ανεπαρκής και απαιτείται να δύναται η συνάρτηση να μεταβάλλει τις τιμές των πραγματικών ορισμάτων. Σε τέτοιες περιπτώσεις χρησιμοποιείται η **κλήση κατ' αναφορά** (call by reference), σύμφωνα με την οποία δε μεταβιβάζονται στις τυπικές παραμέτρους οι τιμές των πραγματικών ορισμάτων αλλά οι διευθύνσεις τους. Κατά συνέπεια, η όποια επεξεργασία υποστούν οι τυπικές παράμετροι θα επηρεάσει τις τιμές των πραγματικών ορισμάτων. Η κλήση κατ' αναφορά χρησιμοποιεί ως πραγματικά ορίσματα τις διευθύνσεις των μεταβλητών και ως τυπικές παραμέτρους δείκτες.

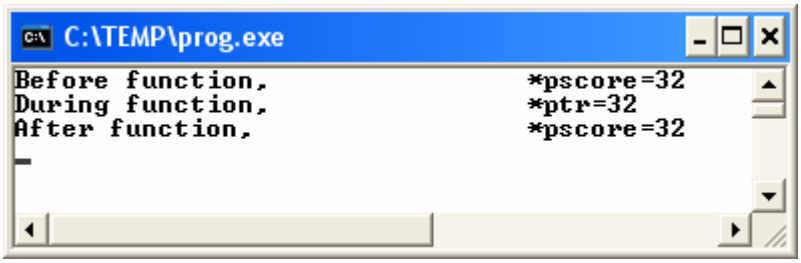
Θα πρέπει να σημειωθεί ότι η κλήση συναρτήσεων με ορίσματα πίνακες είναι κλήση κατ' αναφορά, καθώς το όνομα ενός πίνακα αντιστοιχεί στη διεύθυνση του πρώτου στοιχείου του.

Παρατήρηση: Μπορεί να χρησιμοποιηθεί δείκτης για να αλλαχθεί το περιεχόμενο της θέσης στην οποία δείχνει, αλλά δεν πρέπει να αλλαχθεί ο ίδιος ο δείκτης μέσα στην καλούμενη συνάρτηση. Ο λόγος είναι ότι οι πραγματικές παράμετροι που είναι δείκτες αντιγράφουν μία διεύθυνση στις παραμέτρους της συνάρτησης, αλλά εάν αλλαχθεί η παράμετρος στη συνάρτηση (δηλαδή η διεύθυνση) δε θα αλλαχθεί η πραγματική παράμετρος! Το ακόλουθο παράδειγμα αναδεικνύει το πρόβλημα.

Παράδειγμα 3.2

Να αναλυθεί η λειτουργία του ακόλουθου προγράμματος:

```
#include <stdio.h>
void print(int *ptr);
void main()
{
    int *pscore, num;
    num=32;
    pscore=&num;
    printf( "Before function,\t\t*pscore=%d\n", *pscore);
    print(pscore);
    printf( "After function,\t\t*pscore=%d\n", *pscore);
}
void print(int *ptr)
{
    printf( "During function,\t\t*ptr=%d\n", *ptr);
    ptr=ptr+1;
}
```



Αρχικά δηλώνονται η ακέραια μεταβλητή **num**, η οποία λαμβάνει την τιμή **32**, και ο δείκτης σε ακέραιο **pscore**. Ακολούθως στον **pscore** ανατίθεται η διεύθυνση της **num** και στη συνέχεια καλείται η συνάρτηση ***print()*** με πραγματικό όρισμα τον **pscore**. Η τυπική παράμετρος της ***print()*** είναι ο δείκτης σε ακέραιο **ptr**, ο οποίος λαμβάνει το περιεχόμενο του **pscore**, δηλαδή τη διεύθυνση της **num**.

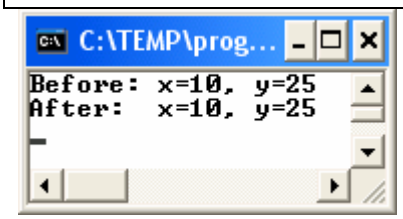
Η συνάρτηση εκτυπώνει το περιεχόμενο της θέσης μνήμης στην οποία δείχνει ο **ptr**, δηλαδή το **32**, και στη συνέχεια ο **ptr** αυξάνεται κατά ένα, δείχνοντας σε θέση μνήμης **4** bytes κάτω από τη θέση μνήμης που έδειχνε προηγουμένως. Αυτή η μεταβολή στο περιεχόμενο του **ptr** δεν έχει νόημα, γιατί με το πέρας της συνάρτησης παύει να ισχύει ο **ptr** και δεν επιδρά στα αποτελέσματα, που παρατίθενται ανωτέρω.

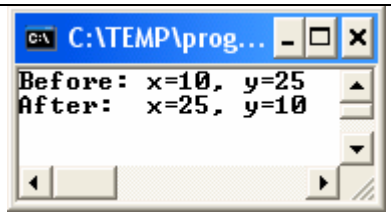
Παράδειγμα 3.3

Να γίνει συγκριτική ανάλυση της λειτουργίας των ακόλουθων προγραμμάτων:

#include <stdio.h> **void swap(int a, int b);** **void main()** **{** **int x=10, y=25;** **printf("Before:x=%d, y=%d\n", x, y);** **swap(x,y);** **printf("After:x=%d, y=%d\n", x, y);** **}**	**#include <stdio.h>** **void swap(int *pa, int *pb);** **void main()** **{** **int x=10, y=25;** **printf("Before:x=%d, y=%d\n", x, y);** **swap(&x, &y);** **printf("After:x=%d, y=%d\n", x, y);** **}**

void swap(int a, int b)	void swap(int *pa, int *pb)
{	{
int temp=a;	int temp=*pa;
a=b;	*pa=*pb;
b=temp;	*pb=temp;
}	}





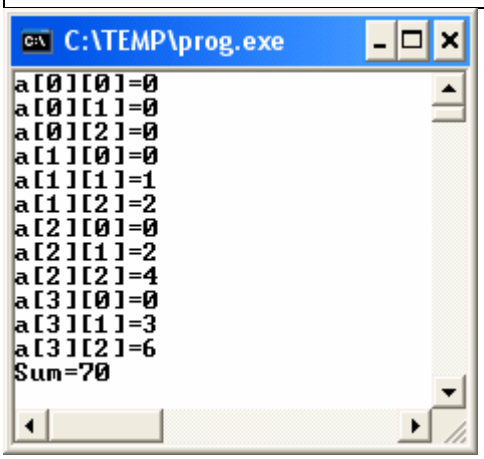
Τα παραπάνω προγράμματα καλούν τη συνάρτηση ***swap()*** για να αντιμετατεθούν τα περιεχόμενα των μεταβλητών **x**, **y**. Στο αριστερό πρόγραμμα χρησιμοποιείται η κλήση κατ' αξία και οι τιμές των **x** και **y** αντιγράφονται στις τυπικές παραμέτρους **a** και **b**, αντίστοιχα. Αυτό που επιτυγχάνει η ***swap()*** είναι να αντιμετατεθούν οι τιμές των **a** και **b**, με αποτέλεσμα μετά το πέρας της ***swap()*** τα **a** και **b** να μην είναι πλέον ενεργά και τα **x** και **y** να μην έχουν αλλάξει. Κατά συνέπεια το αριστερό πρόγραμμα δεν εκτέλεσε επιτυχώς το έργο της αντιμετάθεσης, όπως άλλωστε προκύπτει και από τα αποτελέσματα.

Στο δεξί πρόγραμμα χρησιμοποιείται η κλήση κατ' αναφορά και οι διευθύνσεις των **x** και **y** αντιγράφονται στις τυπικές παραμέτρους **pa** και **pb**, αντίστοιχα, οι οποίες είναι δείκτες ακεραίων. Χρησιμοποιώντας τον τελεστή περιεχομένου (*) και την τοπική μεταβλητή **temp**, η θέση που αντιστοιχεί στη **x** αποκτά το περιεχόμενο της θέσης που αντιστοιχεί στην **y** και τανάπαλιν. Μετά το πέρας της ***swap()*** παύουν να υφίστανται οι δείκτες **pa** και **pb** και το έργο της αντιμετάθεσης έχει επιτευχθεί, όπως φανερώνουν και τα αποτελέσματα.

Παράδειγμα 3.4

Να γραφεί μία συνάρτηση, η οποία θα δέχεται ως ορίσματα: *α)* τη διεύθυνση του πρώτου στοιχείου ενός πίνακα ακεραίων 4x3 και *β)* το πλήθος των στοιχείων του, και θα επιστρέφει το άθροισμα των τετραγώνων των στοιχείων του πίνακα.

```
#include <stdio.h>
#define rows 4
#define columns 3
int add(int *pin, int number)
{
    int k, sum=0;
    for (k=0;k<number;k++) sum=sum+pin[k]*pin[k];
    // Ισοδύναμη έκφραση:
    //  for (k=0;k<number;k++) sum=sum+(*(pin+k))*(*(pin+k));
    return(sum);
}  // τέλος της συνάρτησης
void main()  {
    int i,j;
    int a[rows][columns];
    for (i=0;i<rows;i++)
        for (j=0;j<columns;j++)
        {
            a[i][j]=i*j;  // π.χ.
            printf( "a[%d][%d]=%d\n",i,j,a[i][j] );
        }
    printf( "Sum=%d\n",add(&a[0][0],rows*columns) );
}
```



Παράδειγμα 3.5

Να περιγραφεί αναλυτικά η λειτουργία του ακόλουθου προγράμματος και να δοθούν τα αποτελέσματά του.

```
#include <stdio.h>
void  f1 ( char *e );
void main()
{
      int a = 8, *b;
      f1("tei");
      b = &a;  a = 14;  *b = 13;
      printf( "\n%d\n",a );
}  // Τέλος της main
void  f1 ( char *e )
{
      char *s;
      s = e;
      while (*e)  e++;
      do
      {
            e - - ;
            printf( "%c",*e );
      }
      while( e > s );
}  // Τέλος της συνάρτησης f1
```

```
C:\TEMP\prog.exe
Initially:        s=t e=t
                   e[0]=t  e[1]=e  e[2]=i e[3]=
address(s)=4239887 address(e)=4239887

while loop
iteration 1:  e=t  address(e)=4239887
iteration 2:  e=e  address(e)=4239888
iteration 3:  e=i  address(e)=4239889

do loop
iteration 1:  e=i  address(e)=4239889
iteration 2:  e=e  address(e)=4239888
iteration 3:  e=t  address(e)=4239887

13
```

➢ Στον κορμό του προγράμματος, αρχικά ορίζονται η ακέραια μεταβλητή **a**, στην οποία δίνεται αρχική τιμή **8**, και ο δείκτης σε ακέραιο **b**. Στη συνέχεια καλείται η συνάρτηση ***f1()*** με όρισμα τη συμβολοσειρά **"tei"**, η οποία αποδίδεται στο δείκτη χαρακτήρα **e**, δηλαδή ο **e** θα δείχνει στη διεύθυνση του χαρακτήρα **t**.

➢ Μέσα στη συνάρτηση ***f1()*** ορίζεται ως τοπική μεταβλητή ο δείκτης **s**, στον οποίο αποδίδεται το περιεχόμενο του **e**, επομένως ο **s** θα δείχνει στο χαρακτήρα **t**. Ακολούθως εκτελείται ένας βρόχος ***while*** με τη συνθήκη *το περιεχόμενο της θέσης μνήμης να είναι διάφορο του μηδενικού χαρακτήρα*. Επομένως θα εκτελεσθεί 3 φορές, μεταφέροντας σε κάθε επανάληψη ο **e** κατά ένα byte παρακάτω. Στο τέλος του βρόχου ο **e** θα δείχνει στο **\0** του **"tei"**.

➢ Ακολουθεί ένας βρόχος ***do–while***, στον οποίο ο **e** μεταφέρεται ένα byte ψηλότερα και στη συνέχεια τυπώνεται το περιεχόμενο της διεύθυνσης στην οποία δείχνει. Ο βρόχος διαρκεί όσο ισχύει η συνθήκη **e>s**. Κατά συνέπεια ο βρόχος θα εκτελεσθεί τρεις φορές με τα αποτελέσματα να παρουσιάζονται κατωτέρω. Στο τέλος της τρίτης επανάληψης η συνθήκη καθίσταται ψευδής καθώς καθώς **e=s=4239887**, οπότε τερματίζεται η **f1()** και ο έλεγχος του προγράμματος περνά στη γραμμή

b = &a; a = 14; *b = 13;

Στη γραμμή αυτή ο δείκτης b δείχνει στη διεύθυνση του **a**. Ακολούθως η τιμή του **a** γίνεται **14**. Τέλος, το περιεχόμενο της διεύθυνσης στην οποία δείχνει ο **b** γίνεται **13**, δηλαδή η μεταβλητή **a** έμμεσα αποκτά την τιμή **13**. Η τιμή αυτή αποτυπώνεται στην οθόνη μέσω της τελευταίας εντολής του προγράμματος.

3.5.2 Συναρτήσεις με τύπο επιστροφής δείκτη

Έως τώρα οι δείκτες περνούσαν ως ορίσματα σε συναρτήσεις. Μπορούν όμως και οι συναρτήσεις να επιστρέφουν δείκτες, με την προϋπόθεση ότι ο επιστρεφόμενος δείκτης θα πρέπει να δείχνει σε δεδομένα της καλούσας συνάρτησης (π.χ. της ***main()***). Δεν πρέπει ποτέ να επιστρέφει δείκτης που δείχνει σε τοπική μεταβλητή της καλούμενης συνάρτησης, γιατί όταν τερματισθεί η συνάρτηση οι τοπικές μεταβλητές εξαφανίζονται.

Μία συνάρτηση που επιστρέφει δείκτη έχει την ακόλουθη μορφή:

<τύπος δεδομένων του επιστρεφόμενου δείκτη> *<όνομα συνάρτησης>(*παράμετροι*)

π.χ. στη δήλωση

char *incr(int x)

ο επιστρεφόμενος δείκτης είναι τύπου χαρακτήρα.

Για να αποφευχθούν πιθανά προβλήματα που οφείλονται στις ιδιαιτερότητες των δεικτών, προτείνεται να αποφεύγονται οι συναρτήσεις με επιστρεφόμενο δείκτη, εκτός εάν χρησιμοποιείται η λέξη κλειδί ***static***, όπως φαίνεται στο παράδειγμα 3.6.

