ΤΕΙ ΣΕΡΡΩΝ

Τμήμα Πληροφορικής & Επικοινωνιών

Τομέας Υπολογιστικών Τεχνικών & Συστημάτων

Διαδικαστικός Προγραμματισμός

Δρ. Πάρις Μαστοροκώστας

Αναπληρωτής Καθηγητής

Σέρρες

2006

Κεφάλαιο 3: Τελεστές – εκφράσεις

Κεφάλαιο 4: Έλεγχος ροής – προτάσεις υπό συνθήκη διακλάδωσης

Κεφάλαιο 5: Προτάσεις επανάληψης – βρόχοι

Κεφάλαιο 6: Πίνακες – αλφαριθμητικά

Κεφάλαιο 10: Δείκτες

Βιβλιογραφία

Κεφάλαιο 1

ΕΙΣΑΓΩΓΗ
ΒΑΣΙΚΑ ΣΤΟΙΧΕΙΑ ΠΡΟΓΡΑΜΜΑΤΟΣ

1.1 Εισαγωγή

Αντικείμενο του παρόντος συγγράματος είναι η εισαγωγή του αναγνώστη στη λογική του προγραμματισμού Η/Υ. Το ενδιαφέρον εστιάζεται στον καλούμενο **διαδικαστικό προγραμματισμό** (procedural programming), βασικά στοιχεία του οποίου είναι η δόμηση του προγράμματος και η επαναλαμβανόμενη χρήση υποπρογραμμάτων, τα οποία είτε επιτελούν εργασίες γενικής φύσης είτε απευθύνονται σε ένα τμήμα του συνολικού προβλήματος. Στόχος είναι η κατανόηση των αρχών του προγραμματισμού και η εμπέδωση της φιλοσοφίας του, έτσι ώστε ο αναγνώστης να μπορεί χωρίς δυσχέρειες να προχωρήσει σε άλλες μορφές προγραμματισμού, όπως ο αντικειμενοστραφής προγραμματισμός (object–oriented programming).

Στην προσπάθεια αυτή θα χρησιμοποιηθεί ως πλατφόρμα μία γλώσσα προγραμματισμού **υψηλού επιπέδου**, η γλώσσα C. Ο όρος γλώσσα υψηλού επιπέδου υποδηλώνει ότι δεν είναι κατασκευασμένη για να λειτουργεί σε συγκεκριμένη αρχιτεκτονική υπολογιστή αλλά δύναται να λειτουργήσει σε πληθώρα αρχιτεκτονικών, γεγονός που σημαίνει ότι:

- Οι διάφορες αρχιτεκτονικές υπολογιστών για να λειτουργήσουν χρησιμοποιούν ένα σύνολο εντολών, το οποίο χρησιμοποιεί τη γλώσσα μηχανής της εκάστοτε αρχιτεκτονικής. Επομένως, έαν κάποιος προγραμματίσει κάνοντας χρήση της γλώσσας μηχανής μίας αρχιτεκτονικής, τα προγράμματά του δε θα είναι συμβατά με άλλες αρχιτεκτονικές. Το αντιστάθμισμα αυτού του μειονεκτήματος είναι ότι στο παρελθόν, που οι υπολογιστές είχαν λίγη μνήμη και χαμηλή ταχύτητα, ο προγραμματισμός σε γλώσσα μηχανής χειριζόταν αποδοτικότερα τους πόρους του μηχανήματος.

- Για να είναι μία γλώσσα συμβατή με τις γλώσσες μηχανής διαφόρων αρχιτεκτονικών, θα πρέπει να λαμβάνει χώρα μεταγλώττιση από τη γλώσσα προγραμματισμού στην εκάστοτε γλώσσα μηχανής. Όντως αυτό συμβαίνει και το λογισμικό που επιτελεί τη συγκεκριμένη εργασία ονομάζεται **μεταγλωττιστής** (compiler).

Με βάση τα παραπάνω, οι γλώσσες προγραμματισμού υψηλού επιπέδου είναι *μεταγλωττισμένες* γλώσσες, αποσκοπώντας στο να είναι ευανάγνωστες και κατανοητές.

Σε ό,τι αφορά τη C, παρουσιάζει μία σειρά από ενδιαφέροντα και χρήσιμα χαρακτηριστικά:

- Μπορεί να χρησιμοποιηθεί και ως γλώσσα προγραμματισμού χαμηλού επιπέδου, επιτρέποντας άμεση πρόσβαση στους πόρους του υπολογιστή.
- Είναι σχετικά μικρή και εύκολη στην εκμάθηση.
- Υποστηρίζει δομημένο προγραμματισμό.
- Είναι αποτελεσματική, παράγοντας συμπαγή και γρήγορα στην εκτέλεση προγράμματα.
- Αποτελεί μαζί με τη C++ τις ευρύτερα χρησιμοποιούμενες γλώσσες σε ερευνητικά και αναπτυξιακά προγράμματα, γεγονός που έχει δημιουργήσει μία πολλή μεγάλη εγκατεστημένη βάση εφαρμογών που αναπτύχθηκαν με αυτές τις γλώσσες και πρέπει να συντηρούνται και να εξελίσσονται.

1.2 Ιστορική αναδρομή

Η C επινοήθηκε το 1972 από τον Dennis Ritchie, στα εργαστήρια Bell. Δημιουργήθηκε για να εξυπηρετήσει το λειτουργικό σύστημα Unix, το οποίο έως τότε ήταν γραμμένο σε assembly. Ο δημιουργός του Unix, Ken Thompson, φίλος και συνεργάτης του Ritchie, είχε δημιουργήσει την πρόγονο της C, τη γλώσσα B. Και οι δύο γλώσσες έχουν κοινή καταγωγή από τη γλώσσα BCPL, η οποία είχε αναπτυχθεί από τον Martin Richards κατά το πέρασμά του απο το Τεχνολογικό Ινστιτούτο της Μασσαχουσέττης (MIT) το 1967, στηριζόμενη στη γλώσσα CPL (Cambridge Programming Language) του πανεπιστημίου του Cambridge. Και οι τρεις γλώσσες κατασκευάστηκαν στα πλαίσια του προγράμματος MAC και του απόγονού του Multics, τα οποία στόχευαν στην κατανομή των πόρων των υπολογιστών σε πολλούς χρήστες. Τα δύο αυτά προγράμματα, στα οποία συνέπραξαν το MIT, η General Electric και τα εργαστήρια Bell, απετέλεσαν τη θερμοκοιτίδα πολλών προγραμμάτων λογισμικού, που

κυριαρχούν από τη δεκαετία του 1960 έως σήμερα. Για ενδελεχή μελέτη της ιστορίας του λογισμικού, ο αναγνώστης μπορεί να ανατρέξει στην αναφορά [12].

Η C, ούσα ευέλικτη και αποδοτική χρησιμοποιήθηκε αρχικά για τον προγραμματισμό συστημάτων στο Unix. Το 1974 εμφανίσθηκε από τον Brian Kernighan το πρώτο γραπτό κείμενο για τη γλώσσα, υπό τον τίτλο "Programming in C: A Tutorial". Το 1977 έγινε η πρώτη επίσημη τεκμηρίωση της γλώσσας με το βιβλίο "The C Programming Language" από τους Kernighan και Ritchie. Το βιβλίο αυτό απετέλεσε το «ευαγγέλιο» των προγραμματιστών της C, αποκαλούμενο «Λευκή Βίβλος» ή «πρότυπο K&R».

Με την πάροδο του χρόνου η γλώσσα C άρχισε να χρησιμοποιείται και σε άλλα πεδία εφαρμογών, πέραν του προγραμματισμού συστημάτων. Η εμφάνιση των μεταγλωττιστών της γλώσσας στο MS–DOS και ο μεγάλος αριθμός προγραμμάτων βιβλιοθήκης που κατασκευάστηκαν, οδήγησαν τη γλώσσα στο απόγειό της στα τέλη της δεκαετίας του 1980. Βέβαια η γλώσσα γνώρισε πολλές αλλαγές, οδηγούμενη τελικά στην επονομαζόμενη **ANSI έκδοση**. Η τελευταία ενημέρωση της γλώσσας έγινε το 1999.

Τα τελευταία χρόνια τα ηνία έχει λάβει ο αντικειμενοστρεφής προγραμματισμός και στην κατεύθυνση αυτή συνέβαλλε η γλώσσα C++, που επινοήθηκε το 1983 από το Δανό Bjarne Stroustrup στα εργαστήρια της AT&T (το τμήμα των εργαστήρια Bell που πήγαν στην AT&T όταν η εταιρεία διασπάσθηκε το 1995). Η C++ – ή *C με τάξεις* – μπορεί να θεωρηθεί απόγονος της C, αν και έχει αρκετές διαφορές. Γεγονός είναι ότι αντικατέστησε τη C σε πολύ μεγάλο ποσοστό και αποτελεί σήμερα μία από τις κυρίαρχες γλώσσες προγραμματισμού.

1.3 Εργαλεία ανάλυσης προβλημάτων

Για την ανάλυση ενός προβλήματος και την κατάρτιση του μοντέλου που θα υλοποιηθεί σε κώδικα υπάρχουν τρία εργαλεία:

1. Η **φυσική γλώσσα** (natural language), σύμφωνα με την οποία το πρόβλημα αναλύεται σε απλές προτάσεις της καθομιλουμένης γλώσσας, χρησιμοποιώντας το συντακτικό αυτής.

2. **Το διάγραμμα ροής** (flow chart), σύμφωνα με το οποίο απεικονίζεται γραφικά η εξέλιξη του προβλήματος, με χρήση ειδικών συμβόλων.

3. **Ο ψευδοκώδικας** (pseudocode), ο οποίος μετασχηματίζει τη φυσική γλώσσα σε μία σειρά προτάσεων που χρησιμοποιούν το συντακτικό γλώσσας προγραμματισμού, χωρίς

να ακολουθούν επακριβώς το φορμαλισμό συγκεκριμένης γλώσσας.

Δεν υπάρχουν γενικοί κανόνες για την υιοθέτηση κάποιου από τα τρία εργαλεία. Η χρήση καθενός εκ των τριών εξαρτάται από τον εκάστοτε προγραμματιστή και το συγκεκριμένο πρόβλημα. Μία ενδεικτική περιγραφή της λειτουργίας των ανωτέρω εργαλείων θα γίνει με τη βοήθεια του ακόλουθου παραδείγματος:

Να μετατραπούν οι βαθμοί Farenheit σε βαθμούς Κελσίου, χρησιμοποιώντας την εξίσωση μετασχηματισμού C=(F-32))5/9.

1. Με χρήση φυσικής γλώσσας

Ζήτησε από το χρήστη τη θερμοκρασία σε βαθμούς F

Διάβασε την τιμή που δίνει ο χρήστης

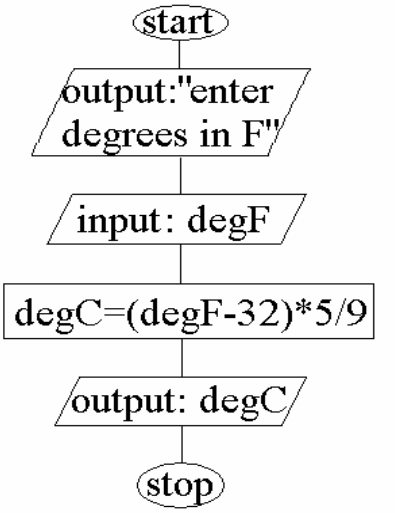
Αποθήκευσε την τιμή σε θέση αποθήκευσης που ονομάζεται degF

Υπολόγισε τους βαθμούς C με χρήση μαθηματικής σχέσης

Αποθήκευσε το αποτέλεσμα σε θέση αποθήκευσης που ονομάζεται degC

Τύπωσε το περιεχόμενο της degC

2. Με χρήση διαγράμματος ροής



2. Με χρήση ψευδοκώδικα

```
print “enter degrees in Farenheit”
read degF
degC = (degF - 32) * 5 / 9
print degC
```

1.4 Στάδια υλοποίησης προγράμματος

Το υπολογιστικό πρόγραμμα είναι μία ακολουθία εντολών, με τις οποίες ο υπολογιστής εκτελεί μία συγκεκριμένη εργασία και επιλύει ένα δοθέν πρόβλημα. Η υλοποίηση ενός προγράμματος περιλαμβάνει τέσσερα στάδια:

1. Η συγγραφή του πηγαίου κώδικα (source code). Στο βήμα αυτό χρησιμοποιείται ένας **συντάκτης προγράμματος** (text editor) για τη συγγραφή του κώδικα. Συνήθως χρησιμοποιείται ο ενσωματωμένος συντάκτης της C. Το αποτέλεσμα είναι ένα αρχείο κειμένου, αναγνώσιμο από οποιονδήποτε συντάκτη (DOS editor, notepad, wordpad, πρόγραμμα επεξεργασίας κειμένου), το οποίο έχει κατάληξη **.c** (ή **.cpp** εάν χρησιμοποιείται η C++).

2. Η μεταγλώττιση του πηγαίου κώδικα (compilation). Η διαδικασία της μεταγλώττισης εκτελείται αυτόματα από το **μεταγλωττιστή** (compiler) και παράγεται ο **τελικός ή αντικείμενος κώδικας** (object code), που είναι ο κώδικας σε γλώσσα μηχανής. Στο στάδιο αυτό ανιχνεύονται τα **συντακτικά σφάλματα** (syntax errors), τα οποία είναι σφάλματα που οφείλονται σε παραβίαση των συντακτικών κανόνων, και αναφέρονται υπό μορφή λίστας (οπότε μπορούν να διορθωθούν προτού εκτελεσθεί το πρόγραμμα). Εάν δεν υπάρχουν συντακτικά σφάλματα, το αρχείο που προκύπτει ονομάζεται έχει το όνομα του αρχείου του πηγαίου κώδικα και κατάληξη **.obj**.

3. Η σύνδεση του αντικείμενου κώδικα. Στο τρίτο στάδιο επιτελείται η διεργασία της σύνδεσης (linking), η οποία είναι πλήρως αυτοματοποιημένη και γίνεται με χρήση του **συνδέτη** (linker). Το αποτέλεσμα της σύνδεσης είναι η δημιουργία του εκτελέσιμου κώδικα ή η αναφορά τυχόν προβλημάτων, όπως για παράδειγμα η αδυναμία εντοπισμού μίας συνάρτησης ή μίας εξωτερικής μεταβλητής. Ο εκτελέσιμος κώδικας αποθηκεύεται σε αρχείο που έχει το όνομα του πηγαίου κώδικα και επέκταση **.exe**. Πλέον το πρόγραμμα έχει ξεφύγει από το περιβάλλον της γλώσσας προγραμματισμού και μπορεί να εκτελεσθεί όπως ένα οποιοδήποτε εκτελέσιμο αρχείο του υπολογιστή.

Ο συνδέτης έχει τη δυνατότητα να συνδέσει περισσότερα του ενός αρχεία αντικείμενου κώδικα. Επιπλέον, αναζητά σε βιβλιοθήκες τα σώματα των συναρτήσεων που ο προγραμματιστής χρησιμοποίησε στο πρόγραμμά του (λεπτομέρειες στην §1.5 και σε επόμενα κεφάλαια).

4. Η εκτέλεση του προγράμματος. Στο τελευταίο στάδιο εκτελείται ή «τρέχει» το

αρχείο **.exe** και ελέγχεται η ορθή λειτουργία του προγράμματος. Τα σφάλματα που ανακύπτουν στο στάδιο αυτό ονομάζονται **σημασιολογικά σφάλματα** (semantic errors). Οφείλονται σε κακή σχεδίαση της λύσης του προβλήματος και, δυστυχώς, αναγνωρίζονται στο χρόνο εκτέλεσης. Χαρακτηριστικό σφάλμα εκτέλεσης είναι η διαίρεση αριθμού με το μηδέν, ενέργεια που δεν ανιχνεύεται ως λανθασμένη κατά τη μεταγλώττιση.

Σε περίπτωση σφάλματος, ο προγραμματιστής επιστρέφει στο πρώτο στάδιο, όπου κάνει τις διορθώσεις, και επαναλαμβάνει τα υπόλοιπα στάδια έως ότου το πρόγραμμα λειτουργήσει επιτυχώς.

1.5 Βασικά στοιχεία προγράμματος

Έστω το ακόλουθο απλό πρόγραμμα:

```
/*****************************************
  This program prints out the sentence "This is a test."
***************************************** */
#include<stdio.h>
void main ()
{
      printf( "This is a test.\n" );
}  // end of main
```

- Η καρδιά ενός προγράμματος της γλώσσας C είναι η λέξη ***main***. Αντιστοιχεί στο κύριο τμήμα του κώδικα και χρησιμοποιείται για να γνωστοποιήσει το σημείο έναρξης εκτέλεσης του προγράμματος. Μετά τη ***main*** ακολουθεί το εισαγωγικό αριστερό άγκιστρο ({) και κατόπιν οι γραμμές του προγράμματος. Η ***main***, και μαζί το πρόγραμμα, τερματίζει με το καταληκτικό δεξί άγκιστρο (}). Στο παρόν πρόγραμμα η ***main*** αποτελείται από μία μόνο πρόταση κλήσης της συνάρτησης ***printf***, η οποία ανήκει στη βασική βιβλιοθήκη συναρτήσεων της C. Η βιβλιοθήκη συναρτήσεων περιλαμβάνει τον κώδικα πρότυπων συναρτήσεων και αποτελείται από μία σειρά αρχείων, τα οποία έχουν την κατάληξη **h**. Τα αρχεία αυτά ονομάζονται **αρχεία κεφαλίδας** (header files) και περιλαμβάνουν συναρτήσεις συναφούς λειτουργίας. Δηλώνονται πριν από τη ***main***, με χρήση της οδηγίας προς τον προεπεξεργαστή ***include*** ως εξής:

#include <όνομα_αρχείου_κεφαλίδας.h>

Στην παρούσα περίπτωση το αρχείο κεφαλίδας είναι το ***stdio.h*** (**st**an**d**ard **i**nput–**o**utput), το οποίο περιλαμβάνει συναρτήσεις σχετιζόμενες με τις συσκευές εισόδου (π.χ. πληκτρολόγιο) και εξόδου (π.χ. οθόνη).

- Η συνάρτηση ***main*** εκτός από προτάσεις και κλήσεις σε συναρτήσεις βιβλιοθήκης μπορεί να καλεί και άλλες συναρτήσεις, οι οποίες δημιουργούνται από το προγραμματιστή. Μία συνάρτηση είναι ένα σύνολο προτάσεων με ένα δεδομένο όνομα, όπως είναι η ***main*** ή η ***printf***. Οι προτάσεις αποτελούν το **σώμα** (body) της συνάρτησης και περικλείονται σε άγκιστρα. Οι συναρτήσεις πριν από τη ***main*** και αναπτύσσουν τον κώδικά τους είτε πάνω είτε κάτω από τη ***main***[1].

- Όλες οι προτάσεις τελειώνουν με **ερωτηματικό** (**;**) (semicolon), το οποίο ονομάζεται **σύμβολο του τερματιστή προτάσεων** (statement terminator symbol). Εξαίρεση αποτελούν οι οδηγίες προς τον προεπεξεργαστή, όπως είναι η ***include***, οι οποίες αρχίζουν πάντοτε με **δίεση** (#) και δεν τελειώνουν με ερωτηματικό**.**

- Η έξοδος του προγράμματος είναι η έξοδος της ***printf***: **This is a test.**

- Οι τρεις πρώτες γραμμές του ανωτέρω προγράμματος είναι **σχόλια** (comments) και δεν αποτελούν τμήμα του κώδικα, καθώς δε λαμβάνονται υπόψη από το μεταγλωττιστή. Το σχόλιο είναι κείμενο ανάμεσα σε /* και */ και μπορεί να τοποθετηθεί οπουδήποτε μέσα στο πρόγραμμα. Μπορεί να επεκταθεί σε περισσότερες από μία γραμμές. Σε περίπτωση που πρέπει να τοποθετηθεί ένα σχόλιο σε κάποιο σημείο μίας γραμμής και να μην επεκταθεί σε άλλη γραμμή, χρησιμοποιείται το σύμβολο // και ακολούθως τοποθετείται το σχόλιο, όπως συμβαίνει στο τέλος του ανωτέρω προγράμματος (το κείμενο **end of main** είναι σχόλιο).

Τα σχόλια πρέπει να χρησιμοποιούνται αφειδώς γιατί καταστούν τον κώδικα ευανάγνωστο και συνεισφέρουν στην επεξήγηση δυσνόητων σημείων. Είναι θεμιτό να ευθυγραμμίζονται τα σύμβολα των σχολίων και να μην τοποθετούνται ποτέ σχόλια μέσα σε σχόλια (ένθετα σχόλια) γιατί μπορεί να δημιουργηθεί πρόβλημα σε μερικούς μεταγλωττιστές.

Παρακάτω παρουσιάζονται μερικές περιπτώσεις σωστών και λανθασμένων σχολίων:

[1] Λεπτομερής ανάλυση του συντακτικού και της λειτουργίας των συναρτήσεων στο αντίστοιχο κεφάλαιο.

α) **/* Αυτό /* το σχόλιο */ είναι λανθασμένο */**

β) /*

Αυτό το

σχόλιο

είναι σωστό

*/

Παρατηρήσεις:

1. Η γλώσσα C διαχωρίζει τα κεφαλαία γράμματα από τα μικρά (case sensitive). Η εντολή ***Printf*** δεν είναι ίδια με την ***printf.*** Όλες οι εντολές στη C γράφονται με μικρά γράμματα!

2. Η σωστή στηλοθεσία είναι πολύ σημαντική καθώς καθιστά τον κώδικα ευανάγνωστο.

3. Η συνάρτηση ***printf*** ανήκει στις **μορφοποιούμενες** συναρτήσεις εισόδου–εξόδου. Ονομάζεται μορφοποιούμενη γιατί δίνει τη δυνατότητα στο χρήστη να μορφοποιήσει την έξοδό της, δυνάμενη να εκτυπώσει μεταβλητές διαφόρων τύπων και με διάφορους τρόπους, χρησιμοποιώντας κατάλληλα σύμβολα. Δυαδική της ***printf*** είναι η ***scanf***, η οποία λαμβάνει πληροφορία από την είσοδο (πληκτρολόγιο).

Η πρόταση **printf("This is a test.\n");** καλεί την ***printf*** για να τυπωθεί το καθορισμένο κείμενο. Τα **ορίσματα εισόδου** (input arguments) περικλείονται από παρενθέσεις και προσδιορίζουν το προς εκτύπωση κείμενο και τη μορφή με την οποία θα εκτυπωθεί. Τέλος, το σύμβολο **\n**, που ανήκει στις **ακολουθίες διαφυγής**, σημαίνει «μετακινήσου στην επόμενη γραμμή». Λεπτομερής περιγραφή της λειτουργίας των συναρτήσεων εισόδου – εξόδου δίνεται στο επόμενο κεφάλαιο.

4. Πέραν της ***include***, μία σημαντική οδηγία προς τον προεπεξεργαστή είναι η ***define***, η οποία αντιστοιχίζει ένα όνομα με μία σταθερά ή με μία σειρά χαρακτήρων. Οποτεδήποτε εμφανίζεται το όνομα μέσα στον κώδικα, αντικαθίσταται αυτόματα με την τιμή της σταθεράς ή τη συμβολοσειρά. Για παράδειγμα, εάν χρησιμοποιηθεί η λέξη **TRUE** στη θέση της τιμής **1** και η λέξη **FALSE** στη θέση της τιμής **0**, θα δοθούν δύο ***define*** ως εξής:

#define TRUE 1

#define FALSE 0

Εάν αντικατασταθεί ολόκληρη φράση, μπορεί να εμφανισθεί στην οθόνη με χρήση της ***printf***:

#define TITLOS "TEI of Serres\nDpt of Informatics and Communications\n"

printf(TITLOS);

Το αποτέλεσμα είναι:

TEI of Serres

Dpt of Informatics and Communications

Μία συνηθισμένη χρήση της define είναι για τον καθορισμό του μεγέθους στοιχείων, όπως είναι η διάσταση ενός πίνακα, τα οποία μπορεί να αλλάξουν κατά την εκτέλεση του προγράμματος. Περισσότερα για αυτό το ζήτημα θα αναφερθούν στο κεφάλαιο 6.

1.6 Λεξιλόγιο της γλώσσας C

Από τα αναφερθέντα στην προηγούμενη παράγραφο γίνεται φανερό ότι η λειτουργία τής C στηρίζεται σε ένα λεξιλόγιο. Το λεξιλόγιο της C περιλαμβάνει τέσσερις μείζονες κατηγορίες λέξεων:

1. Δεσμευμένες λέξεις

2. Λέξεις κλειδιά

3. Τελεστές

4. Αναγνωριστές

1.6.1 Δεσμευμένες λέξεις

Οι δεσμευμένες λέξεις (reserved words) χρησιμοποιούνται από τη C κατά τρόπο αποκλειστικό και πρέπει να αποφεύγεται η χρήση τους ως ονόματα. Αποτελούνται από:

- Ονόματα συναρτήσεων πρότυπης βιβλιοθήκης (runtime function names), όπως ***printf***, ***abs*** κ.λ.π.
- Macro names. Είναι ονόματα που περιέχονται σε αρχεία κεφαλίδας για ορισμό μακροεντολών, π.χ. **EOF**, **INT_MAX**.
- Type names. Είναι ονόματα τύπων σε ορισμένα αρχεία κεφαλίδας, π.χ. **time_t**, **va_list**.
- Ονόματα εντολών προεπεξεργαστή (preprocessor). Είναι ονόματα που χρησιμοποιεί προεπεξεργαστής της C και έχουν προκαθορισμένη σημασία, π.χ. ***include***, ***define***.
- Ονόματα που αρχίζουν με το χαρακτήρα υπογράμμισης _ και έχουν δεύτερο

χαρακτήρα τον ίδιο ή κεφαλαίο γράμμα, π.χ. **_DATE_**, **_FILE_**.

1.6.2 Λέξεις κλειδιά

Οι λέξεις κλειδιά (keywords) είναι λεκτικές μονάδες, οι οποίες είτε μόνες τους είτε με άλλες λεκτικές μονάδες χαρακτηρίζουν κάποια γλωσσική κατασκευή. Π.χ. η λέξη **int** αναπαριστά τον ακέραιο τύπο δεδομένων και το ζεύγος **if–else** χρησιμοποιείται στον έλεγχο ροής προγράμματος.

Οι λέξεις κλειδιά, αν και είναι ένας περιορισμός των γλωσσών, αυξάνουν την αναγνωσιμότητα και αξιοπιστία των προγραμμάτων ενώ ταυτόχρονα επιταχύνουν τη διαδικασία της μεταγλώττισης. Λέξεις κλειδιά όπως **if**, **else**, **for**, **case**, **while**, **do** έχουν γίνει κοινά αποδεκτές, διευκολύνοντας την εκμάθηση των γλωσσών προγραμματισμού.

Στον πίνακα που ακολουθεί παρατίθενται οι λέξεις κλειδιά της C:

auto	**else**	**register**	**union**
break	**enum**	**return**	**unsigned**
case	**extern**	**short**	**void**
char	**float**	**signed**	**volatile**
const	**for**	**sizeof**	**while**
continue	**goto**	**static**	
default	**if**	**struct**	
do	**int**	**switch**	
double	**long**	**typedef**	

Πίνακας 1.1 *Λέξεις κλειδιά της C*

1.6.3 Αναγνωριστές

Οι αναγνωριστές (identifiers) είναι λεκτικές μονάδες που κατασκευάζει ο προγραμματιστής. Αυτές οι λεκτικές μονάδες χρησιμοποιούνται συνήθως ως ονόματα που ο προγραμματιστής δίνει σε δικές του κατασκευές, όπως μεταβλητές, σταθερές, συναρτήσεις και δικούς του τύπους δεδομένων. Ένα όνομα προσδιορίζει μοναδιαία, από το σύνολο των κατασκευών του προγράμματος, την κατασκευή στην οποία αποδόθηκε, εξ ου και το όνομα αναγνωριστής. Περισσότερα στοιχεία για τους αναγνωριστές σημειώνονται στην §2.1.2.

1.7 Κανόνες δημιουργίας ευανώγνωστων προγραμμάτων

- Θα πρέπει να αποφεύγονται ονόματα ενός χαρακτήρα, όπως **i, j, x, y** (εκτός από ειδικές περιπτώσεις που θα εξετασθούν αργότερα).
- Τα ονόματα που χρησιμοποιούνται θα πρέπει να είναι εκφραστικά ονόματα. Συγκεκριμένα:
    1. Η μεταβλητή που αναπαριστά την ταχύτητα θα μπορούσε να ονομασθεί **velocity** και τη μέγιστη τιμή της **max_velocity** ή **maxVelocity**.
    2. Η συνάρτηση που εμφανίζει τα λάθη στην οθόνη μπορεί να λάβει τα ενδεικτικά ονόματα **display_error** ή **displayError**.
- Για καλύτερη αναγνωσιμότητα των μεταβλητών που αποτελούνται από δύο ή περισσότερες λέξεις ο προγραμματιστής θα πρέπει να αποφασίσει εάν θα χρησιμοποιήσει τη μορφή ***display_error*** ή τη μορφή ***displayError***. Η σύμβαση που θα επιλεγεί θα πρέπει να τηρηθεί σε όλο το πρόγραμμα.
- Θα πρέπει να χρησιμοποιούνται μικρά γράμματα για ονόματα μεταβλητών.

Κεφάλαιο 2

ΜΕΤΑΒΛΗΤΕΣ–ΣΤΑΘΕΡΕΣ–Ι/Ο ΚΟΝΣΟΛΑΣ

2.1 Η έννοια της μεταβλητής

Ένα από τα βασικότερα πλεονεκτήματα του υπολογιστή είναι η δυνατότητά του να διαχειρίζεται πληροφορία, σε μορφή αριθμητικών δεδομένων, γραμμάτων ή ακολουθίας γραμμάτων. Τα δεδομένα αποθηκεύονται στη μνήμη και απαιτούν ένα μέσο για να κληθούν από τα προγράμματα, να εισαχθούν σε υπολογισμούς και να δημιουργήσουν νέα δεδομένα. Αρχικά οι προγραμματιστές χειρίζονταν τα δεδομένα δουλεύοντας με τις διευθύνσεις μνήμης στις οποίες ήταν αποθηκευμένα. Με την αύξηση του μεγέθους και της πολυπλοκότητας των προγραμμάτων, αυτός ο τρόπος διαχείρισης έθετε πολλούς περιορισμούς και αποτελούσε πηγή προβλημάτων.

Η λύση στο πρόβλημα της αναφοράς σε δεδομένα και στη διαχείρισή τους δόθηκε με την εισαγωγή της έννοιας των μεταβλητών, οι οποίες είναι φορείς δεδομένων. Το όνομα μίας μεταβλητής είναι άμεσα συνδεδεμένο με τη διεύθυνση στην οποία είναι αποθηκευμένο το δεδομένο. Με τον τρόπο αυτό ο προγραμματιστής μπορεί να χειρισθεί δεδομένα χωρίς να γνωρίζει την ακριβή διεύθυνση της μνήμης όπου αυτά τοποθετούνται.

Επιγραμματικά, η χρήση των μεταβλητών είναι ίδια με εκείνη της άλγεβρας, π.χ. η παράσταση **y=3x+5** ισχύει τόσο στα μαθηματικά όσο και στις γλώσσες προγραμματισμού, με τα **x** και **y** να είναι μεταβλητές. Ωστόσο στον προγραμματισμό η χρήση της είναι **γενικευμένη**: η μεταβλητή είναι μία θέση μνήμης για ένα δεδομένο. Δημιουργείται όταν δηλώνεται και η τιμή της μπορεί να είναι άγνωστη έως ότου χρησιμοποιηθεί από το πρόγραμμα.

2.1.1 Δήλωση μεταβλητής

Οι μεταβλητές δηλώνονται με ***πρόταση ορισμού***, η οποία τελειώνει πάντοτε με (**;**). Η

μορφή της δήλωσης είναι: **data_type var, var, … ;**

π.χ. **int counter1, counter2;**

Οι μεταβλητές δηλώνονται στην αρχή μίας συνάρτησης, συνήθως αμέσως μετά το εισαγωγικό άγκιστρο ({), και οπωσδήποτε πριν από τη χρήση τους.

Η δήλωση γνωστοποιεί στο μεταγλωττιστή το όνομα και τις ιδιότητες της μεταβλητής. Μία δήλωση έχει ως αποτέλεσμα τη σύνδεση του ονόματος της μεταβλητής με:

- τον ανάλογο τύπο δεδομένων, γεγονός που λαμβάνει χώρα στο χρόνο μεταγλώττισης (compile–time)
- μία θέση μνήμης κατάλληλου μεγέθους, γεγονός που λαμβάνει χώρα στο χρόνο εκτέλεσης (run–time)

2.1.2 Ονομασία μεταβλητής

Σε ό,τι αφορά την ονοματολογία, ακολουθούνται οι εξής κανόνες:

- Στην C τα ονόματα των μεταβλητών σχηματίζονται από:
  α) τα γράμματα του αλφαβήτου
  β) τα ψηφία 0 έως 9
  γ) το χαρακτήρα υπογράμμισης __ (underscore)
- Το όνομα πρέπει να ξεκινά με γράμμα ή με χαρακτήρα υπογράμμισης (στη δεύτερη περίπτωση ο επόμενος χαρακτήρας πρέπει να είναι μικρό γράμμα).
- Το όνομα δεν πρέπει να είναι ίδιο με δεσμευμένη λέξη.
- Σημαντικοί είναι μόνο οι πρώτοι 31 χαρακτήρες του ονόματος. Οι υπόλοιποι δε λαμβάνονται υπόψη.
- Τα όνομα μεταβλητής πρέπει να είναι ενδεικτικό της ιδιότητάς της ή του τύπου δεδομένου που αντιπροσωπεύει, έτσι ώστε να δίνει πληροφορία στον προγραμματιστή.

Με βάση τα παραπάνω, ακολουθούν ενδεικτικές περιπτώσεις ονοματολογίας.

- Έγκυρα ονόματα μεταβλητών:

  totalArea **max_amount** **counter1**

  Counter1 **_temp_in_F**

- Μη έγκυρα ονόματα μεταβλητών:

  $product **total%** **3rd**

- Απαράδεκτα ονόματα μεταβλητών:

  l[1] **x2**

  maximum_number_of_students_in_my_class

2.2 Τύποι μεταβλητών

Οι μεταβλητές της γώσσας C ανήκουν σε δύο κατηγορίες. Η κατηγορία των βαθμωτών τύπων περιλαμβάνει τους ακέραιους (integers), οι οποίοι δηλώνονται με την κωδική λέξη **int**, τους πραγματικούς, οι οποίοι μερίζονται στους αριθμούς κινητής υποδιαστολής με κωδική λέξη **float** και τους αριθμούς διπλής ακρίβειας με κωδική λέξη **double**, τη μεταβλητή χαρακτήρα (character, **char**), τους δείκτες (pointers) και τον απαριθμητικό τύπο (enumerated, **enum**).

Στην κατηγορία των συναθροιστικών τύπων ανήκουν οι πίνακες, οι δομές (**struct**) και οι ενώσεις (**union**). Στη συνέχεια του κεφαλαίου γίνεται αναφορά στους βασικούς τύπους της C: **char**, **int**, **float**, **double** και σε επόμενο κεφάλαιο αναπτύσσονται οι πίνακες. Οι άλλοι τύποι μεταβλητών θα μελετηθούν αργότερα.

2.2.1 Ο τύπος του χαρακτήρα

Ο τύπος του χαρακτήρα (**char**) παριστάνει απλούς χαρακτήρες του αλφάβητου της γλώσσας. Βρίσκεται ανάμεσα σε απλά εισαγωγικά (π.χ. **'C'**, **'2'**, **'*'**, **')'**).

- Η δήλωση της μεταβλητής χαρακτήρα ακολουθεί τον εξής φορμαλισμό:

char όνομα_μεταβλητής; π.χ. **char choice**;

Υπάρχει η δυνατότητα ταυτόχρονα με τη δήλωση να αποδοθεί αρχική τιμή στη μεταβλητή:

char choice='A';

- Η εισαγωγή τιμών στις μεταβλητές χαρακτήρα γίνεται με χρήση της συνάρτησης ***scanf*** και του **προσδιοριστή** (specifier) **%c** (character). Η πρόταση

scanf("%c", &ch);

διαβάζει από την κύρια είσοδο (πληκτρολόγιο) ένα χαρακτήρα και τον αποδίδει στη

[1] Στην πορεία ανάγνωσης του κειμένου ο αναγνώστης θα παρατηρήσει ότι οι ονομασίες των μεταβλητών δε συνάδουν πάντοτε με τους κανόνες που αναφέρονται, καθώς χρησιμοποιούνται μεταβλητές με ένα ή δύο γράμματα. Η χρήση τους γίνεται *συγγραφική αδεία*, επιβληθείσα για λόγους πρακτικούς, έτσι ώστε να μη διευρύνεται ο κώδικας.