Παράδειγμα 3.6

```
int *incr();
void main()
{
    int *pscore;
    pscore=incr();
    pscore++;
}
int *incr()
{
    int y;
    y=10;
```

```
        return(&y);
    }
```

Στο παραπάνω πρόγραμμα υπάρχει σφάλμα γιατί όταν τελειώνει η συνάρτηση ***incr()*** η **y** εξαφανίζεται και η τιμή της χάνεται. Ωστόσο, πίσω στη συνάρτηση ο **pscore** θα δείχνει στη διεύθυνση που επέστρεψε από τη συνάρτηση αλλά θα υπάρχει «σκουπίδι» (junk) σ' αυτή τη διεύθυνση, εφόσον το **y** έχει παύσει να ισχύει. Μάλιστα κατά τη μεταγλώττιση ο μεταγλωττιστής δίνει μήνυμα προειδοποίησης (warning) για «ύποπτη μετατροπή δείκτη» (suspicious pointer conversion). Όταν αντικατασταθεί η **int y;** από **static int y;** η μεταγλώττιση εκτελείται επιτυχώς, καθώς η στατική μεταβλητή **y** θα διατηρηθεί και μετά την έξοδο από τη συνάρτηση ***incr()***.

3.6 Δείκτες και δομές

Όπως κάθε μεταβλητή έτσι και μία μεταβλητή τύπου δομής (π.χ. δομή **addressT**) που ορίζεται από τη δήλωση

struct addressT addr1;

έχει διεύθυνση. Η διεύθυνση αυτή μπορεί να ληφθεί εφαρμόζοντας τον τελεστή διεύθυνσης στη μεταβλητή **addr1**. Εάν δηλωθεί ένας δείκτης σε δομή **addressT**, αυτός θα δείχνει στη μεταβλητή **addr1** με τη δήλωση:

stuct addressT *paddr;

paddr=&addr1;

Πλέον ο δείκτης **paddr** θα δείχνει στη δομή **addr1**, παρέχοντας έναν εναλλακτικό τρόπο πρόσβασης στα μέλη της.

Η προσπέλαση ενός μέλους της δομής μέσω ενός δείκτη γίνεται με χρήση του **τελεστή βέλους** ή **έμμεσης προσπέλασης** ή **δείκτη δομής** (structure pointer operator) (αποτελείται από το «μείον» και το «μεγαλύτερο» **->**). Η πρόταση

printf("%s\n",paddr->name);

τυπώνει το μέλος **name** της δομής **addr1** ενώ η πρόταση

paddr->zip_code=61124;

αναθέτει το 61124 στο **zip_code** της δομής **addr1**.

Η έκφραση **paddr->zip_code** είναι ισοδύναμη με την έκφραση **(*paddr).zip_code**. Οι παρενθέσεις είναι απαραίτητες επειδή ο τελεστής τελείας (**.**) έχει μεγαλύτερη προτεραιότητα από τον τελεστή (*).

3.6.1 Δείκτες εντός δομών

Ένας δείκτης μπορεί να αποτελεί μέλος δομής, π.χ.

```
struct struct_typeT
{
    int  *point1;
    char  *point2;
    float var3;
};
void main()
{
    struct struct_typeT deikt;
    int x=10;
    char y;
    deikt.point1=&x; // Ο δείκτης deikt.point1 δείχνει στη διεύθυνση της
                     // ακέραιας μεταβλητής x
    deikt.point2=&y; // Ο δείκτης deikt.point2 δείχνει στη διεύθυνση της
                     // μεταβλητής χαρακτήρα y
    *deikt.point1=13;        // Το περιεχόμενο της διεύθυνσης που
                             //  δείχνει ο δείκτης deikt.point1 γίνεται 13
    . . . . . .
}
```

Καθώς στην εντολή ***deikt.point1=13;** ο τελεστής (**.**) έχει υψηλότερη προτεραιότητα από τον τελεστή (*), προτείνεται να χρησιμοποιούνται παρενθέσεις έτσι ώστε να αποφευχθούν πιθανά σφάλματα:

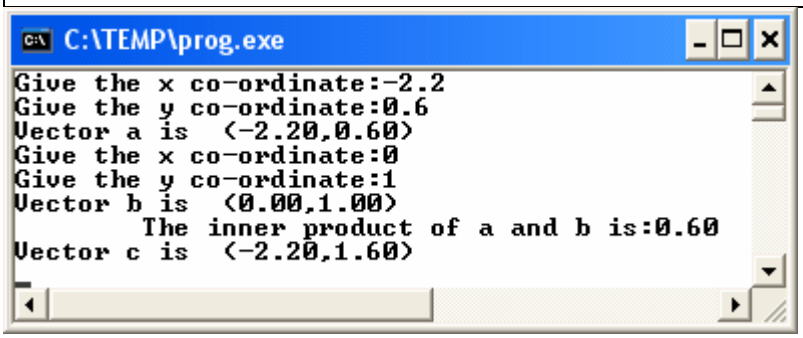
***(deikt.point1)=13;**

Παράδειγμα 3.7

Ο κώδικας που ακολουθεί χρησιμοποιεί δείκτες σε δομές για την ανάγνωση, την άθροιση και τον υπολογισμό του εσωτερικού γινομένου διανυσμάτων.

```
#include<stdio.h>
typedef struct vect      // Ορισμός της δομής vect
{
      float x,y;
} vector; //Τύπου vect
// Δήλωση συναρτήσεων
void prvect(char d, vector v);   // Εκτύπωση διανύσματος, κλήση κατ' αξία
void scanvect( vector *p);  // Ανάγνωση διανύσματος
float inprodr(vector *p, vector *q);// Εσωτερικό γινόμενο διανυσμάτων
vector addvectr(vector *p, vector *q);   // Άθροισμα διανυσμάτων
void main()
{
      vector a,b,c;
      scanvect(&a);
      prvect('a',a);
      scanvect(&b);
      prvect('b',b);
      printf("\tThe inner product of a and b is:%.2f\n",inprodr(&a,&b));
      c=addvectr(&a,&b);
      prvect('c',c);
}     // Τέλος της main
void prvect(char d, vector v)
{
      printf( "Vector %c is  ",d );
      printf( "(%.2f,%.2f)\n",v.x,v.y );
}     // Τέλος της prvect
void scanvect( vector *p)
```

```
{
    printf( "Give the x co-ordinate:" );
    scanf("%f",&p->x);
    printf( "Give the y co-ordinate:" );
    scanf("%f",&p->y);
}   // Τέλος της scanvect
float inprodr(vector *p, vector *q)
{
    return((*p).x*(*q).x+(p->y)*(q->y));
}   // Τέλος της inprodr
vector addvectr(vector *p, vector *q)
{
    vector sum;
    sum.x=(p->x)+(q->x);
    sum.y=(p->y)+(q->y);
    return(sum);
}   // Τέλος της addvectr
```



3.7 Δείκτες και η λέξη κλειδί typedef

Ενδιαφέρον παρουσιάζει η χρήση της λέξης κλειδί **typedef** σε συνδυασμό με δείκτες. Ειδικότερα, συχνά χρησιμοποιείται για να απλοποιήσει τις δηλώσεις μεταβλητών αλφαριθμητικού τύπου, με την πρόταση

typedef char *String;

Κατ' αυτόν τον τρόπο η ακόλουθη δήλωση είναι αφενός μεν έγκυρη αφετέρου δε ευανάγνωστη:

String pch, name[10];

Ο παρακάτω ορισμός είναι περισσότερο χαρακτηριστικός της απλοποίησης των δηλώσεων:

typedef int (*func)(char *, char *);

Η δήλωση καθορίζει ότι το όνομα **func** αναπαριστά τον τύπο που είναι δείκτης σε συνάρτηση, η οποία έχει ως τυπικές παραμέτρους δύο δείκτες σε χαρακτήρες και επιστρέφει ακέραιο. Χρησιμοποιώντας το νέο όνομα, η δήλωση δείκτη σε συνάρτηση αυτής της μορφής, που θα έπρεπε να είναι

int (*pfunction)(char *, char *)

παίρνει την απλή μορφή

func pfunction;

Μετά από αυτή τη δήλωση μπορεί να γραφεί

pfunction=strcmp;

για να δείχνει ο δείκτης **pfunction** στη συνάρτηση ***strcmp()*** της βασικής βιβλιοθήκης.

3.8 Ορίσματα της γραμμής διαταγής

Όπως κάθε συνάρτηση, έτσι κι η ***main()*** μπορεί να δεχθεί παραμέτρους, οι οποίες επιτρέπουν να δίνεται στο καλούμενο πρόγραμμα ένα σύνολο από εισόδους, που καλούνται **ορίσματα γραμμής διαταγής** (command line arguments). Ο μηχανισμός περάσματος ορισμάτων βασίζεται στην ακόλουθη δήλωση της ***main()***:

```
void main(int argc, char *argv[])
{
    ......
}
```

όπου

- **argc** (*argument count*): είναι ο αριθμός των ορισμάτων της γραμμής διαταγής, συμπεριλαμβανομένου και του ονόματος του προγράμματος.
- **argv** (*argument vector*): είναι δείκτης σε πίνακα από δείκτες, που δείχνουν στα ορίσματα της γραμμής διαταγής, τα οποία αποθηκεύονται με τη μορφή

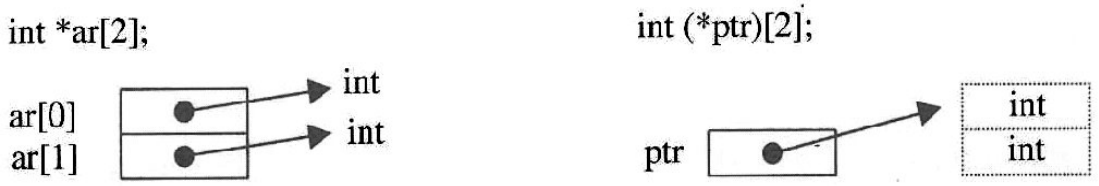
αλφαριθμητικών. Ο πίνακας στον οποίο δείχνει ο **argv** έχει ένα επιπλέον στοιχείο, το **argv[argc]**, το οποίο έχει τιμή **NULL**.

Παρατήρηση: Σε μία έκφραση δήλωσης, ο τελεστής πίνακα έχει μεγαλύτερη προτεραιότητα από τον τελεστή (*). Οι δύο παρακάτω δηλώσεις βασίζονται στο γεγονός αυτό:

int *ar[2];

int (*ptr)[2];

Η πρώτη δήλωση ορίζει έναν πίνακα δύο δεικτών σε ακεραίους και στο χρόνο εκτέλεσης έχει ως αποτέλεσμα, όπως φαίνεται στο ακόλουθο σχήμα, τη δέσμευση δύο θέσεων μνήμης για μελλοντική αποθήκευση δεικτών σε ακεραίους. Αντίθετα, η δεύτερη δήλωση ορίζει ένα δείκτη σε πίνακα δύο ακεραίων, τον οποίο όμως δε δηλώνει και κατά συνέπεια δε δεσμεύει τον απαιτούμενο χώρο.



Παράδειγμα 3.8

Να καταστρωθεί πρόγραμμα που θα τυπώνει τα ορίσματα της γραμμής διαταγής.

Έστω **echo** το όνομα του προγράμματος. Όταν το καλούμε με την εντολή

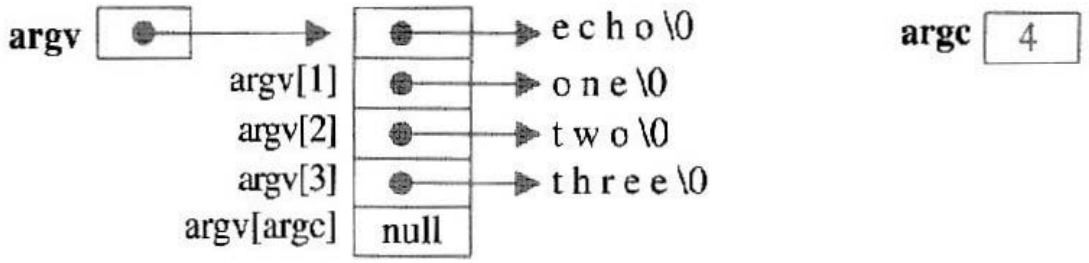
echo one two three

θα πρέπει να εκτελείται και να τυπώνει στην οθόνη **one two three**.

Το σώμα της ***main()*** έχει πρόσβαση στις μεταβλητές **argc** και **argv** όπως παριστάνονται από το ακόλουθο σχήμα, με βάση το οποίο η διαμόρφωση του σώματος της **main** είναι απλή:

```
void main(int argc, char *argv[])
{
    int i;
    for (i=1; i<argc; i++) printf( "%s%s",argv[i], " " );
```

```
        printf( "\n" );
    }
```



Η **printf("%s%s",argv[i], " ");** τυπώνει μετά από κάθε όρισμα ένα κενό. Εάν δεν πρέπει να τυπώνεται το κενό μετά το τελευταίο όρισμα, η πρόταση πρέπει να διαμορφωθεί όπως παρακάτω:

for (i=1; i<argc; i++) printf("%s%s",argv[i],(i<argc-1)? " ":" ");

Εάν χρησιμοποιηθεί η σημειολογία των δεικτών αντί αυτής του πίνακα, η πρόταση μπορεί να γραφεί ως εξής:

while (--argc) printf("%s%s",*++argv,(i<argc-1)? " ":" ");

---

4. Δυναμική διαχείριση μνήμης - δείκτες σε αλφαριθμητικά

4.1 Η έννοια της δυναμικής διαχείρισης μνήμης

Όταν δηλώνεται ένας πίνακας προσδιορίζεται το μέγεθός του, το οποίο αποτελεί το μέγιστο αριθμό στοιχείων που μπορεί να έχει ο πίνακας, και αυτό παραμένει σταθερό καθόλη τη διάρκεια του προγράμματος, ανεξάρτητα από τον πραγματικό αριθμό στοιχείων του πίνακα που θα χρησιμοποιηθούν. Για παράδειγμα, η δήλωση

int array[40];

δεσμεύει 40 τετράδες bytes έως ότου τελειώσει το πρόγραμμα. Αυτός ο τρόπος δέσμευσης μνήμης είναι στατικός και δεν μπορεί να ανταποκριθεί στην περίπτωση που το μέγεθος του πίνακα πρέπει είτε να επιλέγεται είτε να μεταβάλλεται μετά την έναρξη εκτέλεσης του προγράμματος.

Στη C υπάρχουν δομές όπως η στοίβα (stack) ή 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 το 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 διαγράφοναι παλιές. Σε μία τέτοια περίπτωση θα πρέπει να γίνεται *δυναμική διαχείριση της μνήμης*: 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.

Η C υποστηρίζει τη δυναμική διαχείριση μνήμης παρέχοντας ένα σύνολο από συναρτήσεις της βασικής βιβλιοθήκης. Οι συνήθεις συναρτήσεις διαχείρισης μνήμης είναι:

- Οι ***malloc()***, ***calloc()*** για τον καθορισμό του μεγέθους της εκχωρούμενης μνήμης κατά την εκτέλεση.
- Η ***realloc()*** για την αλλαγή του μεγέθους της εκχωρούμενης μνήμης κατά την

εκτέλεση.