μεταβλητή **ch**. Θα πρέπει να προσεχθεί η χρήση του **&** πριν από τη μεταβλητή. Ονομάζεται **τελεστής διεύθυνσης** και προηγείται πάντοτε των μεταβλητών στην εντολή ***scanf***.

- Η εκτύπωση μεταβλητών χαρακτήρα γίνεται με χρήση της συνάρτησης ***printf*** και του προσδιοριστή **%c**. Η πρόταση

printf("The character is %c\n", choice);

θα τυπώσει (θεωρώντας ότι στη **choice** εισήχθη ο **A**)

The character is A

Παρατήρηση: Επειδή κάθε χαρακτήρας του κώδικα ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) αντιστοιχεί σε έναν οκταψήφιο δυαδικό αριθμό, εάν αντί του **%c** χρησιμοποιηθεί ο προσδιοριστής **%d** (decimal), η ***printf*** θα εμφανίσει τον ASCII κωδικό τού χαρακτήρα. Η πρόταση

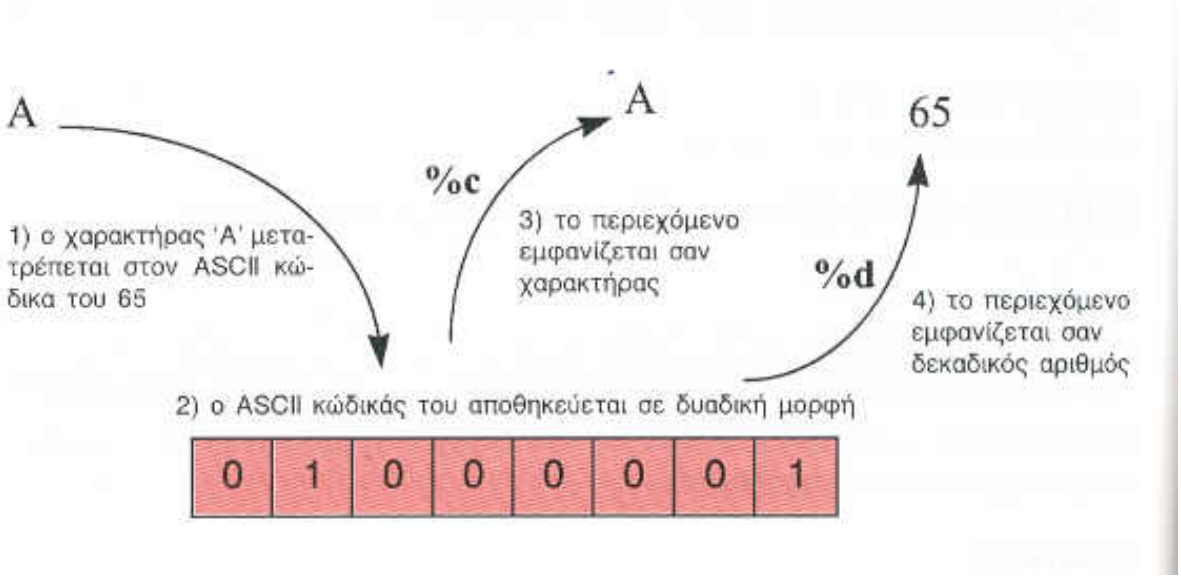
printf("The ASCII Code of %c is %d\n", choice,choice);

θα τυπώσει

The ASCII Code of A is 65

όπου **65** είναι το δεκαδικό ισοδύναμο του δυαδικού κωδικού ASCII για το χαρακτήρα **A**.

- Ο μεταγλωττιστής απαιτεί 1 byte μνήμης για την αποθήκευση της τιμής μίας μεταβλητής χαρακτήρα. Στο σχήμα 2.1 παρουσιάζονται οι διαδικασίες της αποθήκευσης και ανάκλησης για το χαρακτήρα A. Η τιμή της μεταβλητής μετατρέπεται σε ακέραιο (ενέργεια 1 του σχήματος 2.1), ο οποίος αποθηκεύεται (ενέργεια 2). Στην περίπτωση ανάκλησης της τιμής, εκτελείται η αντίστροφη διεργασία. Ο αριθμός μετατρέπεται σε χαρακτήρα μέσω του προσδιοριστή **%c** και ακολούθως είτε τυπώνεται ο χαρακτήρας (ενέργεια 3) είτε τυπώνεται το δεκαδικό ισοδύναμο μέσω του προσδιοριστή **%d** (ενέργεια 4). Σε κάθε περίπτωση, ο μεταγλωττιστής είναι υπεύθυνος για το ότι ο υπολογιστής διαχειρίζεται τα bits και bytes σύμφωνα με τους τύπους που δηλώνονται.



Σχ.2.1 *Αποθήκευση και ανάκληση ASCII χαρακτήρα*

2.2.2 Μη εκτυπούμενοι χαρακτήρες

Οι σταθερές τύπου χαρακτήρα ***«νέα γραμμή (new-line)»*** και ***«στηλοθέτης (tab)»*** ανήκουν στην κατηγορία των μη εκτυπούμενων χαρακτήρων, τους οποίους η C αναπαριστά με τις ***«ακολουθίες διαφυγής (escape sequences)»*** **\n** και **\t**, αντίστοιχα. Η παρακάτω πρόταση δίνεται ως παράδειγμα χρήσης χαρακτήρων διαφυγής:

```
printf( "Write, \" a \\ is a backslash. \"\n" );
```

Η πρόταση θα εμφανίσει στην κύρια έξοδο (οθόνη):

```
Write, " a \ is a backslash. "
```

Οι μη εκτυπούμενοι χαρακτήρες και οι αντίστοιχες ακολουθίες διαφυγής παρατίθενται στον πίνακα 2.1:

Χαρακτήρας	Ακολουθία	Χαρακτήρας	Ακολουθία
συναγερμός (κουδούνι)	**\a**	πλάγια γραμμή	****
οπισθοχώρηση	**\b**	λατινικό ερωτηματικό	**\?**
αλλαγή σελίδας	**\f**	μονό εισαγωγικό	**\'**
νέα γραμμή	**\n**	διπλό εισαγωγικό	**\"**
επαναφορά κεφαλής	**\r**	οκταδικός αριθμός	**\ooo**
οριζόντιος στηλοθέτης	**\t**	δεκαεξαδικός αριθμός	**\xhhh**
κατακόρυφος στηλοθέτης	**\v**		

Πίνακας 2.1 *Μη εκτυπούμενοι χαρακτήρες και αντίστοιχες ακολουθίες διαφυγής*

Παράδειγμα 2.1

Να καταστρωθεί πρόγραμμα που να επιτελεί τα παρακάτω:

Ζήτησε από το χρήστη ένα χαρακτήρα

Πάρε από το χρήστη το χαρακτήρα

Τύπωσε το χαρακτήρα και τον ASCII κωδικό του

Βρες τον επόμενο χαρακτήρα

Τύπωσέ τον μαζί με τον κωδικό του

```
#include <stdio.h>
void main()
{
      char ch,next_ch;
      printf( "Write a character:\t" );
      scanf( "%c",&ch );
      printf( "The ASCII code of char %c is %d\n", ch, ch );
      next_ch=ch+1;      /* βρίσκει τον επόμενο χαρακτήρα */
      printf( "The ASCII code of char %c is %d\n", next_ch, next_ch );
}
```

2.2.3 Ο τύπος του ακεραίου

Ο τύπος του ακεραίου (**int** από τη λέξη integer) χρησιμοποιείται για να παραστήσει ακέραιους αριθμούς. Η περιοχή τιμών εξαρτάται από την αρχιτεκτονική του μηχανήματος. Για έναν υπολογιστή που διαθέτει 2 bytes (16 bits) για κάθε ακέραιο, το σύνολο των δυνατών τιμών είναι $2^{16} = 65536$. Εάν θεωρηθεί ότι τις μισές τιμές θα καταλάβουν αρνητικοί ακέραιοι και τις υπόλοιπες μισές θετικοί, η περιοχή τιμών του τύπου **int** είναι από -32768 έως +32767, με την ακόλουθη αντιστοιχία δυαδικής και δεκαδικής τιμής:

$$(0000000000000000)_2 \quad \rightarrow \quad (-32768)_{10}$$

$$(0111111111111111)_2 \quad \rightarrow \quad (-1)_{10}$$

$$(1000000000000000)_2 \rightarrow (0)_{10}$$

$$(1111111111111111)_2 \rightarrow (+32767)_{10}$$

- Η δήλωση της μεταβλητής ακεραίου ακολουθεί τον εξής φορμαλισμό:

int όνομα_μεταβλητής; π.χ. **int num**;

Υπάρχει η δυνατότητα ταυτόχρονα με τη δήλωση να αποδοθεί αρχική τιμή στη μεταβλητή:

int num=46;

- Εάν πριν από τη λέξη **int** τοποθετηθεί ο προσδιοριστής **long** τότε οι ακέραιοι **long int** εξασφαλίζουν αποθηκευτικό χώρο 32 bits. Αντίστοιχα, ο προσδιοριστής **unsigned** χρησιμοποιείται πριν από τη λέξη **int** για να χαρακτηρίσει τη μεταβλητή χωρίς πρόσημο, η οποία λαμβάνει τιμές από 0 έως 65535 για ακέραιο 16 bits.

Σε περιβάλλοντα 4 bytes (32 bits) όπως τα Windows XP, το σύνολο των δυνατών τιμών είναι $2^{32} = 4.294.967.296$. Έτσι οι προσημασμένοι ακέραιοι αποθηκεύουν τιμές στο διάστημα από -2.147.483.648 έως +2.147.483.647.

Ένας ακέραιος **short int** είναι τουλάχιστον 16 bits και ο **int** είναι τουλάχιστον τόσο μεγάλος όσο ο **short int** (συνήθως είναι 32 bits).

- Η εισαγωγή τιμών στις ακέραιες μεταβλητές γίνεται με χρήση της συνάρτησης ***scanf*** και του προσδιοριστή **%d**. Η πρόταση

scanf("%d", &num);

διαβάζει από το πληκτρολόγιο έναν ακέραιο και τον αποδίδει στη μεταβλητή **num**.

- Η εκτύπωση ακέραιων μεταβλητών γίνεται με χρήση της συνάρτησης ***printf*** και των προσδιοριστών **%d**, **%o**, **%x** για την εμφάνιση σε δεκαδική, οκταδική και δεκαεξαδική μορφή, αντίστοιχα. Οι προσδιοριστές **l** (long), **h** (short), και **u** (unsigned) τοποθετούνται πριν από τους **d**, **o**, **x**. Η πρόταση

printf("dec=%d, octal=%o, hex=%x", num,num,num);

θα τυπώσει (θεωρώντας ότι η **num** λαμβάνει την τιμή **46**):

dec=46, octal=56, hex=2e

Παρατηρήσεις:

1. Υπάρχει η δυνατότητα να καθορισθεί ο αριθμός των ψηφίων που θα τυπωθούν,

τοποθετώντας τον επιθυμητό αριθμό ανάμεσα στο **%** και το **d**. Εάν ο αριθμός είναι μικρότερος από τον απαιτούμενο αριθμό ψηφίων του ακέραιου, η επιλογή δε θα ληφθεί υπόψη. Στην αντίθετη περίπτωση, στις πλεονάζουσες θέσεις θα τοποθετηθούν κενά. Για το λόγο αυτό, ο αριθμός που τοποθετείται στον προσδιοριστή **%d** ονομάζεται **καθοριστικό ελάχιστου πλάτους πεδίου**. Με αυτόν τον τρόπο, σε διαδοχικές ***printf*** θα υπάρξει ευθυγράμμιση των αποτελεσμάτων κατά στήλες (βλ. Παράδειγμα 5.9). Οι προτάσεις

printf("dec=%1d, octal=%4o, hex=%4x", num,num,num);

printf("dec=%4d, octal=%4o, hex=%4", num,num,num);

θα τυπώσουν αντίστοιχα

dec=46, octal= 56, hex= 2e

dec= 46, octal= 56, hex= 2e

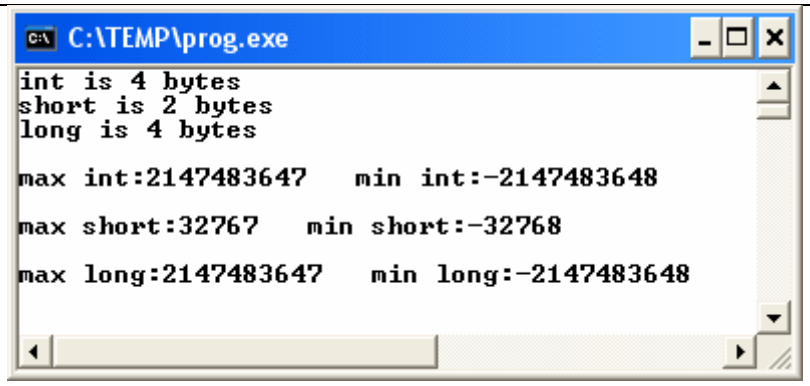
2. Όταν γράφεται ένας αριθμός στον πηγαίο κώδικα χωρίς δεκαδικό ή εκθετικό μέρος, ο μεταγλωττιστής το χειρίζεται ως **ακέραια σταθερά**. Η σταθερά **245** αποθηκεύεται ως **int**, ενώ η σταθερά **100000** αποθηκεύεται ως **long int**. Εάν ορισθεί η σταθερά **8965** ως **8965L**, ο μεταγλωττιστής αναγκάζεται να δεσμεύσει χώρο για **long int**.

Παράδειγμα 2.2

Να καταστρωθεί πρόγραμμα που να εξετάζει το μήκος του τύπου ακεραίου.

```
#include <stdio.h>
#include <climits.h>    // limits.h για παλαιότερα συστήματα
void main()
{
    int number_int=INT_MAX;  // Μέγιστος int: ορίζεται στο climits.h
    short int number_short=SHRT_MAX;  // Μέγιστος short int
    long int number_long=LONG_MAX;  // Μέγιστος long integer
    /* ο τελεστής sizeof δίνει το μέγεθος ενός τύπου δεδομένου
    ή μίας μεταβλητής */
    printf( "int is %d bytes\n",sizeof(int) );
```

```
        printf( "short is %d bytes\n",sizeof(short) );
        printf( "long is %d bytes\n",sizeof(long) );
        printf( "\nmax int:%d  min int:%d\n",number_int,INT_MIN );
        printf( "\nmax short:%d min short:%d\n",SHRT_MAX,SHRT_MIN );
        printf( "\nmax long:%d min long:%d\n",number_long,LONG_MIN );
    }
```



Από τα αποτελέσματα προκύπτει ότι ταυτίζονται τα bytes για **int** και **long**, γιατί στην έκδοση 5 της Borland C++ και με λειτουργικά συστήματα Windows 95 και μεταγενέστερα ο **int** καταλαμβάνει 4 bytes.

2.2.4 Τύποι πραγματικών αριθμών

Οι πραγματικοί αριθμοί είναι οι αριθμοί που διαθέτουν κλασματικό μέρος και εκφράζονται συνήθως στις ακόλουθες μορφές:

Αριθμός με δεκαδικά	*Επιστημονική σημειογραφία*	*Εκθετική σημειογραφία*
123.456	1.23456x102	1.23456e+02
0.00002	2.0x10-5	2.0e-5
50000.0	2.0x104	5.0e+04

Η γλώσσα C διαθέτει δύο τύπους για αναπαράσταση πραγματικών αριθμών. Τον τύπο **float** για αριθμούς κινητής υποδιαστολής απλής ακρίβειας και τον τύπο **double** για αριθμούς κινητής υποδιαστολής διπλής ακρίβειας.

- Η δήλωση της μεταβλητής float ή double ακολουθεί τον εξής φορμαλισμό:

float όνομα_μεταβλητής; π.χ. **float plank=6.63e-34**;

Η χρήση του προσδιοριστή **long** πριν από τον τύπο **double** χρησιμοποιείται για δήλωση μεταβλητής κινητής υποδιαστολής εκτεταμένης ακρίβειας, π.χ.

long double plank;

- Η εισαγωγή τιμών στις μεταβλητές κινητής υποδιαστολής γίνεται με χρήση της συνάρτησης ***scanf*** και του προσδιοριστή **%f** (float). Η πρόταση

scanf("%f", &num);

διαβάζει από το πληκτρολόγιο έναν πραγματικό αριθμό και τον αποδίδει στη μεταβλητή **num**.

- Η εκτύπωση πραγματικών μεταβλητών γίνεται με χρήση της συνάρτησης ***printf*** και των προσδιοριστών **%f** για εμφάνιση σε δεκαδική μορφή, **%e** για εμφάνιση σε εκθετική μορφή, και **%g** για να ανατεθεί στο σύστημα να επιλέξει μεταξύ των δύο προηγουμένων, με προτεραιότητα στη μορφή με το μικρότερο μέγεθος.

- Σε ό,τι αφορά το χώρο που καταλαμβάνουν στη μνήμη, ως συνηθισμένα μεγέθη αναφέρονται για τους μεν **float** τα 32 bits, για τους δε **double** τα 64 bits. Θα πρέπει να σημειωθεί ότι, σε αντιδιαστολή με τους ακέραιους αριθμούς, δεν υπάρχει αντιστοιχία ένα προς ένα ανάμεσα στους πραγματικούς αριθμούς και στις απεικονίσεις τους στις γλώσσες προγραμματισμού. Οι αριθμοί που αντιστοιχούν στους πραγματικούς αριθμούς είναι προσεγγίσεις αυτών και ονομάζονται **αριθμοί μηχανής**, εξαιτίας της ανάγκης να απεικονισθούν με πεπερασμένο αριθμό ψηφίων πραγματικοί αριθμοί που θεωρητικά μπορούν να περιέχουν άπειρο αριθμό κλασματικών ψηφίων. Σε μία μεταβλητή τύπου **float** των 32 bits τα 8 bits χρησιμοποιούνται για τον εκθέτη, ένα για το πρόσημο και τα υπόλοιπα 23 για το κλασματικό μέρος. Η μορφή του αριθμού είναι η ακόλουθη:

$$\pm(\ .\ d_1 d_2 ... d_{23}\ ) \cdot 2^e$$

όπου τα ψηφία $d_1 ... d_{23}$ είναι δυαδικά και το e είναι το δεκαδικό ισοδύναμο του οκταψήφιου δυαδικού εκθέτη. Κατά σύμβαση το $d_1 = 1$. Το δεκαδικό ισοδύναμο της ανωτέρω μορφής είναι:

$$\pm\left[(d_1 \cdot 2^{-1}) + (d_2 \cdot 2^{-2}) + ... + (d_{23} \cdot 2^{-23}) \right] \cdot 2^e = \pm 2^e \cdot \sum_{i=1}^{23} d_i \cdot 2^{-i}$$

Κατά συνέπεια, με βάση το παραπάνω σύστημα απεικόνισης το μέγεθος της κλασματικής

ακρίβειας ενός αριθμού κινητής υποδιαστολής καθορίζεται από τον αριθμό των κλασματικών ψηφίων.

Ο οκταψήφιος δυαδικός εκθέτης αντιστοιχεί σε $2^8 = 256$ δυνατές τιμές. Από αυτές οι 127 δίνονται σε αρνητικούς ακέραιους και οι υπόλοιπες 129 σε θετικούς, με την ακόλουθη αντιστοιχία δυαδικής και δεκαδικής τιμής:

$$e_{\min} = (00000000)_2 \qquad \rightarrow \qquad e_{\min} = (-127)_{10}$$

$$e_{\max} = (11111111)_2 \qquad \rightarrow \qquad e_{\max} = (128)_{10}$$

Με βάση τα παραπάνω, η μέγιστη απόλυτη τιμή πραγματικών αριθμών που μπορεί να επιτευχθεί με μεταβλητή τύπου **float** των 32 bits είναι (βλ. Παράδειγμα 2.3):

$$|\max| = \left[(1 \cdot 2^{-1}) + (1 \cdot 2^{-2}) + ... + (1 \cdot 2^{-23}) \right] \cdot 2^{128} = 3.402823e + 38$$

Στους αριθμούς διπλής ακρίβειας δεσμεύονται 64 bits, εκ των οποίων τα 11 δίνονται στον εκθέτη, ένα στο πρόσημο και 52 στο κλασματικό μέρος.

Παρατηρήσεις:

1. Όπως σημειώθηκε στους ακέραιους, έτσι και στους πραγματικούς υπάρχει η δυνατότητα να καθορισθεί ο αριθμός των ψηφίων που θα εκτυπωθούν, τοποθετώντας τον επιθυμητό αριθμό ανάμεσα στο **%** και το **f**. Μάλιστα ο αριθμός θα είναι της μορφής **a.b**, με το **a** να δηλώνει το συνολικό αριθμό των ψηφίων – συμπεριλαμβανομένου του προσήμου – και το **b** να δηλώνει τον αριθμό των δεκαδικών ψηφίων. Οι προτάσεις

```
float num=46.37;
printf( "num=%8.4f,  num=%12.1f\n", num,num );
```

θα τυπώσουν

num= 46.3700, num= 46.4

Είναι φανερό ότι στην περίπτωση που ο αριθμός των δεκαδικών ψηφίων που ζητούνται είναι μικρότερος από τον απαιτούμενο γίνεται στρογγυλοποίηση (το **37** στρογγυλοποιήθηκε στο **40** και παρελήφθη το **0**).

2. Πραγματικοί αριθμοί όπως οι

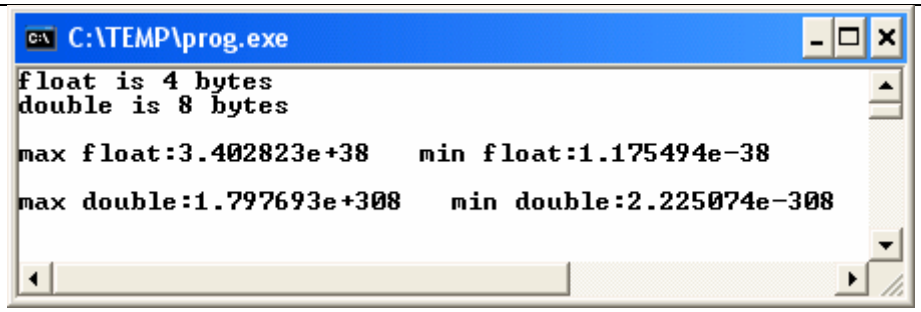
0.12 45.68 9e-5 24e09 0.0034e-08

όταν εμφανίζονται στον πηγαίο κώδικα αποτελούν τις **πραγματικές σταθερές**. Θεωρούνται από το μεταγλωττιστή ως **double** και δεσμεύουν τον αντίστοιχο χώρο.

Παράδειγμα 2.3

Να καταστρωθεί πρόγραμμα που να εξετάζει το μήκος του τύπου κινητής υποδιαστολής.

```
#include <stdio.h>
#include <float.h>      // για τα όρια του float
void main()
{
      float num_float=FLT_MAX;    // Μέγιστος float
      double num_double=DBL_MAX;  // Μέγιστος double
      /* ο τελεστής sizeof δίνει το μέγεθος ενός τύπου δεδομένου
      ή μίας μεταβλητής */
      printf( "float is %d bytes\n",sizeof(float) );
      printf( "double is %d bytes\n",sizeof(double) );
      printf( "\nmax float:%e min float:%e\n",num_float,FLT_MIN);
      printf( "\nmax double:%e,min double:%e\n" num_double,DBL_MIN);
      printf( "\n\nPress any key to continue" );
}
```



2.3 I/O κονσόλας

Η I/O κονσόλας αναφέρεται στις λειτουργίες που γίνονται στο πληκτρολόγιο και στην οθόνη του υπολογιστή. Πέραν των **printf** και **scanf**, που χρησιμοποιήθηκαν προγουμένως (και αποτελούν τη μορφοποιούμενη I/O κονσόλας γιατί μπορούν να διαβάζουν και να τυπώσουν δεδομένα σε διάφορες μορφές), υπάρχει μία σειρά

απλούστερων συναρτήσεων, που αναπτύσσεται στις επόμενες παραγράφους.

2.3.1 Οι συναρτήσεις getche, getch

Η συνάρτηση ***getche*** διαβάζει ένα χαρακτήρα από την κύρια είσοδο. Αναμένει έως ότου πατηθεί ένα πλήκτρο και στη συνέχεια επιστρέφει την τιμή του, εμφανίζοντας στην οθόνη το πλήκτρο που πατήθηκε. Το πρωτότυπο της ***getche*** είναι το ακόλουθο:

int getche(void);

και το αρχείο κεφαλίδας της συνάρτησης βρίσκεται στο ***conio.h***.

Η ***getche*** επιστρέφει μεν έναν ακέραιο αλλά το byte χαμηλής τάξης είναι αυτό που περιέχει τον χαρακτήρα. Η χρήση ακεραίων γίνεται για λόγους συμβατότητας με τον αρχικό μεταγλωττιστή της UNIX C.

Η συνάρτηση ***getch*** αποτελεί παραλλαγή της ***getche*** και βρίσκεται επίσης στο ***conio.h***. Λειτουργεί όπως ακριβώς η ***getche***, με τη διαφορά ότι η ***getch*** δεν εμφανίζει τον πληκτρολογηθέντα χαρακτήρα στην οθόνη.

2.3.2 Η συνάρτηση getchar

Η συνάρτηση ***getchar*** (get character) διαβάζει ένα χαρακτήρα από την κύρια είσοδο και τον επιστρέφει στο πρόγραμμα. Αποτελεί παραλλαγή της ***getche***. Είναι η αρχική συνάρτηση εισόδου χαρακτήρων και βασίζεται στο UNIX. Το πρόβλημα με τη συνάρτηση αυτή είναι ότι κρατά την είσοδο στην περιοχή προσωρινής αποθήκευσης μέχρι να δοθεί επαναφορά κεφαλής. Έτσι, μετά την επιστροφή της ***getchar*** περιμένουν ένας ή περισσότεροι χαρακτήρες στην ουρά εισόδου.

Το πρωτότυπο της ***getchar*** είναι το ακόλουθο:

int getchar(void);

και το αρχείο κεφαλίδας της συνάρτησης βρίσκεται στο ***stdio.h***.

2.3.3 Η συνάρτηση putchar

Η συνάρτηση ***putchar*** (put character) εμφανίζει στην οθόνη το χαρακτήρα που έχει ως όρισμα (π.χ. **c**), στην τρέχουσα θέση του δρομέα. Το πρωτότυπο της ***putchar*** είναι:

int putchar(int c);

Η ***putchar*** επιστρέφει μεν έναν ακέραιο αλλά το byte χαμηλής τάξης είναι αυτό που περιέχει το χαρακτήρα. Όπως συνέβη και με τις προηγούμενες συναρτήσεις, η χρήση

ακεραίων γίνεται για λόγους συμβατότητας με τον αρχικό μεταγλωττιστή της UNIX C. Το αρχείο κεφαλίδας της συνάρτησης ***putchar*** βρίσκεται στο ***stdio.h***.

2.3.4 Η συνάρτηση kbhit

Η συνάρτηση ***kbhit*** (keyboard hit) ελέγχει κατά πόσον ο χρήστης έχει πατήσει κάποιο πλήκτρο. Εφόσον έχει πατήσει κάποιο πλήκτρο η συνάρτηση επιστρέφει ως αληθής, σε αντίθετη περίπτωση επιστρέφει ως ψευδής. Η συνάρτηση ***kbhit*** χρησιμοποιείται κυρίως για να διακόπτει ο χρήστης το πρόγραμμα κατά το δοκούν.

Το πρωτότυπο της ***kbhit*** είναι:

int kbhit(void);

Το αρχείο κεφαλίδας της συνάρτησης ***kbhit*** βρίσκεται στο ***conio.h***. Παραδείγματα όπου γίνεται χρήση της ***kbhit***. θα παρουσιασθούν στη συνέχεια.

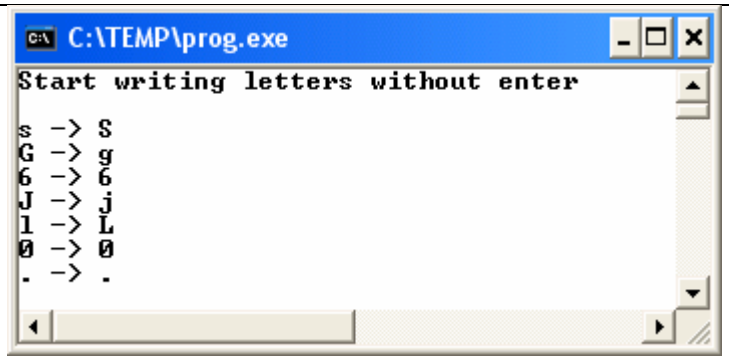
Παράδειγμα 2.4

Το ακόλουθο πρόγραμμα παίρνει χαρακτήρες από το πληκτρολόγιο και μετατρέπει τα κεφαλαία γράμματα σε μικρά και τούμπαλιν. Το πρόγραμμα τερματίζει μόλις πληκτρολογηθεί μία τελεία (**.**). Το αρχείο κεφαλίδας ***ctype.h*** απαιτείται για τη συνάρτηση ***islower***, που αληθεύει εάν το όρισμά της είναι σε μικρά γράμματα, και τις συναρτήσεις ***toupper***, ***tolower***, που μετασχηματίζουν τα γράμματα.[2]

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
void main()
{
    char ch;
    printf( "Start writing letters without enter\n\n" );
    do
    {
        ch=getche();
```

[2] Ο αναγνώστης καλείται να παρακάμψει την επαναληπτική πρόταση και την υπό συνθήκη διακλάδωση. Θα μελετηθούν εκτενώς αργότερα.