- Η ***free()*** για την απελευθέρωση μνήμης.

4.2 Οι συναρτήσεις malloc, calloc και free

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

void * malloc(int size);

Η ***malloc()*** επιστρέφει ένα δείκτη στην αρχή του μπλοκ, στο οποίο γίνεται η εκχώρηση. Πρέπει πάντοτε 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. Δε 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 δεσμευθεί μνήμη για 20 ακεραίους η ***malloc()*** συντάσσεται ως εξής:

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

Στην παραπάνω πρόταση το **size=20*sizeof(int)** και γίνεται μετατροπή τύπου (**int** *) ώστε να ταιριάζει η έξοδος της ***malloc()*** με το 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);
```

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

Η συνάρτηση ***calloc()*** χρησιμοποιείται για τη δέσμευση μνήμης, δεσμεύοντας χώρο για έναν πίνακα **n** στοιχείων, μεγέθους **size** το καθένα. Ορίζεται στα αρχεία κεφαλίδας ***stdlib.h*** και ***alloc.h*** και δηλώνεται ως εξής:

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

Η ***calloc()*** επιστρέφει ένα 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 τη αποδέσμευση μνήμης. Δεσμεύει ένα μπλοκ διαδοχικών θέσεων μνήμης. Ορίζεται στα αρχεία κεφαλίδας ***stdlib.h*** και ***alloc.h*** και δηλώνεται ως εξής:

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

Η ***free()*** δέχεται ως όρισμα ένα δείκτη, ο οποίος δείχνει στην αρχή του μπλοκ που απελευθερώνεται. Για παράδειγμα, εάν έχει δεσμευθεί μνήμη για 20 ακεραίους η ***free()*** συντάσσεται ως εξής:

```
int *pstart;
```

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

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

Παρατηρήσεις:

1. 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 */
```

2. Όταν σε ένα πρόγραμμα γίνουν επαναλαμβανόμενες εκχωρήσεις μνήμης χωρίς τις αντίστοιχες απελευθερώσεις, συμβαίνουν «διαρροές μνήμης»: το πρόγραμμα αυξάνει συνεχώς 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.

3. Δε 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
```

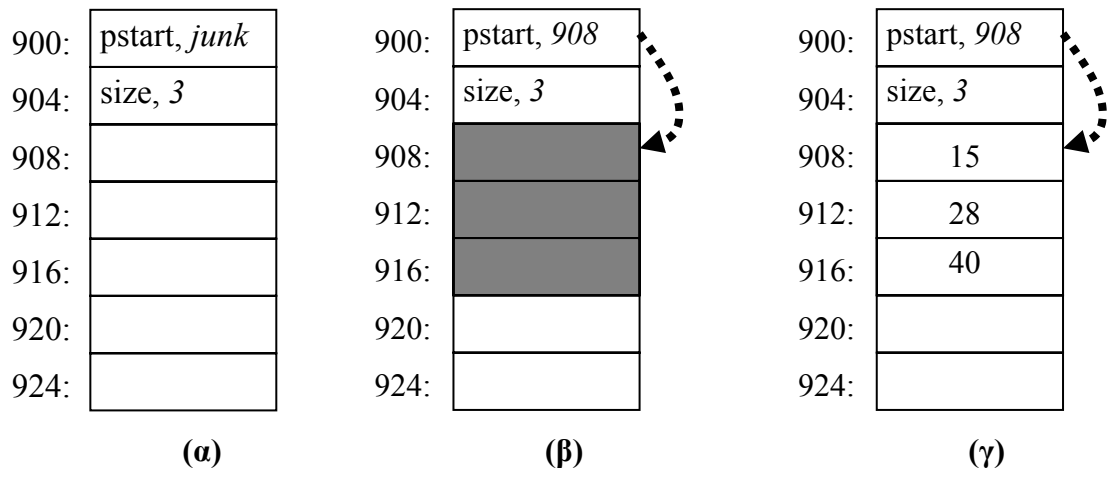
```
free(pmsg);
free(pmessage);   // ΛΑΘΟΣ: Το μπλοκ μνήμης έχει ήδη απελευθερωθεί!
```

Παράδειγμα 4.1

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

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int *pstart;
    int size=3;
    pstart=(int *) malloc (size*sizeof(int));
    *pstart=15;
    *(start+1)=28;
    *(start+2)=*(start+1)+12;
    free(pstart);
}
```

- Δηλώνονται ο δείκτης σε ακέραιο **pstart** και η ακέραια μεταβλητή **size**, η οποία αρχικοποιείται λαμβάνοντας την τιμή **3** (σχήμα 4-1α).
- Η ***malloc()*** αναζητά ένα συνεχές μπλοκ 3x4=12 bytes και όταν το βρει επιστρέφει ένα δείκτη στην αρχή του μπλοκ. Δηλαδή, **επιστρέφει τη διεύθυνση** του πρώτου byte σ' αυτό το μπλοκ. Η διεύθυνση αυτή ανατίθεται στο δείκτη **pstart** (σχήμα 4-1β).
- 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 4-1γ).
- Με χρήση της ***free()*** απελευθερώνεται η δεσμευθείσα μνήμη και ο χάρτης μνήμης επανέρχεται στην αρχική του μορφή (σχήμα 4-1α).



Σχήμα 4-1. *Χάρτης μνήμης*

4.3 Η συνάρτηση realloc

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

void * realloc(void *block, int size);

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

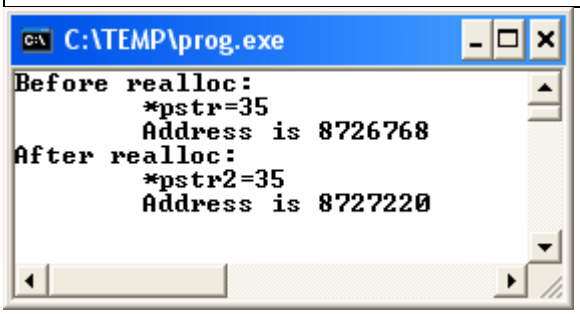
Η λειτουργία της ***realloc()*** αναδεικνύεται μέσα από το παράδειγμα 4.2.

Παράδειγμα 4.2

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. Από τα αποτελέσματα διαπιστώνεται ότι:

α) το νέο μπλοκ μνήμης μετακινήθηκε σε άλλο σημείο της μνήμης και β) τα περιεχόμενα του αρχικού μπλοκ διατηρήθηκαν αναλλοίωτα.

```
#include <stdio.h>
#include <alloc.h>
void main()
{
    int *pstr,*pstr2;
    pstr = (int *) malloc(10*sizeof(int));
    *pstr=35;
    *(pstr+1)=-12;
    printf( "Before 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);
}
```



4.4 Πίνακες δεικτών

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

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

Η πρόταση

char *name[3];

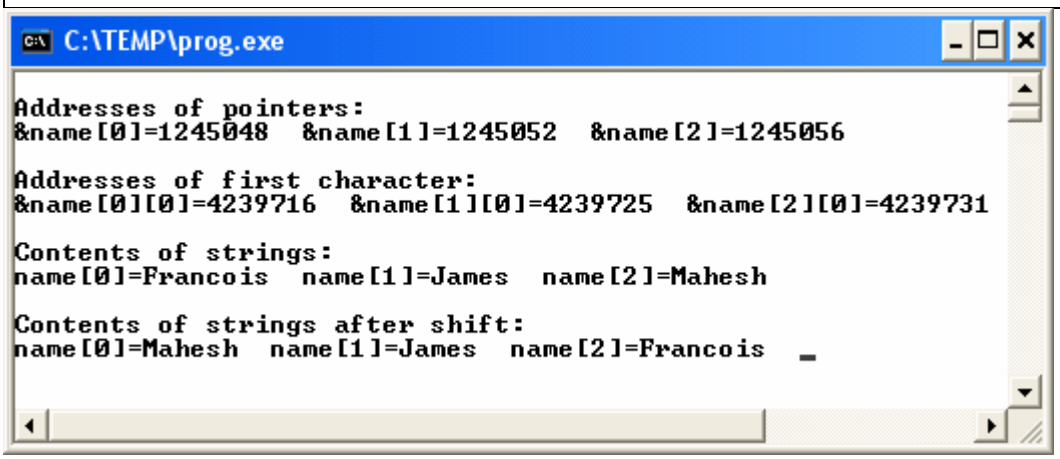
ορίζει τον πίνακα **name** τριών θέσεων, τα στοιχεία του οποίου είναι δείκτες σε χαρακτήρα. Με αυτόν τον τρόπο τα στοιχεία του πίνακα δείχνουν σε αλφαριθμητικά και ο δείκτης (index) του πίνακα επιλέγει ένα αλφαριθμητικό. Η λειτουργία του πίνακα αυτού θα περιγραφεί με τη βοήθεια του ακόλουθου παραδείγματος ενώ περισσότερα στοιχεία για τη χρήση δεικτών σε αλφαριθμητικά θα δοθούν στην §4.6.

Παράδειγμα 4.3

Να γίνει αναλυτική περιγραφή του ακόλουθου προγράμματος και αιτιολόγηση των αποτελεσμάτων.

```
#include<stdio.h>
void main()
{
    int i;
    char  *name[3] = { "Francois","James","Mahesh" };
    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 shift:\n");
        for (i=0; i<3; i++)  printf( "name[%d]=%s  ",i,name[i] );
    }
```



➢ Αρχικά δημιουργείται ο πίνακας δεικτών **name** με στοιχεία τους δείκτες χαρακτήρα, **name[0]**, **name[1]**, **name[2]**, οι οποίοι αποθηκεύονται στα bytes **1245048-1245051**, **1245052-1245055** και **1245056-1245059**, αντίστοιχα.

➢ Σε κάθε δείκτη αποδίδεται η διεύθυνση του πρώτου byte ενός αλφαριθμητικού, τα οποία αποθηκεύονται σε διαφορετικό τμήμα της μνήμης. Ειδικότερα, στο δείκτη **name[0]** αντιστοιχεί το αλφαριθμητικό **"Francois"**, 8 χαρακτήρων, το οποίο αποθηκεύεται στις διευθύνσεις **4239716-4239723**, ενώ στο byte **4239724** αποθηκεύεται ο μηδενικός χαρακτήρας τερματισμού του αλφαριθμητικού. Κατ' αντίστοιχο τρόπο ο **name[1]** δείχνει στο **"James"** και ο **name[2]** στο **"Mahesh"**.

➢ Με τη βοήθεια του δείκτη χαρακτήρα **temp** ανταλλάσονται οι διευθύνσεις των **name[0]** και **name[2]**. Στο τέλος του προγράμματος ο **name[0]** δείχνει στο **"Mahesh"** και ο **name[2]** στο **"Francois"**.

4.5 Δείκτες σε δείκτες

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.

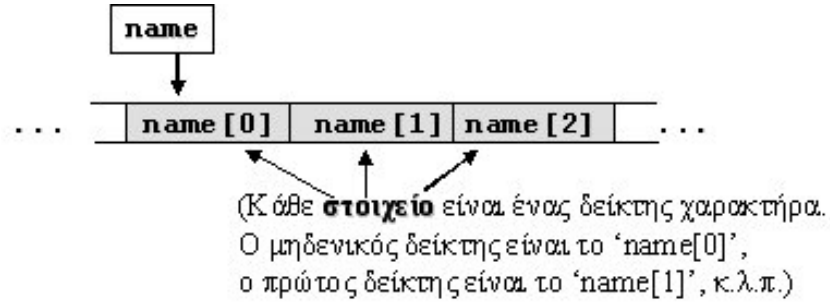
Η έμμεση αναφορά μπορεί να λάβει ένθεση οιουδήποτε βάθους (δείκτης σε δείκτη σε δείκτη κ.λ.π.), ωστόσο θα πρέπει να αποφεύγονται οι υπερβολές γιατί ο κώδικας αφενός μεν θα γίνει δυσανάγνωστος αφετέρου δε θα είναι επιρρεπής σε σφάλματα.

Για την ανάλυση της λειτουργίας των δεικτών σε δείκτες θα χρησιμοποιηθεί το ακόλουθο πρόγραμμα, όπου μελετάται η περίπτωση ***δεικτών σε αλφαριθμητικά***:

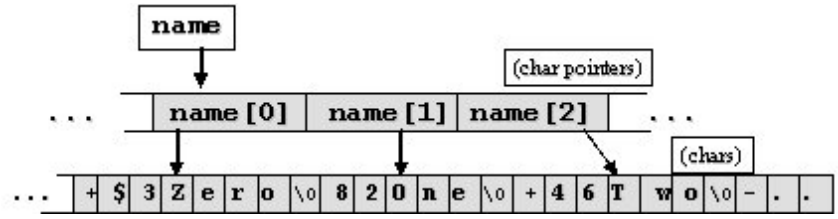
```
    #include <stdio.h>
    void main()
    {
1       char **name;     // pointer-to-(pointers-to char)
2       name=(char **)malloc(3*sizeof(char *));
3       name[0]="Zero";
4       name[1]= "One";
5       name[2]="Two";
6       printf( "%s,%s,%s,",name[0],name[1],name[2]);
7       printf( "%c,%c,%c!\n",name[0][0],name[1][0],name[2][0]);
8       free(name);
    }
```

➢ *Γραμμή 1*: Δηλώνεται ένας δείκτης που δείχνει σε μία λίστα δεικτών σε χαρακτήρα.

➢ *Γραμμή 2*: Δεσμεύεται ένα μπλοκ μνήμης, επαρκές για 3 δείκτες σε χαρακτήρα. Στη συνέχεια ανατίθεται στο **name** ένας δείκτης σε αυτό το μπλοκ. Επομένως ο **name** δείχνει στη δεσμευμένη μνήμη και όχι το ****name**.

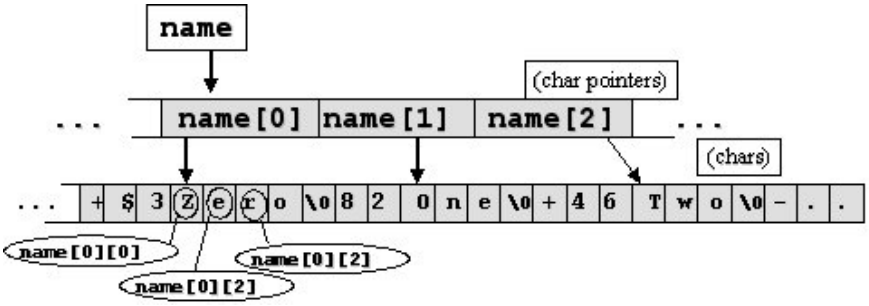


➢ *Γραμμές 3-5*: Κάθε στοιχείο του πίνακα **name** δείχνει σε μία αλφαριθμητική σταθερά. Τα τρία αλφαριθμητικά δεν είναι κατ' ανάγκη τοποθετημένα διαδοχικά στη μνήμη:



➢ *Γραμμή 6*: Χρησιμοποιείται ο πρώτος *δείκτης* (index) του πίνακα για να επιλεγεί αλφαριθμητικό.

➢ *Γραμμή 7*: Ο δεύτερος *δείκτης* επιλέγει χαρακτήρες στο αλφαριθμητικό.



➢ *Γραμμή 8*: Απελευθερώνεται το μπλοκ μνήμης που είχε δεσμευτεί με το δείκτη **name**, αποδεσμεύοντας τις θέσεις μνήμης που καταλάμβαναν οι **name[0]**, **name[1]**, **name[2]**, με αποτέλεσμα τα περιεχόμενα των θέσεων μνήμης **name[0][0]** κ.λ.π. να θεωρούνται ανενεργά («σκουπίδια»).

Η χρήση δεικτών για το χειρισμό αλφαριθμητικών παρουσιάζει μία σειρά πλεονεκτημάτων:

- Μπορούν να εκτελεσθούν πράξεις ακεραίων.

- Μπορούν να χρησιμοποιηθούν ως κινητά ονόματα πινάκων. Ο *δείκτης πίνακα* (array index) [ ] λειτουργεί!

***(arr+n) = arr[n]**

arr+n = &arr[n]

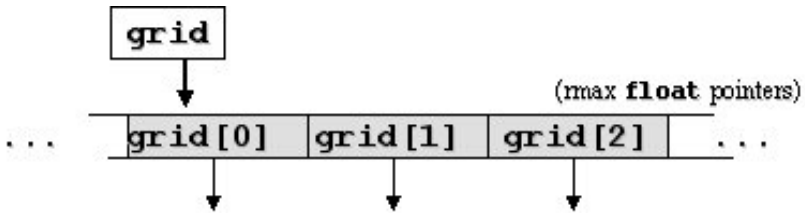
- Οι δείκτες μπορούν να ορίσουν ένα αλφαριθμητικό όπως ακριβώς το ορίζει ένας πίνακας.

4.5.1 Πολυδιάστατοι πίνακες με δεδομένα αριθμητικών τύπων

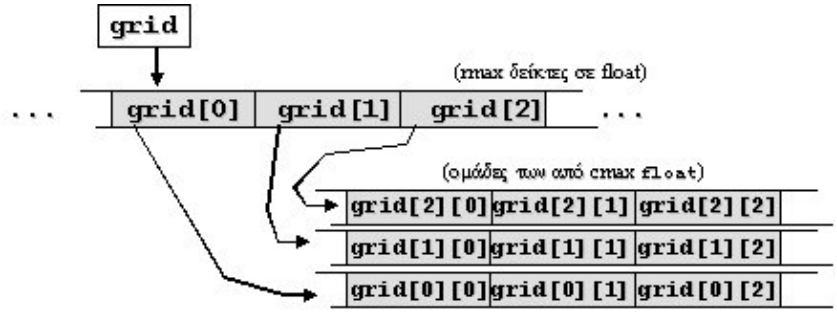
Στην περίπτωση δεικτών σε δείκτες κατά την οποία αποθηκεύονται αριθμητικοί τύποι δεδομένων (int, float, double) η εκχώρηση και αποδέσμευση μνήμης ακολουθούν διαφορετική διαδικασία, η οποία θα περιγραφεί με τη βοήθεια του ακόλουθου προγράμματος:

```
      #include <stdio.h>
      void main()
      {
            int i, rmax=13, cmax=7;
1           float **grid; // pointer-to-(pointers-to-float)
2           grid = (float **)malloc(rmax*sizeof(float *));
3           for(i=0; i<rmax; i++)
            {
                  grid[i]=(float *)malloc(cmax*sizeof(float));
            }
            . . . . . .
4           for(i=0; i<rmax; i++)
            {
                  free(grid[i]);
            }
5           free(grid);
      }
```

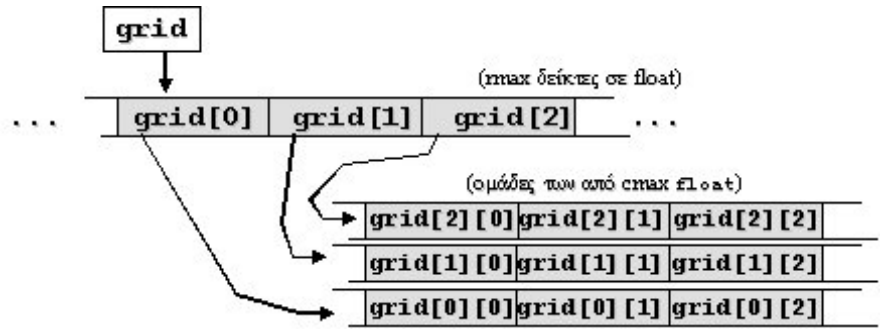
- *Γραμμή 1*: Δηλώνεται ένας δείκτης που δείχνει σε μία λίστα δεικτών σε float.
- *Γραμμή 2*: Δεσμεύεται ένα μπλοκ μνήμης, επαρκές για **rmax** δείκτες σε float.



- *Γραμμή 3:* Δημιουργείται ένα μπλοκ από **cmax** αριθμούς κινητής υποδιαστολής για κάθε δείκτη σε float.



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



Μακροσκοπικά η προσπέλαση πινάκων με χρήση δεικτών είναι απολύτως ίδια με την προσπέλαση των κλασικών πινάκων, π.χ. το στοιχείο **grid[1][2]** αντιστοιχεί στη δεύτερη γραμμή και τρίτη στήλη του πίνακα. Η διαφορά έγκειται στο ότι στους πίνακες με δείκτες υπάρχουν **rmax** ομάδες από **cmax** στοιχεία και οι ομάδες δεν καταλαμβάνουν κατ' ανάγκη διαδοχικές θέσεις μνήμης. Όταν ζητείται το στοιχείο **grid[1][2]** ουσιαστικά ζητείται να προσπελασθεί ο δεύτερος δείκτης της λίστας των δεικτών και να ληφθεί η τρίτη τιμή των float, στους οποίους δείχνει ο συγκεκριμένος

δείκτης.

Παράδειγμα 4.4

Με χρήση των ***malloc*** και ***free*** να γραφεί τμήμα κώδικα για τη δέσμευση κι απελευθέρωση μνήμης ενός πίνακα ακεραίων 25x32, για τον οποίο χρησιμοποιείται ο δείκτης ****pinakas**. Σε κάθε εκχώρηση μνήμης να γίνεται έλεγχος για την ύπαρξη διαθέσιμης μνήμης.

```
int **pinakas;
// malloc:
pinakas=(int **)malloc(25*sizeof(int *));      assert(pinakas!=NULL);
for (i=0; i<25; i++)
{
      pinakas[i]=(int *)malloc(32*sizeof(int));
      assert(pinakas[i]!=NULL);
}
// free:
for (i=24; i>=0; i--)     free( pinakas[i] );
free( pinakas );
```

4.6 Δείκτες και συναρτήσεις αλφαριθμητικών

4.6.1 Η συνάρτηση εύρεσης χαρακτήρα σε αλφαριθμητικό

Η συνάρτηση ***strchr(str1,ch)*** βρίσκει το χαρακτήρα **'ch'** μέσα στο string **str1**. Στην πρώτη εύρεση του **'ch'** επιστρέφει ένα δείκτη σε χαρακτήρα. Δέχεται δύο ορίσματα: το πρώτο όρισμα είναι το όνομα του αλφαριθμητικού και το δεύτερο όρισμα είναι ο χαρακτήρας. Η συνάρτηση ***strchr(str1,ch)*** ορίζεται στο αρχείο κεφαλίδας ***string.h***.

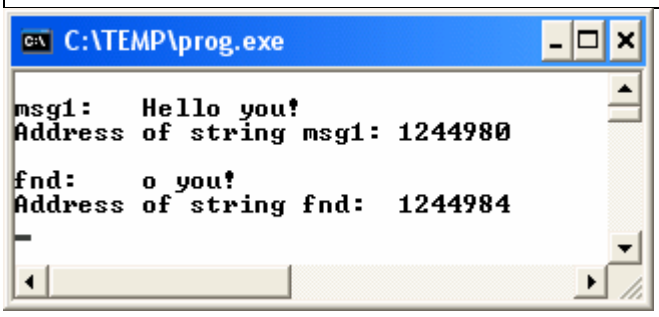
Θα πρέπει να σημειωθεί ότι δείκτης που επιστρέφει η συνάρτηση ουσιαστικά

αντιστοιχεί στο τμήμα του αλφαριθμητικού που αρχίζει από τον υπό εύρεση χαρακτήρα, όπως φαίνεται στο παράδειγμα που ακολουθεί.

Παράδειγμα 4.5

Εύρεση του χαρακτήρα **'o'** μέσα στο αλφαριθμητικό **"Hello you!"**.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char msg1[81]={ "Hello you!" };
    char *fnd;
    printf( "\nmsg1:\t%s\n",msg1 );
    printf( "Address of string msg1:\t%d\n",msg1 );
    fnd = strchr(msg1,'o');
    printf( "\nfnd:\t%s\n",fnd );
    printf( "Address of string fnd:\t%d\n",fnd );
}
```



4.6.2 Η συνάρτηση εύρεσης αλφαριθμητικού σε αλφαριθμητικό

Η συνάρτηση ***strstr(str1,str2)*** βρίσκει το αλφαριθμητικό **str2** μέσα στο string **str1**. Στην πρώτη εύρεση του **str2** επιστρέφει ένα δείκτη σε χαρακτήρα. Δέχεται δύο ορίσματα: το πρώτο όρισμα είναι το όνομα του αλφαριθμητικού στο οποίο θα γίνει η

αναζήτηση και και το δεύτερο όρισμα είναι το όνομα του προς εύρεση αλφαριθμητικού. Και η συνάρτηση ***strstr(str1,str2)*** ορίζεται στο αρχείο κεφαλίδας ***string.h***.

Ο δείκτης που επιστρέφει η συνάρτηση δείχνει στο τμήμα του αλφαριθμητικού **str1** που αρχίζει από τον πρώτο χαρακτήρα του **str2** και φθάνει έως το τέλος του **str1**, όπως φαίνεται στο παράδειγμα που ακολουθεί.

Παράδειγμα 4.6

Εύρεση του αλφαριθμητικού **"yo"** μέσα στο αλφαριθμητικό **"Hello you!"**.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char msg1[81]={ "Hello you!" };
    char *fnd;
    printf( "\nmsg1:\t%s\n",msg1 );
    printf( "Address of string msg1:\t%d\n",msg1 );
    fnd = strstr(msg1,"yo");
    printf( "\nfnd:\t%s\n",fnd );
    printf( "Address of string fnd:\t%d\n",fnd );
}
```

Παράδειγμα 4.7

Να δοθούν τα πρωτότυπα των συναρτήσεων:

α) ***strlen()*** **(**μήκος αλφαριθμητικού).

β) ***strcpy()*** (αντιγραφή ενός αλφαριθμητικού σε ένα άλλο).

α) Θεωρώντας ότι το όρισμα που δέχεται η συνάρτηση είναι δείκτης σε χαρακτήρα, η δήλωση διαμορφώνεται ως εξής:

int strlen(char *s);

Το σώμα της συνάρτησης με πίνακες και με δείκτες δίνεται παρακάτω:

int strlen(char str[ ]) **{** **int i=0;** **while (str[i++]);** **return(i-1);** **}**	**int strlen(char *s)** **{** **char *p=s;** **while (*s!='\0') s++;** **return(s-p);** **}**

Ο δείκτης **s** δείχνει στη διεύθυνση του αλφαριθμητικού (στον πρώτο χαρακτήρα). Με την εντολή **char *p=s;** ο δείκτης **p** αποκτά το περιεχόμενο του **s**, δείχνοντας και αυτός στον πρώτο χαρακτήρα. Ακολούθως ο βρόχος ***while*** εκτελείται όσο ο **s** δε δείχνει στο μηδενικό χαρακτήρα: σε κάθε επανάληψη ο **s** δείχνει στον επόμενο χαρακτήρα. Όταν ο βρόχος τερματίζεται, ο **s** δείχνει στο μηδενικό χαρακτήρα, οπότε η εντολή **s-p** εκτελεί αφαίρεση δεικτών και δίνει τον αριθμό των στοιχείων μεταξύ των δύο δεικτών, δηλαδή το πραγματικό μήκος του αλφαριθμητικού, δηλαδή χωρίς το μηδενικό χαρακτήρα.

β) Ενεργώντας αντίστοιχα με το προηγούμενο σκέλος του παραδείγματος και θεωρώντας ότι τα ορίσματα που δέχεται η συνάρτηση είναι δείκτες σε χαρακτήρα, η δήλωση διαμορφώνεται ως εξής:

void strcpy(char *s, char *t);

Το σώμα της συνάρτησης με πίνακες και με δείκτες δίνεται παρακάτω:

void strcpy(char s[ ], char t[ ]) **{** **int i=0;** **while ((s[i]=t[i])!='\0') i++;** **}**	**void strcpy(char *s, char *t)** **{** **while ((*s=*t)!='\0')** **{** **s++;** **t++;** **}** **}**

5. Αρχεία

5.1 Γενικά

Τα **αρχεία** (files) μπορούν να θεωρηθούν ως σύνθετοι τύποι δεδομένων, οι οποίοι δεν αποθηκεύουν τα δεδομένα τους στην κύρια μνήμη αλλά σε εξωτερικά μέσα αποθήκευσης, όπως οι σκληροί δίσκοι, οι δισκέτες, τα cd κ.λ.π. Με τον τρόπο αυτό τα δεδομένα ενός αρχείου δεν εκλείπουν με το πέρας του προγράμματος στο οποίο δημιουργήθηκαν αλλά διατηρούνται στα μέσα αποθήκευσης και μπορούν να ανακτηθούν και να τροποποιηθούν ανά πάσα στιγμή.

Η C θεωρεί κάθε αρχείο ως μία σειριακή ακολουθία από bytes. Το τέλος ενός αρχείου σηματοδοτείται από το ***τέλος αρχείου*** (end-of-file, **EOF**), που είναι ένας ακέραιος με τιμή -1.