```
            printf( " -> ",ch );
            if (islower(ch)) putchar(toupper(ch));
            else putchar(tolower(ch));
            printf( "\n" );
        }
        while (ch!='.');        // τέλος προγράμματος με '.'
    }
```



Κεφάλαιο 3

ΤΕΛΕΣΤΕΣ – ΕΚΦΡΑΣΕΙΣ

3.1 Ορισμοί – σημειογραφίες

Οι τελεστές (operators) είναι σύμβολα ή λέξεις που αναπαριστούν συγκεκριμένες διεργασίες, οι οποίες εκτελούνται επί ενός ή περισσότερων δεδομένων. Τα δεδομένα καλούνται **τελεστέοι** (operands) και μπορούν να είναι μεταβλητές, σταθερές ή ακόμη και κλήσεις συναρτήσεων.

Οι τελεστές χρησιμοποιούνται για το σχηματισμό εκφράσεων. Μία έκφραση, εν γένει, αποτελείται από έναν ή περισσότερους τελεστέους και από έναν ή περισσότερους τελεστές. Κάθε έκφραση έχει μία τιμή, η οποία υπολογίζεται με ορισμένους κανόνες. Για παράδειγμα, στην έκφραση **num+12** ο χαρακτήρας + αναπαριστά τη διεργασία της πρόσθεσης των δύο τελεστέων, οι οποίοι είναι η μεταβλητή **num** και η σταθερά **12**.

Οι τελεστές ταξινομούνται, ανάλογα με τον αριθμό των τελεστέων στους οποίους δρουν, σε μοναδιαίους (unary), δυαδικούς (binary) και τριαδικούς (ternary). Μία δεύτερη κατηγοριοποίηση επιτελείται με βάση τη διεργασία που εκτελούν, οδηγώντας στις κατηγορίες του πίνακα 3.1:

Κατηγορία	Ενδεικτικοί τελεστές
Αριθμητικοί	+ - * /
Λογικοί	&& || !
Συσχετιστικοί	> >= == !=
Διαχείρισης bits	>> & ! ^
Διαχείρισης μνήμης	& [] . ->

Πίνακας 3.1 *Κατηγορίες τελεστών*

Τα σύμβολα των συνηθέστερων δυαδικών τελεστών στη C παρατίθενται στον πίνακα που ακολουθεί:

Δυαδικός τελεστής	Σύμβολο	Δυαδικός τελεστής	Σύμβολο
Μικρότερο	<	πρόσθεση	+
μικρότερο ή ίσο	<=	αφαίρεση	-
Ίσο	= =	πολλαπλασιασμός	*
Διάφορο	!=	διαίρεση πραγματικών	/
Μεγαλύτερο	>	πηλίκο διαίρεσης ακεραίων	/
Μεγαλύτερο ή ίσο	>=	υπόλοιπο διαίρεσης ακεραίων	%

Πίνακας 3.2 *Σύμβολα δυαδικών τελεστών*

Στη συνέχεια του κεφαλαίου θα μελετηθούν οι συνηθέστεροι των τελεστών ενώ για ενδελεχή μελέτη ο αναγνώστης μπορεί να ανατρέξει στην αναφορά [3].

3.1.1 Σημειογραφία

Η γλώσσα C δίνει τη δυνατότητα να επιτυγχάνονται διαφορετικές λειτουργίες στην ίδια έκφραση ανάλογα με τη θέση των τελεστών ανάμεσα στους τελεστέους. Για το λόγο αυτό έχουν αναπτυχθεί τρεις σημειογραφίες για τους δυαδικούς τελεστές:

- Η σημειογραφία ***ένθεσης*** ή ***ένθετου τελεστή*** (infix notation), όταν ο τελεστής τοποθετείται μεταξύ των τελεστέων στους οποίους ενεργεί, όπως στην έκφραση **x+y**.
- Η σημειογραφία ***πρόθεσης*** ή ***προπορευόμενου τελεστή*** (prefix notation), όταν αυτός τοποθετείται πριν από τους τελεστέους, όπως στην έκφραση **+x y**.
- Η σημειογραφία ***παρελκόμενου τελεστή*** (postfix notation), όταν ο τελεστής τοποθετείται μετά από τους τελεστέους, όπως στην έκφραση **x y+**.

Παρατηρήσεις:

1. Οι τελεστές είναι προτιμότερο να μη χρησιμοποιούνται σε μεικτούς τύπους. Για παράδειγμα, η έκφραση

out_int = my1_int + my2_int

δεν παρουσιάζει κανένα πρόβλημα, σε αντιδιαστολή με τον ακόλουθο μεικτό τύπο

out_float = my_double / my_int

2. Στη C υπάρχει διάκριση ανάμεσα στη διαίρεση ακεραίων και στη διαίρεση αριθμών κινητής υποδιαστολής. Στη διαίρεση ακεραίων το αποτέλεσμα είναι το πηλίκο, π.χ. **5/2=2**. Για να ληφθεί ως αποτέλεσμα αριθμός κινητής υποδιαστολής, τουλάχιστον ένας από τους

τελεστέους πρέπει να είναι αριθμός κινητής υποδιαστολής: **5.0/2 υπολογίζεται ως 2.5**

3.2 Κατηγορίες εκφράσεων της C – προτεραιότητα και προσεταιριστικότητα

Οι εκφράσεις τής C μπορούν να καταταγούν στις παρακάτω κατηγορίες:

- ***Σταθερές εκφράσεις***. Είναι εκφράσεις που περιέχουν μόνο σταθερές τιμές.
- ***Ακέραιες εκφράσεις και εκφράσεις κινητής υποδιαστολής***. Είναι εκφράσεις, οι οποίες μετά από όλες τις άμεσες και έμμεσες μετατροπές τύπων δίνουν αποτέλεσμα ακέραιου τύπου ή τύπου κινητής υποδιαστολής, αντίστοιχα.
- ***Εκφράσεις δείκτη***. Είναι εκφράσεις με τιμή μία διεύθυνση. Περιλαμβάνουν μεταβλητές δείκτη, τον τελεστή διεύθυνσης (**&**), αλφαριθμητικές σταθερές και ονόματα πινάκων.

Ο υπολογισμός μίας έκφρασης δεν είναι πάντοτε απλή υπόθεση, ιδιαίτερα στην περίπτωση που υπάρχουν ένθετες (nested) εκφράσεις, δηλαδή εκφράσεις που είναι φωλιασμένες μέσα σε άλλες. Στην έκφραση

(((n+5)<=a) && q)

η έκφραση **n+5** είναι φωλιασμένη στην έκφραση **(n+5)<=a**, η οποία με τη σειρά της είναι φωλιασμένη στη συνολική έκφραση. Μία άλλη περίπτωση δυσχέρειας στον υπολογισμό είναι η διαδοχική παράθεση τελεστών: **7*8-2**, η οποία μπορεί να υπολογισθεί είτε ως **(7*8)-2=54** είτε ως **7*(8-2)=42**, οδηγώντας σε διαφορετικά αποτελέσματα.

Για να αντιμετωπισθούν οι ανωτέρω δυσχέρειες έχει υιοθετηθεί μία σειρά εφαρμογής των τελεστών, η επονομαζόμενη **εφαρμοστική σειρά** (applicative order), η οποία στηρίζεται στις έννοιες της **προτεραιότητας** (precedence) και της **προσεταιριστικότητας** (associativity) των τελεστών.

Οι τελεστές ταξινομούνται σε επίπεδα προτεραιότητας, με τη σύμβαση ότι οι τελεστές υψηλότερου επιπέδου προτεραιότητας δρούν επί των τελεστέων πριν από τους τελεστές χαμηλότερου επιπέδου.

Η ύπαρξη περισσότερων τελεστών στο ίδιο επίπεδο προτεραιότητας επιβάλλει τον προσδιορισμό της **κατεύθυνσης εφαρμογής**, με την κατεύθυνση από τα αριστερά προς τα δεξιά να είναι ευρύτερα χρησιμοποιούμενη. Ένας τελεστής καλείται **αριστερής προσεταιριστικότητας** (left associative), όταν σε εκφράσεις που περιέχουν πολλά στιγμιότυπα του τελεστή η ομαδοποίηση γίνεται από τα αριστερά προς τα δεξιά. Έτσι, η έκφραση **10-8-2** υπολογίζεται ως **(10-8)-2**. Οι τελεστές +,-,*,/ είναι όλοι αριστερής

προσεταιριστικότητας.

Για τη C παράδειγμα **δεξιάς προσεταιριστικότητας** αποτελεί η ύψωση σε δύναμη και ο τελεστής ανάθεσης (=). Στην έκφραση **num1=num2=10** εφαρμόζεται πρώτα ο δεξιός τελεστής ανάθεσης, με αποτέλεσμα η **num2** να αποκτήσει την τιμή 10, και ακολούθως εφαρμόζεται ο αριστερός τελεστής ανάθεσης, έτσι ώστε η **num1** εξισώνεται με τη **num2** και αποκτά την τιμή **10**.

Η προτεραιότητα και το είδος προσεταιριστικότητας των τελεστών παρατίθενται στον πίνακα 3.3, όπου οι τελεστές έχουν τοποθετηθεί με σειρά φθίνουσας προτεραιότητας:

Τελεστές	**Είδος προσεταιριστικότητας**
() [] ->	από αριστερά προς τα δεξιά
! ~ ++ -- + - * & (τύπος) sizeof	από δεξιά προς τα αριστερά
*** / % *(αριθμητικοί τελεστές)***	από αριστερά προς τα δεξιά
+ - *(αριθμητικοί τελεστές)*	»
<< >>	»
< <= > >=	»
**== != **	»
&	»
^	»
|	»
&&	»
||	»
?:	από δεξιά προς τα αριστερά
= += -= *= %= &= ^= |= <<= >>=	»
,	από αριστερά προς τα δεξιά

Πίνακας 3.3 *Προτεραιότητα και προσεταιριστικότητα τελεστών*

Με βάση τα προαναφερθέντα, είναι προφανές ότι με τους κανόνες προτεραιότητας και προσεταιριστικότητας δεν είναι απαραίτητη η χρήση παρενθέσεων για τον προσδιορισμό του τρόπου υπολογισμού της τιμής των εκφράσεων. Ωστόσο, οι παρενθέσεις χρησιμοποιούνται για τους ακόλουθους λόγους:

- Για να προσδιορισθεί συγκεκριμένη σειρά εφαρμογής, όπως στην έκφραση **(2-3)*4**.
- Για να καταστεί μία έκφραση ευανάγνωστη, όπως στην έκφραση **2-(3*4)**, παρά το

γεγονός ότι στην τελευταία περίπτωση αποτελεί πλεονασμό.

Στην περίπτωση ένθετων παρενθέσεων ο μεταγλωττιστής εφαρμόζει πρώτα τις εσωτερικές παρενθέσεις. Για παρενθέσεις όμως που βρίσκονται στο ίδιο βάθος ένθεσης, δεν ορίζεται η σειρά υπολογισμού.

Παράδειγμα 3.1

Με χρήση του πίνακα 3.3 να υπολογισθούν οι εκφράσεις:

α) **x = 17 - 2 * 8**

β) **y = 17 - 2 - 8**

α) **x =17-(2*8) , x=1**

β) **y = (17-2)-8, y=7**

3.3 Τελεστές αύξησης και μείωσης

Ο τελεστής αύξησης (increment operator) συμβολίζεται ++. Με χρήση αυτού του τελεστή η έκφραση **num = num + 1;** γίνεται **num ++;**

Αντίστοιχα, ο τελεστής μείωσης (decrement operator) συμβολίζεται **--** και η έκφραση **num = num - 1;** γίνεται **num --;**

Παράδειγμα 3.2

Προπορευόμενοι και παρελκόμενοι τελεστές μοναδιαίας αύξησης και μείωσης: να υπολογισθούν οι τιμές των **x** και **y** στις ακόλουθες διαδοχικές εκφράσεις.

Πρόταση	τιμή x	Τιμή y
int x = 10, y = 20;	**10**	**20**
++ x;	**11**	**20**
y = --x;	**10**	**10**
y = x-- + y;	**9**	**20**

y = y – x++;	**10**	**11**

Παράδειγμα 3.3

Να προσδιορισθεί η τιμή των **x** και **z** μετά την εκτέλεση κάθε μίας από τις παρακάτω προτάσεις, θεωρώντας ότι πριν την εκτέλεση της κάθε πρότασης οι τιμές των **x** και **y** είναι το **10** και το **20** αντίστοιχα.

α) **z = ++x + y;**

β) **z = --x + y;**

γ) **z = x++ + y;**

δ) **z = x-- + y;**

Στην περίπτωση του προπορευόμενου τελεστή, το σύστημα πρώτα εκτελεί την αύξηση ή μείωση και μετά χρησιμοποιεί τη νέα τιμή της μεταβλητής στον υπολογισμό της τιμής της έκφρασης (προτάσεις *α* και *β*). Αντίθετα, στην περίπτωση του παρελκόμενου τελεστή το σύστημα πρώτα χρησιμοποιεί την τιμή της μεταβλητής για τον υπολογισμό της τιμής της έκφρασης και μετά εκτελεί την αύξηση ή μείωση της τιμής της μεταβλητής (προτάσεις *γ* και *δ*).

Πρόταση	τιμή x	τιμή z
Z = ++x + y;	**11**	**31**
Z = --x + y;	**9**	**29**
Z = x++ + y;	**11**	**30**
Z = x-- + y;	**9**	**30**

3.4 Τελεστές ανάθεσης

Οι τελεστές ανάθεσης (assignment operators) εκτελούν κάποια πράξη ανάμεσα στους τελεστέους και εκχωρούν το αποτέλεσμα σε έναν από τους τελεστέους:

- **x* = 10;** εκτελεί την πράξη του πολλαπλασιασμού μεταξύ των **x** και **10** και εκχωρεί το αποτέλεσμα στον τελεστέο **x**. Αντιστοιχεί στην πρόταση **x = x * 10;**

- **x* = y + 1;** Αντιστοιχεί στην πρόταση **x = x * (y + 1);** κι <u>όχι</u> στην πρόταση
  x = x * y + 1;

Τελεστές ανάθεσης δημιουργούν κι οι τελεστές διαχείρισης δυαδικών ψηφίων (bitwise operators). Οι τελεστές αυτοί είναι: **>>= <<= &= ^= |=** .

Οι τελεστές ανάθεσης μαζί με τους τελεστές αύξησης/μείωσης γίνονται αιτία δημιουργίας παρενεργειών (side effects), για το λόγο αυτό αναφέρονται και ως **παρενεργοί τελεστές** (side effect operators). Οι παρενέργειες αυτές έχουν ως αποτέλεσμα την απροσδιόριστη συμπεριφορά του συστήματος ως προς τον τρόπο υπολογισμού της τιμής της μεταβλητής **i** σε εκφράσεις όπως: **i = n[i++];** ή **i = ++i + 1;**

3.5 Συσχετιστικοί τελεστές

Οι συσχετιστικοί τελεστές (relational operators) συγκρίνουν δύο τελεστέους. Οι βασικοί τελεστές της κατηγορίας αυτής παρατίθενται στον πίνακα 3.4:

Τελεστής	Δράση
<	μικρότερο από
>	μεγαλύτερο από
<=	μικρότερο ή ίσον από
>=	μεγαλύτερο ή ίσον από
==	Ίσο
!=	διάφορο

Πίνακας 3.4 *Συσχετιστικοί τελεστές*

Το αποτέλεσμα της χρήσης των συσχετιστικών τελεστών είναι είτε **ΑΛΗΘΕΣ** (true) είτε **ΨΕΥΔΕΣ** (false). Για παράδειγμα, η τιμή της έκφρασης **(3 < 2)** είναι ψευδής ενώ η τιμή της έκφρασης **(2 = = 2)** είναι αληθής.

Στη C (και σε πολλές άλλες γλώσσες προγραμματισμού) η τιμή ΑΛΗΘΗΣ αντιστοιχεί στον ακέραιο **1** και η τιμή ΨΕΥΔΗΣ αντιστοιχεί στον ακέραιο **0**.

<u>Παρατήρηση:</u> Συγκρίνοντας τους αριθμητικούς με τους συσχετιστικούς τελεστές προκύπτει ότι και οι δύο χρησιμοποιούν αριθμητικές εισόδους, π.χ. **(num + 10)** και **(num < 10)**, όπου **num** είναι μία ακέραια μεταβλητή. Ωστόσο οι αριθμητικοί τελεστές μπορούν να δώσουν ως έξοδο οποιοδήποτε αριθμό (για κάθε τιμή του **num** η πρόταση **(num + 10)** δίνει μία άλλη τιμή) ενώ οι συσχετιστικές τελεστές έχουν δίτιμη έξοδο (για κάθε τιμή του

num μικρότερη του 10 η πρόταση **(num < 10)** δίνει TRUE (1) και για όλες τις άλλες τιμές του num δίνει FALSE (0)).

3.6 Λογικοί τελεστές

Οι λογικοί τελεστές δρουν επί ενός ή δύο τελεστέων και λειτουργούν με βάση τη δίτιμη άλγεβρα Boole. Τόσο οι είσοδοι όσο και οι έξοδοι μπορούν να λάβουν μόνο δύο τιμές, TRUE και FALSE. Οι τελεστές της κατηγορίας αυτής παρατίθενται στον πίνακα 3.5 ενώ στον πίνακα 3.6 περιγράφεται ο τρόπος λειτουργίας τους (πίνακας αληθείας):

Τελεστής	Δράση
&&	Λογικό AND
||	Λογικό OR
!	Λογικό NOT

Πίνακας 3.5 *Λογικοί τελεστές*

p	q	p&&q	p||q	!p
T	**T**	**T**	**T**	**F**
T	**F**	**F**	**T**	
F	**T**	**F**	**T**	**T**
F	**F**	**F**	**F**	

Πίνακας 3.6 *Πίνακας αληθείας*

Παράδειγμα 3.4

Για x = 10 και y = -8 να υπολογισθούν οι εκφράσεις:

α) **(x+5) < (12-y)**

β) **(x>5) || (y>10)**

Αντιακθιστώντας τις αριθμητικές τιμές προκύπτει:

α) **(10+5) < (12- (-8)) → 15< 20 → TRUE**

β) **(10>5) || (-8>10) → (TRUE) || (FALSE) → TRUE**

3.7 Μετατροπές τύπων

Όταν ένας τελεστής έχει τελεστέους διαφορετικών τύπων δεδομένων, αυτοί μετατρέπονται σε ενιαίο τύπο. Η μετατροπή είτε γίνεται αυτόματα από τον υπολογιστή, οπότε καλείται **υπονοούμενη** (implicit conversion), είτε άμεσα από τον προγραμματιστή, οπότε καλείται **ρητή** (explicit conversion).

3.7.1 Υπονοούμενες μετατροπές

Οι υπονοούμενες μετατροπές διευκολύνουν την εργασία του προγραμματιστή, ο οποίος όμως θα πρέπει σε κάθε περίπτωση να γνωρίζει τις συνέπειες μίας μετατροπής. Για παράδειγμα, η έκφραση

3.0 + 1/2

δε δίνει τιμή **3.5**, όπως πιθανόν να ήταν αναμενόμενο, αλλά **3.0**.

Η διαδικασία της αυτόματης μετατροπής στηρίζεται στους ακόλουθους κανόνες:

- Σε κάθε πράξη που υπάρχουν δύο τύποι δεδομένων, ο στενότερος τύπος μετατρέπεται στον ευρύτερο χωρίς να υπάρχει απώλεια πληροφορίας.
- Οι τύποι της γλώσσας ταξινομούνται ανάλογα με το μέγεθος της μνήμης που απαιτούν για αποθήκευση, όπως παρακάτω

char < int < long < float < double

Ο τύπος **unsigned** ακολουθεί τον αντίστοιχο προσημασμένο τύπο.

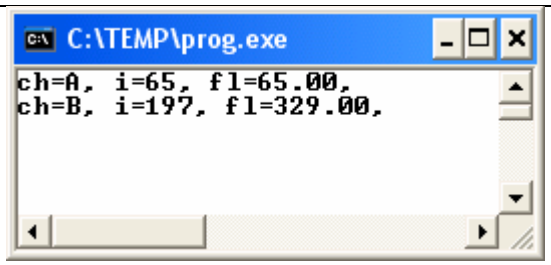
- Όλοι οι μεταγλωττιστές της C όταν υπολογίζουν αριθμητικές εκφράσεις μετατρέπουν αυτόματα τον τύπο **char** σε **int** και το **float** σε **double**.

Παράδειγμα 3.5

Να μελετηθεί το ακόλουθο πρόγραμμα:

```
#include <stdio.h>
void main()
{
    char ch;
    int i;
```

```
        float fl;
        fl=i=ch='A';  // (1)
        printf( "ch=%c, i=%d, fl=%2.2f,\n",ch,i,fl );
        ch=ch+1;  // (2)
        i=fl+2*ch;  // (3)
        fl=2.0*ch+i;  // (4)
        printf( "ch=%c, i=%d, fl=%2.2f,\n",ch,i,fl );
}
```



(1): Ο χαρακτήρας **'A'** αποθηκεύεται ως χαρακτήρας στη μεταβλητή **ch**. Η μεταβλητή **i** λαμβάνει την τιμή του ακέραιου από τη μετατροπή του **'A'** (**65**), ενώ η μεταβλητή **fl** λαμβάνει την τιμή του αριθμού κινητής υποδιαστολής που προέρχεται από το **65** (**65.00**).

(2): Η πρόσθεση της μονάδας γίνεται στην ακέραια τιμή του **'A'**. Το αποτέλεσμα (**66**) αντιστοιχεί στο χαρακτήρα **'B'**, ο οποίος αποθηκεύεται στη μεταβλητή **ch**.

(3): Η πράξη δίνει **2*66+65.00=197.00**. Το αποτέλεσμα μετατρέπεται σε **int** (**197**) και αποθηκεύεται στη μεταβλητή **i**.

(4): Η πράξη δίνει **2.0*66+197=329.00** (οι αριθμοί int μετατρέπονται σε **float**). Το αποτέλεσμα αποθηκεύεται στη μεταβλητή **fl**.

3.7.2 Ρητές μετατροπές – τελεστής typecast

Εκτός από τις αυτόματες μετατροπές η C επιτρέπει ρητές μετατροπές μίας τιμής σε ένα διαφορετικό τύπο δεδομένων. Η διαδικασία ονομάζεται **προσαρμογή** ή **εκμαγείο** (casting) και ο τελεστής μετατροπής τύπου ή ***cast*** τελεστής, όπως αποκαλείται, είναι

μοναδιαίος κι έχει τη μορφή ***(τύπος δεδομένων)***, π.χ. ***(float)***. Τοποθετείται μπροστά από μία έκφραση για να μετατρέψει την τιμή της στον περικλειόμενο σε παρενθέσεις τύπο:

```
int i,j;
float f1,f2,f3;
i=5; j=2;
fl = i/j + 0.5;                 /* Αποτέλεσμα: 2.5 */
f2 = (float)i/(float)j + 0.5;   /* Αποτέλεσμα: 3.0 */
f3 = i/j + 0.5;                 /* Αποτέλεσμα: 2.5 */
```

3.8 Τελεστής sizeof

Ο τελεστής ***sizeof*** είναι μοναδιαίος και δρα σε δύο τύπους δεδομένων:

α) σε έκφραση, π.χ. **sizeof(x+y)**

β) σε τύπο δεδομένων, πχ. **sizeof(int)**

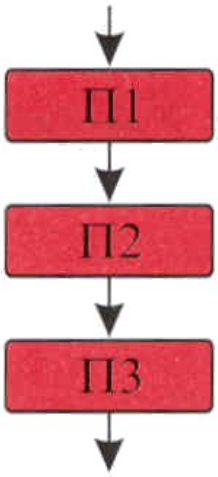
Σε κάθε περίπτωση επιστρέφει τον αριθμό των bytes που η τιμή τής έκφρασης ή ο τύπος των δεδομένων καταλαμβάνει στη μνήμη. Προσοχή θα πρέπει να δοθεί στο γεγονός ότι το σύστημα δεν υπολογίζει την τιμή της έκφρασης κι έτσι πιθανή ύπαρξη παρενεργειών από τους τελεστές δε δημιουργεί παρενέργειες στη λειτουργία της ***sizeof***.

Κεφάλαιο 4

ΕΛΕΓΧΟΣ ΡΟΗΣ – ΠΡΟΤΑΣΕΙΣ ΥΠΟ ΣΥΝΘΗΚΗ ΔΙΑΚΛΑΔΩΣΗΣ

4.1 Έλεγχος ροής

Τα προγράμματα αποτελούνται από προτάσεις, οι οποίες εκτελούνται με κάποια σειρά. Ο πιο συνηθισμένος τρόπος εκτέλεσης είναι ο **ακολουθιακός**: δύο ή περισσότερες προτάσεις βρίσκονται διατεταγμένες η μία μετά την άλλη και εκτελούνται διαδοχικά, όπως φαίνεται στο σχήμα 4.1.



Σχ. 4.1 *Ακολουθιακή εκτέλεση προτάσεων*

Ωστόσο ορισμένες φορές επιβάλλεται να γίνουν λογικές επιλογές (με χρήση λογικών τελεστών και τελεστών συσχέτισης). Εάν π.χ. περιγραφόταν η σωστή συμπεριφορά ενός πεζού μπροστά σε ένα φωτεινό σηματοδότη, θα προέκυπτε η ακόλουθη πρόταση:

ΕΑΝ στο σηματοδότη βρίσκεται ο ΓΡΗΓΟΡΗΣ

ΤΟΤΕ μπορείς να διασχίσεις την οδό

ΑΛΛΙΩΣ περίμενε αλλαγή του σηματοδότη

Για να επιτευχθεί οποιαδήποτε διαφοροποίηση από την ακολουθιακή εκτέλεση απαιτούνται ειδικές κατασκευές. Ορισμένες από αυτές τις κατασκευές διασφαλίζουν ταυτόχρονα τη δόμηση του προγράμματος, με κύριο στόχο: η δομή του πηγαίου κώδικα να μας βοηθά να κατανοήσουμε τι κάνει το πρόγραμμα. Οι κατασκευές διακρίνονται σε δύο βασικές κατηγορίες:

1) την ***υπό συνθήκη διακλάδωση*** (conditional branching)
2) την ***επανάληψη*** (looping)

Η πρώτη κατηγορία θα αποτελέσει αντικείμενο του παρόντος κεφαλαίου ενώ η δεύτερη κατηγορία θα μελετηθεί στο επόμενο κεφάλαιο.

4.2 Επιλεκτική εκτέλεση προτάσεων

Στις γλώσσες προγραμματισμού μία πρόταση διακλάδωσης περιέχει έναν αριθμό υποπροτάσεων, από τις οποίες επιλέγεται για εκτέλεση μόνο μία. Η πρόταση ***if***, που συναντάται σε πολλές γλώσσες προγραμματισμού, είναι η πλέον γνωστή πρόταση αυτής της κατηγορίας κι έχει την παρακάτω μορφή:

if E then Π1 else Π2

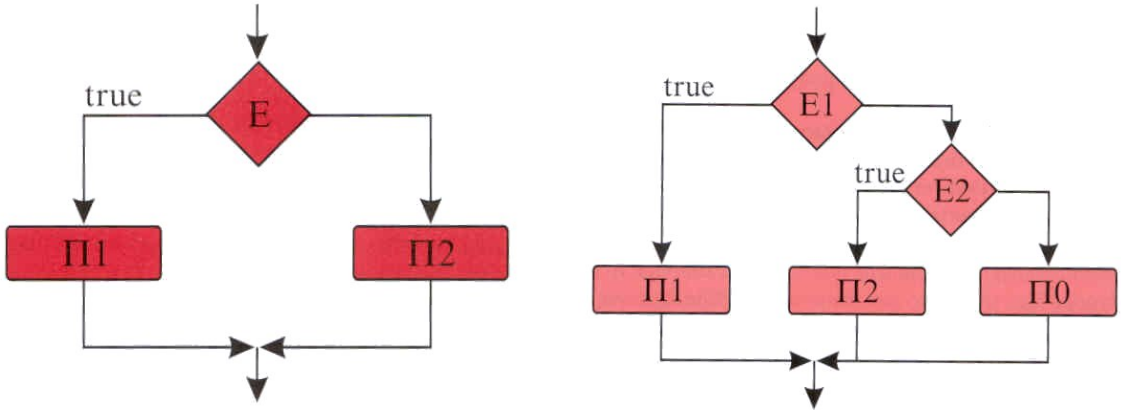
Στο σχήμα 4.2α αναπαριστάται η παραπάνω πρόταση. Είναι προφανές ότι είναι πρόταση μίας εισόδου – μίας εξόδου. Ο έλεγχος του προγράμματος εισέρχεται από το σημείο στην κορυφή, υπολογίζεται η τιμή της λογικής πρότασης **E** και, έαν είναι αληθής, επιλέγεται για εκτέλεση η πρόταση **Π1**, διαφορετικά η **Π2**. Σε κάθε περίπτωση, ο έλεγχος μεταφέρεται στο ένα και μοναδικό σημείο εξόδου στο κάτω μέρος του διαγράμματος.

Στο σχήμα 4.2β παρουσιάζεται η γενίκευση της προηγούμενης περίπτωσης, η επιλεκτική εκτέλεση πολλών προτάσεων, όπου μάλιστα υπάρχει ένθεση, δηλαδή υπάρχει διακλάδωση μέσα σε διακλάδωση. Ο φορμαλισμός για την περίπτωση αυτή είναι ο ακόλουθος:

if E1 then Π1
else if E2 then Π2
else Π0

Οι λογικές εκφράσεις υπολογίζονται σειριακά και η πρώτη που θα δώσει αληθή τιμή οδηγεί στην εκτέλεση της αντίστοιχης πρότασης. Εάν καμία από τις λογικές εκφράσεις δε δώσει αληθή τιμή, εκτελείται η **Π0**.

Η πρόταση ***if*** βασίζει την επιλογή της σε λογική έκφραση. Υπάρχουν κατασκευές στις οποίες η απόφαση επιλογής και εκτέλεσης πρότασης βασίζεται σε άλλου τύπου εκφράσεις, όπως συμβαίνει με τη ***switch*** της C, στην οποία η επιλογή γίνεται μέσα από ένα σύνολο αμοιβαία αποκλειόμενων επιλογών, όπως θα αναλυθεί σε επόμενη παράγραφο.



(α) *Δύο προτάσεις* **(β)** *Πολλές προτάσεις με ένθεση*

Σχ. 4.1 *Επιλεκτική εκτέλεση προτάσεων*

Παράδειγμα 4.1

Να περιγραφεί με ψευδοκώδικα η διεργασία που πρέπει να ακολουθήσει ο υπολογιστής για να διαπιστώσει κατά πόσο ένα δεδομένο έτος είναι δίσεκτο ή όχι. Να χρησιμοποιηθεί η κατασκευή **if – else**.

Εάν αναπαρασταθεί το έτος με την ακέραια μεταβλητή **year** και ο τελεστής υπολοίπου (modulo) με το σύμβολο **%**, η περιγραφή μπορεί να γίνει ως ακολούθως:

IF ((year % 400) == 0) THEN το έτος είναι δίσεκτο
ELSE IF ((year % 100) == 0) THEN το έτος δεν είναι δίσεκτο
ELSE IF ((year % 4) == 0) THEN το έτος είναι δίσεκτο
ELSE το έτος δεν είναι δίσεκτο

4.3 Υπό συνθήκη διακλάδωση if – else

Στη C η υπό συνθήκη διακλάδωση ***if*** έχει στη γενική περίπτωση την ακόλουθη σύνταξη:

```
if (συνθήκη)
{
    προτάσεις;
}
else
{
    προτάσεις;
}
```

Η ***if*** αποτελείται από τρία τμήματα:

- **Το τμήμα της συνθήκης**, που ακολουθεί τη λέξη ***if***.
- **Το αληθές τμήμα**, που ακολουθεί τη λέξη ***if*** και εκτελείται όταν η συνθήκη είναι αληθής.
- **Το ψευδές τμήμα** – εφόσον υπάρχει – που ακολουθεί τη λέξη ***else*** και εκτελείται όταν η συνθήκη είναι ψευδής.

Όταν τα ***if***, ***else*** ακολουθεί μία μόνο πρόταση, τα άγκιστρα περιττεύουν, ωστόσο είναι ορθή προγραμματιστική τακτική να τοποθετούνται πάντοτε, αφενός μεν για να καταστήσουν τον κώδικα ευανάγνωστο αφετέρου δε για να αποτρέψουν λάθη σε περίπτωση που προστεθούν κι άλλες προτάσεις στο αληθές ή το ψευδές τμήμα.

Παρατηρήσεις:

1. Μερικές φορές δεν υπάρχει ***else***, δηλαδή δεν υπάρχει ψευδές τμήμα:

```
if (gas_tank_empty == TRUE) fill_up_tank();
```

Εάν η συνθήκη είναι ψευδής (π.χ. το ντεπόζιτο της βενζίνης είναι άδειο) δε γίνεται καμία ενέργεια.

2. Όταν υπάρχουν **περισσότερα** από δύο τμήματα και απαιτούνται ένθετες προτάσεις **if/else**, το ζεύγος

```
else { if (συνθήκη) { προτάσεις; } }
```

μπορεί να ατικατασταθεί με την περισσότερο ευανάγνωστη μορφή:

```
else if (συνθήκη) { προτάσεις; }
```

Η ανωτέρω μορφή ονομάζεται **κλίμακα *if – else – if***. Στη γενική περίπτωση έχει την ακόλουθη σύνταξη:

```
if (συνθήκη)
{
      προτάσεις;
}
else if (συνθήκη)
{
      προτάσεις;
}
. . . . . .
else if (συνθήκη)
{
      προτάσεις;
}
else
{
      προτάσεις;
}
```

Παράδειγμα 4.2

Να ελεγχθεί κατά πόσον τα ακόλουθα τμήματα κώδικα είναι λειτουργικά ισοδύναμα.

```
int maria, petros;                          int maria, petros;
maria = 15;                                 maria = 15;
petros = maria +3;                          petros = maria +3;
if ( maria < 2 )     maria *=2;             if ( maria < 20 )    maria *=2;
else petros = 1700;                         if ( maria >= 20 ) petros = 1700;
printf( "petros = %d\n",petros );           printf( "petros = %d\n",petros );
```

Οι πρώτες πέντε γραμμές των δύο τμημάτων κώδικα είναι ίδιες, στο τέλος των οποίων η μεταβλητές **maria** και **petros** έχουν λάβει τις τιμές **30** και **18**, αντίστοιχα. Στην έκτη γραμμή υπάρχει διαφοροποίηση: στο πρώτο τμήμα κώδικα το ***else*** δε θα εκτελεσθεί,

καθώς η συνθήκη του ***if*** ήταν αληθής, και η μεταβλητή **petros** θα διατηρήσει την τιμή της (**18**). Στο δεξί τμήμα κώδικα όμως, η διακλάδωση ***if*** είναι καινούρια, η συνθήκη είναι αληθής (**maria=30>20**) και η μεταβλητή **petros** θα λάβει νέα τιμή (**1700**). Κατά συνέπεια τα δύο τμήματα κώδικα δεν είναι λειτουργικά ισοδύναμα.

4.4 Ο υποθετικός τελεστής

Ο υποθετικός τελεστής (**?:**) αποτελείται από δύο σύμβολα. Ανήκει στην κατηγορία των τελεστών που αποτελούνται από συνδυασμό συμβόλων και δεν ακολουθούν καμία από τις postfix, prefix ή infix σημειογραφίες. Όταν τα σύμβολα ή οι λέξεις του τελεστή είναι διάσπαρτα στους τελεστέους στους οποίους εφαρμόζεται ο τελεστής, λέμε ότι ο τελεστής είναι σε **μεικτή σημειογραφία** (mixfix notation).

Η έκφραση που σχηματίζει ο υποθετικός τελεστής έχει τη μορφή:

εκφρ1 ? εκφρ2 : εκφρ3

Ουσιαστικά ο υποθετικός τελεστής υλοποιεί μία υποθετική πρόταση. Η τιμή της παραπάνω έκφρασης είναι η τιμή της **εκφρ2**, εάν η **εκφρ1** είναι αληθής, αλλιώς είναι η τιμή της **εκφρ3**.

Η **εκφρ1** αποτελεί τη συνθήκη ελέγχου. Έτσι η έκφραση

x>z ? x : z

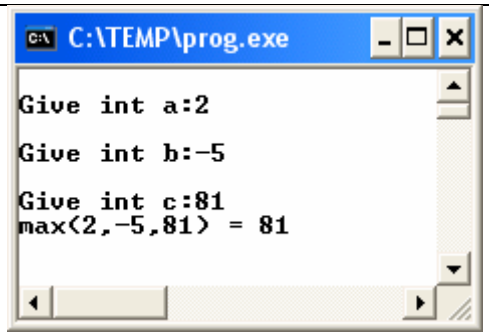
Έχει τιμή **x**, εάν το **x>z** είναι αληθές, διαφορετικά έχει τιμή **z**.

Παράδειγμα 4.3

Να γραφεί πρόγραμμα που υπολογίζει το μέγιστο ανάμεσα σε τρεις ακέραιους. Ακολούθως να τροποποιηθεί ο κορμός του προγράμματος κάνοντας χρήση του υποθετικού τελεστή.

```
#include <stdio.h>
void main()
{
    int a,b,c;
```

```
        printf( "\nGive int a:" );      scanf("%d",&a);
        printf( "\nGive int b:" );      scanf("%d",&b);
        printf( "\nGive int c:" );      scanf("%d",&c);
        if (a>b)
        {
            if (a>c) printf( "max(%d,%d,%d) = %d\n",a,b,c,a );
            else printf( "max(%d,%d,%d) = %d\n",a,b,c,c );
        }
            else if (b>c) printf( "max(%d,%d,%d) = %d\n",a,b,c,b );
            else printf( "max(%d,%d,%d) = %d\n",a,b,c,c );
    }
```



Χρησιμοποιώντας τον υποθετικό τελεστή, οι προτάσεις διακλάδωσης συνοψίζονται στην ακόλουθη πρόταση:

```
max=(a>b?a:b) > c ? (a>b?a:b):c;
printf( "max(%d,%d,%d) = %d\n",a,b,c, max );
```

4.5 Υπό συνθήκη διακλάδωση switch

Αν και η κλίμακα ***if – else – if*** μπορεί να πραγματοποιεί ελέγχους διαφόρων ειδών, είναι δύσχρηστη καθώς δεν παρέχει εποπτεία στον προγραμματιστή και καθυστερεί στην εκτέλεση. Για τις περιπτώσεις πολλαπλής διακλάδωσης η C διαθέτει την πολυκλαδική εντολή ***switch***, η οποία έχει την ακόλουθη σύνταξη:

```
    switch(έκφραση)
    {
```

```
        case (σταθ.-έκφρ. 1):
            προτάσεις;
        break;
        case (σταθ.-έκφρ. 2):
            προτάσεις;
        break;
        . . . . . .
        case (σταθ.-έκφρ. N):
            προτάσεις;
        break;
        default:
            προτάσεις;
        break;
}
```

Η πρόταση ***switch*** επιτρέπει τον προσδιορισμό απεριόριστου αριθμού διαδρομών, ανάλογα με την τιμή της έκφρασης. Υπολογίζεται η έκφραση και η τιμή της συγκρίνεται διαδοχικά με τις σταθερές εκφράσεις (***σταθ.-έκφρ. 1, σταθ.-έκφρ. 2, …***). Ο έλεγχος μεταφέρεται στις προτάσεις που είναι κάτω από τη ***σταθ.-έκφρ.*** με την οποία ισούται η τιμή της ***έκφρασης***. Εάν δεν ισούται με καμία από τις σταθερές εκφράσεις, ο έλεγχος μεταφέρεται στις προτάσεις που ακολουθούν την ετικέτα ***default***, εάν βέβαια αυτή υπάρχει, αλλιώς στην πρόταση που ακολουθεί το σώμα της ***switch***.

Η πρόταση ελέγχου ***break***, η οποία υποδηλώνει άμεση έξοδο από τη ***switch***, είναι προαιρετική. Εάν αυτή λείπει, μετά την εκτέλεση των προτάσεων που ακολουθούν την επιλεγείσα ετικέτα ***case*** θα ακολουθήσει η εκτέλεση των προτάσεων και των επόμενων ***case*** ετικετών. Βέβαια στην πράξη η ***break*** συναντάται σχεδόν πάντοτε, ακόμη και μετά τις προτάσεις της ετικέτας ***default***. Το τελευταίο γίνεται για να προστατευθούμε από το δύσκολο στην ανεύρεση σφάλμα που θα προκύψει εάν προστεθεί μελλονικά μία νέα ***case*** και ταυτόχρονα παραληφθεί να προστεθεί πριν από αυτή η ***break***.

Θα πρέπει να σημειωθεί ότι η ***switch*** διαφέρει από την ***if*** στο ότι ελέγχει μόνο την ισότητα, ενώ η παράσταση με συνθήκη της ***if*** μπορεί να είναι οιουδήποτε τύπου.

Η λειτουργία της ***switch*** διέπεται από το ακόλουθο σύνολο κανόνων:

- Κάθε ***case*** πρέπει να έχει μία ***int*** ή ***char*** σταθερά έκφραση.

- Δύο ***case*** δεν μπορούν να έχουν την ίδια τιμή.
- Οι προτάσεις κάτω από την ετικέτα ***default*** εκτελούνται όταν δεν ικανοποιείται καμία από τις ***case*** ετικέτες.
- Η ***default*** δεν είναι απαραίτητα η τελευταία ετικέτα.

Παράδειγμα 4.4

Να γραφεί πρόγραμμα, το οποίο να δίνει τη δυνατότητα στο χρήστη να εισάγει δύο αριθμούς και στη συνέχεια να εκτελεί επί αυτών επιλεκτικά μία από τις τέσσερις αριθμητικές πράξεις.

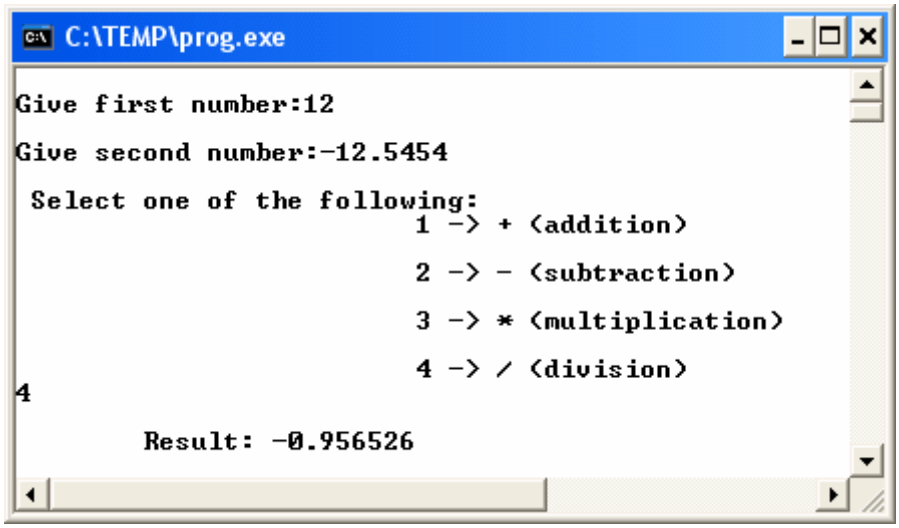
Χρησιμοποιώντας δομημένα Ελληνικά η διεργασία περιγράφεται ως εξής:

Πάρε δύο αριθμούς
Ενημέρωσε το χρήστη για δυνατές επιλογές
Πάρε την επιλογή του χρήστη
Ανάλογα με την επιλογή
    Εκτέλεσε την αντίστοιχη πράξη
Εμφάνισε το αποτέλεσμα
Τερμάτισε

Ο κώδικας του προγράμματος είναι ο ακόλουθος:

```
#include <stdio.h>
#include <conio.h>
#define ADD 1
#define SUB 2
#define MUL 3
#define DIV 4
void main()  {
    float num1, num2,result;
    int choice;
    printf( "\nGive first number:"); scanf("%f",&num1 );
    printf( "\nGive second number:"); scanf("%f",&num2 );
    printf( "\n Select one of the following:" );
```

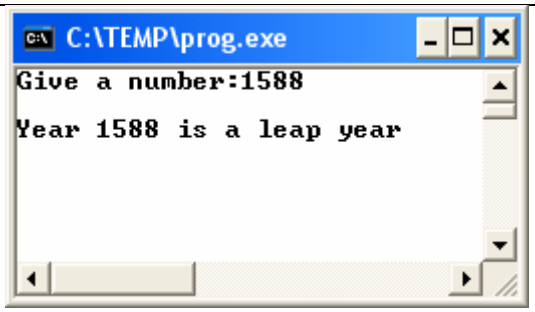
```
        printf( "\n\t\t\t 1 -> + (addition)\n" );
        printf( "\n\t\t\t 2 -> - (subtraction)\n" );
        printf( "\n\t\t\t 3 -> * (multiplication)\n" );
        printf( "\n\t\t\t 4 -> / (division)\n" );
        scanf( "%d",&choice );
        switch(choice)
        {
            case 1:
                result=num1+num2;
            break;
            case 2:
                result=num1-num2;
            break;
            case 3:
                result=num1*num2;
            break;
            case 4:
                if (num2) result=num1/num2; // num2 != 0
                else printf( "\t\t ERROR: division by 0" );
            break;
            default:
            printf( "This selection is not supported" );
            break;
        } // Τέλος της switch
        printf( "\n\tResult: %f\n",result );
    } // Τέλος της main
```



Παράδειγμα 4.5

Να γραφεί πρόγραμμα που να υλοποιεί το παράδειγμα 4.1.