5.1.1 Τα κανάλια stdin, stdout, stderr

Κάθε φορά που ξεκινά η εκτέλεση ενός προγράμματος, ο υπολογιστής ανοίγει αυτόματα:

- Το κανάλι καθιερωμένης εισόδου ***stdin*** (standard input), το οποίο χρησιμοποιείται για ανάγνωση από την κονσόλα. Όταν γίνεται χρήση των ***scanf()***, ***gets()*** για να αναγνωσθούν δεδομένα από το πληκτρολόγιο είναι σαν να γίνεται ανάγνωση από το «αρχείο» ***stdin***.
- Το κανάλι καθιερωμένης εξόδου ***stdout*** (standard output) και το κανάλι σφαλμάτων ***stderr*** (standard errors), τα οποία χρησιμοποιούνται για εκτύπωση στην κονσόλα. Όταν γίνεται χρήση των ***printf()***, ***puts()*** για να εκτυπωθούν δεδομένα στην οθόνη είναι σαν να γράφονται τα δεδομένα στο «αρχείο» ***stdin***.

Τα ανωτέρω κανάλια ή ροές (streams) αποτελούν τα μέσα επικοινωνίας των αρχείων με τα προγράμματα, καθώς επειδή αποτελούν **δείκτες αρχείου** (file pointers, **FILE ***) μπορούν να χρησιμοποιηθούν σε οποιαδήποτε συνάρτηση χρησιμοποιεί μία μεταβλητή τύπου **FILE**. Έτσι, το κανάλι ***stderr*** μπορεί να ανακατευθυνθεί και να γράφονται τα μηνύματα λάθους σε αρχείο αντί να εμφανίζονται στην οθόνη.

- Με την εντολή **program_name < filename** ορίζεται ως κύρια είσοδος αντί του πληκτρολογίου το αρχείο **filename**.
- Με την εντολή **program_name > filename** ορίζεται ως κύρια έξοδος αντί για την οθόνη το αρχείο **filename**.

Οι παραπάνω εντολές δίνονται από τη γραμμή διαταγής (command line).

Παρατήρηση: Τα ***stdin***, ***stdout***, ***stderr*** δεν είναι μεταβλητές αλλά σταθερές και δεν μπορούν να αλλαχθούν. Όπως ο υπολογιστής δημιουργεί αυτόματα αυτούς τους δείκτες αρχείου στην αρχή του προγράμματος, έτσι και τους αποσύρει αυτόματα στο τέλος του προγράμματος. Δε θα πρέπει να κλείσουν αυτά τα κανάλια με παρέμβαση του χρήστη.

5.1.2 Η ενδιάμεση μνήμη - δείκτης αρχείου

Για ανάγνωση και εγγραφή σε συσκευές εισόδου/εξόδου (input/output, **I/O**), όπως ο σκληρός δίσκος, τα λειτουργικά συστήματα χρησιμοποιούν **ενδιάμεση μνήμη** (buffers), η οποία είναι περιοχή της κύριας μνήμης όπου τα δεδομένα αποθηκεύονται προσωρινά πριν σταλούν στον τελικό τους στόχο. Έτσι επιταχύνονται τα προγράμματα γιατί ελαχιστοποιείται ο αριθμός των προσβάσεων στις I/O συσκευές.

Οι μονάδες I/O επιτρέπουν στο λειτουργικό σύστημα να έχει πρόσβαση μόνο σε καθορισμένου μεγέθους τμήματα, τα ονομαζόμενα **blocks**, μεγέθους 512 ή 1024 bytes. Επομένως, ακόμη κι αν θέλουμε να διαβάσουμε μόνο ένα χαρακτήρα από ένα αρχείο, το λειτουργικό σύστημα διαβάζει όλο το μπλοκ στο οποίο βρίσκεται αποθηκευμένος ο χαρακτήρας. Έτσι, με τη χρήση του buffer, εάν χρειασθούμε άλλους χαρακτήρες από το ίδιο μπλοκ δεν επιστρέφουμε στη συσκευή αλλά τους διαβάζουμε από το buffer.

Το νήμα που κρατάει ενωμένο το **σύστημα I/O με ενδιάμεση αποθήκευση**, δηλαδή με χρήση της ενδιάμεσης μνήμης, είναι ο δείκτης αρχείου. Ο δείκτης αρχείου δείχνει σε πληροφορίες που καθορίζουν διάφορα ζητήματα του αρχείου, όπως είναι το όνομά του, η κατάστασή του και η τρέχουσα θέση του. Ουσιαστικά ο δείκτης αρχείου κατονομάζει ένα συγκεκριμένο αρχείο στο μέσο αποθήκευσης (π.χ. σκληρός δίσκος) και χρησιμοποιείται από το σχετικό κανάλι για να κατευθύνει τις συναρτήσεις του συστήματος I/O εκεί όπου πρέπει να ενεργήσουν. Ο τύπος του δείκτη αρχείου

(**FILE**) ορίζεται στο ***stdio.h***. Για την ανάγνωση ή την εγγραφή αρχείου πρέπει να χρησιμοποιούνται δείκτες αρχείου. Μία μεταβλητή δείκτη αρχείου δηλώνεται ως εξής:

FILE *fp;

Για λόγους συμβατότητας στο συμβολισμό, έχει επικρατήσει τα ονόματα των δεικτών αρχείου να αρχίζουν από ***f*** (**f**ile).

5.1.3 Κατηγορίες αρχείων

Η C υποστηρίζει δύο κατηγορίες αρχείων, ανάλογα με τον τρόπο που αποθηκεύονται τα δεδομένα:

- Τα **δυαδικά αρχεία** (binary files), τα οποία αποθηκεύουν κάθε τύπο δεδομένου: char, int, float, double, απαριθμητικό τύπο κ.λ.π. Ο τρόπος αποθήκευσης είναι ίδιος με εκείνον της κύριας μνήμης, δηλαδή ο μεταγλωττιστής δεν κάνει μεταγλώττιση των bytes, απλά διαβάζει και γράφει bits, ακριβώς όπως αυτά εμφανίζονται. Για παράδειγμα, ο αριθμός 12345678 εγγράφεται σε δυαδικό αρχείο ως ακέραιος, απαιτώντας 4 bytes.

  Τα δυαδικά αρχεία συνήθως δεν είναι αναγνώσιμα από τους συντάκτες (editors) και αναγιγνώσκονται μέσα από προγράμματα (π.χ. τα εκτελέσιμα αρχεία είναι δυαδικά). Ορισμένες φορές δεν είναι *φορητά* (δεν ανοίγουν σε όλα τα μηχανήματα).

- Τα **αρχεία κειμένου** (text files), στα οποία τα δεδομένα αποθηκεύονται ως μία ακολουθία από bytes χαρακτήρων. Ο αριθμός 12345678 εγγράφεται ως αλφαριθμητικό σε αρχείο κειμένου, απαιτώντας 9 bytes (1 για κάθε χαρακτήρα κι ένα για το χαρακτήρα τερματισμού '**0**').

  Τα αρχεία κειμένου είναι αναγνώσιμα από τους συντάκτες. Μάλιστα τα προγράμματα της C αποθηκεύονται ως αρχεία κειμένου (π.χ. αρχεία .**h**, .**c**). Τέλος, τα αρχεία κειμένου είναι φορητά σε κάθε υπολογιστή.

Ο τρόπος αποθήκευσης των δεδομένων δεν είναι η μοναδική διαφορά ανάμεσα στις δύο κατηγορίες αρχείων. Υπάρχουν διαφορές ανάμεσα στον τρόπο ερμηνείας του χαρακτήρα νέας γραμμής και του χαρακτήρα τέλους του αρχείου, οι οποίες θα

μελετηθούν παρακάτω. Οι δύο μορφές αρχείων χρησιμοποιούνται εξίσου αποτελεσματικά, απλώς έχουν διαφορετικό πεδίο εφαρμογών.

5.2 Άνοιγμα - κλείσιμο αρχείου

Για να επικοινωνήσει ένα πρόγραμμα με ένα αρχείο θα πρέπει το τελευταίο να δηλωθεί μέσα στο πρόγραμμα. Η δήλωση γίνεται με τη διαδικασία ανοίγματος του αρχείου, η οποία ακολουθεί τον εξής φορμαλισμό:

pF=fopen("filename","mode");

όπου ο δείκτης αρχείου **pF** έχει δηλωθεί προηγουμένως με την εντολή **FILE *pF;**

- Η συνάρτηση ***fopen()*** δεσμεύει τους απαραίτητους πόρους από το λειτουργικό σύστημα, δημιουργεί το κανάλι επικοινωνίας και επιστρέφει στο πρόγραμμα που την κάλεσε ένα δείκτη **pF**, που δείχνει σε δομή τύπου **FILE**. Σε περίπτωση σφάλματος, όταν είτε δεν υπάρχει ένα αρχείο προς ανάγνωση είτε δεν υπάρχει αποθηκευτικός χώρος για τη δημιουργία νέου αρχείου προς εγγραφή, επιστρέφεται το **NULL**. Όλες οι προσπελάσεις γίνονται μέσω του δείκτη. Ο δείκτης **pF** είναι το όνομα του αρχείου μέσα στο πρόγραμμα. Ένα από τα πεδία της δομής **FILE** είναι ο **δείκτης θέσης αρχείου** (file position indicator), ο οποίος δείχνει στο byte από όπου ο επόμενος χαρακτήρας πρόκειται να διαβασθεί ή όπου ο επόμενος χαρακτήρας πρόκειται να εγγραφεί.

- Η συμβολοσειρά **filename** είναι το φυσικό όνομα του αρχείου, με το οποίο το αρχείο αποθηκεύεται στη συσκευή αποθήκευσης. Εάν δοθεί μόνο ένα όνομα, το αρχείο θα αποθηκευθεί ή θα αναζητηθεί στον τρέχοντα κατάλογο. Υπάρχει η δυνατότητα να δοθεί ολόκληρο το μονοπάτι μέσα στη συσκευή αποθήκευσης:

"c:\\teiser\\prgrmmng\\myfile.txt"

Για καθορισμό του ονόματος του αρχείου από το χρήστη κατά τη διάρκεια εκτέλεσης του προγράμματος μπορεί να χρησιμοποιηθεί ο ακόλουθος κώδικας:

```
char *filename;                    // εναλλακτικά char filename[30];
printf( "Enter filename -> " );
scanf( "%s",file_name );           // Ανάγνωση του ονόματος του αρχείου
pF=fopen( file_name,"r" );
```

- Η συμβολοσειρά **mode** ελέγχει το είδος της πρόσβασης στο αρχείο (εγγραφή, ανάγνωση κ.λ.π.). Για παράδειγμα, εάν τεθεί

pF=fopen("myfile.txt","r");

τότε το αρχείο **myfile.txt** θα χρησιμοποιηθεί για ανάγνωση.

Παράμετροι προσδιορισμού του τρόπου πρόσβασης σε αρχεία κειμένου:

- **'r'**: **άνοιγμα αρχείου για ανάγνωση**. Ο δείκτης θέσης αρχείου βρίσκεται στην αρχή του κειμένου.
- **'w'**: **δημιουργία νέου αρχείου για εγγραφή**. Εάν το αρχείο υπάρχει ήδη, το μέγεθός του θα μηδενισθεί και τα περιεχόμενα θα διαγραφούν. Ο δείκτης θέσης αρχείου τίθεται στην αρχή του αρχείου.
- **'a'**: **άνοιγμα υπάρχοντος αρχείου κειμένου**, στο οποίο όμως μπορούμε να γράψουμε μόνο στο τέλος του αρχείου (προσάρτηση σε αρχείο).
- **'r+'**: **άνοιγμα υπάρχοντος αρχείου κειμένου για ανάγνωση και εγγραφή**. Ο δείκτης θέσης αρχείου τίθεται στην αρχή του αρχείου.
- **'w+'**: **δημιουργία νέου αρχείου για ανάγνωση και εγγραφή**. Εάν το αρχείο υπάρχει ήδη, το μέγεθός του θα μηδενισθεί και τα περιεχόμενα θα διαγραφούν.
- **'a+'**: **άνοιγμα υπάρχοντος αρχείου ή δημιουργία νέου σε append μορφή**. Μπορούμε να διαβάσουμε δεδομένα από οποιοδήποτε σημείο του αρχείου, αλλά μπορούμε να γράψουμε δεδομένα μόνο στη θέση του δείκτη **EOF**.

*Οι προσδιοριστές για τα δυαδικά αρχεία είναι ίδιοι, με τη διαφορά ότι έχουν ένα **b** που τους ακολουθεί. Έτσι, για να ανοίξουμε ένα δυαδικό αρχείο προς ανάγνωση, θα πρέπει να χρησιμοποιήσουμε τον προσδιοριστή **'rb'**.*

Το κλείσιμο ενός αρχείου γίνεται μετά το τέλος της χρήσης της συνάρτησης ***fclose()***:

fclose(pF);

Όταν το αρχείο κλείσει σωστά επιστρέφεται το **0** ενώ σε περίπτωση σφάλματος επιστρέφεται το **EOF**.

Παρατηρήσεις:

1. Ο δείκτης **FILE** χειρίζεται κατά τρόπο αποκλειστικό το αρχείο και σε δείκτες τέτοιου τύπου δεν επιτρέπεται 'αριθμητική δεικτών'!!!

Π.χ.

```
fclose( pF );       // σωστό
fclose( pF+1 );     // ΛΑΘΟΣ
```

2. Η συνάρτηση ***fopen()*** δεσμεύει μνήμη. Εάν αμεληθεί να απελευθερωθεί με χρήση της ***fclose()*** θα υπάρξει μεγάλη διαρροή μνήμης. Για το λόγο αυτό θα πρέπει πάντοτε να γίνεται έλεγχος κατά πόσον μία ***fopen()*** συνοδεύεται από την αντίστοιχη ***fclose()***.

3. Η συνάρτηση ***fcloseall()*** κλείνει όλα τα αρχεία που είναι ανοικτά τη στιγμή της εφαρμογής της. Προτείνεται να τοποθετείται στο τέλος των προγραμμάτων έτσι ώστε να τερματίζονται όλα τα αρχεία που παραμένουν ανοικτά εκ παραδρομής.

5.3 Ανάγνωση - εγγραφή χαρακτήρων σε αρχεία

5.3.1 Η συνάρτηση εγγραφής χαρακτήρων putc

Η συνάρτηση ***putc()*** χρησιμοποιείται για την εγγραφή χαρακτήρων σε ένα κανάλι που έχει ανοίξει προηγουμένως με την ***fopen()***. Το πρωτότυπο της συνάρτησης είναι το εξής:

int putc(int ch, FILE *pF);

όπου **pF** είναι ο δείκτης αρχείου που επιστρέφεται από την ***fopen()*** και **ch** είναι ο προς εγγραφή χαρακτήρας. Για ιστορικούς λόγους ο **ch** ονομάζεται **int** αλλά χρησιμοποιεί μόνο ένα byte, το byte χαμηλής τάξης. Η ***putc()*** ορίζεται στο ***stdio.h***.

Εάν η λειτουργία της συνάρτησης επιτύχει, επιστρέφεται ο χαρακτήρας που ενεγράφη. Εάν αποτύχει, θα επιστρέψει το **EOF**.

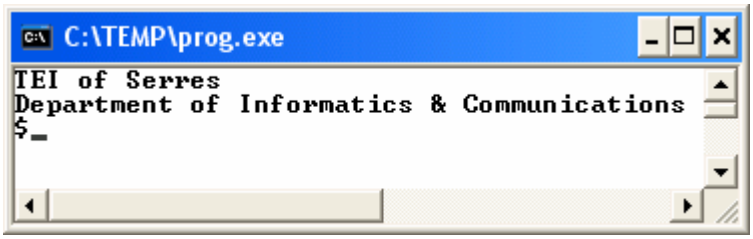
Παράδειγμα 5.1

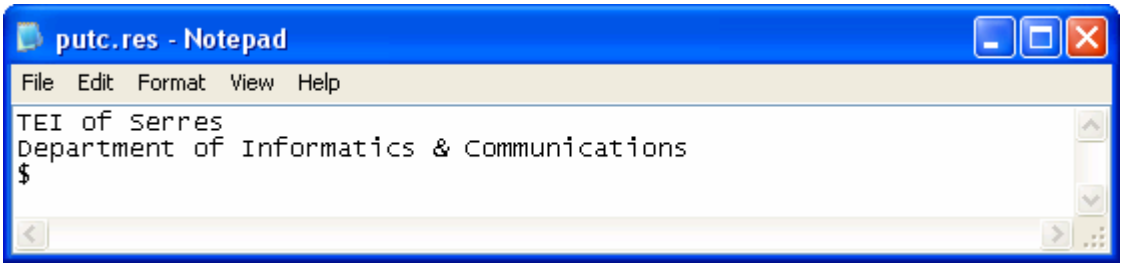
Να καταστρωθεί πρόγραμμα, το οποίο διαβάζει χαρακτήρες από το

πληκτρολόγιο και τους γράφει σε αρχείο, έως ότου πληκτρολογήσουμε το σύμβολο του δολαρίου (**$**).

```
#include<stdio.h>
void main()
{
    FILE *pF; char ch;
    pF=fopen( "putc.res","w" );
    if (pF==NULL) printf( "\t\tFILE ERROR: Exit program\n" );
    else
    {
        do
        {
            ch=getchar();
            putc(ch,pF);
        }
        while (ch!='$');
    }
    fclose( pF );
}
```

Ακολουθούν η έξοδος στην οθόνη (οι χαρακτήρες που πληκτρολογήθηκαν) και το προκύπτον αρχείο **putc.res**.





5.3.2 Η συνάρτηση ανάγνωσης χαρακτήρων getc

Η συνάρτηση ***getc()*** είναι συμπληρωματική της ***putc()*** και χρησιμοποιείται για την ανάγνωση χαρακτήρων από ένα κανάλι που έχει ανοίξει προηγουμένως με την ***fopen()***. Το πρωτότυπο της συνάρτησης είναι το εξής:

int getc(FILE *pF);

όπου **pF** ο δείκτης αρχείου που επιστρέφεται από την ***fopen()***. Για ιστορικούς λόγους η ***getc()*** επιστρέφει έναν ακέραιο αλλά τα bytes υψηλής τάξης είναι μηδέν, άρα μόνο το byte χαμηλής τάξης περιέχει πληροφορία. Η ***getc()*** ορίζεται στο ***stdio.h***.

Η συνάρτηση ***getc()*** επιστρέφει ένα **EOF** όταν ο υπολογιστής φθάσει στο τέλος του αρχείου. Έτσι, για να διαβάσουμε ένα αρχείο κειμένου έως το σημάδι τέλους αρχείου, μπορούμε να χρησιμοποιήσουμε τον ακόλουθο κώδικα:

```
ch = getc(pF);
while (ch!=EOF)  ch=getc(pF);
```

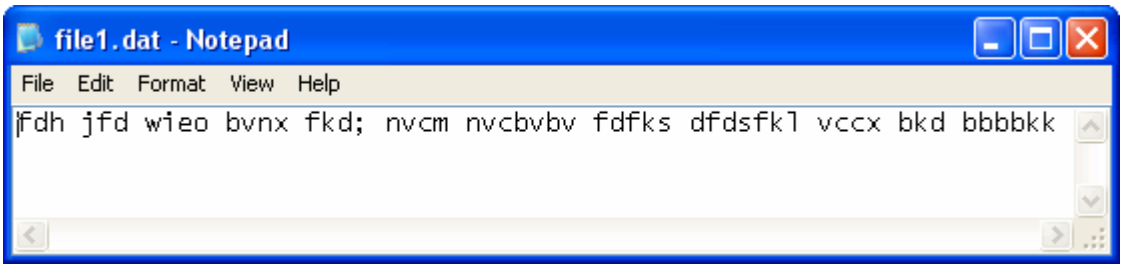
Παράδειγμα 5.2

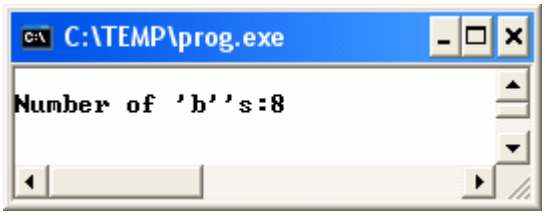
Να γραφεί πρόγραμμα, το οποίο θα βρίσκει πόσες φορές υπάρχει ο χαρακτήρας **'b'** στο αρχείο **file1.dat**.

```
#include <stdio.h>
void  main()
{
    FILE *f1;
    char get_char;
    int sum_b=0;
    f1=fopen( "file1.dat","r" );
    while (get_char!=EOF)
    {
        get_char=getc(f1);
        if (get_char=='b') sum_b++;
    }
```

```
        fclose( f1 );
        printf( "\nNumber of 'b''s:%d",sum_b);
    }
```

Ακολουθούν ένα τυχαίο αρχείο **file1.dat**, από το οποίο γίνεται η ανάγνωση των δεδομένων, και τα αποτελέσματα του μετρητή.





Παράδειγμα 5.3

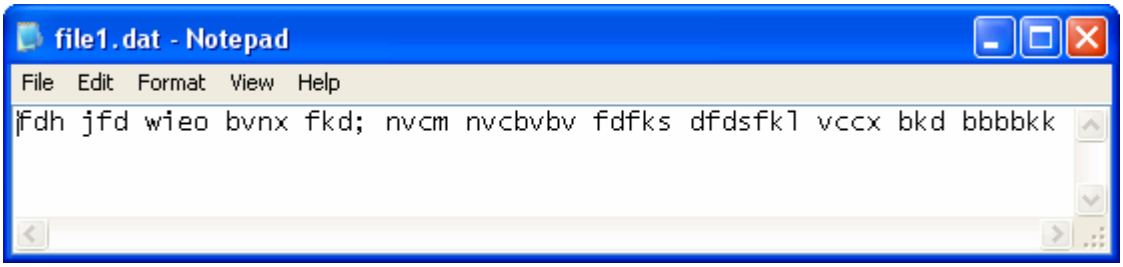
Να καταστρωθεί πρόγραμμα, το οποίο να δίνει τον αριθμό των λέξεων που περιέχονται σε ένα αρχείο ASCII. Το πρόγραμμα θα πρέπει να χειρίζεται τους *λευκούς χαρακτήρες* (κενά, νέες γραμμές, στηλοθέτες) ως πραγματικούς χαρακτήρες. Δηλαδή, εάν υπάρχει μία συμβολοσειρά από κενά ή χαρακτήρες επιστροφής, το πρόγραμμα τους διαβάζει και αναμένει για τον πρώτο πραγματικό (μη λευκό) χαρακτήρα. Όλο αυτό το μετρά ως λέξη. Κατόπιν διαβάζει τους πραγματικούς χαρακτήρες έως την εμφάνιση του επόμενου λευκού χαρακτήρα.

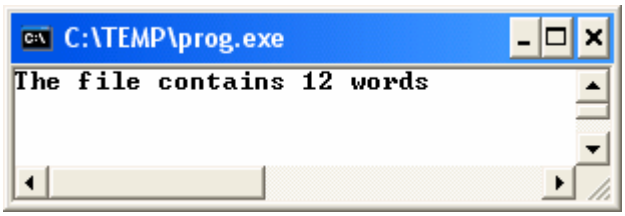
Μία μεταβλητή (σημαία) θα πρέπει να ελέγχει κατά πόσον το πρόγραμμα βρίσκεται στο μέσο μίας λέξης ή στο μέσο κάποιου κενού.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>      // Για την exit()
```

```
void main()
{
    FILE *fptr;
    char ch,string[81];
    int white=1;  // Σημαία λευκού χαρακτήρα
    int count=0;  // Μετρητής λέξεων
    if ((fptr=fopen( "file1.txt","r" ))==NULL)
    {
        printf( "ERROR: can't open file" );
        exit(1);
    }
    while ((ch=getc(fptr))!=EOF)   // Ανάγνωση χαρακτήρων έως EOF
    {
        switch(ch)
        {
            case ' ':  // Έλεγχος για λευκούς χαρακτήρες
            case '\t':
            case '\n':
                white++;
            break;
            default:  // Μη λευκοί χαρακτήρες, μέτρηση λέξεων
                if (white)
                {
                    white=0;
                    count++;
                }
            break;
        }
    }
    fclose( fptr );
    printf( "The file contains %d words\n",count );
}
```

Ακολουθούν ένα τυχαίο αρχείο **file1.dat**, από το οποίο γίνεται η ανάγνωση των δεδομένων, και τα αποτελέσματα του μετρητή.





5.4 Μορφοποιούμενες συναρτήσεις εισόδου - εξόδου σε αρχεία

5.4.1 Η συνάρτηση fprintf

Η συνάρτηση ***fprintf()*** χρησιμοποιείται για εγγραφή σε ένα αρχείο. Έχει τους ίδιους μορφολογικούς κανόνες με την ***printf()*** με τη διαφορά ότι το πρώτο όρισμα είναι ο δείκτης αρχείου στο οποίο θα γίνει η εγγραφή:

fprintf(pF, *ορίσματα*);

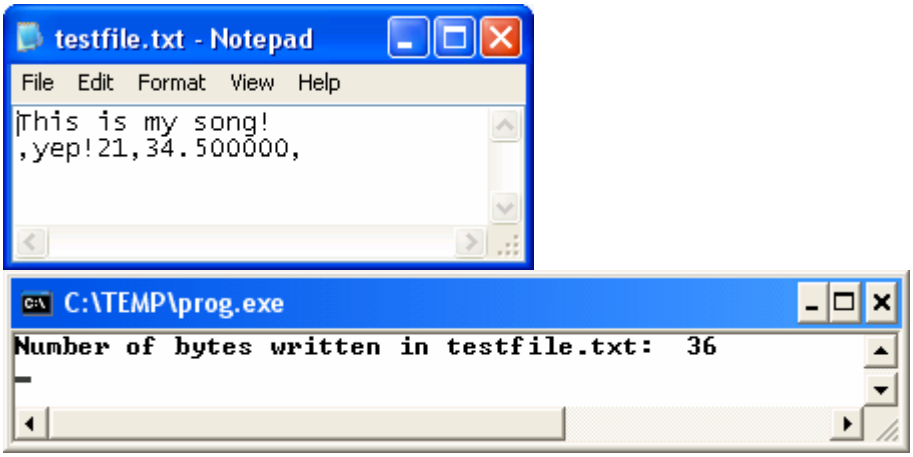
Η ***fprintf()*** επιστρέφει έναν ακέραιο, ο οποίος είναι ο αριθμός των bytes που ενεγράφησαν. Σε περίπτωση σφάλματος επιστρέφει το **EOF**.

<u>Παρατήρηση:</u> Η **printf(*ορίσματα*);** ισοδυναμεί με την **fprintf(stdout, *ορίσματα*);**, δηλαδή με την ***fprintf()*** που έχει κανάλι εξόδου την οθόνη αντί για το αρχείο.

Παράδειγμα 5.4

Να περιγραφεί η λειτουργία του ακόλουθου προγράμματος:

```
#include <stdio.h>
void main()
{
    int cnt;
    FILE *pF;
    char *filename="testfile.txt", msg[40]="This is my song!\n";
    pF=fopen( filename,"w" );
    cnt=fprintf( pF,"%s,yep!%d,%f,\n",msg,21,34.5 );
    printf( "Number of bytes written in %s:  %d\n",filename,cnt );
    fclose( pF );
}
```



Αρχικά ορίζεται η ακέραια μεταβλητή **cnt** και ο δείκτης αρχείου **pF**. Ακολουθεί ο ορισμός ενός δείκτη χαρακτήρων **filename**, ο οποίος δείχνει στο αλφαριθμητικό **"testfile.txt"**, και το αλφαριθμητικό **msg** που έχει αρχικοποιηθεί.

Ο δείκτης **pF** ορίζεται να δείχνει σε αρχείο με φυσικό όνομα το αλφαριθμητικό στο οποίο δείχνει ο **filename**, δηλαδή το **"testfile.txt"**. Η συνάρτηση ***fprintf()*** θα τυπώσει στο αρχείο **"testfile.txt"**:

- Τη συμβολοσειρά **msg** (17 bytes μαζί με το byte του μηδενικού χαρακτήρα) στο τέλος της οποίας δηλώνεται αλλαγή γραμμής
- Τους χαρακτήρες **",yep!"** (5 bytes)Τ
- ον ακέραιο **21** (4 bytes)

- Το **κόμμα** (1 byte)
- Τον αριθμό κινητής υποδιαστολής **34.5** (8 bytes)
- Το **κόμμα** (1 byte)

Συνολικά θα εγγραφούν στο αρχείο 36 bytes, όπως φαίνεται και στα αποτελέσματα.

Στο τέλος του προγράμματος το αρχείο **"testfile.txt"** κλείνει με χρήση της ***fclose()***.

5.4.2 Η συνάρτηση fscanf

Η συνάρτηση ***fscanf()*** χρησιμοποιείται για ανάγνωση από ένα αρχείο. Έχει τους ίδιους μορφολογικούς κανόνες με την ***scanf()*** με τη διαφορά ότι το πρώτο όρισμα είναι ο δείκτης αρχείου από το οποίο θα γίνει η ανάγνωση:

fscanf(pF, *ορίσματα*);

Η ***fscanf()*** επιστρέφει έναν ακέραιο, ο οποίος είναι ο αριθμός των στοιχείων που ανεγνώσθησαν. Σε περίπτωση σφάλματος θα επιστραφεί το **EOF** εάν επιχειρηθεί ανάγνωση στο τέλος του αρχείου, ή το **0** εάν δεν υπάρχουν δεδομένα προς ανάγνωση.

<u>Παρατήρηση:</u> Η **scanf(*ορίσματα*);** ισοδυναμεί με την **fscanf(stdin, *ορίσματα*);**, δηλαδή με την ***fscanf()*** που έχει κανάλι εισόδου το πληκτρολόγιο αντί για το αρχείο.

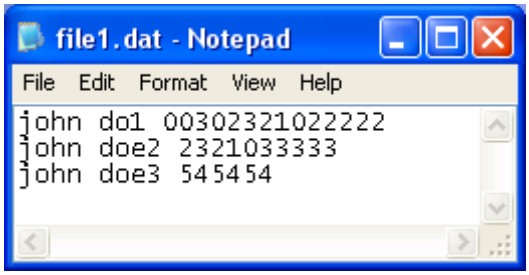
Παράδειγμα 5.5

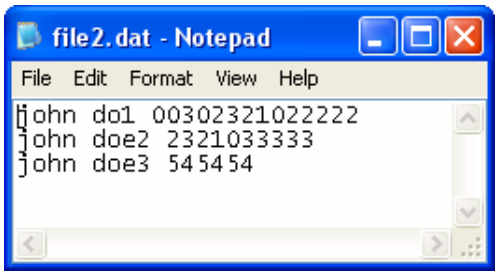
Να γραφεί πρόγραμμα, το οποίο θα δημιουργεί πίνακα 3 στοιχείων, με στοιχεία δομές. Κάθε δομή θα έχει ως μέλη το όνομα, το επώνυμο και το τηλέφωνο ενός ανθρώπου. Το πρόγραμμα θα διαβάζει τα περιεχόμενα του πίνακα από το αρχείο **file1.dat**, θα τα αποδίδει στον πίνακα και θα τα γράφει σε ένα άλλο αρχείο, το **file2.dat**.