```
#include <stdio.h>
void main()  {
      int year;
      printf( "Give a number:" );
      scanf( "%d",&year );
      if ( ((year%400)==0) || ( !((year%100)==0) && ((year%4)==0) ) )
            printf( "\nYear %d is a leap year",year );
      else printf( "\nYear %d is not a leap year",year );
}
```



Κεφάλαιο 5

ΠΡΟΤΑΣΕΙΣ ΕΠΑΝΑΛΗΨΗΣ - ΒΡΟΧΟΙ

5.1 Γενικά

Οι προτάσεις επανάληψης αποτελούν ένα ισχυρό εργαλείο προγραμματισμού καθώς μπορούν να κωδικοποιήσουν και να συμπυκνώσουν επαναλαμβανόμενες λειτουργίες, δημιουργώντας ένα βρόχο (loop). Ορίζονται ως προτάσεις που **επαναλαμβάνουν ένα μπλοκ εντολών είτε για όσες φορές το επιθυμούμε είτε έως ότου πληρωθεί μία συνθήκη τερματισμού**. Η πλήρωση του κριτηρίου τερματισμού οδηγεί στην περάτωση του βρόχου.

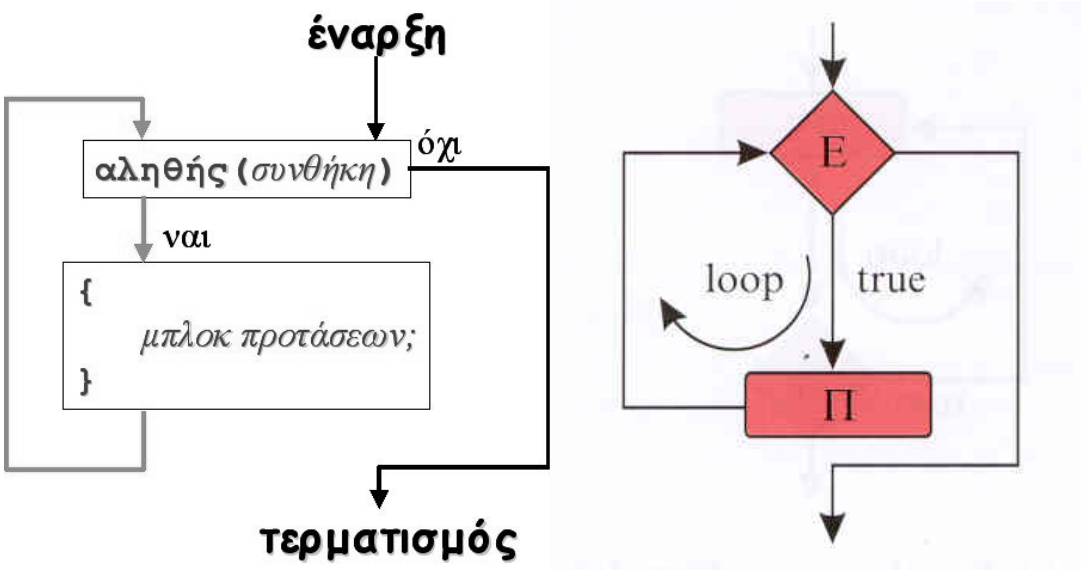
Εάν δεν υπάρχει συγκεκριμένος αριθμός επαναλήψεων ή συνθήκη τερματισμού, ο βρόχος θα εκτελείται αενάως και θα καλείται **ατέρμων βρόχος** (infinite loop), γεγονός που οδηγεί ως επί το πλείστον σε σφάλμα.

Εάν ο βρόχος τελειώνει μετά το πέρας ενός ορισμένου αριθμού επαναλήψεων τότε καλείται **βρόχος οδηγούμενος από μετρητή**. Εάν περατώνεται με την πλήρωση ενός κριτηρίου τερματισμού ονομάζεται **βρόχος οδηγούμενος από γεγονός**. Επιπρόσθετα, στις περισσότερες γλώσσες προγραμματισμού υπάρχει μία δεύτερη κατηγοριοποίηση των προτάσεων επανάληψης: *α)* εκείνες στις οποίες ο έλεγχος του κριτηρίου τερματισμού γίνεται στην αρχή του βρόχου, επονομαζόμενες **βρόχοι με συνθήκη εισόδου** (pre-test loops), και *β)* εκείνες στις οποίες ο έλεγχος του κριτηρίου τερματισμού γίνεται στο τέλος του βρόχου, επονομαζόμενες **βρόχοι με συνθήκη εξόδου** (post-test loops).

Στις παραγράφους που ακολουθούν πρώτα θα παρουσιασθούν οι μορφές βρόχων που κυριαρχούν στις γλώσσες προγραμματισμού και στη συνέχεια θα εστιασθούμε στις επαναληπτικές προτάσεις που χρησιμοποιούνται στη C.

5.1.1 Βρόχος while-do

Ο βρόχος ***while-do*** είναι βρόχος με συνθήκη εισόδου, δυνάμενος να οδηγείται τόσο από μετρητή όσο και από γεγονός. Η λειτουργία του απεικονίζεται στο μπλοκ διάγραμμα του σχήματος 5.1α ενώ το διάγραμμα ροής παρατίθεται στο σχήμα 5.1β.



(α) *Μπλοκ διάγραμμα* **(β)** *Διάγραμμα ροής*

Σχ. 5.1 *Βρόχος while-do*

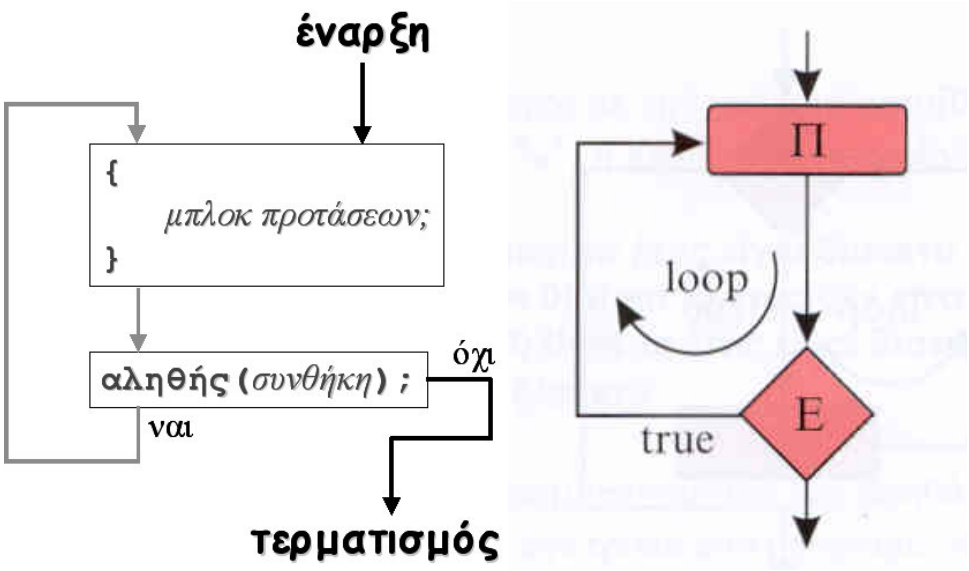
Όπως προκύπτει από το σχήμα 5.1, ο βρόχος θα εκτελείται – δηλαδή η πρόταση Π – για όσες επαναλήψεις η συνθήκη Ε είναι αληθής. Κατά συνέπεια απαιτείται: *α)* πριν την πρώτη επανάληψη η συνθήκη Ε να είναι αληθής και *β)* κατά τη διάρκεια εκτέλεσης του βρόχου να υπάρχει η δυνατότητα, μέσω της Π, να μπορεί να καταστεί η Ε ψευδής για να τερματισθεί ο βρόχος.

5.1.2 Βρόχος do-while

Ο βρόχος ***do-while*** είναι βρόχος με συνθήκη εξόδου, δυνάμενος κι αυτός να οδηγείται τόσο από μετρητή όσο και από γεγονός. Στα σχήματα 5.2α και 5.2β παρουσιάζονται το μπλοκ διάγραμμα και το διάγραμμα ροής αντίστοιχα.

Από το σχήμα 5.2 καθίσταται φανερό ότι ο βρόχος ***do-while*** διαφέρει από το βρόχο ***while-do*** στο σημείο ελέγχου της συνθήκης τερματισμού. Ο βρόχος ***do-while*** επιτρέπει την εκτέλεση της πρώτης επανάληψης πριν προχωρήσει στον έλεγχο της συνθήκης, γεγονός που σημαίνει ότι δεν απαιτείται να είναι αληθής η συνθήκη πριν από την εκτέλεση του βρόχου. Μπορεί να γίνει αληθής μέσα στην Π και, φυσικά, πρέπει να

υπάρχει η δυνατότητα, μέσω της Π, να μπορεί να καταστεί η Ε ψευδής για να τερματισθεί ο βρόχος.



(α) *Μπλοκ διάγραμμα* **(β)** *Διάγραμμα ροής*

Σχ. 5.2 *Βρόχος do-while*

5.2 Βρόχοι με συνθήκη εισόδου στη C

5.2.1 Βρόχος while

Ο βρόχος ***while*** είναι βρόχος με συνθήκη εισόδου, οδηγούμενος από γεγονός. Η λειτουργία του περιγράφεται εποπτικά από το σχήμα 5.1α και η σύνταξή του είναι η ακόλουθη:

```
while (συνθήκη)
{
    προτάσεις, μέσα στις οποίες θα αλλάζει η συνθήκη;
}
```

Η λειτουργία της πρότασης επανάληψης ***while*** μπορεί να μορφοποιηθεί σε δομημένα Ελληνικά ως εξής:

Έλεγξε τη συνθήκη.
Εάν είναι αληθής
    Προχώρησε στις προτάσεις
    Ξεκίνησε από την αρχή

Αλλιώς σταμάτησε

Ο βρόχος ***while*** είναι κατάλληλος στις περιπτώσεις που δεν είναι γνωστός εκ των προτέρων ο αριθμός των επαναλήψεων. Εκτελείται καθόσον η συνθήκη παραμένει αληθής. Όταν η συνθήκη καταστεί ψευδής, ο έλεγχος του προγράμματος παρακάμπτει το περιεχόμενο του βρόχου και προχωρά στην επόμενη εντολή. Επιπρόσθετα, για το βρόχο while ισχύουν οι παρατηρήσεις της §5.1.1.

Θα πρέπει να σημειωθεί ότι εάν το σώμα του βρόχου αποτελείται από μία πρόταση, δεν απαιτούνται {}.

Παράδειγμα 5.1

Να περιγραφεί η λειτουργία του ακόλουθου τμήματος κώδικα και να δοθεί το διάγραμμα ροής της πρότασης επανάληψης.

```
int count=30;
int limit=40;
while (count<limit)
{
  count++;
  printf( "count is %d\n",count );
}
<επόμενη πρόταση>;
```

Στις πρώτες δύο γραμμές δηλώνονται και αρχικοποιούνται οι ακέραιες μεταβλητές **count** και **limit**. Ακολουθεί η πρόταση επανάληψης ***while***, η οποία καθορίζει ως συνθήκη η μεταβλητή **count** να είναι μικρότερη της **limit**, γεγονός που αληθεύει. Κατά συνέπεια ο έλεγχος εισέρχεται στο βρόχο, η μεταβλητή **count** αυξάνεται κατά μία μονάδα και τυπώνεται στην οθόνη η φράση

count is 31

Ο βρόχος εκτελείται συνολικά 10 φορές:

count is 31

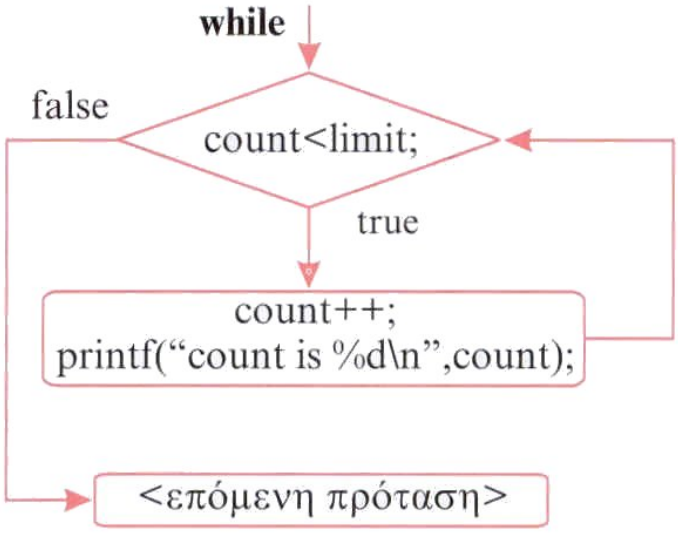
count is 32

. . . . . .

count is 40

Στην ενδέκατη επανάληψη η **count** έχει λάβει την τιμή 40 και είναι ίση με τη **limit**, οπότε η συνθήκη καθίσταται ψευδής και ο έλεγχος προσπερνά το βρόχο και προχωρά στην επόμενη πρόταση. Θα πρέπει να σημειωθεί ότι εάν η συνθήκη ήταν εξαρχής ψευδής (π.χ. **count=45**), ο βρόχος δε θα εκτελείτο ούτε μία φορά.

Το διάγραμμα ροής απεικονίζεται στο σχήμα 5.3:



Σχ. 5.3 *Διάγραμμα ροής της πρότασης επανάληψης*

Παράδειγμα 5.2

Δίνονται οι παρακάτω δύο προτάσεις:

α) **while (++count<12) Π1**

β) **while (count++<12) Π1**

Να περιγραφεί ο τρόπος με τον οποίο ο υπολογιστής τις εκτελεί, εντοπίζοντας τη διαφορά τους, εάν υπάρχει.

Υπάρχει διαφορά μεταξύ των προτάσεων κι αυτή εντοπίζεται στον αριθμό επαναλήψεων. Η *(α)* χρησιμοποιεί την προθεματική σημειογραφία ενώ η *(β)* τη μεταθεματική. Στην πρόταση *(α)* πρώτα αυξάνεται η τιμή τής **count** και η νέα τιμή της συγκρίνεται με το **12**, ενώ στη *(β)* πρώτα συγκρίνεται η τιμή τής **count** με το **12** και στη

συνέχεια αυξάνεται η τιμή της. Αυτό σημαίνει πως η πρόταση Π1 θα εκτελεσθεί μία φορά παραπάνω στην περίπτωση *(β)*.

5.2.2 Βρόχος for

Ο βρόχος ***for*** είναι βρόχος με συνθήκη εισόδου, οδηγούμενος από μετρητή. Η λειτουργία του περιγράφεται εποπτικά από το σχήμα 5.1α και η σύνταξή του είναι η ακόλουθη:

```
for (αρχική τιμή μετρητή; συνθήκη; βήμα μετρητή)
{
    προτάσεις;
}
```

Η λειτουργία της πρότασης επανάληψης ***for*** μπορεί να μορφοποιηθεί σε δομημένα Ελληνικά ως εξής:

Αρχικοποίησε το μετρητή
Έλεγξε τη συνθήκη
Εάν είναι αληθής
    Εκτέλεσε τις προτάσεις
    Ενημέρωσε το μετρητή
    Επάνελθε στον έλεγχο της συνθήκης
Αλλιώς ενημέρωσε το μετρητή και σταμάτησε

Μία τυπική εκτέλεση του βρόχου ***for*** είναι η ακόλουθη, κατά την οποία σε κάθε επανάληψη θα τυπώνεται η μεταβλητή **n**:

```
for (n=0; n<10; n++) printf( "n=%d\n",n );
```

Ωστόσο ο βρόχος ***for*** στη γλώσσα C παρέχει μεγάλη ευελιξία καθώς οι εκφράσεις μέσα στις παρενθέσεις μπορούν να έχουν πολλές παραλλαγές. Ενδεικτικά αναφέρονται μερικές από τις παραλλαγές ενώ για ενδελεχή μελέτη ο αναγνώστης μπορεί να ανατρέξει στην αναφορά [18].

- Μπορεί να χρησιμοποιηθεί ο τελεστής μείωσης για μέτρηση προς τα κάτω:

```
for (n=10; n>0; n- -) printf( "n=%d\n",n );
```

- Το βήμα καθορίζεται από το χρήστη:

for (n=0; n<60; n=n+13) printf("n=%d\n",n);

- Χρησιμοποιώντας την ιδιότητα ότι κάθε χαρακτήρας του κώδικα ASCII έχει μία ακέραια τιμή, ο μετρητής μπορεί να είναι μεταβλητή χαρακτήρα. Το παρακάτω τμήμα κώδικα θα τυπώνει τους χαρακτήρες από το '**a**' έως το '**z**' μαζί με τον ASCII κωδικό τους:

  for (n='a'; n<'z'; n++) printf("n=%d, the ASCII value is %d\n",n,n);

- Ο μετρητής μπορεί να αυξάνει κατά γεωμετρική πρόοδο:

  for (n=0; n<60.0; n=1.2*n) printf("n=%f\n",n);

Θα πρέπει να σημειωθεί ότι εάν το σώμα του βρόχου αποτελείται από μία πρόταση, δεν απαιτούνται **{}**.

Παράδειγμα 5.3

Να περιγραφεί η λειτουργία του ακόλουθου τμήματος κώδικα και να δοθεί το διάγραμμα ροής της πρότασης επανάληψης.

```
int count, max_count=30;
for (count=0; count<max_count; count++)
{
      printf( "count is %d\n",count );
      <άλλες προτάσεις>;
}
<επόμενη πρόταση>;
```

Στην πρώτη γραμμή δηλώνονται οι ακέραιες μεταβλητές **count** και **max_count**, και αποδίδεται τιμή στη **max_count**. Ακολουθεί η πρόταση επανάληψης ***for***, η οποία έχει μετρητή τη μεταβλητή **count**, βήμα τη μονάδα και θα εκτελείται καθόσον ο μετρητής έχει τιμή μικρότερη της **max_count**.

Ο βρόχος εκτελείται συνολικά 30 φορές:

count is 0

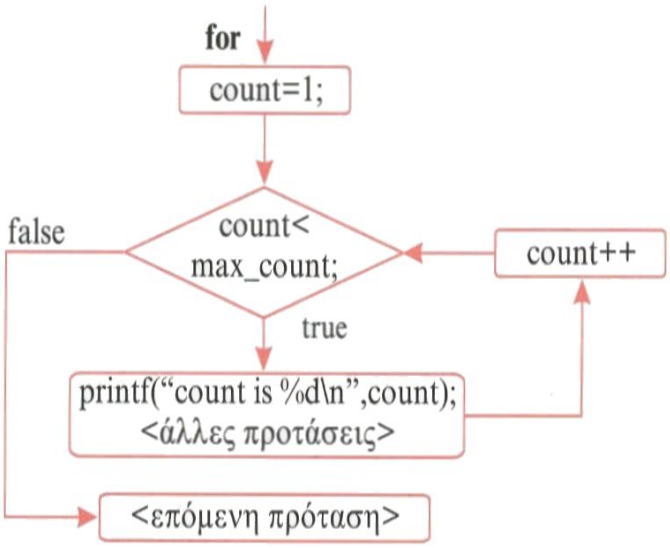
count is 1

. . . . . .

count is 29

Στο τέλος της τριακοστής επανάληψης ο μετρητής έχει λάβει την τιμή 30 και στον έλεγχο της συνθήκης στην τριακοστή πρώτη επανάληψη η τελευταία είναι ψευδής, οπότε ο έλεγχος προσπερνά το βρόχο και προχωρά στην επόμενη πρόταση.

Το διάγραμμα ροής απεικονίζεται στο σχήμα 5.4:



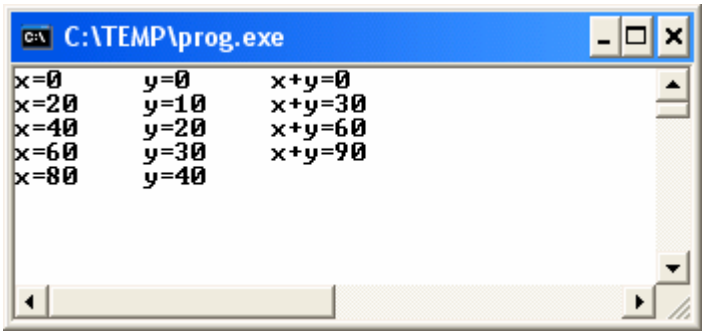
Σχ. 5.4 *Διάγραμμα ροής της πρότασης επανάληψης*

Παράδειγμα 5.4

Η πρόταση επανάληψης ***for*** μπορεί να χρησιμοποιεί περισσότερες της μίας μεταβλητές ελέγχου του βρόχου. Στο ακόλουθο πρόγραμμα τόσο η μεταβλητή **x** όσο και η **y** ελέγχουν το βρόχο:

```
#include <stdio.h>
void main()
{
      int x,y;
      for (x=0,y=0; x+y<100; x=x+20,y=y+10)
                  printf( "x=%d\ty=%d\tx+y=%d\n",x,y,x+y );
      printf( "x=%d\ty=%d ",x,y );
}
```

Το παραπάνω πρόγραμμα τυπώνει τους αριθμούς 0 έως 90 σε βήματα του 30. Σε κάθε εκτέλεση του βρόχου το **x** αυξάνει κατά 20 και το **y** κατά 10. Μετά το τέλος του βρόχου τα **x** και **y** έχουν διατηρήσει τις τιμές που τους δόθηκαν πριν τερματίσει ο βρόχος, όπως φαίνεται στα αποτελέσματα.



5.2.3 Ο τελεστής κόμμα (,)

Στο παράδειγμα 5.4 οι μεταβλητές για το έλεγχο της πρότασης επανάληψης διαχωρίζονταν με τον **τελεστή κόμμα (,)**. Ο τελεστής κόμμα επιτρέπει την παράθεση περισσότερων της μίας εκφράσεων σε θέσεις όπου επιτρέπεται μία έκφραση. Η τιμή της έκφρασης είναι η τιμή της δεξιότερης των εκφράσεων. Συνήθως περιπλέκει τον κώδικα και για αυτό το λόγο η χρήση του είναι περιορισμένη, εκτός από την πρόταση ***for***, στην οποία συνηθίζεται να χρησιμοποιείται ως συνθετικό των εκφράσεων αρχικοποίησης και ανανέωσης. Για παράδειγμα, η πρόταση

for (i=0,j=10; i<8; i++,j++) t[j]=s[i];

αντιγράφει τα οκτώ πρώτα στοιχεία του πίνακα **s** στον **s**, ξεκινώντας από το ενδέκατο στοιχείο του.

Θα πρέπει να αποφεύγονται προτάσεις όπως η

for (ch=getchar(),j=0; ch!= '.'; j++,putchar(ch),ch=getchar())

Η πρόταση, αν και είναι συμπαγής ως προς τον κώδικα, μειώνει σε μεγάλο βαθμό την αναγνωσιμότητά του.

5.2.4 Μετασχηματισμός βρόχων while – for

Ένας βρόχος ***for*** μπορεί να μετασχηματισθεί σε βρόχο ***while*** και τανάπαλιν με βάση την ακόλουθη φόρμα μετασχηματισμού:

```
for (αρχική τιμή μετρητή; συνθήκη; βήμα μετρητή)
{
    προτάσεις;
}
```

↓

```
αρχική τιμή μετρητή;
while (συνθήκη)
{
    προτάσεις;
    βήμα μετρητή;
}
```

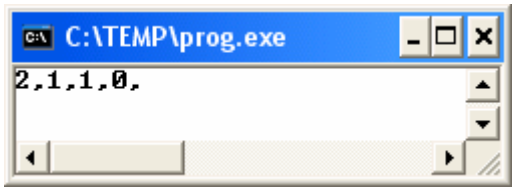
Παράδειγμα 5.5

Τι εμφανίζεται στην οθόνη του υπολογιστή από την εκτέλεση του βρόχου:

```
for (n = 4; n > 0;  n - - )      printf( "%d,",n/2 );
```

Να γραφεί εκ νέου ο παραπάνω κώδικας αντικαθιστώντας τη ***for*** με ***while***.

Ο παραπάνω κώδικας εκτελεί ένα βρόχο ***for*** με φθίνον βήμα τη μονάδα, σε κάθε επανάληψη του οποίου τυπώνεται το πηλίκο διαίρεσης ακεραίων. Τα αποτελέσματα είναι τα εξής:



Με βάση τη φόρμα μετασχηματισμού της §5.2.4, ο κώδικας με χρήση του βρόχου ***while*** είναι ο ακόλουθος:

```
n=4;
while (n>0)
{
      printf( "%d,",n/2 );
      n- -;
}
```

5.3 Βρόχος με συνθήκη εξόδου στη C (do – while)

Ο βρόχος ***do–while*** είναι βρόχος με συνθήκη εξόδου. Η λειτουργία του περιγράφεται εποπτικά στο σχήμα 5.2α και η σύνταξή του είναι η ακόλουθη:

```
do
{
      προτάσεις, μέσα στις οποίες θα αλλάζει η συνθήκη;
}
while (συνθήκη);
```

Η λειτουργία της πρότασης επανάληψης ***do–while*** μπορεί να μορφοποιηθεί σε δομημένα Ελληνικά ως εξής:

Εκτέλεσε τις προτάσεις
Έλεγξε τη συνθήκη
Εάν είναι αληθής
    Ξεκίνησε από την αρχή
Αλλιώς σταμάτησε

Είναι φανερό ότι ο βρόχος ***do–while*** εκτελείται τουλάχιστον μία φορά, καθώς ο έλεγχος της συνθήκης έπεται του σώματος του βρόχου. Δεν είναι συχνή η χρήση του – στατιστικά χρησιμοποιείται μόνο στο 5% των περιπτώσεων χρήσης βρόχου – καθώς αφενός μεν είναι προτιμότερο να εξετάζεται ένας βρόχος προτού εκτελεσθεί παρά μετά, αφετέρου δε σε πολλές χρήσεις είναι σημαντικό να μπορεί να παραληφθεί τελείως ο βρόχος εφόσον δεν ικανοποιείται εξαρχής ο έλεγχος. Επιπρόσθετα, για το βρόχο while ισχύουν οι παρατηρήσεις της §5.1.2.

Θα πρέπει να σημειωθεί ότι εάν το σώμα του βρόχου αποτελείται από μία πρόταση, δεν απαιτούνται {}.

Παράδειγμα 5.6

Να περιγραφεί η λειτουργία του ακόλουθου τμήματος κώδικα και να δοθεί το διάγραμμα ροής της πρότασης επανάληψης.

```
int count=30;
int limit=40;
do
{
  count++;
  printf( "count is %d\n",count);
}
while (count<limit);
<επόμενη πρόταση>;
```

Στις πρώτες δύο γραμμές δηλώνονται και αρχικοποιούνται οι ακέραιες μεταβλητές **count** και **limit**. Ακολουθεί η πρόταση επανάληψης ***do–while***, η οποία καθορίζει ως συνθήκη η μεταβλητή **count** να είναι μικρότερη της **limit**, γεγονός που αληθεύει. Κατά συνέπεια ο έλεγχος εισέρχεται στο βρόχο, η μεταβλητή **count** αυξάνεται κατά μία μονάδα, τυπώνεται στην οθόνη η φράση

count is 31

και στη συνέχεια διενεργείται ο έλεγχος της συνθήκης. Ο βρόχος εκτελείται συνολικά 10 φορές:

count is 31

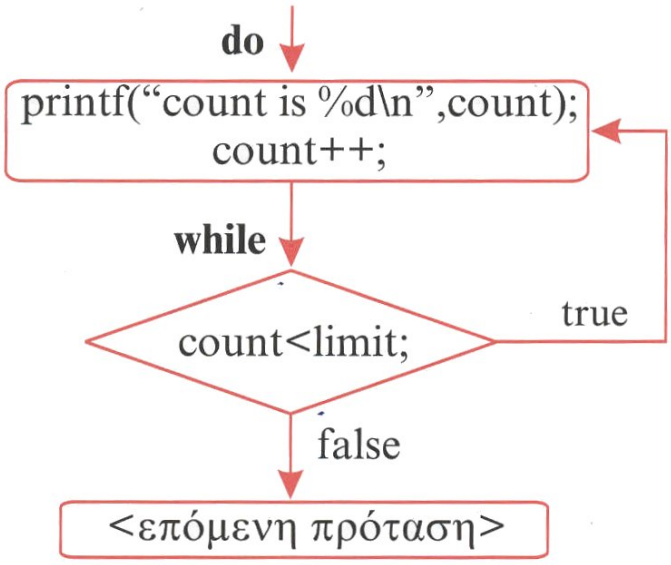
count is 32

. . . . . .

count is 40

Στο τέλος της δέκατης επανάληψης η συνθήκη είναι ψευδής και ο βρόχος τερματίζεται. Θα πρέπει να σημειωθεί ότι εάν η συνθήκη ήταν εξαρχής ψευδής (π.χ. **count=45**), ο βρόχος θα εκτελείτο μία φορά.

Το διάγραμμα ροής απεικονίζεται στο σχήμα 5.5:



Σχ. 5.5 *Διάγραμμα ροής της πρότασης επανάληψης*

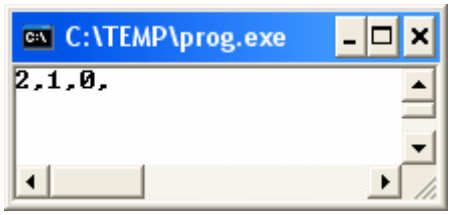
Παράδειγμα 5.7

Τι εμφανίζεται στην οθόνη του υπολογιστή από την εκτέλεση του βρόχου:

for (n=4; n > 1; n- -) printf("%d,",10%n);

Να γραφεί εκ νέου ο παραπάνω κώδικας αντικαθιστώντας τη ***for*** με ***do–while***.

Ο παραπάνω κώδικας εκτελεί ένα βρόχο ***for*** με φθίνον βήμα τη μονάδα, σε κάθε επανάληψη του οποίου τυπώνεται το υπόλοιπο διαίρεσης ακεραίων. Τα αποτελέσματα είναι τα εξής:



Ο κώδικας με χρήση του βρόχου ***do–while*** είναι ο ακόλουθος:

```
n=4;
do
{
```

```
        printf( "%d,",10%n );
        n- -;
    }
    while (n-1);
```

Θα πρέπει να σημειωθεί ότι στη συνθήκη ελέγχου του βρόχου ο μετρητής δεν είναι το **n** αλλά το **n-1** (δηλαδή η **n-1** να είναι αληθής, επομένως **n-1>0⇔n>1**), καθώς πρέπει να ληφθεί υπόψη η πρώτη επανάληψη, η οποία εκτελείται ούτως ή άλλως.

5.4 Ένθετοι βρόχοι

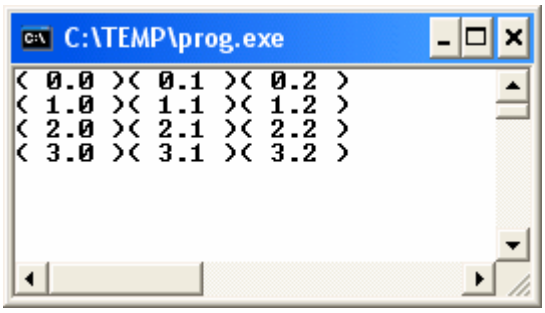
Ένθετος ή φωλιασμένος βρόχος (nested loop) ονομάζεται ο βρόχος που περικλείεται σε έναν άλλο. Ο εσωτερικός βρόχος λογίζεται ως μία πρόταση του εσωτερικού, κατά συνέπεια πρώτα θα εκτελείται ολόκληρος ο εσωτερικός βρόχος και μετά θα εκτελείται η επόμενη επανάληψη του εξωτερικού. Η C δε θέτει κανένα περιορισμό στην ένθεση των προτάσεων ελέγχου ροής, επιτρέποντας την πολλαπλή ένθεση. Ο συνολικός αριθμός επαναλήψεων σε έναν πολλαπλό βρόχο είναι το γινόμενο του αριθμού των επαναλήψεων όλων των επιμέρους βρόχων.

Παράδειγμα 5.8

Να περιγραφεί η λειτουργία του ακόλουθου τμήματος κώδικα.

```
for ( i=0; i<4; i++ )
{
    for ( j=0; j<3; j++ )
    {
        printf( "( %d.%d )", i, j );
    }
    printf( "\n" );
}
```

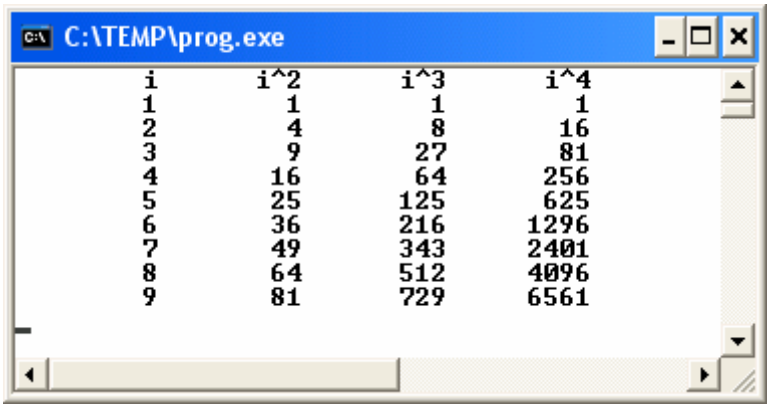
Ο κώδικας παρουσιάζει ένα διπλό βρόχο. Ο εξωτερικός βρόχος ***for*** έχει μετρητή το **i** και σώμα που αποτελείται από δύο προτάσεις: *α)* τον εσωτερικό βρόχο ***for*** με μετρητή το **j** και *β)* την πρόταση ***printf("\n");***. Σε κάθε επανάληψη του εσωτερικού βρόχου θα εκτελούνται όλες οι επαναλήψεις του ένθετου και στη συνέχεια θα εκτελείται η ***printf("\n");***, όπως προκύπτει από τα αποτελέσματα:



Παράδειγμα 5.9

Το πρόγραμμα που ακολουθεί εμφανίζει τις τέσσερις πρώτες ακέραιες δυνάμεις των αριθμών έως 9.

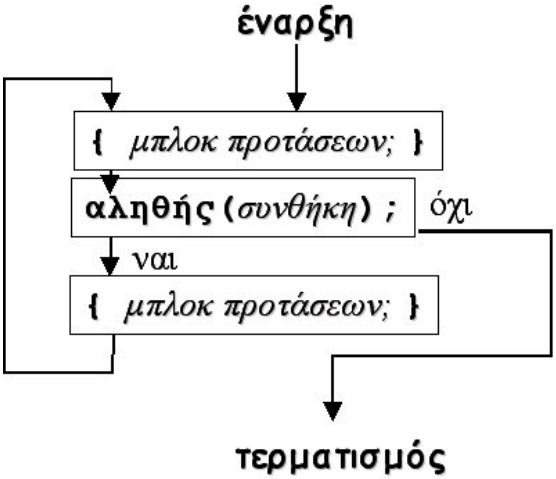
```
# include <stdio.h>
void main()  {
      int i,j,k,temp;
      printf( "      i     i^2     i^3     i^4\n" );
      for ( i=1; i<10; i++ )  {
            for ( j=1; j<5; j++ )  {
                  temp=1;
                  for ( k=0; k<j; k++ )  temp=temp*i;
                  printf( "%9d",temp );
            }   // Τέλος του βρόχου j
            printf( "\n" );
      }   // Τέλος του βρόχου i
}
```



5.5 Διακοπτόμενοι βρόχοι στη C

5.5.1 Η κωδική λέξη break

Στο προηγούμενο κεφάλαιο η λέξη ***break*** χρησιμοποιήθηκε στην πρόταση διακλάδωσης ***switch***. Όμως πέραν αυτής, η ***break*** έχει και δεύτερη χρήση, η οποία σχετίζεται με τις προτάσεις επανάληψης. Χρησιμοποιείται για να τερματίζει αμέσως την εκτέλεση ενός βρόχου, μεταβιβάζοντας τον έλεγχο του προγράμματος στην εντολή που βρίσκεται αμέσως μετά το βρόχο. Εάν η ***break*** βρίσκεται μέσα σε ένθετο βρόχο, τότε επηρεάζεται μόνο ο εσώτερος βρόχος. Η λειτουργία της ***break*** περιγράφεται εποπτικά στο σχήμα 5.6:



Σχ. 5.6 *Μπλοκ διάγραμμα της break*

Παράδειγμα 5.10

Το πρόγραμμα που ακολουθεί εμφανίζει στην οθόνη τους αριθμούς 1 έως 10 και στη συνέχεια τερματίζεται γιατί η ***break*** υπερφαλαγγίζει τη συνθήκη ελέγχου του βρόχου **t<100**.