```
#include <stdio.h>
#include <assert.h>
#define N 3
```

```
// Ορισμός δομής:
typedef struct
{
    char name[30];
    char surname[30];
    char phone_num[15];
    // Εάν ο phone_num ήταν ακέραιος, θα ήταν δεκαψήφιος ακέραιος.
}   struct_type;
void main()
{
    struct_type id[N];
    FILE *f1;
    int i;
    f1=fopen( "file1.dat","r" );  assert( f1!=NULL );
    for (i=0; i<N; i++)
      fscanf( f1,"%s %s %d\n",id[i].name,id[i].surname,id[i].phone_num
    );
    fclose( f1 );
    f1=fopen( "file2.dat","w" );  assert( f1!=NULL );
    for (i=0; i<N; i++)
     fprintf( f1,"%s %s %d\n",id[i].name,id[i].surname,id[i].phone_num
    );
    fclose( f1 );
}
```

Ακολουθούν ένα τυχαίο αρχείο **file1.dat**, από το οποίο γίνεται η ανάγνωση του πίνακα δομών, και το προκύπτον αρχείο **file2.dat**.





5.5 Ανάγνωση - εγγραφή σε δυαδικά αρχεία

Αν και η χρήση των ***fprintf()***, ***fscanf()*** είναι συχνά ο πιο εύκολος τρόπος για να γράφουμε ή να διαβάζουμε μία συλλογή δεδομένων σε ένα αρχείο δίσκου, δεν είναι πάντοτε και ο πιο αποτελεσματικός. Επειδή γράφουμε φορμαρισμένα δεδομένα ASCII, όπως δηλαδή αυτά θα εμφανίζονταν στην οθόνη, και όχι δυαδικά, κάνουμε περισσότερα πράγματα σε κάθε κλήση και καταλαμβάνουμε περισσότερο χώρο. Έτσι, εάν ενδιαφέρει η ταχύτητα ή το μέγεθος του αρχείου, θα πρέπει πιθανώς να χρησιμοποιήσουμε δυαδικά αρχεία.

Πέραν των ζητημάτων ταχύτητας και αποθηκευτικού χώρου, η χρήση μορφοποιούμενων συναρτήσεων ανάγνωσης–εγγραφής παρουσιάζει ένα άλλο πρόβλημα: δεν υπάρχει <u>άμεσος</u> τρόπος ανάγνωσης και εγγραφής πολύπλοκων τύπων δεδομένων, όπως πίνακες και δομές, καθώς με τις συναρτήσεις αυτές κάθε φορά γράφεται/διαβάζεται ένα στοιχείο του πίνακα ή της δομής. Για ανάγνωση και εγγραφή τέτοιων τύπων δεδομένων με μία μόνο πρόταση χρησιμοποιείται το ζεύγος των συναρτήσεων ***fread()/fwrite()***.

5.5.1 Η συνάρτηση fread

Η συνάρτηση ***fread()*** χρησιμοποιείται για την ανάγνωση μπλοκ δεδομένων από ένα αρχείο. Ορίζεται στο ***stdio.h*** και έχει το ακόλουθο πρωτότυπο:

int fread(void *buffer, int length, int num_items, FILE *fp);

όπου

- **buffer** είναι ένας δείκτης σε μία περιοχή της μνήμης η οποία θα δεχθεί τα δεδομένα που διαβάζονται από το αρχείο.

- **length** είναι το μέγεθος του τύπου των δεδομένων που θα αναγνωσθούν. Για τον προσδιορισμό τους συνήθως χρησιμοποιείται η ***sizeof***.
- **num_items** είναι ο αριθμός των στοιχείων (μήκους **length** bytes το καθένα) που θα αναγνωσθούν.
- **fp** είναι ο δείκτης του προς ανάγνωση αρχείου.
- Η ***fread()*** επιστρέφει έναν ακέραιο, ο οποίος είναι ο αριθμός των στοιχείων (όχι των
bytes) που ανεγνώσθησαν επιτυχώς.

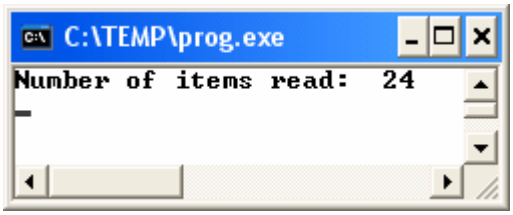
Θα πρέπει να σημειωθεί ότι για να λειτουργήσει επιτυχώς η ***fread()*** θα πρέπει η περιοχή προσωρινής αποθήκευσης που καθορίζεται από το **buffer** αφενός μεν να αποθηκεύει ίδιου τύπου με τα προς ανάγνωση δεδομένα, αφετέρου δε να έχει επαρκή μνήμη.

Παράδειγμα 5.6

Να περιγραφεί η λειτουργία του ακόλουθου προγράμματος:

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
void main()
{
    FILE *pF;
    int buf[40];
    int i,cnt, n=24;
    pF=fopen( "file.dat","rb" );    assert(pF!=NULL);
    cnt=fread(buf, sizeof(int), n, pF);
    if (cnt!=n)
    {
        printf( "ERROR");
        exit(-1);
```

```
        }
        printf( "Number of items read:  %d\n",cnt );
        fclose( pF );
    }
```



Αρχικά ορίζεται ο **buffer** ως πίνακας ακεραίων με μέγεθος **40** και ο αριθμός των προς ανάγνωση δεδομένων, **n**, με τιμή **24**. Ακολούθως ανοίγει για ανάγνωση το δυαδικό αρχείο **"file.dat"**, το οποίο περιλαμβάνει 32 ακεραίους. Με χρήση της ***fread()*** διαβάζονται τα δεδομένα, η **cnt** γίνεται ίση με την **n**, όπως φαίνεται στην έξοδο στην οθόνη. Σε περίπτωση σφάλματος έχει ληφθεί πρόνοια για έξοδο από το πρόγραμμα. Στο τέλος το προγράμματος κλείνει το αρχείο **"file.dat"**.

5.5.2 Η συνάρτηση fwrite

Η συνάρτηση ***fwrite()*** χρησιμοποιείται για την εκτύπωση μπλοκ δεδομένων σε ένα αρχείο. Ορίζεται στο ***stdio.h*** και έχει το ακόλουθο πρωτότυπο:

int fwrite(void *buffer, int length, int num_items, FILE *fp);

όπου τα ορίσματα είναι ακριβώς τα ίδια με εκείνα της ***fread()*** (εξυπακούεται ότι στο buffer θα αποθηκεύονται πλέον τα προς εγγραφή δεδομένα και η έξοδος θα επιστρέφει τον αριθμό των στοιχείων που ενεγράφησαν επιτυχώς).

Παρατηρήσεις:

1. Στην περίπτωση εγγραφής αλφαριθμητικών, ένα συχνό σφάλμα που γίνεται είναι να χρησιμοποιείται η ***strlen()*** για τον υπολογισμό των χαρακτήρων του αλφαριθμητικού, παραλείποντας όμως τον τερματιστή του (το μηδενικό χαρακτήρα **'\0'**).

Για παράδειγμα, στον ακόλουθο κώδικα η εγγραφή του αλφαριθμητικού είναι ατελής:

```
int cnt;
FILE *pF;
char msg[40]="This is my song!";
pF=fopen( "music.mdi","wb" );
cnt=fwrite(msg, sizeof(char), strlen(msg),pF);
```

Το σφάλμα διορθώνεται με την προσθήκη μίας μονάδας στη ***strlen()***, ώστε να περιληφθεί ο χαρακτήρας τερματισμού του αλφαριθμητικού:

```
cnt= fwrite(msg, sizeof(char),1+strlen(msg),pF);
```

2. Οι εντολές ***fread()/fwrite()*** μπορούν να χρησιμοποιηθούν με τον ίδιο τρόπο και σε αρχεία κειμένου.

Παράδειγμα 5.7

Να περιγραφεί η λειτουργία του ακόλουθου προγράμματος:

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
void main()
{
    FILE *pF;
    int buf[40];
    int i,cnt, n=32;
    pF=fopen( "file.dat","wb" );   assert(pF!=NULL);
    for (i=1; i<=n; i++) buf[i]=2*i;
    cnt = fwrite(buf, sizeof(int), n, pF);
    if(cnt!=n)
    {
        printf( "ERROR");
        exit(-1);
```

```
        }
        printf( "Number of items written:  %d\n",cnt );
        fclose( pF );
    }
```

Αρχικά ορίζεται ο **buffer** ως πίνακας ακεραίων με μέγεθος **40** και ο αριθμός των προς εγγραφή δεδομένων, **n**, με τιμή **32**. Ακολούθως ανοίγει για εγγραφή το δυαδικό αρχείο **"file.dat"**. Με χρήση της ***fwrite()*** εκτυπώνονται τα δεδομένα και η **cnt** γίνεται ίση με την **n**, όπως φαίνεται στην έξοδο στην οθόνη. Σε περίπτωση σφάλματος έχει ληφθεί πρόνοια για έξοδο από το πρόγραμμα. Στο τέλος το προγράμματος κλείνει το αρχείο **"file.dat"**.

Παράδειγμα 5.8

Να γραφεί κώδικας για το παράδειγμα 5.5 με χρήση δυαδικών αρχείων και των συναρτήσεων ***fread()/fwrite()***.

```
#include <stdio.h>
#include <assert.h>
#define N 3
// Ορισμός δομής:
typedef struct
{
    char name[30];
    char surname[30];
    char phone_num[15];
```

```
}    struct_type;
void main()
{
     struct_type id[N];
     FILE *f1;
     int i;
     f1=fopen( "file1.dat","rb" );  assert( f1!=NULL );
     for (i=0; i<N; i++)  fread(&id[i],sizeof(id[i]),1,f1);
     fclose( f1 );
     f1=fopen( "file2.dat","wb" );  assert( f1!=NULL );
     for (i=0; i<N; i++)  fwrite(&id[i],sizeof(id[i]),1,f1);
     fclose( f1 );
}
```

5.5.3 Η συνάρτηση feof

Όταν ανοίγει ένα δυαδικό αρχείο, είναι πιθανόν ο υπολογιστής να διαβάσει μία ακέραια τιμή ίση με το **EOF**. Σε μία τέτοια περίπτωση θα δηλωθεί μία συνθήκη τέλους αρχείου, ακόμη κι αν ο υπολογιστής δεν έχει φθάσει στο φυσικό τέλος του αρχείου. Για να λύσει αυτό το πρόβλημα η C περιλαμβάνει τη συνάρτηση ***feof()***, η οποία καθορίζει πού βρίσκεται το σημείο τέλους αρχείου όταν διαβάζονται δυαδικά δεδομένα. Η συνάρτηση ***feof()*** λαμβάνει ως όρισμα ένα δείκτη αρχείου και επιστρέφει **1** εάν ο υπολογιστής έχει φθάσει στο τέλος του αρχείου ή **0** εάν ο υπολογιστής δεν έχει φθάσει στο τέλος του αρχείου. Η ***feof()*** ορίζεται στο ***stdio.h***.

Έτσι, για να διαβάσουμε ένα δυαδικό αρχείο έως το σημάδι τέλους αρχείου, μπορούμε να χρησιμοποιήσουμε τον ακόλουθο κώδικα:

```
ch=getc(fp);
while (!feof(fp))   ch=getc(fp);
```

5.6 Τυχαία προσπέλαση δυαδικού αρχείου

Έως τώρα τα αρχεία προσπελαύνονταν σειριακά, δηλαδή για να βρεθεί ένα δεδομένο σε ένα αρχείο θα έπρεπε να προσπελασθούν πρώτα όλα τα προηγούμενά του. Επιπρόσθετα, για να ενημερωθεί μία εγγραφή του αρχείου (π.χ. ένα όνομα), θα έπρεπε να διαβασθεί όλο το αρχείο, να γίνει η αλλαγή και κατόπιν να ξαναγραφεί.

Γίνεται φανερό ότι η σειριακή προσπέλαση δεν ενδείκνυται σε μεγάλα αρχεία ή σε αρχεία που προσπελαύνονται και τροποποιούνται συχνά. Για αυτές τις περιπτώσεις η C παρέχει τη δυνατότητα **τυχαίας προσπέλασης** (random access), με την οποία υπάρχει πρόσβαση σε οιοδήποτε σημείο ενός αρχείου. Η τυχαία προσπέλαση στηρίζεται στο γεγονός ότι κάθε ανοικτό αρχείο έχει έναν δείκτη θέσης αρχείου, ο οποίος καθορίζει σε ποιο σημείο του αρχείου θα γίνει ανάγνωση ή εγγραφή. Η θέση αυτή δίνεται ως αριθμός bytes από την αρχή του αρχείου και είναι μία μεταβλητή της δομής **FILE**. Όταν το αρχείο ανοίγει για ανάγνωση η θέση αυτή είναι **0**. Όταν ανοίγει για προσάρτηση είναι το τέλος του αρχείου. Καθορίζοντας το δείκτη θέσης αρχείου μπορούμε να έχουμε προσπέλαση σε οιοδήποτε σημείο του αρχείου. Αυτή είναι η έννοια της τυχαίας προσπέλασης.

5.6.1 Η συνάρτηση fseek

Η συνάρτηση ***fseek()*** αποτελεί το εργαλείο για την εκτέλεση λειτουργιών τυχαίας ανάγνωσης και εγγραφής. Ορίζεται στο ***stdio.h*** και έχει το ακόλουθο πρωτότυπο:

int fseek(FILE *fptr, long offset, int origin)

όπου

- **fptr** είναι ένας δείκτης αρχείου που επιστρέφεται από την ***fopen()***.
- **offset** είναι ο αριθμός των bytes που δηλώνουν την απόσταση της νέας θέσης από το **origin**.
- **origin** είναι το σημείο αφετηρίας και μπορεί να έχει μία από τις τρεις ακόλουθες τιμές:

Αφετηρία	Όνομα
αρχή του αρχείου	**SEEK_SET (0)**
τρέχουσα θέση	**SEEK_CUR (1)**

τέλος αρχείου	**SEEK_END (2)**

Έτσι, για να βρεθεί π.χ. το **offset** από την τρέχουσα θέση του, το **origin** θα πρέπει να λάβει την τιμή **SEEK_CUR**. Σε περίπτωση επιτυχίας η ***fseek()*** επιστρέφει **0**, ενώ εάν αποτύχει επιστρέφει **μη μηδενική** τιμή.

Θα πρέπει να σημειωθεί ότι ο δείκτης θέσης αρχείου επανατοποθετείται στην αρχή με τη συνάρτηση

rewind(fptr);

Παράδειγμα 5.9

Για τη διαχείριση των στοιχείων των φοιτητών ορίζεται ο πίνακας **student_list[size]** με στοιχεία τύπου δομής **Student**:

```
typedef struct
{
    int AM;
    int year;
    char firstname[15];
    char lastname[30];
}   Student;
```

Για τη διαχείριση μεμονωμένων φοιτητών ορίζεται η μεταβλητή **svar**, επίσης τύπου δομής **Student**.

Ζητείται να γραφούν συναρτήσεις που να επιτελούν τα ακόλουθα:

- ***save_data()***: Αποθήκευση των δεδομένων του πίνακα **student_list** στο αρχείο **students.dat**.
- ***read_data()***: Ανάγνωση των δεδομένων από το αρχείο και αποθήκευσή τους στον πίνακα **student_list**.
- ***read_student()***: Προσπέλαση ενός συγκεκριμένου φοιτητή στο αρχείο και αποθήκευση των στοιχείων του σε μία μεταβλητή τύπου **Student**.
- ***save_student()***: Αποθήκευση των στοιχείων ενός συγκεκριμένου φοιτητή στο αρχείο.

Υποθέτουμε ότι το δυαδικό αρχείο έχει ανοίξει κανονικά στη ***main()***, υπάρχει

ένας έγκυρος δείκτης **fptr** σε αυτό και το **size** έχει καθορισθεί με **#define**.

- ***save_data()***

  Η συνάρτηση καλείται στη ***main()*** ως εξής:

```
save_data(fptr, student_list);
```

και έχει το ακόλουθο σώμα:

```
void save_data( FILE *fp, Student list[ ] ) // ή ισοδύναμα Student *list
{
    int i;
    rewind(fp);
    for ( i=0; i<size; i++ )  fwrite(&list[i],sizeof(Student),1,fp);
}
```

Αντί του βρόχου ***for*** θα μπορούσε να γραφεί:

```
fwrite(list,sizeof(Student),size,fp);
```

- ***read_data()***

  Η συνάρτηση καλείται στη ***main()*** ως εξής:

```
read_data(fptr, student_list);
```

και έχει το ακόλουθο σώμα:

```
void save_data( FILE *fp, Student *list )
{
    int i;
    rewind(fp);
    fread(list,sizeof(Student),size,fp);
}
```

- ***read_student()***

  Η συνάρτηση καλείται στη ***main()*** ως εξής:

```
read_student(fptr, &svar, i);
```

όπου **svar** είναι μία μεταβλητή τύπου **Student** (στη θέση της θα μπορούσε να είναι η **&student_list[i]**), **i** είναι η θέση του στον πίνακα **student_list**. Το σώμα της είναι το ακόλουθο:

```
void read_student( FILE *fp, Student *s, int k )
```

```
{
    int i;
    fseek(fp,k*sizeof(Student),SEEK_SET);
    fread(s,sizeof(Student),1,fp);
}
```

- ***save_student()***

Η συνάρτηση καλείται στη ***main()*** ως εξής:

save_student(fptr, &student_list[i], i);

όπου **i** είναι ο αύξων αριθμός του στον πίνακα **student_list**. Το σώμα της είναι το ακόλουθο:

```
void save_student( FILE *fp, Student *s, int k )
{
    int i;
    fseek(fp,k*sizeof(Student),SEEK_SET);
    fwrite(s,sizeof(Student),1,fp);
}
```

Η συνάρτηση ***save_student()*** χρησιμοποιείται όταν έχουμε κάνει αλλαγές στα στοιχεία ενός φοιτητή και θέλουμε να ενημερώσουμε την εγγραφή του στο αρχείο.

Η συνάρτηση ***read_student()*** χρησιμοποιείται για να φέρουμε τα στοιχεία του i-στού φοιτητή από το αρχείο και πιθανώς να τα αλλάξουμε αργότερα.

5.7 Ανάγνωση - εγγραφή χαρακτήρων με χρήση των fread/fwrite

Το ζεύγος ***fread()/fwrite()*** μπορεί να επιτελέσει τη λειτουργία των ***getc()/putc()***, όχι μόνο για ένα αλλά για οιονδήποτε αριθμό χαρακτήρων. Η λειτουργία θα περιγραφεί με τη βοήθεια του ακόλουθου κώδικα:

```
FILE *pFin,*pFout;
```

```
char buf[100];
int cnt;
pFin=fopen("src.txt","r");
pFout=fopen("dest.txt","w");
if ((pFin==NULL) || (pFout==NULL)  exit(-1);    // Σφάλμα
cnt=fread(buf,sizeof(char),100,pFin);
while(cnt==100)
{
   fwrite(buf,sizeof(char),100,pFout);
   cnt=fread (buf,sizeof(char),100,pFin );
}
if (cnt!=0) fwrite(buf,sizeof(char),cnt,pFout);
fclose(pFout);
fclose(pFin);
```

- Ορίζονται οι δείκτες αρχείου **pFin** και **pFout** και ο buffer χαρακτήρων 100 θέσεων. Ο **pFin** χρησιμοποιείται για να ανοίξει το αρχείο ανάγνωσης **"src.txt"** και ο **pFout** για να ανοίξει το αρχείο εγγραφής **"dest.txt"**. Εάν υπάρξει σφάλμα στο άνοιγμα ενός από τα δύο αρχεία, το πρόγραμμα τερματίζεται (**exit(-1)**).
- Με την πρόταση

cnt=fread(buf,sizeof(char),100,pFin);

ζητείται να αναγνωσθούν 100 χαρακτήρες από το **"src.txt"** με χρήση του buffer. Η ***fread()***επιστρέφει στη **cnt** τον αριθμό των χαρακτήρων που ανεγνώσθησαν.

- Η συνθήκη ***while*** ελέγχει κατά πόσον γέμισε ο buffer. Εάν γέμισε γράφουμε <u>ολόκληρο το buffer</u> στο αρχείο εγγραφής **"dest.txt"** και ακολούθως επαναλαμβάνουμε την ανάγνωση.
- Η πρόταση

if (cnt!=0) fwrite(buf,sizeof(char),cnt,pFout);

γράφει στο **"dest.txt"** τα περιεχόμενα του buffer που μπορεί να παρέμειναν.

- Το πρόγραμμα ολοκληρώνεται με κλείσιμο των αρχείων εγγραφής και ανάγνωσης.

5.8 Ανάγνωση - εγγραφή γραμμή ανά γραμμή

Η C δίνει τη δυνατότητα ανάγνωσης και εγγραφής γραμμή ανά γραμμή με το ζεύγος συναρτήσεων ***fgets()/fputs()***. Οι συναρτήσεις ορίζονται στο ***stdio.h*** και έχουν τα ακόλουθα πρωτότυπα:

char *fgets(char *str, int length, FILE *fp);

char *fputs(char *str, FILE *fp);

Η συνάρτηση ***fputs()*** λειτουργεί όπως ακριβώς η ***puts()***, με τη διαφορά ότι η ***fputs()*** γράφει στο κανάλι που καθορίζεται. Η συνάρτηση ***fgets()*** διαβάζει ένα αλφαριθμητικό από το καθορισμένο κανάλι έως ότου διαβάσει είτε ένα χαρακτήρα νέας γραμμής είτε αριθμό χαρακτήρων ίσο με **length-1**. Εάν η ***fgets()*** διαβάσει ένα χαρακτήρα νέας γραμμής, ο τελευταίος θα αποτελέσει τμήμα του αλφαριθμητικού (σε αντίθεση με τη ***gets()***). Ωστόσο, μόλις τερματίσει η ***fgets()***, το αλφαριθμητικό που θα προκύψει θα έχει στο τέλος του ένα μηδενικό (null).

Παράδειγμα 5.10

Στον ακόλουθο κώδικα

```
char buf[100];
FILE *pFin,*pFout;
......
while(fgets(buf,100,pFin)!=NULL) fputs(buf,pFout);
```

η πρόταση **fgets(buf,100,pFin)**:

- θα διαβάσει ένα αλφαριθμητικό από το αρχείο που καθορίζει ο δείκτης **pFin** και θα το αποδώσει στο **buf**
- θα σταματήσει μετά τη **νέα-γραμμή '\n'** ή τον 99° χαρακτήρα
- θα τοποθετήσει ακολούθως στο **buf** το null **'\0'**
- θα επιστρέψει το δείκτη του **buf** σε περίπτωση επιτυχίας ή NULL εάν ο **pFin** είναι άδειος

Η πρόταση **fputs(buf,pFout)** θα εγγράψει στο αρχείο που καθορίζει ο δείκτης **pFout** το περιεχόμενο του **buf**.

Παράδειγμα 5.11

Να γραφεί πρόγραμμα, το οποίο θα λαμβάνει ακέραιους από αρχείο **file1.dat** και θα τους αποδίδει σε πίνακα τεσσάρων ακεραίων (**int array[4];**). Στη συνέχεια θα καλείται η συνάρτηση **void pwr(int *array_address, int array_size)**, η οποία θα μεταβάλλει τις τιμές των στοιχείων του πίνακα, υψώνοντας κάθε τιμή στοιχείου του πίνακα array στο τετράγωνο. Η ***main()*** θα τελειώνει με την εγγραφή των νέων τιμών του **array** στο αρχείο **file2.dat**. Η ανάγνωση από αρχείο και η εγγραφή σε αρχείο να γίνει με χρήση των συναρτήσεων **fread()**, **fwrite()**.

```
#include <stdio.h>
void pwr(int *array_address, int array_size);
void main()
{
    int i,array[4];
    FILE *f1;
    f1=fopen( "file1.dat","r" );
    fread( array,sizeof(int),4,f1 );
     fclose( f1 );
    pwr(array,4);
    f1=fopen( "file2.dat","w" );
    fwrite( array,sizeof(int),4,f1 );
    fclose( f1 );
}
void pwr(int *array_address, int array_size)
{
    int i;
    for (i=0;i<array_size;i++)
    {
        array_address[i]=array_address[i]*array_address[i];
        // ή ισοδύναμα:
        // *(array_address+i)=*(array_address+i)*(*(array_address+i));
```

```
        }
    }
```

Βιβλιογραφία

Α. Έντυπη

1 *Modern C++ Design: Generic Programming and Design Patterns*, A Alexandrescu, Addison-Wesley, 2001.

2 *C Προγραμματισμός*, H M Deitel, Prentice-Hall, 2003.

3 *C: How to Program*, H M Deitel, P J Deitel, Εκδόσεις Γκιούρδα, 1996.

4 *The Pragmatic Programmer: From Journeyman to Master*, A Hunt, D Thomas, Addison-Wesley, 1999.

5 *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, Κλ Θραμπουλίδης, Εκδόσεις Τζιόλα, 2002.

6 *A Book on C*, A Kelley. I Pohl, Addison-Wesley, 1997.

7 *The Practice of Programming*, B W Kernighan, Addison-Wesley, 1999.

8 *Η γλώσσα προγραμματισμού C*, B W Kernighan. D M Ritchie, Εκδόσεις Κλειδάριθμος, 1990.

9 *C Programming: A Modern Approach*, K N King, W.W. Norton & Company, 1996.

10 *The Art of Computer Programming*, D E Knuth, Addison-Wesley, 1997.

11 *Χρήση και Προγραμματισμός Turbo C++*, R Lafore, Εκδόσεις Γκιούρδα, 1992.

12 *Αντικειμενοστρεφής Προγραμματισμός με την Turbo C++*, R Lafore, Εκδόσεις Κλειδάριθμος, 1992.

13 *Go To*, S Lohr, Basic Books, 2001.

14 *Προγραμματισμός ΙΙ: Παρουσιάσεις Διαλέξεων*, Π Μαστοροκώστας, ΑΤΕΙ Σερρών, 2004.

15 *C++ Primer Plus*, S Prata, SAMS, 2002.

16 *Εγχειρίδιο εκμάθησης Turbo C*, H Schildt, Εκδόσεις Κλειδάριθμος, 1989.

17 *The C++ Programming Language*, B Stroustrup, Addison-Wesley, 1997.

18 *Expert C Programming*, P Van der Linden, Prentice Hall, 1994.

19 *C Βήμα προς Βήμα*, M Waite, S Prata, Εκδόσεις Γκιούρδα, 1996.

20 *Πλήρης Οδηγός Χρήσης της C*, M Waite, S Prata, D Martin, Εκδόσεις Γκιούρδα, 2000.

B. Ηλεκτρονική

1 http://www.cs.northwestern.edu/academics/courses/110/

2 http://www.stanford.edu/class/cs106a/

3 http://gd.tuwien.ac.at/languages/c/programming-bbrown/

4 http://gd.tuwien.ac.at/languages/c/programming-bbrown/

5 http://www.csd.uoc.gr/~hy150/EssentialC.pdf

6 http://www.cuj.com/

7 http://pw1.netcom.com/~tjensen/ptr/pointers.htm

8 http://www.acm.uiuc.edu/webmonkeys/book/c_guide/