```
# include <stdio.h>
void main()
{
      int t;
      for ( t=0; t<100; t++ )  {
            printf( "%d ",t );
            if (t==10) break;
      }
}
```

5.5.2 Η πρόταση continue

Η εντολή ***continue*** μεταφέρει τον έλεγχο της ροής στην αρχή του βρόχου, παραλείποντας την εκτέλεση του υπόλοιπου τμήματος του σώματος του βρόχου και προχωρώντας στην επόμενη επανάληψη.

Στους βρόχους ***while*** και ***do–while*** η εντολή ***continue*** υποχρεώνει τον έλεγχο του προγράμματος να περάσει κατευθείαν στη συνθήκη ελέγχου και να προχωρήσει κατόπιν στην επεξεργασία του βρόχου. Στην περίπτωση της ***for*** ο υπολογιστής εκτελεί πρώτα το τμήμα του βρόχου και κατόπιν τη συνθήκη ελέγχου, προτού συνεχισθεί η εκτέλεση του βρόχου.

Παράδειγμα 5.11

Το πρόγραμμα που ακολουθεί εμφανίζει στην οθόνη μόνο τους άρτιους αριθμούς.

```
# include <stdio.h>
void main()
{
      int x;
      for ( x=0; x<100; x++ )  {
            if (x%2) continue;
            printf( "%d",x );
            <προτάσεις>;
      }
}
```

Κάθε φορά που παράγεται ένας περιττός αριθμός ενεργοποιείται η εντολή διακλάδωσης και εκτελείται η ***continue***, παρακάμπτεται η ***printf*** και οι υπόλοιπες προτάσεις, οπότε ο έλεγχος προχωρά στην επόμενη επανάληψη.

5.5.3 Η πρόταση goto

Η πρόταση **ρητής διακλάδωσης**

goto <ετικέτα>;

μεταφέρει τον έλεγχο στην πρόταση που σημειώνεται με την ετικέτα ως

<ετικέτα>: πρόταση

Η εντολή ***goto*** **<u>πρέπει να αποφεύγεται</u>** γιατί οδηγεί σε κώδικα «σπαγγέτι» και αίρει τα πλεονεκτήματα του δομημένου προγραμματισμού. Μπορεί να χρησιμοποιηθεί σε περιπτώσεις εξόδου από πολύ βαθιά ενσωματωμένη δομή, όπου μία προσεκτική χρήση της ***goto*** μπορεί να δώσει πιο συμπαγή κώδικα. Θα πρέπει να σημειωθεί ότι η C είναι δομημένη κατά τρόπο ώστε να μην απαιτεί ποτέ τη χρήση τής ***goto***, σε αντιδιαστολή με άλλες γλώσσες προγραμματισμού, όπως η FORTRAN και η BASIC, στις οποίες επιβάλλεται η χρήση της ***goto*** σε ορισμένες περιπτώσεις.

Παράδειγμα 5.12

Έστω το ακόλουθο τμήμα κώδικα:

```
for (. . .) {
    for (. . .) {
        while (. . .) {
            if (. . .) goto label13;
            . . . . . .
        }
    }
}
label13: printf( "ERROR!!!" );
```

Η απαλοιφή της ***goto*** θα υποχρέωνε τον κώδικα να εκτελέσει έναν αριθμό από πρόσθετους ελέγχους. Η χρήση μίας ***break*** δε θα ήταν αρκετή γιατί θα προκαλούσε έξοδο μόνο από τον εσώτερο βρόχο. Εάν τοποθετούνταν έλεγχοι σε όλους τους βρόχους, ο κώδικας θα έπαιρνε την ακόλουθη σωστή αλλά δύσχρηστη μορφή:

```
done=0;
for (. . .) {
    for (. . .) {
        while (. . .) {
            if (. . .) {
                done=1;
                break;   // έξοδος από τη while
            } // τέλος του if
            . . . . . .
        } // τέλος της while
        if (done) break;
```

```
} // τέλος του εσωτερικού βρόχου for
if (done) break;
} // τέλος του εξωτερικού βρόχου for
```

5.6 Κανόνες για τη χρήση των προτάσεων επανάληψης

1. Τοποθετείτε πάντοτε το σώμα των προτάσεων επανάληψης μία θέση στηλογνώμονα δεξιότερα, για αύξηση της αναγνωσιμότητας του κώδικα. Στην περίπτωση δε που το σώμα αποτελείται από περισσότερες της μίας προτάσεις, περικλείετε αυτές σε άγκιστρα.

2. Αποφεύγετε τη χρήση της πρότασης διακλάδωσης ***goto***. Καταστρέφει τη δόμηση του προγράμματος και τις περισσότερες φορές προδίδει αδυναμία κατασκευής δομημένου κώδικα.

3. Προτιμήστε το βρόχο επανάληψης συνθήκης εισόδου (***while***) από τον αντίστοιχο συνθήκης εξόδου (***do-while***) γιατί οδηγεί σε πιο ευανάγνωστο κώδικα.

4. Αποφεύγετε κατά το δυνατόν τη χρήση των ***break*** και ***continue*** σε βρόχους επανάληψης, επειδή διακόπτουν την κανονική ροή ελέγχου και καθιστούν την παρακολούθησή της δύσκολη.

5. Ελέγξτε σχολαστικά και βεβαιωθείτε ότι κάθε συνθήκη βρόχου επανάληψης οδηγεί στην έξοδο μετά από πεπερασμένες επαναλήψεις, έτσι ώστε να μη δημιουργούνται ατέρμονες βρόχοι.

Κεφάλαιο 6

ΠΙΝΑΚΕΣ – ΑΛΦΑΡΙΘΜΗΤΙΚΑ

6.1 Μονοδιάστατοι πίνακες

Ο πίνακας είναι μία συλλογή μεταβλητών ίδιου τύπου, οι οποίες είναι αποθηκευμένες σε διαδοχικές θέσεις μνήμης. Χρησιμοποιείται για την αποθήκευση και διαχείριση δεδομένων κοινού τύπου και αποτελεί, μαζί με τους δείκτες, από τα πλέον ισχυρά εργαλεία της γλώσσας C.

- Η δήλωση του πίνακα ακολουθεί τον εξής φορμαλισμό:

τύπος_δεδομένου όνομα_πίνακα[μέγεθος]

Διακρίνονται τρία τμήματα: *α)* ο τύπος δεδομένου (float, int, char, double), *β)* το όνομα του πίνακα και *γ)* ο αριθμός των στοιχείων που απαρτίζουν τον πίνακα. Έτσι, μία τυπική δήλωση ενός πίνακα 31 στοιχείων κινητής υποδοαστολής είναι η ακόλουθη:

float temp[31];

- Η αναφορά σε στοιχείο πίνακα γίνεται με συνδυασμό του ονόματος και ενός δείκτη (index), ο οποίος εκφράζει τη σειρά τού στοιχείου μέσα στον πίνακα:

temp[0]: ***πρώτο*** στοιχείο του πίνακα

temp[1]: ***δεύτερο*** στοιχείο του πίνακα

temp[30]: ***τελευταίο*** (τριακοστό πρώτο) στοιχείο του πίνακα

- Η απόδοση αρχικής τιμής κατά τη δήλωση του πίνακα γίνεται με χρήση του τελεστή ανάθεσης ως εξής:

float temp[5] = {1,2,-4.2,6,8}: αρχικοποιούνται και τα 5 στοιχεία του πίνακα **temp**.

float temp[5] = {1,2,-4.2}: αρχικοποιούνται τα 3 πρώτα στοιχεία του πίνακα **temp**, δηλαδή τα **temp[0]**, **temp[1]**, **temp[2]**.

- Η ανάγνωση και εκτύπωση ενός πίνακα γίνονται κατά στοιχείο, με τους κανόνες που

ισχύουν για κάθε τύπο δεδομένου:

```
for ( i=0; i<ar_size; i++ )
{
    scanf( "%f",&ar[i] );
    printf( "ar[%d]=%f",i,ar[i] );
}
```

Παρατηρήσεις:

1) Όταν αποδίδονται αρχικές τιμές μπορεί να παραληφθεί το μέγεθος του πίνακα. Ο υπολογιστής θα υπολογίσει αυτόματα πόσα είναι τα στοιχεία του πίνακα από τον αριθμό των αρχικών τιμών που δίδονται. Η παρακάτω δήλωση

char d_ar[ ] = {'a', 'b', 'c', 'd'};

έχει ως αποτέλεσμα τη δημιουργία ενός πίνακα χαρακτήρων (***char***) τεσσάρων στοιχείων με αρχικές τιμές:

d_ar[0] = 'a'
d_ar[1] = 'b'
d_ar[2] = 'c'
d_ar[3] = 'd'

2) Το γεγονός ότι οι δείκτες των στοιχείων ενός πίνακα ξεκινούν από το **0** κι όχι από το **1** μπορεί αρχικά να προκαλέσει σύγχυση αλλά αντανακλά τη φιλοσοφία της C, η οποία επιδιώκει να παραμείνει ο προγραμματισμός κοντά στην αρχιτεκτονική του υπολογιστή. Το **0** αποτελεί το σημείο εκκίνησης για τους υπολογιστές. Εάν η αρίθμηση των στοιχείων πίνακα ξεκινούσε από το **1**, όπως π.χ. συμβαίνει στη FORTRAN, ο μεταγλωττιστής θα έπρεπε να αφαιρέσει τη μονάδα από κάθε αναφορά σε δείκτη στοιχείου για να ληφθεί η πραγματική διεύθυνση ενός στοιχείου. Επομένως, η επιλογή της C παράγει πιο αποτελεσματικό κώδικα.

3) Υπάρχει διαφορά ανάμεσα στη δήλωση πίνακα και στην αναφορά στοιχείου πίνακα. Σε μία δήλωση, ο δείκτης καθορίζει το μέγεθος του πίνακα. Σε μία αναφορά στοιχείου πίνακα, ο δείκτης προσδιορίζει το στοιχείο του πίνακα στο οποίο αναφερόμαστε. Π.χ. στη δήλωση **int temp[31];** το **31** δηλώνει τον αριθμό των στοιχείων του πίνακα. Αντίθετα στη **temp[13]=21;** το **13** δηλώνει το συγκεκριμένο στοιχείο ($14^{ο}$) του πίνακα, στο οποίο αναφερόμαστε και αποδίδουμε την τιμή **21**.

4) Μπορεί να βρεθεί το μέγεθος σε bytes ενός πίνακα χρησιμοποιώντας τον τελεστή ***sizeof***. Για παράδειγμα, εάν θεωρηθεί ο πίνακας **int ar[5];** η έκφραση **sizeof(ar)** δίνει τιμή **20** επειδή ο πίνακας αποτελείται από 5 ακεραίους των 4 bytes.

Στη ***sizeof*** θα πρέπει να περιλαμβάνεται μόνο το όνομα του πίνακα. Αν περιληφθεί δείκτης ενός στοιχείου, τότε θα εξαχθεί το μέγεθος του στοιχείου. Για παράδειγμα, η έκφραση **sizeof(ar[0])** δίνει τιμή **4**.

Χρησιμοποιώντας ένα συνδυασμό των παραπάνω μπορεί να βρεθεί ο αριθμός των στοιχείων του πίνακα. Η έκφραση **sizeof(ar)/sizeof(ar[0])** δίνει **5**, τον αριθμό δηλαδή των στοιχείων του πίνακα **ar**.

6.2 Πολυδιάστατοι πίνακες

Οι πολυδιάστατοι πίνακες είναι πίνακες, τα στοιχεία των οποίων είναι επίσης πίνακες. Η πρόταση **int array[4][12];** δηλώνει τη μεταβλητή **array** ως πίνακα 4 στοιχείων, όπου το κάθε στοιχείο είναι πίνακας 12 στοιχείων ακεραίων. Η C δε θέτει περιορισμό στον αριθμό των διαστάσεων των πινάκων.

Αν και ο πολυδιάστατος πίνακας αποθηκεύεται στη μνήμη ως μία ακολουθία στοιχείων μίας διάστασης, μπορούμε να το θεωρούμε ως πίνακα πινάκων. Για παράδειγμα, έστω το επόμενο «μαγικό τετράγωνο», του οποίου οι γραμμές οριζόντια, κάθετα και διαγώνια δίνουν το ίδιο άθροισμα:

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Για να αποθηκευθεί το τετράγωνο αυτό σε πίνακα θα μπορούσε να γίνει η ακόλουθη δήλωση:

```
int magic[5][5]= {    {17, 24, 1, 8, 15},
                      {23, 5, 7, 14, 16},
```

```
                    {4, 6, 13, 20, 22},
                    {10, 12, 19, 21, 3},
                    {11, 18, 25, 2, 9}
                };
```

Από τον προηγούμενο κώδικα γίνεται αντιληπτό ότι στην απόδοση των αρχικών τιμών οι τιμές των στοιχείων κάθε γραμμής περικλείονται σε άγκιστρα.

- Για την αναφορά σε στοιχείο ενός πολυδιάστατου πίνακα θα πρέπει να καθορισθούν τόσοι δείκτες όσοι είναι αναγκαίοι. Έτσι, η έκφραση

magic[1]

αναφέρεται στη δεύτερη γραμμή του πίνακα, ενώ η

magic[1][3]

αναφέρεται στο τέταρτο στοιχείο της δεύτερης γραμμής του πίνακα.

- Οι πολυδιάστατοι πίνακες αποθηκεύονται κατά γραμμές, που σημαίνει ότι ο τελευταίος δείκτης θέσης μεταβάλλεται ταχύτερα κατά την προσπέλαση των στοιχείων. Για παράδειγμα, ο πίνακας που δηλώνεται ως:

```
int ar[2][3]= {    {0, 1, 2},
                   {3, 4, 5}
              };
```

αποθηκεύεται όπως φαίνεται ακολούθως:

Σχ. 6.1 *Αποθήκευση πολυδιάστατου πίνακα*

6.3 Αρχικοποίηση πολυδιάστατου πίνακα

Για την αρχικοποίηση ενός πολυδιάστατου πίνακα κάθε γραμμή αρχικών τιμών περικλείεται σε άγκιστρα. Εάν δεν υπάρχουν οι αναγκαίες αρχικές τιμές, τα επιπλέον στοιχεία λαμβάνουν αρχική τιμή 0. Έτσι, στη δήλωση και αρχικοποίηση του ακόλουθου πίνακα

```
int ar[5][3]= {   {1, 2, 3},
                  {4},
                  {5, 6, 7}
              };
```

η μεταβλητή ***ar*** δηλώνεται ως πίνακας 5 γραμμών και 3 στηλών. Ωστόσο έχουν αποδοθεί τιμές μόνο για τις 3 πρώτες γραμμές του πίνακα και μάλιστα για τη δεύτερη γραμμή έχει ορισθεί η τιμή μόνο του πρώτου στοιχείου. Η παραπάνω δήλωση έχει ως αποτέλεσμα τη δημιουργία του ακόλουθου πίνακα:

```
1  2  3
4  0  0
5  6  7
0  0  0
0  0  0
```

Εάν δε συμπεριληφθούν τα ενδιάμεσα άγκιστρα:

```
int ar[5][3]= {   1, 2, 3,
                  4,
                  5, 6, 7 };
```

το αποτέλεσμα είναι ο ακόλουθος πίνακας, που είναι σαφώς διαφορετικός, οπότε δημιουργείται πρόβλημα:

```
1  2  3
4  5  6
7  0  0
0  0  0
0  0  0
```

Παρατηρήσεις:

1. Όπως και με τους πίνακες μίας διάστασης, έτσι και στους πολυδιάστατους πίνακες εάν αμεληθεί να δοθεί το μέγεθος του πίνακα, ο μεταγλωττιστής θα το καθορίσει

αυτόματα με βάση τον αριθμό αρχικών τιμών που παρουσιάζονται. Ωστόσο στους πολυδιάστατους πίνακες μπορεί να παραληφθεί ο αριθμός των στοιχείων μόνο της πρώτης διάστασης, καθώς ο μεταγλωττιστής μπορεί να τον υπολογίσει από τον αριθμό των αρχικών τιμών που διατίθενται. Η παρακάτω δήλωση αξιοποιεί τη δυνατότητα αυτή του μεταγλωττιστή:

```
int ar[ ][3][2]=    {    {    {1, 1}, {0, 0}, {1, 1}    },
                         {    {0, 0}, {1, 2}, {0, 1}    }
                    };
```

Με την παραπάνω δήλωση ο **ar** δηλώνεται αυτόματα ως πίνακας 2x3x2.

2. Η παρακάτω δήλωση:

```
int ar[ ][ ]={ 1, 2, 3, 4, 5, 6};
```

είναι ανεπίτρεπτη επειδή ο μεταγλωττιστής δεν μπορεί να γνωρίζει τι είδους θα ήταν αυτός ο πίνακας. Θα μπορούσε να το θεωρήσει είτε πίνακα 2x3 είτε 3x2.

3. Η παρακάτω δήλωση:

```
printf( "%d",array[1,2] );
```

είναι λανθασμένη στη γλώσσα C αλλά δεν εντοπίζεται από το μεταγλωττιστή και οδηγεί σε ανεπιθύμητα αποτελέσματα.. Η σωστή είναι

```
printf( "%d",array[1][2] );
```

Παράδειγμα 6.1

Έστω πίνακας που αναπαριστά τις μέσες θερμοκρασίες των μηνών των τελευταίων τριών ετών. Να δοθούν: *(α)* βρόχος για τον υπολογισμό των μέσων ετήσιων θερμοκρασιών των τριών ετών, *(β)* βρόχος για τον υπολογισμό των μέσων μηνιαίων θερμοκρασιών των τριών ετών.

α)

```
#define MONTHS 12
#define YEARS 3
. . . . . .
int year, month;
```

```
float subtotal, temp[YEARS][MONTHS], avg_temp[YEARS];
. . . . . .
// ο temp θεωρείται αρχικοποιημένος
for ( year=0; year<YEARS; year++ )
{
      subtotal=0.0;
      for ( month=0 ; month<MONTHS ; month++ )
            subtotal+=temp[year][month];
      avg_temp[year]=subtotal/MONTHS;
}
. . . . . .
```

β)

```
#define YEARS 3
#define MONTHS 12
. . . . . .
int year, month;
float subtotal, temp[YEARS][MONTHS], avg_temp[MONTHS];
. . . . . .
// ο temp θεωρείται αρχικοποιημένος
for ( month=0; month<MONTHS; month++ )
{
      subtotal=0.0;
      for ( year=0; year<YEARS; year++ )
      subtotal+=temp[year][month];
      avg_temp[month]=subtotal/YEARS;
}
. . . . . .
```

6.4 Αποθήκευση των πινάκων στη μνήμη

Η πρόταση δήλωσης **int ar[5];** έχει ως αποτέλεσμα τη δέσμευση χώρου στη μνήμη για την αποθήκευση 5 μεταβλητών ακέραιου τύπου. Έστω ότι μέσα στον κώδικα του προγράμματος υπάρχουν οι παρακάτω προτάσεις ανάθεσης, που αποδίδουν τιμές σε στοιχεία του πίνακα:

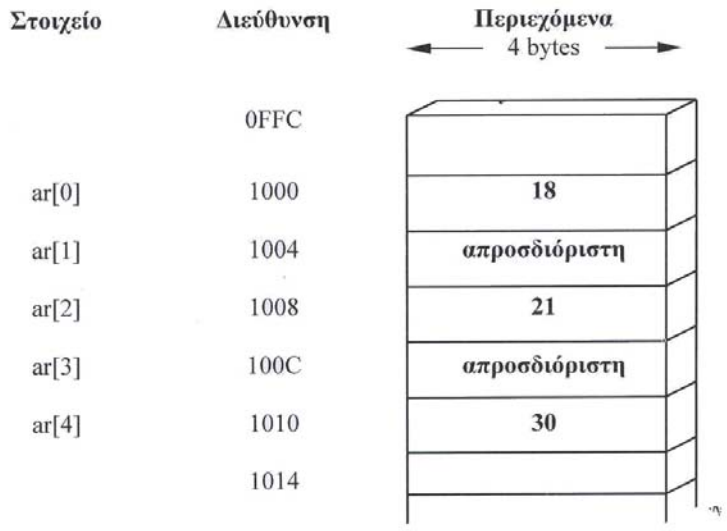
ar[0] = 18;

ar[2] = 21;

ar[4] = ar[0] + 12;

Στο σχήμα 6.2 φαίνεται η μορφή που έχει η μνήμη, η οποία έχει δεσμευτεί για την αποθήκευση του πίνακα **ar**, θεωρώντας 32 bits για κάθε ακέραιο και πως η πρώτη διεύθυνση είναι η 1000. Γίνεται αντιληπτό ότι τα **ar[1]** και **ar[3]** έχουν απροσδιόριστες τιμές. Τα περιεχόμενα αυτών των θέσεων μνήμης είναι ο,τιδήποτε έχει μείνει στις θέσεις αυτές από προηγούμενη αποθήκευση. Οι απροσδιόριστες τιμές αναφέρονται ως «σκουπίδια» (junk) και πολλές φορές αποτελούν αιτία πρόκλησης λαθών.

Για την αποφυγή προβλημάτων αυτής της μορφής πρέπει να αποδίδονται αρχικές τιμές στους πίνακες ή να υπάρχει συνεχής αντίληψη του τι περιλαμβάνει ένας πίνακας. Θα πρέπει να σημειωθεί ότι η ANSI C διασφαλίζει ότι οι καθολικές μεταβλητές (αρχικοποιούνται από το σύστημα με την τιμή 0. Αυτό συμβαίνει βέβαια και για όλα τα στοιχεία πίνακα, ο οποίος έχει δηλωθεί ως καθολική μεταβλητή.



Σχ. 6.2 *Αποθήκευση πίνακα στη μνήμη*

6.5 Το αλφαριθμητικό

Το αλφαριθμητικό ή **συμβολοσειρά (string)** είναι ένας πίνακας χαρακτήρων που τερματίζει με το **μηδενικό (null) χαρακτήρα**. Ο μηδενικός χαρακτήρας έχει ASCII κωδικό **0** και αναπαρίσταται από την ακολουθία διαφυγής **\0**.

- Η δήλωση του αλφαριθμητικού ακολουθεί τον εξής φορμαλισμό:

char όνομα[μήκος]

Διακρίνονται τρία τμήματα: *α)* ο τύπος δεδομένου, ο οποίος είναι πάντοτε char, *β)* το όνομα του αλφαριθμητικού και *γ)* το μήκος του. Έτσι, μία τυπική δήλωση ενός αλφαριθμητικού 30 χαρακτήρων έχει την ακόλουθη μορφή:

char book_title[30]

- Τα αλφαριθμητικά μπορούν να εμφανίζονται μέσα στον κώδικα όπως οι αριθμητικές σταθερές, αποτελώντας τις **αλφαριθμητικές σταθερές**. Η αλφαριθμητική σταθερά απαντήθηκε σε προηγούμενο κεφάλαιο, όπου και σημειώθηκε ότι οι χαρακτήρες της περικλείονται σε διπλά εισαγωγικά. Για την αποθήκευσή της χρησιμοποιείται ένας πίνακας χαρακτήρων, με το μεταγλωττιστή να θέτει αυτόματα στο τέλος του αλφαριθμητικού ένα μηδενικό χαρακτήρα για να προσδιορίσει το τέλος του. Έτσι, η αλφαριθμητική σταθερά **"Hello"** απαιτεί για αποθήκευση 6 bytes, όπως φαίνεται παρακάτω:

'H' 'e' 'l' 'l' 'o' '\0'

Παρατήρηση: Θα πρέπει να σημειωθεί η διαφορά ανάμεσα στη σταθερά χαρακτήρα **'A'** και την αλφαριθμητική σταθερά **"A"**. Η πρώτη απαιτεί 1 byte για αποθήκευση, ενώ η δεύτερη απαιτεί ένα byte για το χαρακτήρα **A** κι ένα byte για το **null**.

6.6 Αρχικοποίηση αλφαριθμητικού

Η ανάθεση τιμής με τη δήλωση ακολουθεί το γενικό κανόνα απόδοσης αρχικής τιμής σε πίνακα:

char isbn[ ] = {'0', '-', '4', '9', '-', '7', '4', '3', '-', '3', '\0'};

Στην πράξη χρησιμοποιείται η ακόλουθη εναλλακτική και πιο συμπαγής μορφή με χρήση αλφαριθμητικής σταθεράς:

char isbn[ ] = "0-49-743-3";

Θα πρέπει να προσεχθεί ότι στη δήλωση με λίστα στοιχείων ο προγραμματιστής πρέπει να περιλάβει ως τελευταία τιμή το **null**. Στο δεύτερο τρόπο απόδοσης αρχικής τιμής, αυτή την εργασία την εκτελεί αυτόματα ο μεταγλωττιστής.

6.7 Είσοδος – έξοδος αλφαριθμητικών

6.7.1 Εισαγωγή αλφαριθμητικού

Η εισαγωγή αλφαριθμητικού από την κύρια είσοδο γίνεται με τη μορφοποιούμενη συνάρτηση ***scanf*** και τον προσδιοριστή **%s**. Η πρόταση

scanf("%s", isbn);

διαβάζει την κύρια είσοδο ως αλφαριθμητικό και αποθηκεύει την τιμή στη μεταβλητή **isbn**. Δε χρειάζεται ο τελεστής **&** πριν από το όνομα της μεταβλητής **isbn** όπως συνέβαινε με τους άλλους τύπους δεδομένων, γιατί το όνομα του αλφαριθμητικού αναπαριστά τη διεύθυνση του πρώτου στοιχείου του.

Εναλλακτικά, η εισαγωγή αλφαριθμητικού μπορεί να γίνει με χρήση της συνάρτησης ***gets***, η γενική μορφή της οποίας είναι

gets(όνομα_πίνακα_χαρακτήρων)

Καλείται η ***gets*** με το όνομα του πίνακα χαρακτήρων ως όρισμα, χωρίς δείκτη. Με την επιστροφή από τη ***gets*** το αλφαριθμητικό θα έχει περασθεί στον πίνακα χαρακτήρων. Η ***gets*** θα διαβάζει χαρακτήρες από το πληκτρολόγιο έως ότου πατηθεί το ENTER.

Παρατηρήσεις:

1. Το αλφαριθμητικό αποθηκεύεται σε έναν πίνακα χαρακτήρων μαζί με το τελικό μηδενικό χαρακτήρα. Κατά συνέπεια, όταν δηλώνεται το μέγεθος του πίνακα, έστω N, σε αυτόν μπορεί να αποθηκευθεί αλφαριθμητικό μέγιστου μήκους N-1.

2. Θα πρέπει να σημειωθεί ότι τόσο η ***scanf*** όσο και η ***gets*** δεν εκτελούν έλεγχο ορίων στον πίνακα χαρακτήρων με τον οποίο καλούνται. Εάν π.χ. δηλωθεί **char isbn[30]** και το αλφαριθμητικό είναι μεγαλύτερο από το μέγεθος του **isbn**, ο πίνακας θα ξεπερασθεί.

6.7.2 Εκτύπωση αλφαριθμητικού

Η εκτύπωση αλφαριθμητικής σταθεράς γίνεται με την ***printf*** χωρίς τη χρήση προσδιοριστή. Απλώς της δίνεται η προς εκτύπωση αλφαριθμητική σταθερά:

printf("Hello");

Η εκτύπωση αλφαριθμητικού γίνεται με την ***printf*** χρησιμοποιώντας τον προσδιοριστή **%s**. Η παρακάτω πρόταση

printf("The ISBN code is: %s", isbn);

θα έχει ως αποτέλεσμα να τυπωθεί στην οθόνη η πρόταση

The ISBN code is: 0-49-743-3

Εναλλακτικά, η εκτύπωση αλφαριθμητικής σταθεράς και αλφαριθμητικού μπορεί να γίνει με χρήση της συνάρτησης ***puts***, η γενική μορφή της οποίας είναι

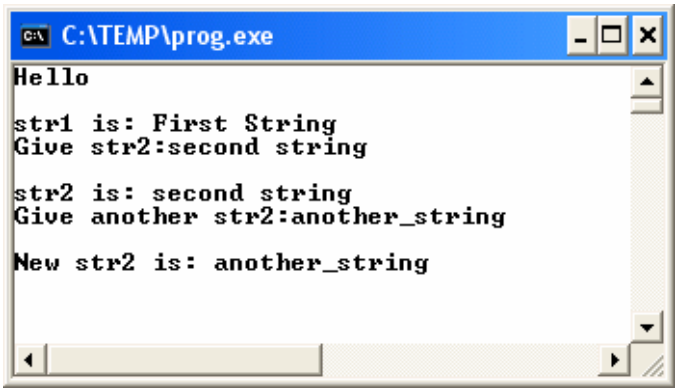
puts(όνομα_πίνακα_χαρακτήρων)

Καλείται η ***puts*** με το όνομα του πίνακα χαρακτήρων ως όρισμα, χωρίς δείκτη, π.χ. **puts(isbn)**. Βέβαια, η ***puts*** παρουσιάζει το μειονέκτημα ότι δεν παρέχει δυνατότητες μορφοποίησης της εξόδου.

Παράδειγμα 6.2

Στο ακόλουθο πρόγραμμα γίνεται εισαγωγή και εκτύπωση αλφαριθμητικών με όλους τους τρόπους που περιγράφηκαν ανωτέρω. Θα πρέπει να προσεχθεί η χρήση της ***define*** για την εισαγωγή αλφαριθμητικής σταθεράς.

```
#include <stdio.h>
#define STA "Hello"
void main()  {
      char str1[ ]= "First String";
      char str2[81];
      puts(STA);
      printf( "\nstr1 is: %s\nGive str2:",str1 );
      gets(str2);
      printf( "\nstr2 is: %s\nGive another str2:",str2 );
      scanf( "%s",str2 );
      printf( "\nNew str2 is: " );
      puts(str2);
}
```



6.8 Συναρτήσεις αλφαριθμητικών

Η C υποστηρίζει μία ποικιλία συναρτήσεων για το χειρισμό των αλφαριθμητικών. Οι συναρτήσεις αυτές βρίσκονται στο αρχείο κεφαλίδας **<string.h>**. Οι πιο συνηθισμένες παρουσιάζονται στον ακόλουθο πίνακα, στον οποίο η δεύτερη και η τρίτη στήλη περιέχουν τα ονόματα των συναρτήσεων όταν η λειτουργία τους επιδρά σε ολόκληρο το αλφαριθμητικό ή στους πρώτους *n* χαρακτήρες, αντίστοιχα:

Λειτουργία	**Όλοι οι χαρακτήρες**	**Οι *n* πρώτοι χαρακτήρες**
Εύρεση μήκους string	**strlen()**	
Αντιγραφή string	**stcpy()**	**strncpy()**
Συνένωση 2 strings	**strcat()**	**strncat()**
Σύγκριση 2 strings	**strcmp()**	**strncmp()**
Εύρεση χαρακτήρα σε string	**strchr()**	**strrchr()**
Εύρεση string σε string	**strstr()**	

6.8.1 Η συνάρτηση μήκους αλφαριθμητικού

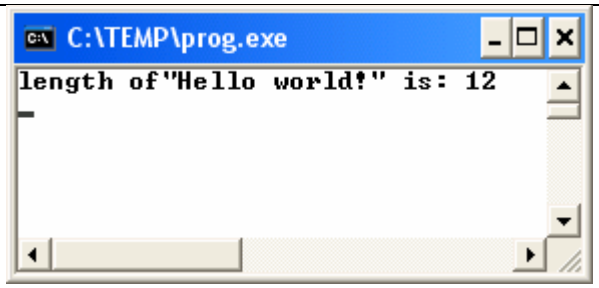
Η συνάρτηση ***strlen()*** επιστρέφει τον αριθμό χαρακτήρων του αλφαριθμητικού, χωρίς να συμπεριλαμβάνει το μηδενικό χαρακτήρα. Το παρακάτω τμήμα κώδικα

```
char name[12] = "abcd";
printf( "%d\n", strlen(name) );
```

θα τυπώσει τον αριθμό των χαρακτήρων του αλφαριθμητικού **name**, δηλαδή 4 κι όχι 12, που είναι ο αριθμός των στοιχείων του πίνακα χαρακτήρων **name**.

Παράδειγμα 6.3

```
#include <stdio.h>
#include <string.h>
void main(
{
      int cnt;
      char msg[20] = { "Hello world!" };
      cnt = strlen(msg);
      printf( "length of\"%s\" is: %d\n",msg,cnt );
}
```



6.8.2 Η συνάρτηση αντιγραφής αλφαριθμητικού

Η συνάρτηση ***strcpy()*** αντιγράφει ένα αλφαριθμητικό από έναν πίνακα σε έναν άλλο. Δέχεται δύο ορίσματα που είναι τα ονόματα των αλφαριθμητικών. ***Το όνομα του πίνακα προορισμού πρέπει να είναι το πρώτο όρισμα, ενώ το δεύτερο όρισμα προσδιορίζει τον πίνακα πηγής***. Στο παρακάτω τμήμα κώδικα

```
char name1[12] = "abcd";
char name2[12] = "ef";
strcpy(name1,name2);
printf( "%s\n", name1 );
```

η πρόταση **strcpy(name1,name2);** αντιγράφει το περιεχόμενο του πίνακα **name2** στον πίνακα **name1**. Έτσι στην οθόνη θα εμφανισθεί το «**ef**».

Για να αντιγραφούν οι πρώτοι *n* χαρακτήρες του **name2** χρησιμοποιείται η σύνταξη

strncpy(name1,name2,n), όπου το τρίτο όρισμα είναι ο αριθμός των προς αντιγραφή χαρακτήρων.

6.8.3 Η συνάρτηση συνένωσης αλφαριθμητικών

Η συνάρτηση ***stcat()*** δέχεται δύο ορίσματα που είναι τα ονόματα των αλφαριθμητικών, τα οποία και συνενώνει. Συγκεκριμένα, ***προσθέτει στο τέλος του αλφαριθμητικού, που προσδιορίζεται από το πρώτο όρισμα, τα στοιχεία του αλφαριθμητικού που προσδιορίζεται από το δεύτερο όρισμα***. Στο παρακάτω τμήμα κώδικα

```
char name1[12] = "abcd";
char name2[12] = "ef";
strcat(name1,name2);
printf( "%sn", name1 );
```

προστίθεται στο τέλος του πίνακα **name1** το περιεχόμενο του πίνακα **name2**. Έτσι στην οθόνη θα εμφανισθεί το «**abcdef**».

Για να προστεθούν οι πρώτοι ***n*** χαρακτήρες του **name2** χρησιμοποιείται η σύνταξη **strncat(name1,name2,n)**, όπου το τρίτο όρισμα είναι ο αριθμός των προστιθέμενων χαρακτήρων.

Παράδειγμα 6.4

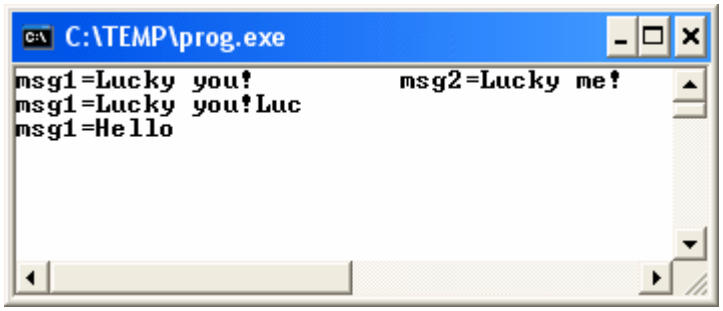
Να περιγραφεί η λειτουργία του ακόλουθου προγράμματος:

```
#include <stdio.h>
#include <string.h>
void main()
{
    char msg1[20] ,msg2[20];
    strcpy( msg1, "Lucky you!" );
    strcpy( msg2, "Lucky me!" );
    printf( "msg1=%s\t\tmsg2=%s\n", msg1,msg2 );
    strncat( msg1,msg2,3 );
    printf( "msg1=%s\n", msg1 );
```

```
        strcpy( msg1,"Hello" );
        printf( "msg1=%s\n", msg1 );
    }
```

Δηλώνονται δύο πίνακες χαρακτήρων 20 θέσεων και με χρήση της συνάρτησης αντιγραφής αλφαριθμητικών τους αποδίδονται τα περιεχόμενα **"Lucky you!"** και **"Lucky me!"**, αντίστοιχα.

Ακολούθως οι τρεις πρώτοι χαρακτήρες του **msg2** προσαρτώνται στο τέλος του **msg1**. Τέλος, στο **msg1** αντιγράφεται το αλφαριθμητικό **"Hello"**.



6.8.4 Η συνάρτηση σύγκρισης αλφαριθμητικών

Η συνάρτηση ***strcmp(name1,name2)*** δέχεται δύο ορίσματα που είναι τα ονόματα των αλφαριθμητικών, τα οποία και συγκρίνει. Η έξοδός της είναι ένας ακέραιος αριθμός, ο οποίος λαμβάνει την τιμή **0** εφόσον τα αλφαριθμητικά είναι όμοια.

Για να συγκριθούν οι πρώτοι ***n*** χαρακτήρες των αλφαριθμητικών χρησιμοποιείται η σύνταξη **strncmp(name1,name2,n)**, όπου το τρίτο όρισμα είναι ο αριθμός των προς σύγκριση χαρακτήρων.

Παράδειγμα 6.5

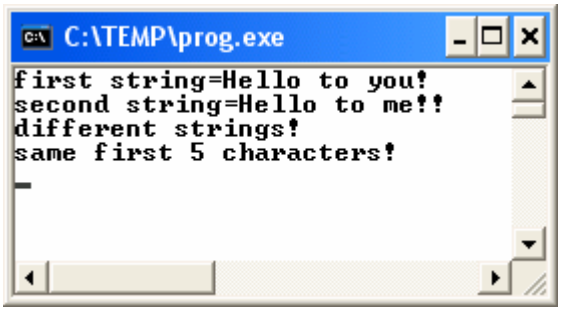
Να περιγραφεί η λειτουργία του ακόλουθου προγράμματος:

```
#include <stdio.h>
#include <string.h>
#define N 5
void main()
```

```
{
    char msg1[81] = {"Hello to you"};
    char msg2[81] = {"Hello to me!"};
    int diff;
    printf( "first string=%s!\n",msg1 );
    printf( "second string=%s!\n",msg2 );
    diff = strcmp(msg1,msg2);
    if(diff==0) printf( "same strings!\n" );
    else printf( "different strings!\n" );
    diff = strncmp(msg1,msg2,N);
    if(diff==0) printf( "same first %d characters!\n",N );
    else printf( "different first %d characters!\n",N );
}
```

Δηλώνονται δύο πίνακες χαρακτήρων 81 θέσεων, στους οποίους αποδίδονται τα περιεχόμενα **"Hello to you"** και **"Hello to me!"**, αντίστοιχα.

Ακολούθως συγκρίνονται τα δύο αλφαριθμητικά με χρήση της ***strcmp()*** και κατόπιν συγκρίνονται οι πέντε πρώτοι χαρακτήρες τους με χρήση της ***strncmp()***.



Κεφάλαιο 7

ΔΗΜΙΟΥΡΓΙΑ ΠΡΟΓΡΑΜΜΑΤΩΝ ΠΑΡΑΔΕΙΓΜΑΤΑ

7.1 Γενικά

Στο παρόν κεφάλαιο θα χρησιμοποιηθεί το σύνολο των στοιχείων της C, που παρατέθηκαν στα προηγούμενα κεφάλαια, με άμεσο στόχο την υλοποίηση απλών προβλημάτων και απώτερο την εισαγωγή στη λογική της προγραμματιστικής ανάλυσης. Επιπρόσθετα, τα παραδείγματα που θα αναπτυχθούν στη συνέχεια αποτελούν μία ενδεικτική αναφορά για τον αναγνώστη, καθώς αποτελούν επιλεγμένα θέματα εξετάσεων.

Πρόβλημα 7.1

Να γραφεί πρόγραμμα, το οποίο να δέχεται ως είσοδο κείμενο, να απαριθμεί τις εμφανίσεις των ψηφίων 0-9, τα λευκά διαστήματα και τους υπόλοιπους χαρακτήρες και στη συνέχεια να τυπώνει τα αποτελέσματα. Το πρόγραμμα ολοκληρώνεται όταν δοθεί ο χαρακτήρας τελεία '**.**'.

Το παραπάνω πρόβλημα μπορεί να μορφοποιηθεί σε δομημένα Ελληνικά ως εξής:

Για κάθε χαρακτήρα του κειμένου που είναι διάφορος της '. '
  ελέγχεται ο τύπος του χαρακτήρα
  αν είναι ένας από τους ' ', '\t', '\n'
    αυξάνεται κατά μία μονάδα ο απαριθμητής των διαστημάτων
  αν είναι ψηφίο
    αυξάνεται κατά μία μονάδα ο απαριθμητής που αντιστοιχεί στο ψηφίο

σε κάθε άλλη περίπτωση

αυξάνεται κατά μία μονάδα ο απαριθμητής των υπόλοιπων χαρακτήρων

Αναπαράσταση δεδομένων:

- Όσον αφορά στις μεταβλητές, απαιτείται ένας απαριθμητής για τα διαστήματα, τον οποίο ονομάζουμε **n_white**, ένας απαριθμητής για τους λοιπούς χαρακτήρες, ο **n_other**, και δέκα απαριθμητές για τα ψηφία. Για την τελευταία περίπτωση μπορούμε να δηλώσουμε δέκα ανεξάρτητες μεταβλητές αλλά είναι προτιμότερο να επιλεχθεί ένας πίνακας δέκα θέσεων, όπως ο **int n_digit[10]**, ο οποίος οδηγεί σε πιο συμπαγή και δομημένο κώδικα.

- Για την εκτύπωση των αριθμών των εμφανίσεων των δέκα ψηφίων, στην περίπτωση του πίνακα απαριθμητών, ο αντίστοιχος κώδικας θα έχει τη μορφή

```
for (i=0;i<10;i++)
    printf( "Digit %d has been encountered \t %d \t times\n",i,n_n_digit[i] );
```

Αναπαράσταση διεργασιών:

- Για τη διεργασία «πάρε χαρακτήρα» χρησιμοποιείται η συνάρτηση ***getchar***, η οποία επιστρέφει το χαρακτήρα που διαβάζει από την κύρια είσοδο. Αυτός ο χαρακτήρας πρέπει να αποθηκευθεί σε μία μεταβλητή τύπου χαρακτήρα για περαιτέρω επεξεργασία.

Τα παραπάνω οδηγούν στη δήλωση

```
char ch;
```

και στην έκφραση

```
ch=getchar();
```

η τιμή της οποίας είναι η τιμή του αριστερού τελεστέου της έκφρασης.

- Για την ανίχνευση του τέλους της εισαγωγής χαρακτήρων χρησιμοποιούμε το συσχετιστικό τελεστή **!=** οδηγούμαστε στην έκφραση

```
(ch=getchar())!='.'
```

Η έκφραση γίνεται ψευδής όταν αναγνωσθεί ο χαρακτήρας τελεία. Μπορεί επομένως να χρησιμοποιηθεί ως έκφραση μίας πρότασης ***while***, που θα οδηγεί στην επανάληψη του συνόλου των ενεργειών που το πρόγραμμα πρέπει να εκτελεί για κάθε χαρακτήρα.

Διαμόρφωση της ροής ελέγχου:

Με βάση τα προηγούμενα, η περιγραφή διαμορφώνεται ως εξής:

while ((ch=getchar())!='.')
{
ελέγχεται ο τύπος του χαρακτήρα
αν είναι ένας από τους ' ', '\t', '\n'
αυξάνεται κατά μία μονάδα ο απαριθμητής των διαστημάτων
αν είναι ψηφίο
αυξάνεται κατά μία μονάδα ο απαριθμητής που αντιστοιχεί στο ψηφίο
σε κάθε άλλη περίπτωση
αυξάνεται κατά μία μονάδα ο απαριθμητής των υπόλοιπων χαρακτήρων
}

Το σώμα της ***while*** αποτελεί κλασική περίπτωση επιλογής από αμοιβαία αποκλειόμενες ενέργειες, γεγονός που οδηγεί στη χρήση της πρότασης ***switch***. Η έκφραση ανάλογα με την τιμή της οποίας θα γίνει η επιλογή της κατάλληλης ενέργειας είναι η απλή έκφραση **ch**.

Α) **εάν είναι ένας από τους ' ', '\t', '\n'**
αυξάνεται κατά μία μονάδα ο απαριθμητής n_white

ή εκφρασμένη στη C

case ' ':
case '\t':
case '\n':
n_white++;
break;

Β) **εάν ο χαρακτήρας είναι ψηφίο**
αυξάνεται κατά μία μονάδα ο απαριθμητής που αντιστοιχεί στο ψηφίο

μία πρώτη μορφή του κώδικα είναι η παρακάτω

```
case '0':
    n_digit++;
break;
. . . . . .
case '9':
    n_white++;
break;
```

Ο κώδικας αυτός δεν εκμεταλλεύεται τη δήλωση των απαριθμητών των ψηφίων ως πίνακα χαρακτήρων. Για το λόγο αυτό, θα προσπαθήσουμε να ενοποιήσουμε τα ***case*** ώστε να έχουν μία παραμετρική πρόταση, που σε κάθε περίπτωση θα οδηγεί στην αύξηση του κατάλληλου απαριθμητή. Θεωρούμε την έκφραση

ch-'o'

και εξετάζουμε την τιμή της για τιμές τής **ch** από το σύνολο {'0', '1', …, '9'}. Είναι προφανές ότι, εάν το **ch** είναι **'0'**, η έκφραση έχει τιμή 0 οπότε και η πρόταση

n_digit[ch-'0']++;

έχει ως αποτέλεσμα την αύξηση του απαριθμητή που αντιστοιχεί στο ψηφίο **'0'**.

Αντίστοιχα, η παραπάνω πρόταση για **ch='8'** θα αυξήσει τον απαριθμητή που αντιστοιχεί στο **'0'**. Κατά αυτόν τον τρόπο οδηγούμαστε στον ακόλουθο συμπαγή κώδικα:

```
case '0':
case '1':
. . . . . ....
case '9':
    n_digit[ch-'0']++;
break;
```

Γ) Για τα υπόλοιπα στοιχεία έχουμε την κλασική περίπτωση χρήσης της εντολής ***default***, οπότε προκύπτει:

```
default:
    n_other++;
break;
```

Το τελευταίο σημείο που πρέπει να προσεχθεί είναι η αρχικοποίηση των μεταβλητών.

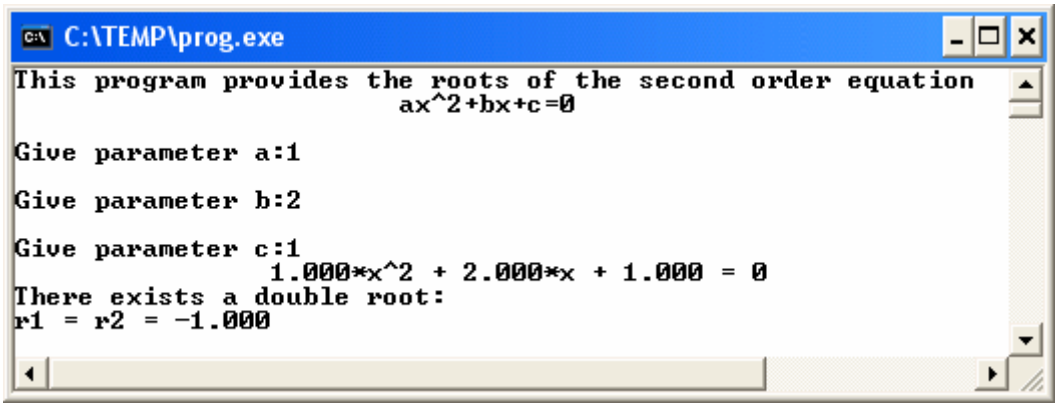
Πρόβλημα 7.2

Να γραφεί πρόγραμμα που επιλύει δευτεροβάθμιες εξισώσεις. Να δέχεται ως είσοδο τους συντελεστές της εξίσωσης και να ελέγχει το είδος των ριζών (απλές πραγματικές, συζυγείς μιγαδικές ή διπλή πραγματική). Θεωρείστε την περίπτωση πραγματικών συντελεστών.

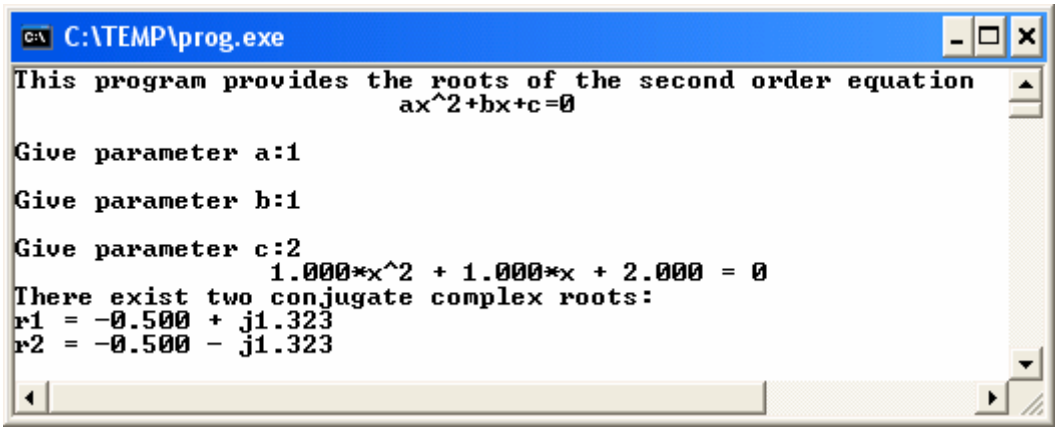
```
#include<stdio.h>
#include<conio.h>
#include<math.h>          // για sqrt(), fabs()
void main()
{
     float a,b,c, D, r1,r2, r,im;
     printf( "This program provides the roots of the second order
              equation\n" );
     printf( "\t\t\tax^2+bx+c=0\n" );
     printf( "\nGive parameter a:");     scanf("%f",&a );
     if (a==0) // Έλεγχος για α=0, οπότε η εξίσωση γίνεται α/θμια
     {
          printf( "For a 2nd order equation, a!=0. Try again\n" );
          printf( "\nGive parameter a:");     scanf("%f",&a );
     }  //τέλος του if
     printf( "\nGive parameter b:");     scanf("%f",&b );
     printf( "\nGive parameter c:");     scanf("%f",&c );
     printf( "\t\t%.3f*x^2 + %.3f*x + %.3f = 0\n",a,b,c ) ;

     D = b*b-4*a*c;    // Διακρίνουσα
     if(D<0)  // Εάν Δ<0, οι ρίζες είναι συζυγείς μιγαδικές
     {
```

```
            printf( "There exist two conjugate complex roots:\n" );
            r=-b/(2*a);
            im=sqrt(-D)/(2*a);
            printf( "r1 = %.3f + j%.3f\n",r,im );
            printf( "r2 = %.3f - j%.3f",r,im );
        }  //τέλος του if
        else if (fabs(D)<1e-10)    // fabs → float,   abs → integer
        {                    // Εάν Δ=0, υπάρχει διπλή ρίζα
            printf( "There exists a double root:\n" );
            printf( "r1 = r2 = %.3f\n", -b/(2*a) );
        } //τέλος του else if
        else // Σε κάθε άλλη περίπτωση υπάρχουν δύο πραγματικές ρίζες
        {
            printf( "There exist two real roots:\n" );
            r1=(-b+sqrt(D))/(2*a);
            r2=(-b-sqrt(D))/(2*a);
            printf("r1=%.3f\nr2=%.3f\n",r1,r2 );
        } //τέλος της else
    } //τέλος της main
```

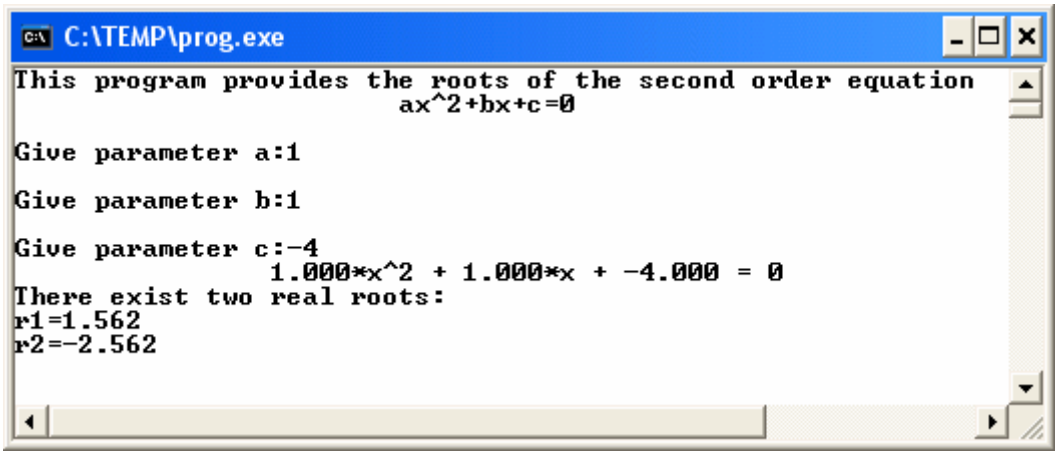
Αποτέλεσμα για διπλή πραγματική ρίζα:



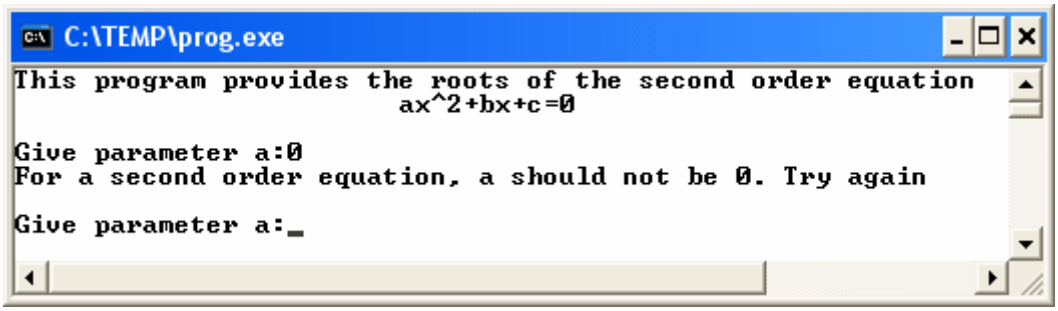
Αποτέλεσμα για συζυγείς μιγαδικές ρίζες:



Αποτέλεσμα για απλές πραγματικές ρίζες:



Αποτέλεσμα για α=0:



Πρόβλημα 7.3

Να γραφεί πρόγραμμα, χωρίς χρήση της εντολής ***goto***, με το οποίο θα εισάγονται από το πληκτρολόγιο δύο ζεύγη πραγματικών αριθμών ($x1$,$y1$), ($x2$,$y2$). Από τα ζεύγη αυτά θα υπολογίζονται οι συντελεστές a, b της εξίσωσης της ευθείας $y=ax+b$. Θα πρέπει να λαμβάνεται πρόνοια ώστε σε περίπτωση που $x2=x1$ να ζητείται εκ νέου το σημείο ($x2$,$y2$).

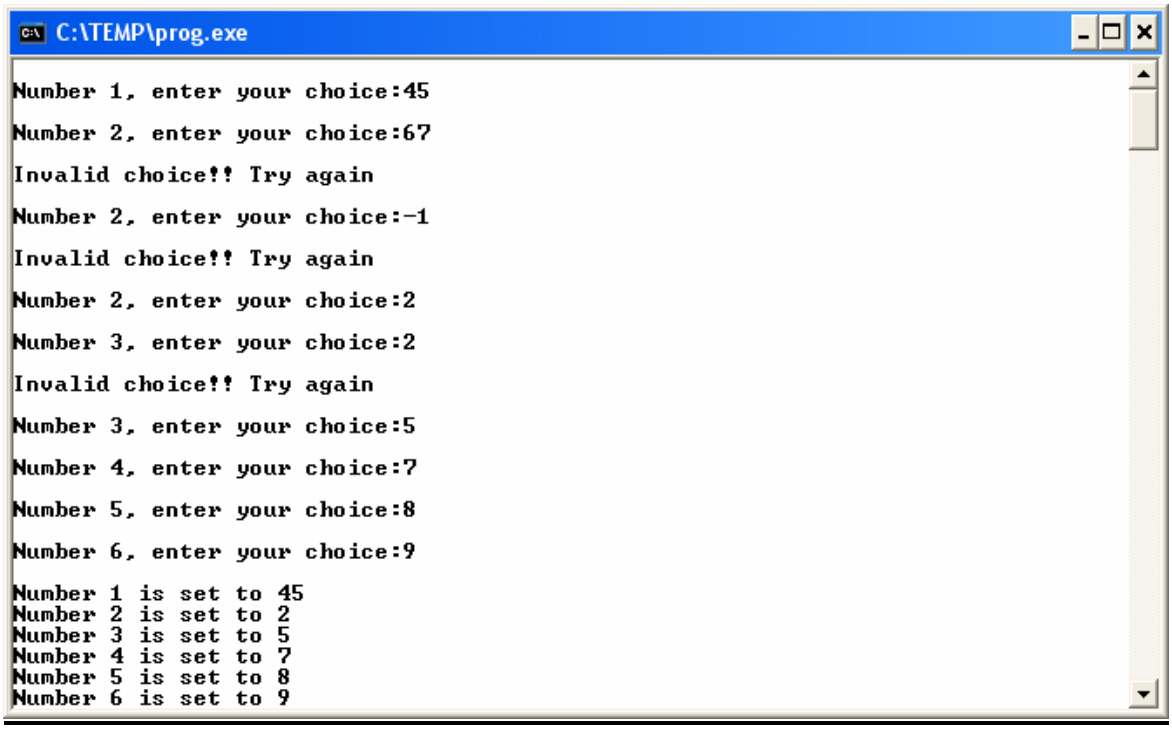
Οι συντελεστές *a*, *b* να εμφανίζονται στην οθόνη.

```
#include <stdio.h>
void main()
{
    float x1,y1,x2,y2,a,b;
    printf( "\nGive x1:"); scanf( "%f",&x1 );
    printf( "\nGive y1:"); scanf( "%f",&y1 );
    printf( "\nGive x2:"); scanf( "%f",&x2 );
    while (x2==x1)
    {
        printf( "n\ERROR! x2=x1, give x2 again, x2:");
        scanf( "%f",&x2 );
    }
    printf( "\nGive y2:"); scanf( "%f",&y2 );
    a=(y2-y1)/(x2-x1);
    b=y2-x2*a;
    printf( "y=ax+b, a=%f, b=%f\n",a,b );
}  //τέλος της main
```

Πρόβλημα 7.4

Να γραφεί πρόγραμμα, το οποίο θα δέχεται από το πληκτρολόγιο διαδοχικά 6 ακέραιους αριθμούς του ΛΟΤΤΟ. Θα πρέπει να λαμβάνεται πρόνοια ώστε όταν είτε ένας αριθμός δεν ανήκει στο [1 49] είτε επαναλαμβάνεται αριθμός που δόθηκε προηγουμένως, να ζητείται νέα τιμή γι' αυτόν. Οι αριθμοί θα αποθηκεύονται σε πίνακα ακεραίων και θα εμφανίζονται στην οθόνη μετά το πέρας της εισαγωγής τους.

```
#include <stdio.h>
#define N 6
void main()
{
      int lotto[6];
      int j,deikt,i=0;
      while (i<N)
      {
            deikt=0;
            printf( "\nNumber %d, enter your choice:",i+1 );
            scanf( "%d",&lotto[i] );
            if ((lotto[i]<1) || (lotto[i]>49)) deikt++;   // [1,49]
            for (j=0;j<i;j++) if (lotto[j]==lotto[i]) deikt++;  //Να μην επαναληφθεί
                                                                //προηγούμενος αριθμός
            if (deikt) printf( "\nInvalid choice!! Try again\n" );
            else i++;
      }  //τέλος της while
      for (i=0;i<N;i++) printf( "\nNumber %d is set to %d",i+1,lotto[i] );
}  //τέλος της main
```

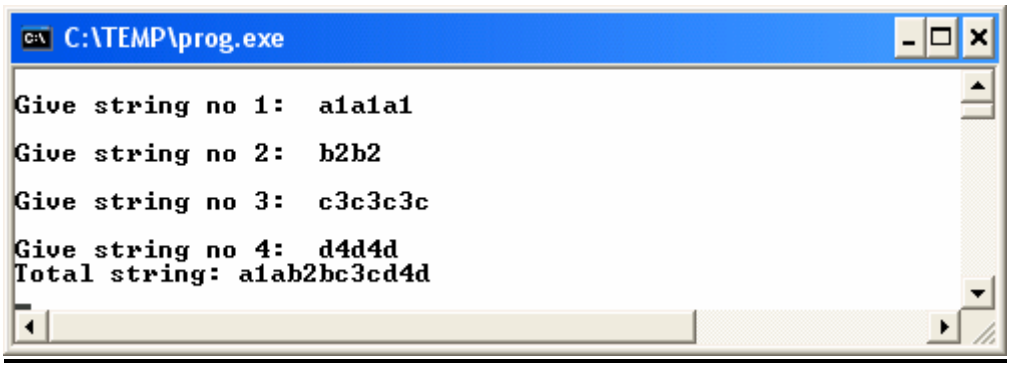


Πρόβλημα 7.5

Να γραφεί πρόγραμμα, με το οποίο θα εισάγονται από το πληκτρολόγιο 4 αλφαριθμητικά σε πίνακα αλφαριθμητικών, μήκους 7 χαρακτήρων το καθένα. Στη συνέχεια θα λαμβάνονται οι τρεις πρώτοι χαρακτήρες κάθε αλφαριθμητικού και θα συνενώνονται σε ένα νέο αλφαριθμητικό, το οποίο και θα τυπώνεται.

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
void main()
{
    int i;
    char str[4][8], str_total[13];     //στις διαστάσεις των πινάκων θα πρέπει να
                                       //ληφθεί υπόψη και ο '\0'
    for (i=0;i<4;i++)
    {
```

```
            printf( "\nGive string no %d:  ",i+1 );
            scanf( "%s",str[i] );
            if (i==0) strncpy(str_total,str[i],3);
            else strncat(str_total,str[i],3);
        }
        printf( "Total string: %s\n",str_total );
} //τέλος της main
```

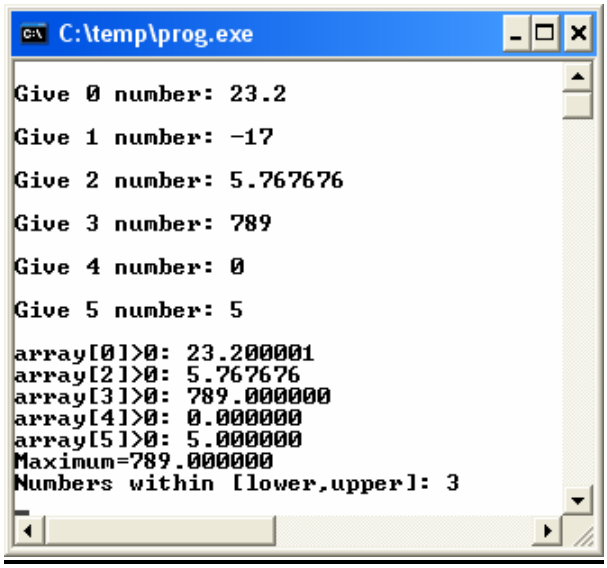


Πρόβλημα 7.6

Να γραφεί πρόγραμμα με το οποίο θα εισάγονται 6 πραγματικοί αριθμοί από το πληκτρολόγιο, θα αποθηκεύονται στον πίνακα **array[]** και θα τυπώνονται: α) οι θετικοί εξ αυτών, β) ο μεγαλύτερος, γ) ο αριθμός των στοιχείων του **array[]**, τα οποία έχουν τιμές στο διάστημα [1.05 50.8].

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define N 6
#define lower 1.05
#define upper 50.8
void main()
```

```
{
    float array[N],maxim;
    int i,count=0;
    for (i=0;i<N;i++)
    {
        printf( "\nGive %d number: ",i );
        scanf( "%f",&array[i] );
    }
    maxim=array[0];
    printf( "\n" );
    for (i=0;i<N;i++)    /* i=0 για να μπουν όλα σε ένα βρόχο, μόνο για το
                            μέγιστο θα ήταν i=1 */
    {
        if (array[i]==fabs(array[i])) printf( "array[%d]>0: %f\n",i,array[i] );
        if (array[i]>maxim) maxim=array[i];
        if ((array[i]>=lower) && (array[i]<=upper)) count++;
    }
    printf( "Maximum=%f\n",maxim );
    printf( "Numbers within [lower,upper]: %d\n",count );
} //τέλος της main
```



Πρόβλημα 7.7

Να γραφεί πρόγραμμα με το οποίο θα εισάγονται 6 χαρακτήρες από το πληκτρολόγιο, θα αποθηκεύονται στον πίνακα **array[]** και θα τυπώνονται διαδοχικά τα ακόλουθα:

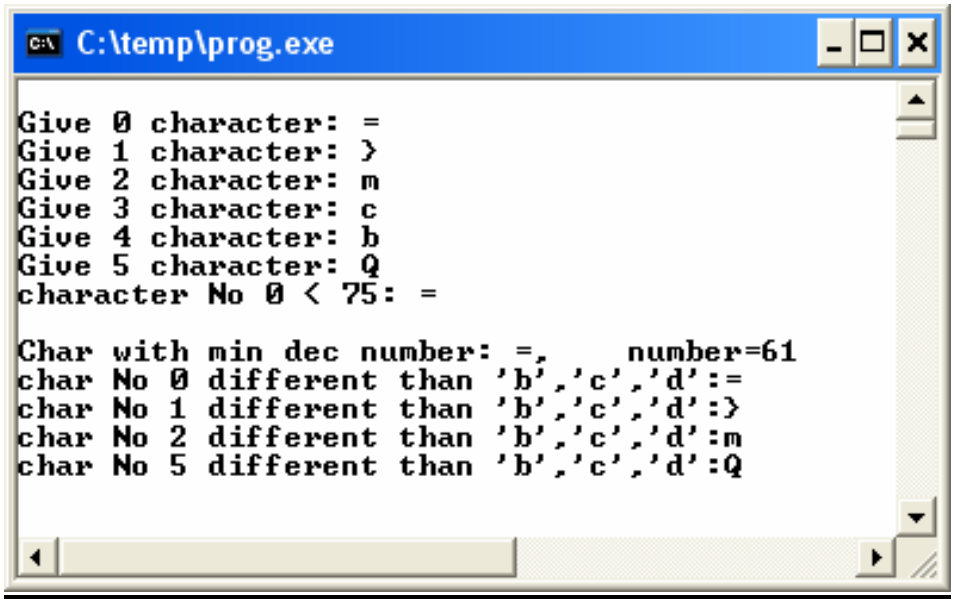
α) οι χαρακτήρες με δεκαδικό ισοδύναμο μικρότερο του 75

β) ο χαρακτήρας με το μικρότερο δεκαδικό ισοδύναμο

γ) όσοι χαρακτήρες είναι διάφοροι των χαρακτήρων '**b**', '**c**', '**d**' (υλοποίηση αποκλειστικά με χρήση της εντολής ***switch-case***)

```
#include <stdio.h>
#include <conio.h>
#define N 6
void main()
{
      char array[N];
      int minim,i;
      for (i=0;i<N;i++)
      {
            printf( "\nGive %d character: ",i );
            array[i]=getche();
      }
      printf( "\n" );
      minim=array[0];
      for (i=0;i<N;i++)
      {
            if (array[i]<75) printf("character No %d < 75: %c\n",i,array[i] );
            if (array[i]<minim) minim=array[i];
      }
      printf( "Char with min dec number: %c, number=%d\n",minim,minim);
      for (i=0;i<N;i++)
```

```
        {
            switch (array[i])
            {
                case 'b':
                case 'c':
                case 'd':
                break;
                default:
                  printf("char No %d different than 'b','c','d':%c\n",i,array[i]);
                break;
            } //τέλος της switch
        } // τέλος της for
    } //τέλος της main
```



Πρόβλημα 7.8

Να γραφεί πρόγραμμα, το οποίο θα υπολογίζει τα εμβαδά του τριγώνου, του τραπεζίου και του κύκλου, μετά από επιλογή του χρήστη. Συγκεκριμένα:

α) Θα ζητά από το χρήστη να επιλέξει το γεωμετρικό σχήμα για το οποίο θα εξαχθεί το εμβαδόν, αντιστοιχώντας την επιλογή κάθε σχήματος σε κάποιον ακέραιο και κάνοντας έλεγχο λανθασμένης απάντησης (απαιτείται βρόχος για τον έλεγχο).

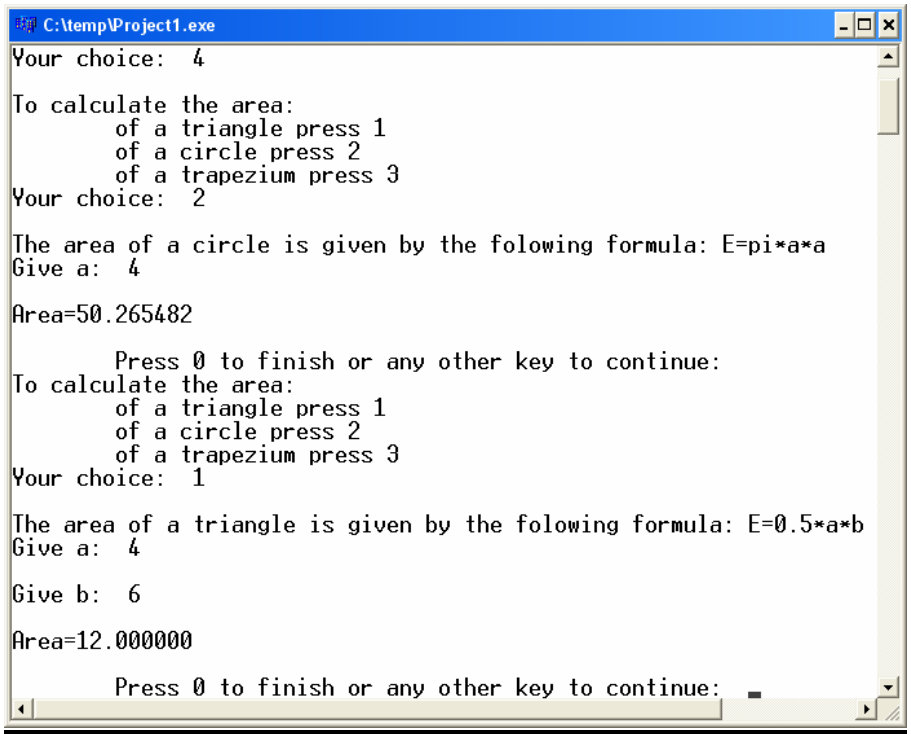
β) Για το επιλεγέν σχήμα θα ζητά από το χρήστη τις απαιτούμενες διαστάσεις και θα τυπώνει το αποτέλεσμα στην οθόνη.

γ) Η διαδικασία θα επαναλαμβάνεται όσες φορές επιθυμεί ο χρήστης, με κριτήριο τερματισμού τον ακέραιο **0**.

Όπου απαιτηθούν προτάσεις διακλάδωσης, αυτές να υλοποιηθούν **<u>αποκλειστικά</u>** με χρήση των ***switch-case*.**

```
#include <stdio.h>
#include <math.h> // για το M_PI
void main()
{
    char sel2;
    int select;
    float a,b,c;
    do
    {
        do
        {
            printf("\nTo calculate the area:");
            printf("\n\tof a triangle press 1\n\tof a circle press
            printf("\n\tof a trapezium press 3");
            printf("\nYour choice:  ");
            scanf("%d",&select);
        } while ((select!=1) && (select!=2) && (select!=3));
```

```
            switch (select)
            {
                case 1:
                    printf("\nThe area of a triangle is given by");
                    printf("the following formula: E=0.5*a*b\nGive a:  ");
                    scanf("%f",&a);
                    printf("\nGive b:  ");
                    scanf("%f",&b);
                    printf("\nArea=%f\n",0.5*a*b);
                break;
                case 2:
                    printf("\nThe area of a circle is given by");
                    printf("the following formula: E=pi*a*a\nGive a:  ");
                    scanf("%f",&a);
                    printf("\nArea=%f\n",M_PI*a*a);
                break;
                default:
                    printf("\nThe area of a trapezium is given by");
                    printf("the following formula: E=0.5*(a+b)*c\nGive a:  ");
                    scanf("%f",&a);
                    printf("\nGive b:  ");
                    scanf("%f",&b);
                    printf("\nGive c:  ");
                    scanf("%f",&c);
                    printf("\nArea=%f\n",0.5*(a+b)*c);
                break;
            }
            printf("\n\tPress 0 to finish or any other key to continue:  ");
            sel2=getch();
        } while (sel2!='0');
    }
```



Πρόβλημα 7.9

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

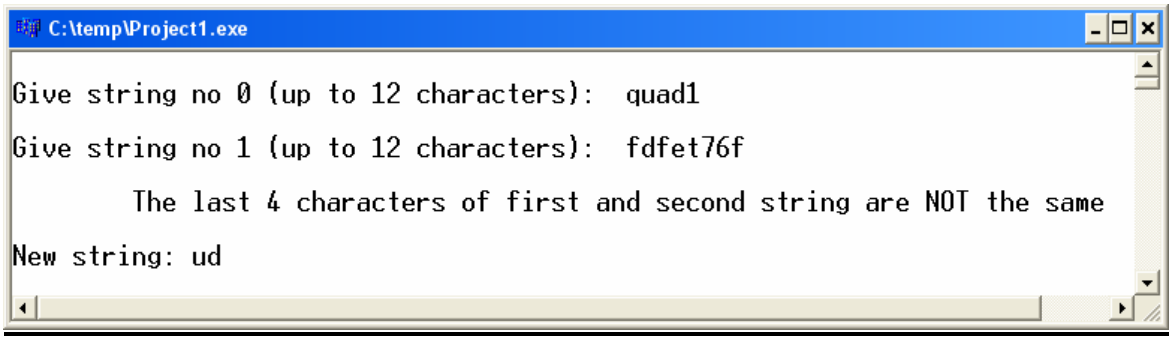
α) Θα δέχεται από το πληκτρολόγιο δύο αλφαριθμητικά (μέγιστου μήκους 12) και θα τα αποθηκεύει σε πίνακα 2 θέσεων.

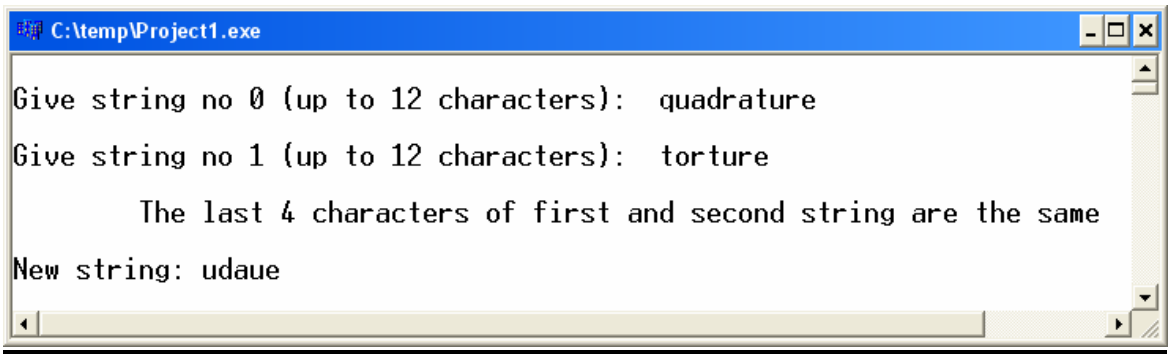
β) Θα ελέγχει εάν οι τελευταίοι 4 χαρακτήρες του πρώτου αλφαριθμητικού είναι ίδιοι με τους τελευταίους 4 χαρακτήρες του δεύτερου αλφαριθμητικού (ελέγχοντας πρώτα εάν τα αναγνωσθέντα αλφαριθμητικά έχουν μήκος μεγαλύτερο ή ίσο του 4).

γ) Θα αντιγράφει σε νέο πίνακα χαρακτήρων κατάλληλου μήκους τους χαρακτήρες του πρώτου αλφαριθμητικού **που βρίσκονται σε άρτια θέση** και θα τυπώνει το νέο πίνακα χαρακτήρων στην οθόνη **ως αλφαριθμητικό**.

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
void main()
{
```

```
        int i,j,length[2],sum;
        char str[2][13],str1[7];
        //A
        for (i=0;i<2;i++)
        {
            printf("\nGive string no %d (up to 12 characters):  ",i);
            scanf("%s",str[i]);
            length[i]=strlen(str[i]);
        }
        //B
        if ((length[0]<4) || (length[1]<4))
          printf("\nError!! Strings should have more at least 4 characters!!
Program aborted");
        else
        {
            sum=0;
            for (i=0;i<4;i++)
            if (str[0][strlen(str[0])-i]==str[1][strlen(str[1])-i]) sum++;
            if (sum==4)
               printf("\n\tThe last 4 characters of first and second string are the
same\n");
            else
            {
              printf("\n\tThe last 4 characters of first and second string");
              printf(" are NOT the same\n");
            }
        }
        //C
        for (i=1;i<strlen(str[0]);i=i+2)  str1[i/2]=str[0][i];
        str1[i/2]='\0'; //για να καταστεί αλφαριθμητικό το str1
        printf("\nNew string: %s",str1);
    }
```



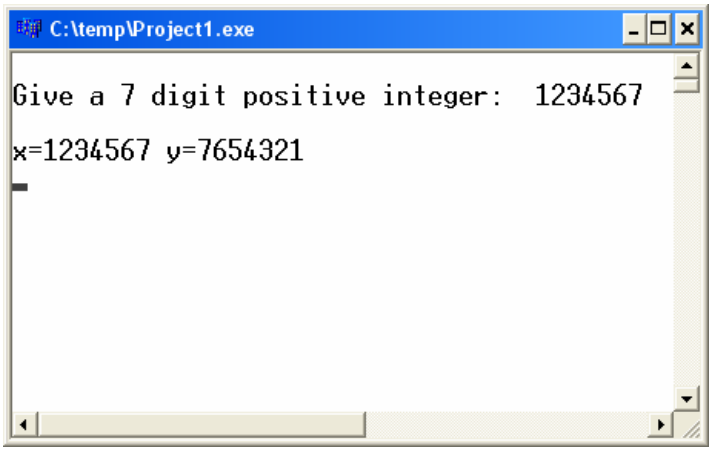


Πρόβλημα 7.10

Να γραφεί πρόγραμμα, το οποίο θα δέχεται από το πληκτρολόγιο ένα θετικό ακέραιο αριθμό N ψηφίων $x = x_{n-1}x_{n-2}...x_1x_0$, θα αποθηκεύει τα ψηφία του $x_k, k = 0,1,...,n-1$ σε πίνακα και θα εμφανίζει στην οθόνη τον αριθμό με αντεστραμμένα τα ψηφία του, δηλαδή τον αριθμό $y = x_0x_1...x_{n-2}x_{n-1}$.

```
#include <stdio.h>
#define N 7
void main()
{
    int x,y,arr[N],z[N],i;
    printf("\nGive a %d digit positive integer:  ",N);
    scanf("%d",&x);
    z[0]=1;
    for (i=1;i<N;i++) z[i]=z[i-1]*10;
    y=x;
```

```
        for (i=(N-1);i>=1;i--)
        {
            arr[i]=y/z[i];
            y=y%z[i];
        }
        arr[0]=y;
        y=arr[N-1];
        for (i=1;i<N;i++) y=y+z[i]*arr[N-(i+1)];
        printf("\nx=%d y=%d\n",x,y);
    }
```



Κεφάλαιο 8

ΔΟΜΕΣ
ΑΠΑΡΙΘΜΗΤΙΚΟΙ ΤΥΠΟΙ ΔΕΔΟΜΕΝΩΝ

8.1 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;

Παρατηρήσεις:

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

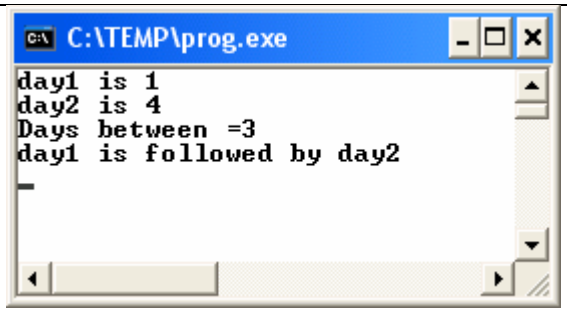
#define True 1

3. Οι μεταβλητές απαριθμητικού τύπου παρουσιάζουν το μειονέκτημα ότι δεν μπορούν να χρησιμοποιηθούν άμεσα στις συναρτήσεις εισόδου–εξόδου (***scanf()***, ***printf()***). Οι συναρτήσεις αυτές τυπώνουν τον ακέραιο στον οποίο αντιστοιχεί το συμβολικό όνομα, όπως φαίνεται στο Παράδειγμα 8.1.

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 8.1

```
#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" );
}
```



8.2 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**.

8.3 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 πριν τον ορισμό της δομής.
- Πίνακες.
- Άλλες δομές, οι οποίες θα πρέπει να έχουν ορισθεί πριν τον ορισμό της δομής.

Ο ορισμός της δομής έχει την ακόλουθη μορφή:

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

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

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;

[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 το χρήστη.

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
```

```
{
    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 8.2

Μία επιχείρηση πώλησης αυτοκινήτων διαθέτει τα ακόλουθα αυτοκίνητα:

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];
    . . . . . .
}
```

8.4 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 }
};
```

8.5 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"

Παρατήρηση: 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.

8.6 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]

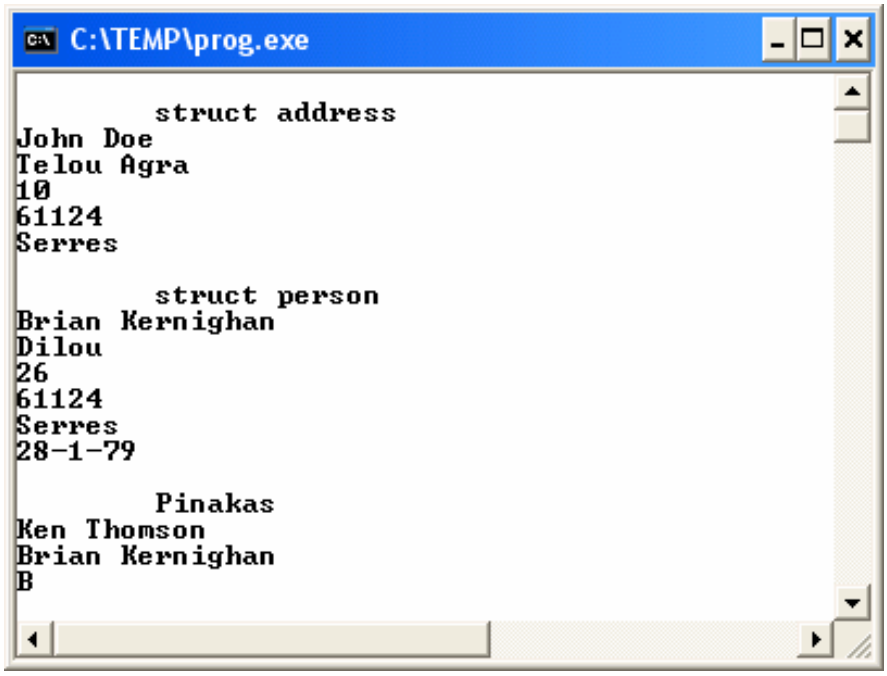
αντίστοιχα.

Παράδειγμα 8.3

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 8.4

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:

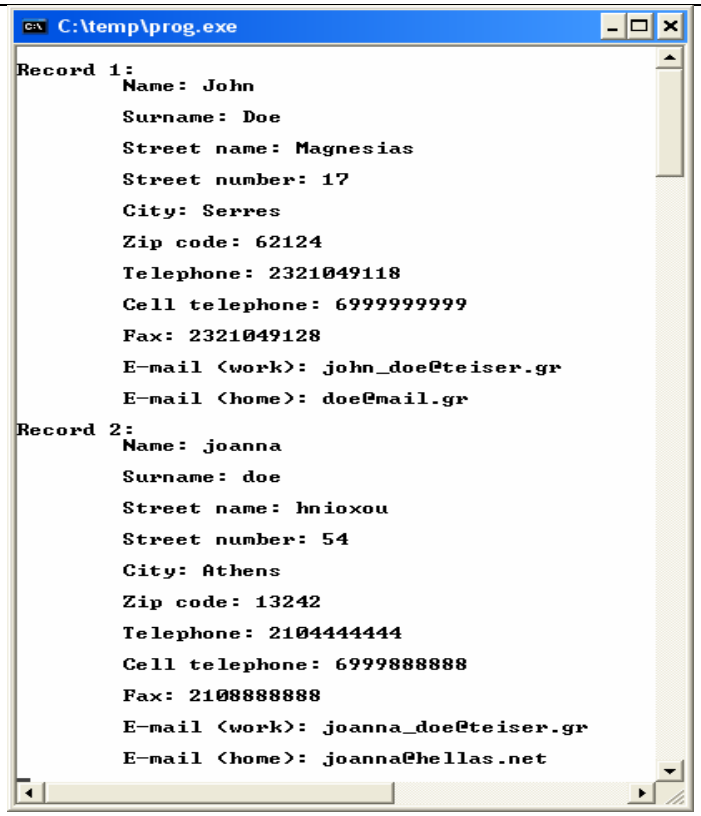
```
struct teleT
{
    char wr_no[ ];     // σταθερό τηλέφωνο
    char cell_no[ ];   // κινητό τηλέφωνο
    char fax_no[ ];
} ;
```

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
```



Κεφάλαιο 9

ΣΥΝΑΡΤΗΣΕΙΣ

9.1 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

9.2 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 9.1 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()
 {
      . . . . . .
 }
```

Σχ. 9.1 *Γενική μορφή προγράμματος στη C*

Μία συνάρτηση περιλαμβάνει τρεις φάσεις:

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

9.2.1 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**).

9.2.2 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()***.

9.2.3 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 9.1. Ωστόσο αυτή είναι μία ριψοκίνδυνη τακτική και θα πρέπει να αποφεύγεται.

Παράδειγμα 9.1

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 9.2

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 9.3

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) );
}
```

9.3 Είδη και εμβέλεια μεταβλητών

9.3.1 Τοπικές μεταβλητές

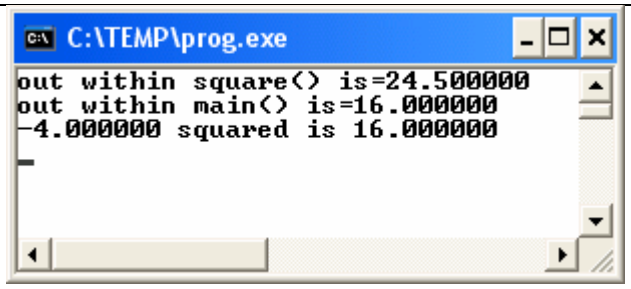
Στο παράδειγμα 9.2 η συνάρτηση ***F_to_C()*** δημιουργεί στον κορμό της τη μεταβλητή **ratio**. Η μεταβλητή αυτή είναι ενεργή μόνο μέσα στο τμήμα κώδικα στο οποίο ορίζεται (σώμα της συνάρτησης) και παύει να υφίσταται μετά το πέρας της συνάρτησης. Μεταβλητές τέτοιου είδους καλούνται **τοπικές μεταβλητές** (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 9.4

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);
}
```



9.3.2 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.

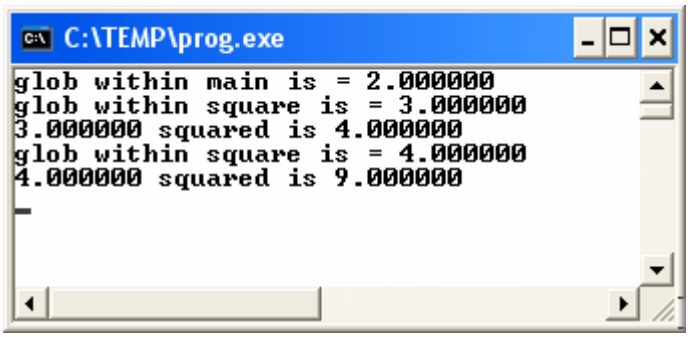
Παράδειγμα 9.5

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**.



9.3.3 Εμβέλεια μεταβλητών

Στις §9.3.1 και 9.3.2 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 το όνομα με τη μικρότερη εμβέλεια αποκρύπτει το όνομα με τη μεγαλύτερη.

Παράδειγμα 9.6

Να προσδιορισθεί η εμβέλεια των μεταβλητών στον ακόλουθο πηγαίο κώδικα:

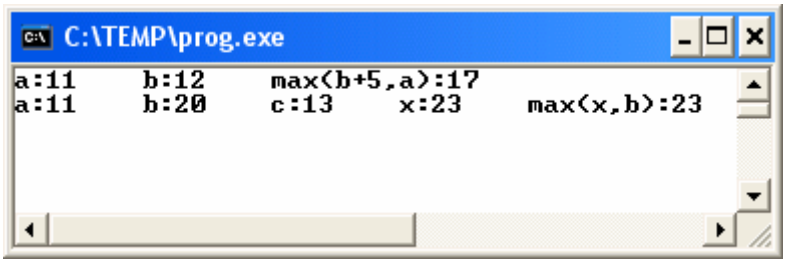
```
1   #include <stdio.h>
2   int max(int a, int b);
3   static void func(int x);
4   int a; static int b;
5   void main()  {
6         b=12; a=b--;
7         printf( "a:%d\tb:%d\tmax(b+5,a):%d\n",a,b,max(b+5,a) );
8         func(a+b);
9   }     // τέλος της main
10  int c=13;
11  int max(int a, int 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()***.



9.3.4 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 9.7

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

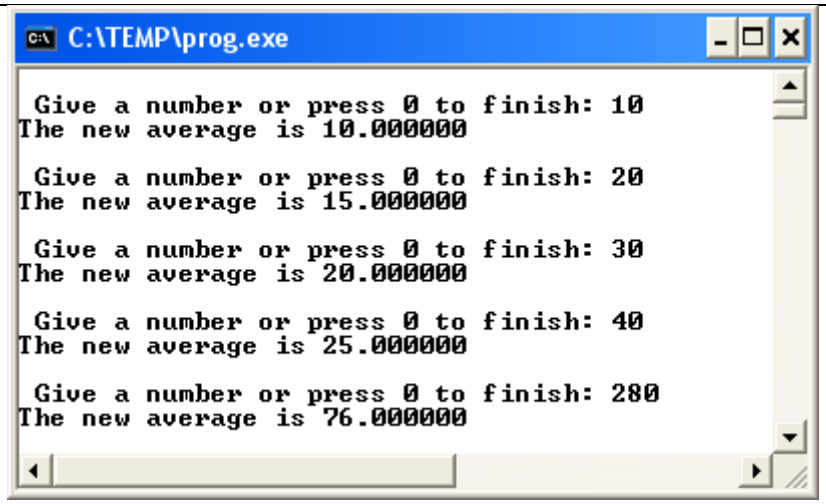
είσοδο στο πρόγραμμα. Αντίθετα η **num** ως τοπική μεταβλητή περιορισμένης διάρκειας αρχικοποιείται σε κάθε ενεργοποίηση της συνάρτησης **func**.

Παράδειγμα 9.8

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
```



9.4 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[ ])
```

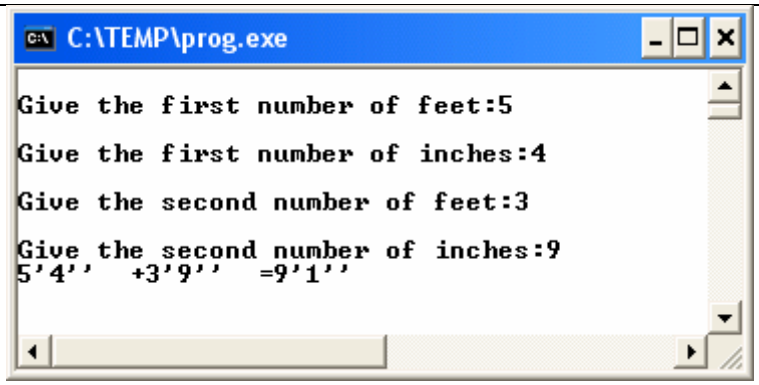
```
{
    int i;
    for (i=0; i<=9; i++)
    printf( "%d ",num[i] );
}
```

Παράδειγμα 9.9

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
```



9.5 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
int sum(int n)
{
    int i, total=0;
    for (i=0; i<=n; i++)
        total+=i;
    return(total);
}
```

```
// Με αναδρομή
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)

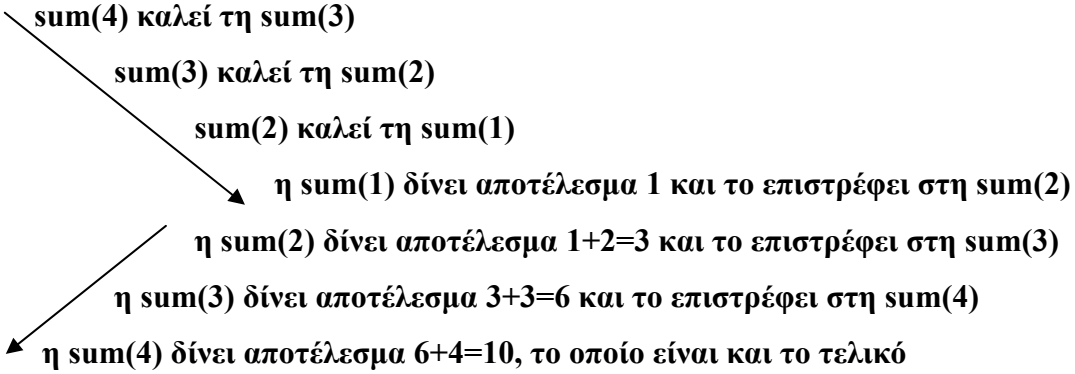
(1+…+(n-2)) = (1+…+(n-3)) + (n-2)

(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.

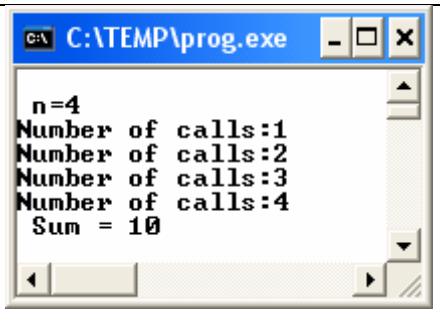
Την πρώτη φορά καλείται η **sum**() με όρισμα **n**. Με την πρόταση **return(sum(n-1)+n)** η συνάρτηση **sum**() καλεί τον ίδιο της τον εαυτό με διαφορετικό όμως όρισμα **(n-1)**. Η ενεργοποίηση αυτή θα προκαλέσει με τη σειρά της νέα ενεργοποίηση και αυτό θα συνεχισθεί έως ότου προκληθεί διακοπή. Η διακοπή 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:



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.

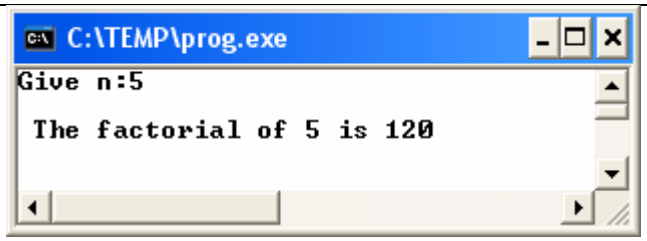
Παράδειγμα 9.10

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
```



Κεφάλαιο 10

ΔΕΙΚΤΕΣ

10.1 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.

10.2 Δήλωση δείκτη

Η δήλωση μίας μεταβλητής δείκτη ακολουθεί τον εξής φορμαλισμό:

<τύπος_δεδομένων> * <όνομα_δείκτη>;

π.χ.

int *pnum;

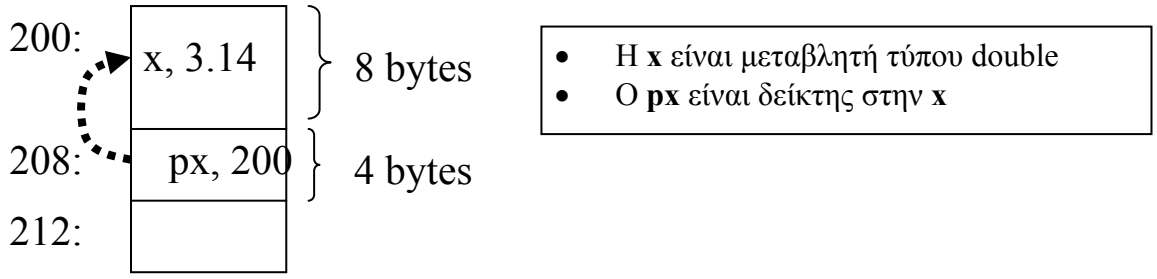
- Όταν ένας δείκτης έχει αποθηκευμένη μία διεύθυνση έχει επικρατήσει να λέγεται ότι *ο δείκτης «δείχνει» στη διεύθυνση*. Στη δήλωση δείκτη ο **<τύπος_δεδομένων>** αφορά στο είδος των δεδομένων που αποθηκεύονται στη διεύθυνση που «δείχνει» ο δείκτης. Ο τύπος της κανονικής μεταβλητής πρέπει να δηλώνεται 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 10.1 απεικονίζεται ο τρόπος λειτουργίας των δεικτών, με τη **x** να είναι μία κανονική μεταβλητή και τον **px** να είναι δείκτης που δείχνει στη **x**.



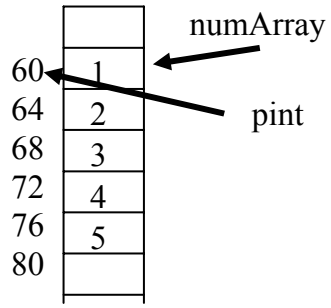
Σχ. 10.1 *Απεικόνιση του τρόπου λειτουργίας των δεικτών*

10.3 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 10.2 απεικονίζεται η διαδικασία ανάθεσης τιμής στο δείκτη **pint**.

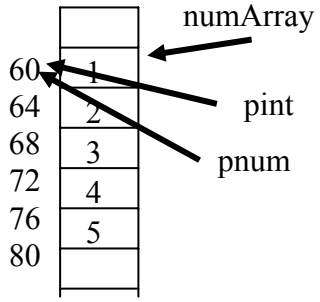
```
int  numArray[5] = {1,2,3,4,5};
int *pint;
pint = numArray;
```



Σχ. 10.2 *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 10.3.

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



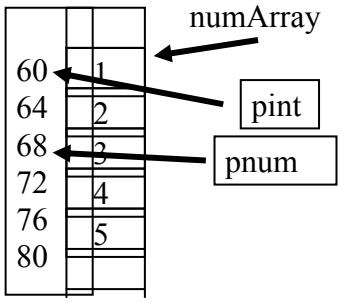
Σχ. 10.3 *Ανάθεση τιμής σε δείκτη με χρήση άλλων δεικτών ίδιου τύπου*

- **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** 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. 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 10.4.

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



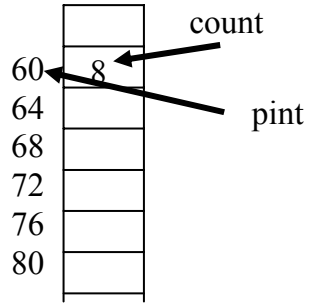
Σχ. 10.4 *Ανάθεση τιμής σε δείκτη με χρήση αριθμητικής δεικτών*

- **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 10.5.

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



Σχ. 10.5 *Ανάθεση τιμής σε δείκτη με χρήση του τελεστή διεύθυνσης*

Παρατήρηση: 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).

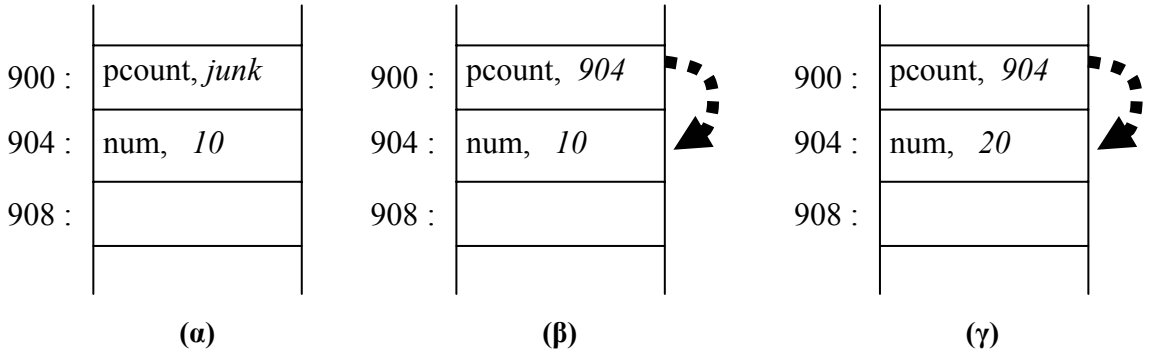
10.4 Προσπέλαση μεταβλητής με χρήση δείκτη

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 10.6α 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 10.6β). 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 10.6γ).



Σχ. 10.6 *Προσπέλαση μεταβλητής με χρήση δείκτη*

Παράδειγμα 10.1

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 10.7α).

➢ *Γραμμή 2*: Δήλωση πίνακα ακεραίων πέντε στοιχείων (**a**) και αρχικοποίηση αυτού (σχήμα 10.7β).

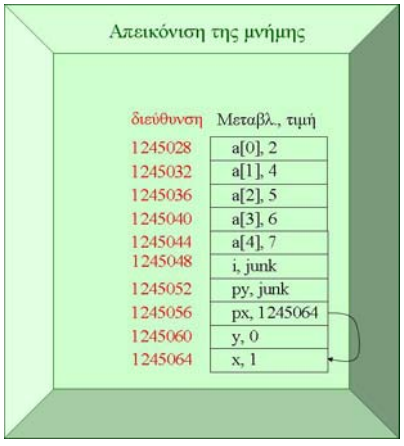
Σχ. 10.7α

Σχ. 10.7β

➢ *Γραμμή 3*: Δήλωση ακέραιας μεταβλητής (**i**) (σχήμα 10.7γ).

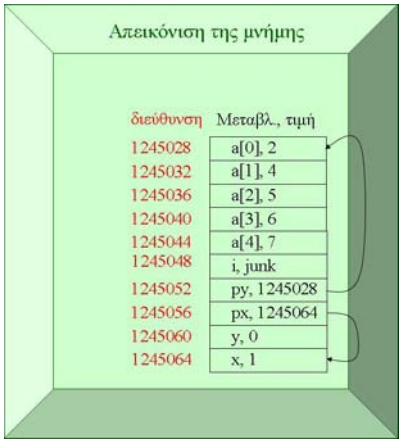
➢ *Γραμμή 4*: Ανάθεση της διεύθυνσης της μεταβλητής **x** στο δείκτη **px**, δηλαδή το περιεχόμενο του **px** είναι η διεύθυνση - το πρώτο byte – του **x** (σχήμα 10.7δ).

Σχ. 10.7γ



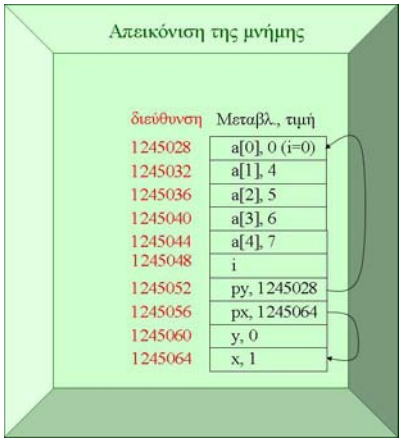
Σχ. 10.7δ

➢ *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 10.7ε).

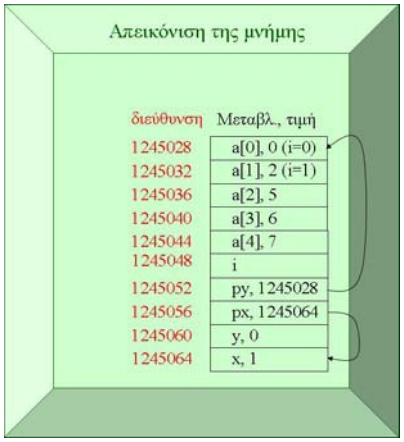


Σχ. 10.7ε

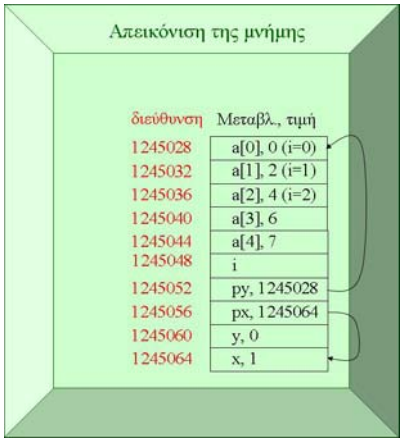
➢ *Γραμμή 6*: Επαναληπτική έκφραση, σε κάθε επανάληψη της οποίας λαμβάνουν χώρα τα ακόλουθα: Η τιμή **2*i** τοποθετείται σε θέση μνήμης με διεύθυνση **2*i** bytes μετά τη διεύθυνση που δείχνει ο **py** (σχήματα 10.7στ–10.7ι). Για παράδειγμα, εάν **i=3**, η τιμή **6** αποθηκεύεται στη θέση μνήμης **py+i=1245028+3*4=1245028+12=1245040**, θέση που καταλαμβάνει το στοιχείο **a[3]**. Με αυτόν τον τρόπο αποδίδεται στο **a[3]** η τιμή **6**.



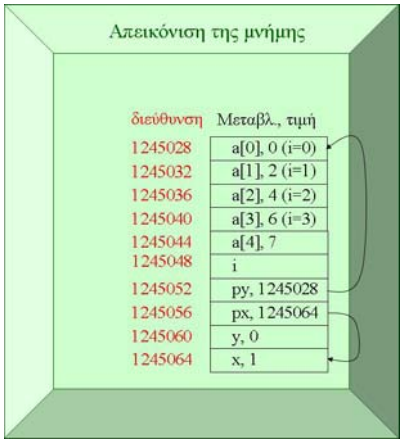
Σχ. 10.7στ



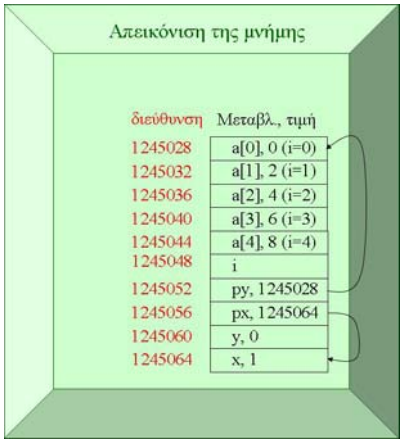
Σχ. 10.7ζ



Σχ. 10.7η



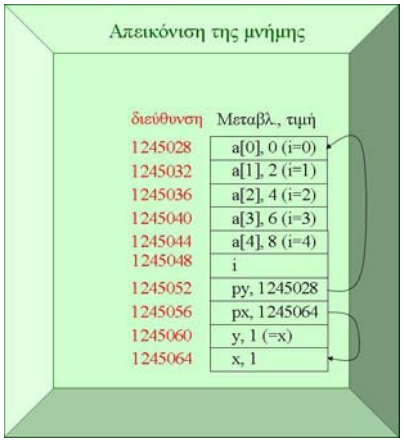
Σχ. 10.7θ



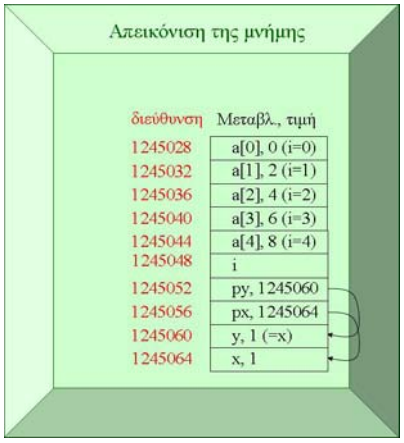
Σχ. 10.7ι

➢ *Γραμμή 7:* Το περιεχόμενο της θέσης στην οποία δείχνει ο **px**, δηλαδή το **1**, αποδίδεται στο **y** (σχήμα 10.7ια).

➢ *Γραμμή 8:* Ανάθεση της διεύθυνσης της μεταβλητής **y** στο δείκτη **py** (σχήμα 10.7ιβ).

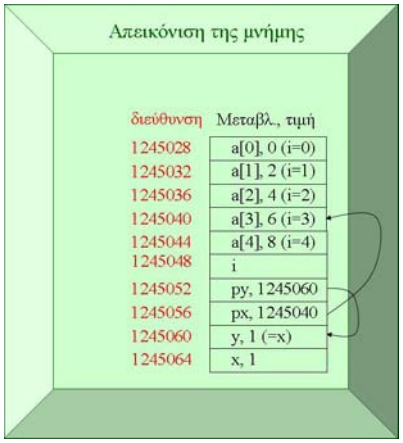


Σχ. 10.7ια



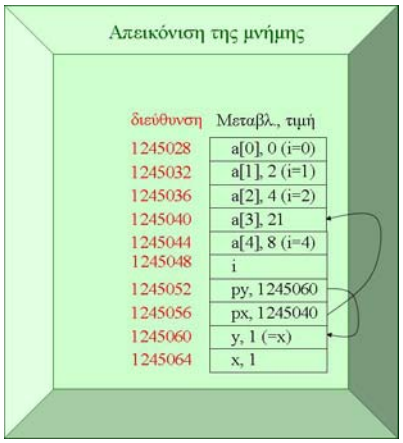
Σχ. 10.7ιβ

➢ *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 10.7ιγ).



Σχ. 10.7ιγ

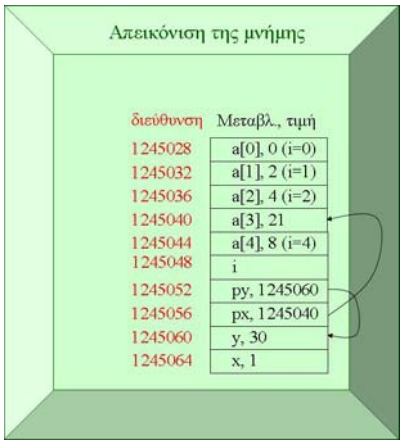
➢ *Γραμμή 10*: Το περιεχόμενο της διεύθυνσης στην οποία δείχνει ο **px** γίνεται **21** (σχήμα 10.7ιδ).



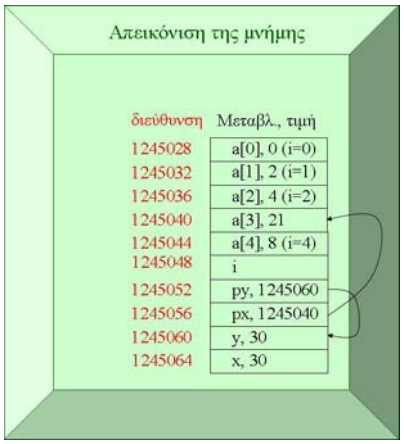
Σχ. 10.7ιδ

➢ *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 10.7ιε).

➢ *Γραμμή 12*: Η μεταβλητή **x** γίνεται ίση με τη μεταβλητή **y** (σχήμα 10.7ιστ).



Σχ. 10.7ιε



Σχ. 10.7ιστ

10.5 Δείκτες και συναρτήσεις

10.5.1 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 10.3, 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.

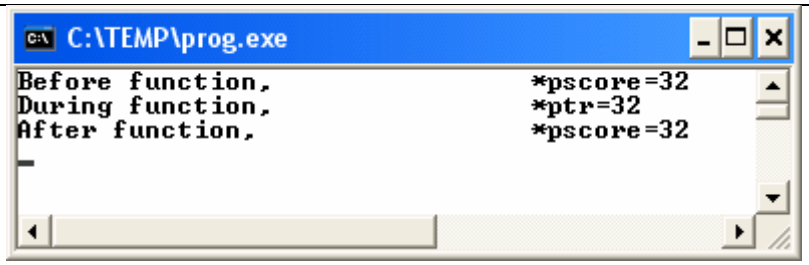
Παρατήρηση: Μπορεί να χρησιμοποιηθεί δείκτης για να αλλαχθεί το περιεχόμενο της θέσης στην οποία δείχνει, αλλά δεν πρέπει να αλλαχθεί ο ίδιος ο δείκτης μέσα στην καλούμενη συνάρτηση. Ο λόγος είναι ότι οι πραγματικές παράμετροι που είναι δείκτες αντιγράφουν μία διεύθυνση στις παραμέτρους της συνάρτησης,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

Παράδειγμα 10.2

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;
}
```



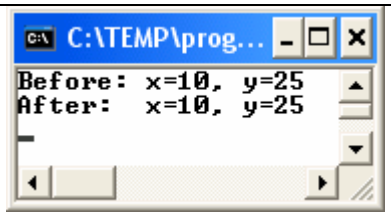
Αρχικά δηλώνονται η ακέραια μεταβλητή **num**, η οποία λαμβάνει την τιμή **32**, και ο δείκτης σε ακέραιο **pscore**. Ακολούθως στον **pscore** ανατίθεται η διεύθυνση της **num** και στη συνέχεια καλείται η συνάρτηση ***print()*** με πραγματικό όρισμα τον **pscore**. Η τυπική παράμετρος της ***print()***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

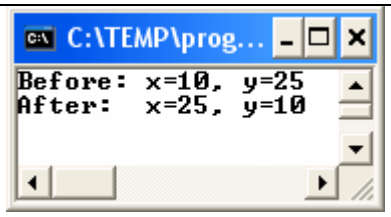
Παράδειγμα 10.3

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);
}
void swap(int a, int b)
{
   int temp=a;
   a=b;
   b=temp;
}
```




```
#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 *pa, int *pb)
{
   int temp=*pa;
   *pa=*pb;
   *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

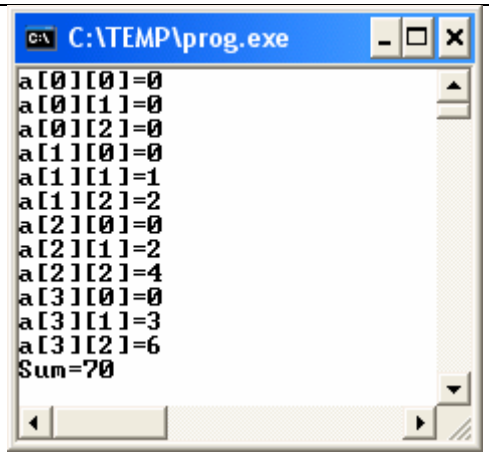
μεταβλητή **temp**, η θέση που αντιστοιχεί στη **x** αποκτά το περιεχόμενο της θέσης που αντιστοιχεί στην **y** και τανάπαλιν. Μετά το πέρας της ***swap()*** παύουν να υφίστανται οι δείκτες **pa** και **pb** και το έργο της αντιμετάθεσης έχει επιτευχθεί, όπως φανερώνουν και τα αποτελέσματα.

Παράδειγμα 10.4

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) );
    }
```

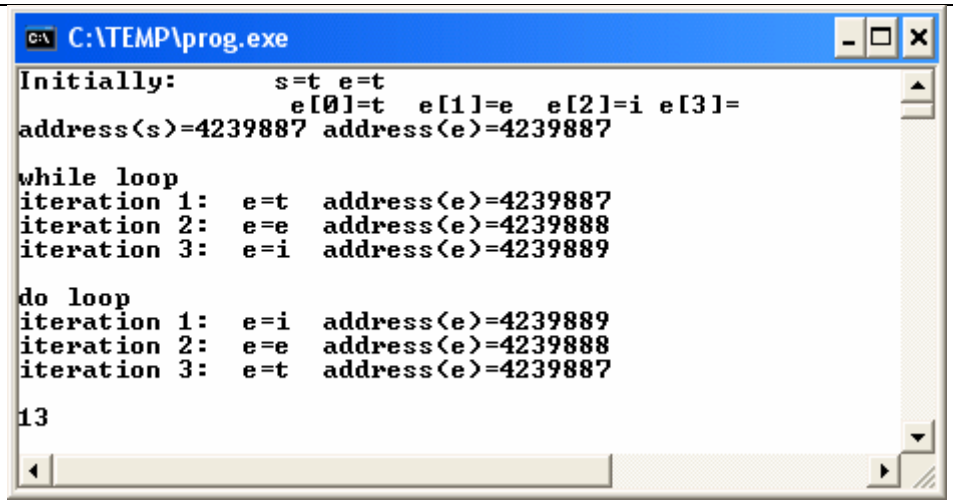


Παράδειγμα 10.5

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
```



➢ 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.

10.5.2 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 10.6.

Παράδειγμα 10.6

```
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()***.

10.6 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 (*).

10.6.1 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:

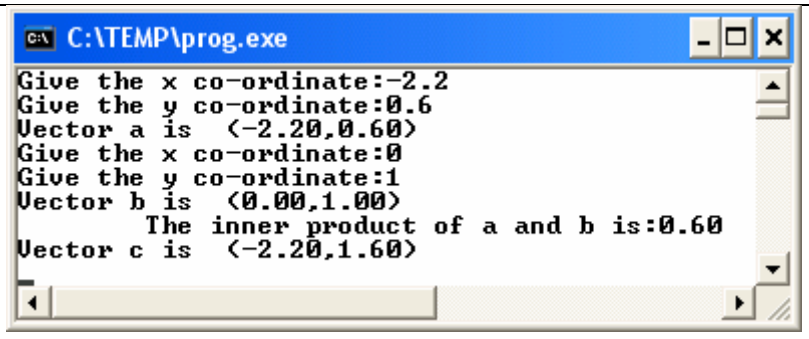
***(deikt.point1)=13;**

Παράδειγμα 10.7

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
```



Παράδειγμα 10.8

Να αναπτυχθεί πρόγραμμα που να λαμβάνει από το πληκτρολόγιο τα στοιχεία ενός εργαζόμενου και να δημιουργεί πίνακα εργαζόμενων, με τύπο δεδομένου κατάλληλη

δομή. Η διαδικασία θα επαναλαμβάνεται για 10 διαφορετικούς εργαζόμενους, όσο είναι και το μέγεθος του πίνακα. Οι πληροφορίες που διαβάζονται για κάθε εργαζόμενο είναι:

Ονοματεπώνυμο: Όνομα και επώνυμο ξεχωριστά (*σε μεταβλητή τύπου δομής*)

Διεύθυνση: Όνομα οδού, αριθμός οδού, πόλη, ταχ. Κώδικας (*σε μεταβλητή τύπου δομής*)

Τηλέφωνα: Τηλέφωνο εργασίας, κινητό (*σε μεταβλητή τύπου δομής*)

Θέση: Τίτλος, κωδικός αριθμός εργαζόμενου, τομέας της επιχείρησης στον οποίο εργάζεται, αριθμός γραφείου, ονοματεπώνυμο προϊσταμένου, ημερομηνία πρόσληψης (ημέρα, μήνας, έτος), μισθός (*σε μεταβλητή τύπου δομής – θα απαιτηθεί ένθετη δομή για την ημερομηνία πρόσληψης*)

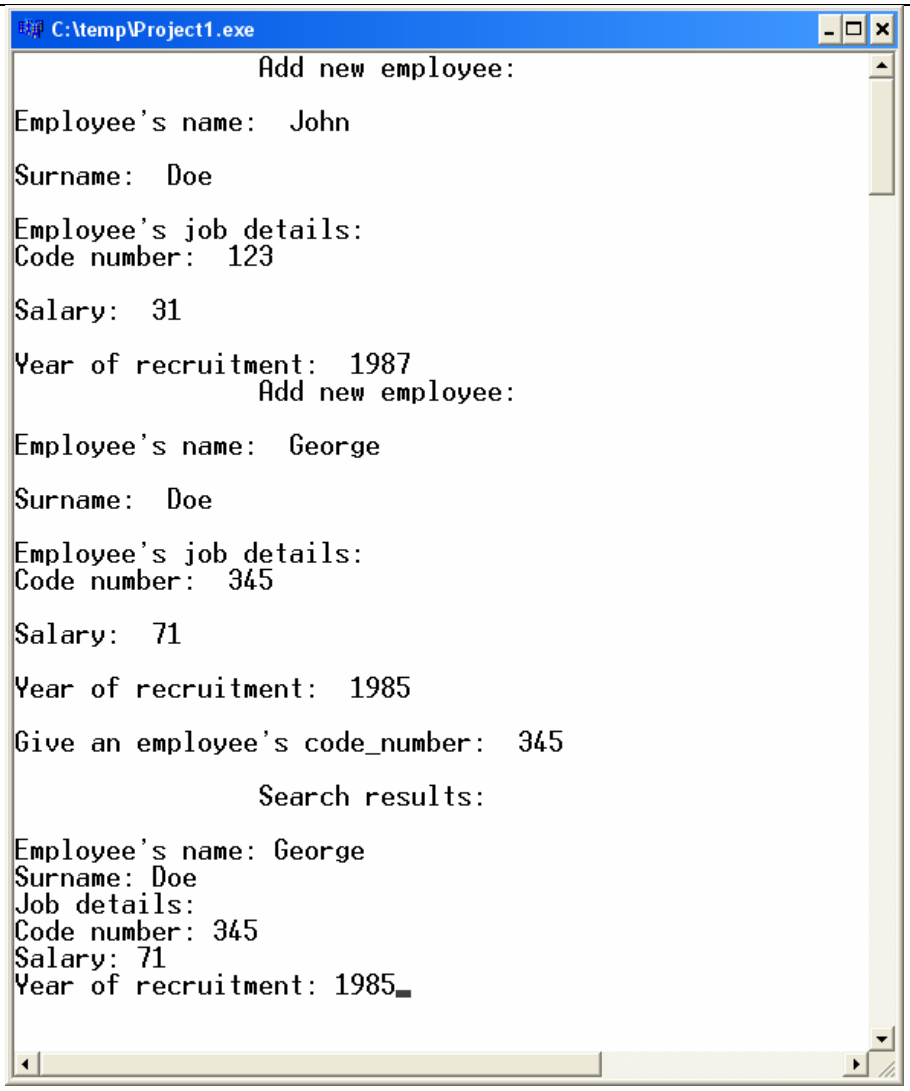
Στη συνέχεια να δίνεται κάποιος κωδικός αριθμός εργαζόμενου από το πληκτρολόγιο και να αναζητείται στον πίνακα. Αν υπάρχει, τότε να εμφανίζονται στην οθόνη οι πληροφορίες του αντίστοιχου εργαζόμενου, αλλιώς να εμφανίζεται ένα ανάλογο μήνυμα.

```
#include<conio.h>
#include<stdio.h>
#define N 10  //Αριθμός υπαλλήλων
struct nmT
{
    char name[40],surname[40];
};
struct addressT
{
    char street_name[40], city[40];
    int street_number,zip_code;
};
struct teleT
{
    char office_number[14],home_number[14];
};
struct hiredateT
{
    int year,month,date;
```

```
};
struct job_descriptionT
{
      char title[40], sector[100], boss_name[40];
      int code_number,office_number,salary;
      hiredateT hire;
};
struct personnelT
{
      nmT nm;
      addressT addr;
      teleT tele;
      job_descriptionT job;
};

void get_employee(personnelT *pers_ptr);
void search_employee(int i, personnelT person[N]);

main()
{
      personnelT pers[N];
      int i;
      for (i=0;i<N;i++) get_employee(&pers[i]);
      printf("\nGive an employee's code_number:  ");
      scanf("%d",&i);
      search_employee(i,pers);
}

void get_employee(personnelT *pers_ptr)
{
      printf("\t\tAdd new employee:\n");
      printf("\nEmployee's name:  ");
```

```
        scanf("%s",pers_ptr->nm.name);
        printf("\nSurname:  ");
        scanf("%s",pers_ptr->nm.surname);
        printf("\t\nEmployee's job details:");
        printf("\nCode number:  ");
        scanf("%d",&pers_ptr->job.code_number);
        printf("\nSalary:  ");
        scanf("%d",&pers_ptr->job.salary);
        printf("\nYear of recruitment:  ");
        scanf("%d",&pers_ptr->job.hire.year);
}

void search_employee(int code, personnelT person[N])
{
        int j=0;
        int index=-1;
        while ((j<N) && (index==-1))
        {
                if (code==person[j].job.code_number) index=j;
                j++;
        }
        if (index==-1)
          printf("\nThe given code does not match to an existing one\n");
        else
        {
                printf("\n\t\tSearch results:\n");
                printf("\nEmployee's name: %s",person[index].nm.name);
                printf("\nSurname: %s",person[index].nm.surname);
                printf("\t\nJob details:");
                printf("\nCode numer: ");
                printf("%d",person[index].job.code_number);
                printf("\nSalary: %d",person[index].job.salary);
```

```
            printf("\nYear of recruitment: ");
            printf("%d",person[index].job.hire.year);
        }
    }
```



Παράδειγμα 10.9

Να γραφεί πρόγραμμα κωδικοποίησης δεδομένων, το οποίο θα επιτελεί τα παρακάτω:

Θα διαβάζει ***N*** τετραψήφιους αριθμούς $x = x_3x_2x_1x_0$ από το πληκτρολόγιο. Μετά από κάθε ανάγνωση αριθμού, ο x θα μετασχηματίζεται στον κωδικοποιημένο αριθμό y, ο οποίος θα εμφανίζεται στην οθόνη. Ακολουθείται η εξής μέθοδος μετασχηματισμού:

- Για κάθε αριθμό της μορφής $x_3x_2x_1x_0$, όπου ως x_k, k=3,2,1,0 συμβολίζονται τα τέσσερα ψηφία, υπολογίζεται ο αριθμός $y_3y_2y_1y_0$ όπου το ψηφίο y_k προκύπτει ως το υπόλοιπο της διαίρεσης του αριθμού (x_k +7) με το 10.
- Ακολούθως αντιμετατίθεται το πρώτο ψηφίο του αριθμού $y_3y_2y_1y_0$ με το τρίτο και το δεύτερο με το τέταρτο. Κατά συνέπεια ο αριθμός $x_3x_2x_1x_0$ μετασχηματίζεται στον αριθμό $y_1y_0y_3y_2$.

Ο μετασχηματισμός θα υλοποιηθεί με τη συνάρτηση ***int transform(int x)***, η οποία θα δέχεται τον αριθμό $x_3x_2x_1x_0$ και θα επιστρέφει τον αριθμό $y_1y_0y_3y_2$. Μέσα στη συνάρτηση ***transform()*** θα χρησιμοποιηθούν οι ακόλουθες συναρτήσεις:

α) Η συνάρτηση ***void give_digits(int x, int *arr)***, η οποία δέχεται ως εισόδους έναν τετραψήφιο θετικό ακέραιο $x_3x_2x_1x_0$ και έναν δείκτη που δείχνει σε πίνακα τεσσάρων θέσεων, στον οποίο αποθηκεύονται τα ψηφία του αριθμού $x_3x_2x_1x_0$.

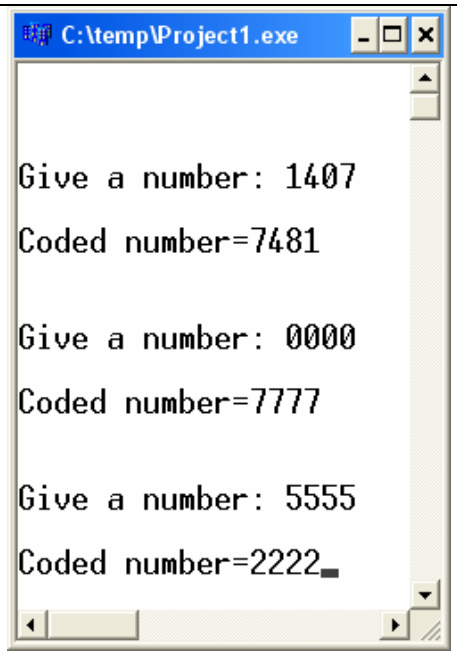
β) Η συνάρτηση ***void swap_digits(int *arr)***, η οποία δέχεται ως είσοδο δείκτη, ο οποίος δείχνει στον πίνακα που βρίσκονται αποθηκευμένα τα ψηφία y_0, y_1, y_2, y_3 και αντιμεταθέτει τις τιμές τους σύμφωνα με την ανωτέρω μέθοδο μετασχηματισμού.

β) Η συνάρτηση ***int get_digits(int *arr)***, η οποία δέχεται ως είσοδο δείκτη που δείχνει σε πίνακα τεσσάρων θέσεων, στον οποίο βρίσκονται αποθηκευμένα τα τέσσερα ψηφία y_0, y_1, y_2, y_3, και επιστρέφει τον αριθμό $y_3y_2y_1y_0$.

```
#include <stdio.h>
#define N 3
void give_digits(int x, int *arr);
int get_digits(int *arr);
int transform(int x);
void swap_digits(int *arr);
void main()
```

```
{
    int x,y,i=0;
    for (i=0;i<N;i++)
    {
        printf("\n\n\nGive a number: ");
        scanf("%d",&x);
        printf("\nCoded number=%d",transform(x));
    }
}

void give_digits(int x, int *arr)
{
    int y;
    arr[3]=x/1000;
    y=x%1000;
    arr[2]=y/100;
    y=y%100;
    arr[1]=y/10;
    arr[0]=y%10;
}

int get_digits(int *arr)
{
    return(arr[0]+arr[1]*10+arr[2]*100+arr[3]*1000);
}

int transform(int x)
{
    int y,arr_x[4],arr_y[4],i,temp;
    give_digits(x,arr_x);
    for (i=0;i<4;i++) arr_y[i]=(arr_x[i]+7)%10;
    swap_digits(arr_y);
```

```
        return(get_digits(arr_y));
}

void swap_digits(int *arr)
{
        int temp;
        temp=arr[0];
        arr[0]=arr[2];
        arr[2]=temp;
        temp=arr[3];
        arr[3]=arr[1];
        arr[1]=temp;
}
```



10.7 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.

10.8 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 σχήμα 10.8, 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.

Σχ. 10.8

Παράδειγμα 10.10

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

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

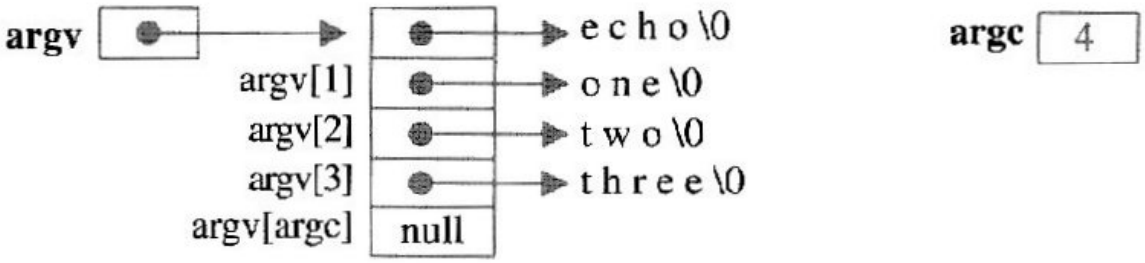
echo one two three

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)? " ":" " );
```

Κεφάλαιο 11

ΔΥΝΑΜΙΚΗ ΔΙΑΧΕΙΡΙΣΗ ΜΝΗΜΗΣ – ΔΕΙΚΤΕΣ ΣΕ ΑΛΦΑΡΙΘΜΗΤΙΚΑ

11.1 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.

11.2 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 <u>πάντοτε</u> 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 11.1

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;
    *(pstart+1)=28;
    *(pstart+2)=*(pstart+1)+12;
    free(pstart);
}
```

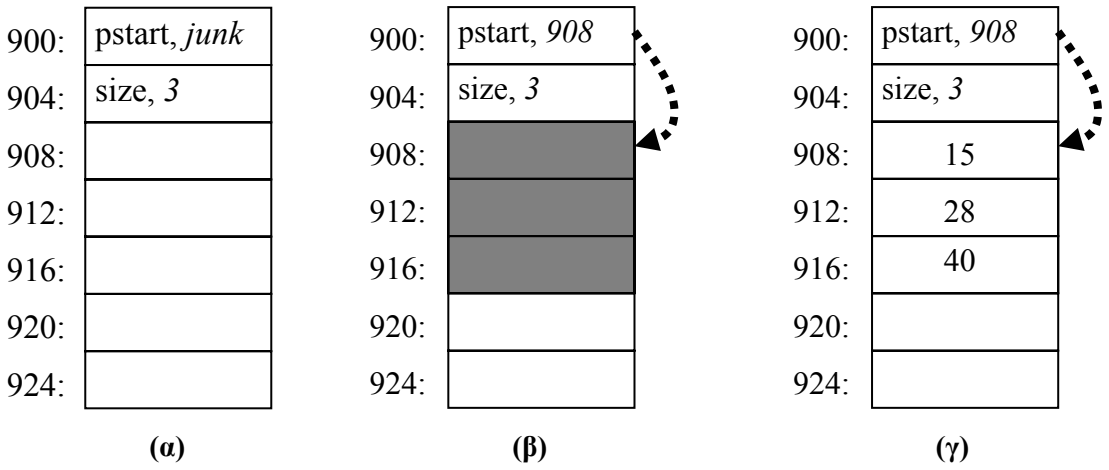
➢ Δηλώνονται ο δείκτης σε ακέραιο **pstart** και η ακέραια μεταβλητή **size**, η οποία αρχικοποιείται λαμβάνοντας την τιμή **3** (σχήμα 11.1α).

➢ 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 11.1β).

➢ 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 11.1γ).

➢ Με χρήση της ***free()*** απελευθερώνεται η δεσμευθείσα μνήμη και ο χάρτης μνήμης

επανέρχεται στην αρχική του μορφή (σχήμα 11.1α).



Σχ. 11.1 *Χάρτης μνήμης*

11.3 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()***.

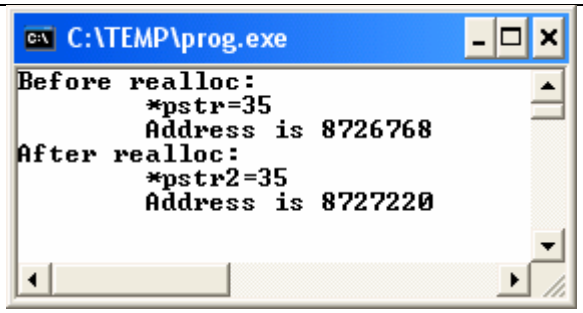
Η λειτουργία της ***realloc()*** αναδεικνύεται μέσα από το παράδειγμα 11.2.

Παράδειγμα 11.2

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);
}
```



11.4 Πίνακες δεικτών

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

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

Η πρόταση

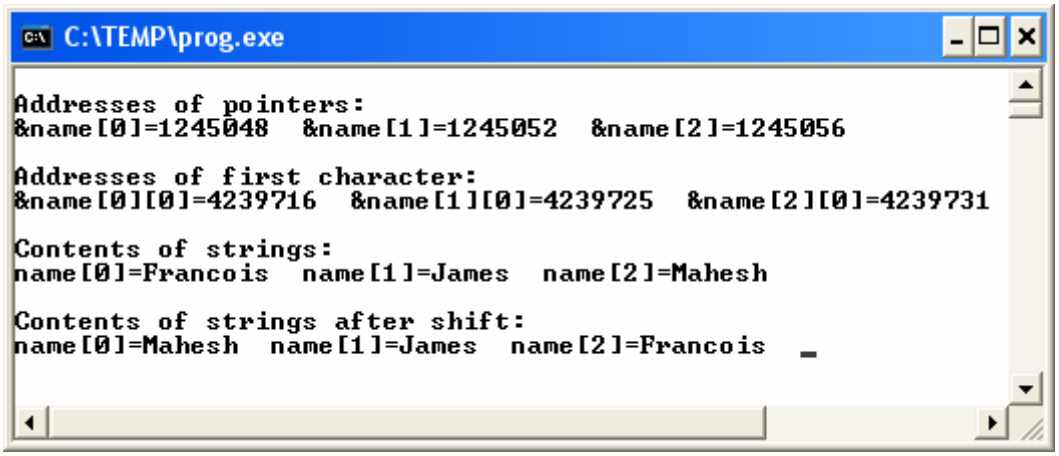
char *name[3];

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 §11.6.

Παράδειγμα 11.3

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"**.

11.5 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.

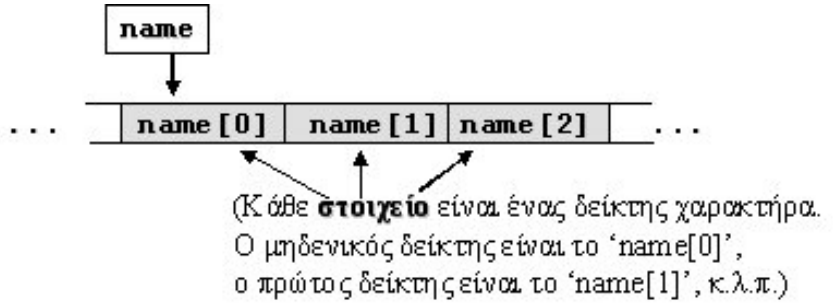
Η έμμεση αναφορά μπορεί να λάβει ένθεση οιουδήποτε βάθους (δείκτης σε δείκτη σε δείκτη κ.λ.π.), ωστόσο θα πρέπει να αποφεύγονται οι υπερβολές γιατί ο κώδικας αφενός μεν θα γίνει δυσανάγνωστος αφετέρου δε θα είναι επιρρεπής σε σφάλματα.

Για την ανάλυση της λειτουργίας των δεικτών σε δείκτες θα χρησιμοποιηθεί το

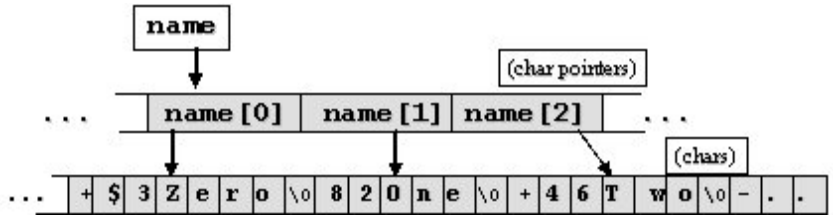
ακόλουθο πρόγραμμα, όπου μελετάται η περίπτωση ***δεικτών σε αλφαριθμητικά***:

```
      #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**.

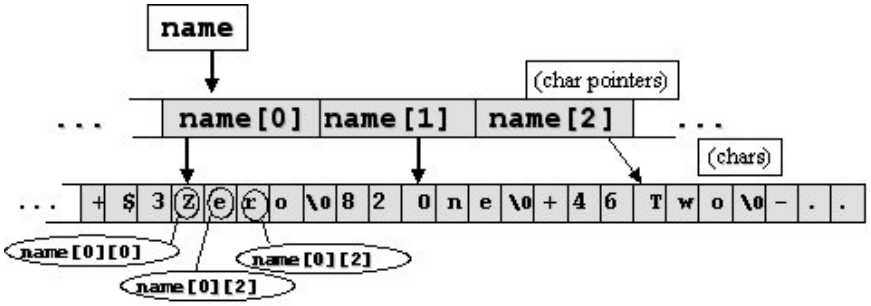


- *Γραμμές 3-5*: Κάθε στοιχείο του πίνακα **name** δείχνει σε μία αλφαριθμητική σταθερά. Τα τρία αλφαριθμητικά δεν είναι κατ' ανάγκη τοποθετημένα διαδοχικά στη μνήμη:



➢ *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.

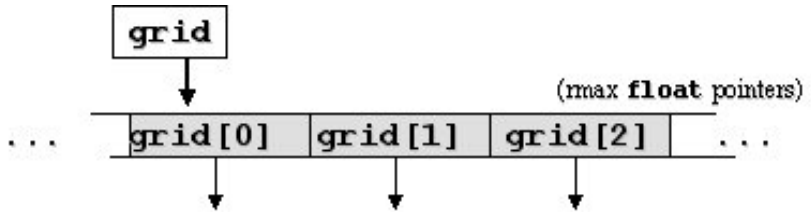
11.5.1 Πολυδιάστατοι πίνακες με δεδομένα αριθμητικών τύπων

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:

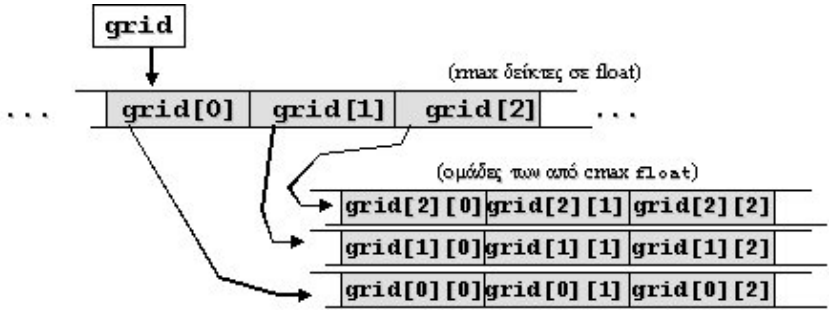
```
#include <stdio.h>
void main()
```

```
        {
            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.

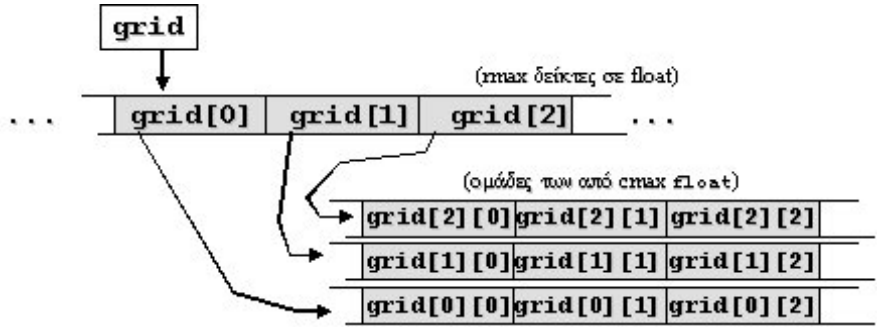


- *Γραμμή 3:* Δημιουργείται ένα μπλοκ από **cmax** αριθμούς κινητής υποδιαστολής για κάθε δείκτη σε float.



- *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 11.4

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 );
```

11.6 Συναρτήσεις οριζόμενες από το χρήστη για τη δέσμευση/αποδέσμευση μνήμης

Στην περίπτωση που στο πρόγραμμα γίνεται επανειλημμένα δέσμευση και αποδέσμευση μνήμης, μπορούν να ορισθούν συναρτήσεις που θα δεσμεύουν και θα αποδεσμεύουν μνήμη για πίνακες float, int κ.λ.π.

Οι συναρτήσεις δέσμευσης μνήμης θα έχουν παραμέτρους τα μεγέθη της μνήμης που ζητείται να δεσμευθεί και θα επιστρέφουν το δείκτη που θα διαχειρίζεται τη μνήμη. Οι συναρτήσεις αποδέσμευσης μνήμης θα έχουν παραμέτρους το δείκτη που διαχειρίσθηκε τη μνήμη και το μέγεθος αυτής. Δε θα έχουν επιστρεφόμενη τιμή.

Παράδειγμα 11.5

Οι ακόλουθες συναρτήσεις δεσμεύουν/αποδεσμεύουν μνήμη για ένα δισδιάστατο πίνακα αριθμών κινητής υποδιαστολής:

```
float **allocate_2(int size1, int size2)
{
      int i;
      float **deikt;
      deikt=(float **)malloc(size1*sizeof(float *)); assert(deikt!=NULL);
      for (i=0;i<size1;i++)
      {
            deikt[i]=(float *)malloc(size2*sizeof(float));
            assert(deikt[i]!=NULL);
      }
      return(deikt);
}

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

Καλούνται ως εξής:

```
float **s;
s=allocate_2(4,2000);        // δέσμευση μνήμης για πίνακα s[4][2000]
free_2(s,4);                 // αποδέσμευση μνήμης του πίνακα s[4][2000]
```

11.7 Δείκτες και συναρτήσεις αλφαριθμητικών

11.7.1 Η συνάρτηση εύρεσης χαρακτήρα σε αλφαριθμητικό

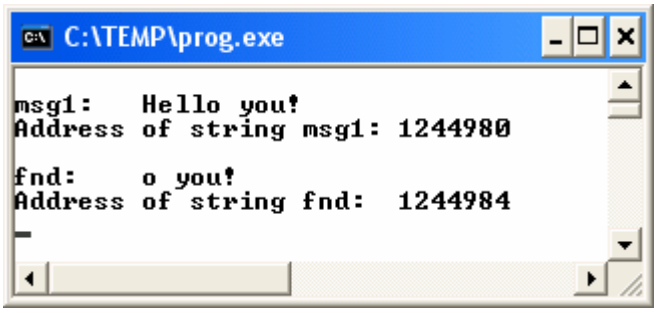
Η συνάρτηση ***strchr(str1,ch)*** βρίσκει το χαρακτήρα **'ch'** μέσα στο string **str1**. Στην πρώτη εύρεση του **'ch'** επιστρέφει ένα δείκτη σε χαρακτήρα. Δέχεται δύο ορίσματα: το πρώτο όρισμα είναι το όνομα του αλφαριθμητικού και το δεύτερο όρισμα είναι ο χαρακτήρας. Η συνάρτηση ***strchr(str1,ch)*** 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 11.6

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 );
}
```



11.7.2 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 11.7

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 11.8

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++;
    }
}
```

Κεφάλαιο 12

ΑΡΧΕΙΑ

12.1 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.

12.1.1 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.

12.1.2 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).

12.1.3 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.

12.2 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.

12.3 Ανάγνωση – εγγραφή χαρακτήρων σε αρχεία

12.3.1 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**.

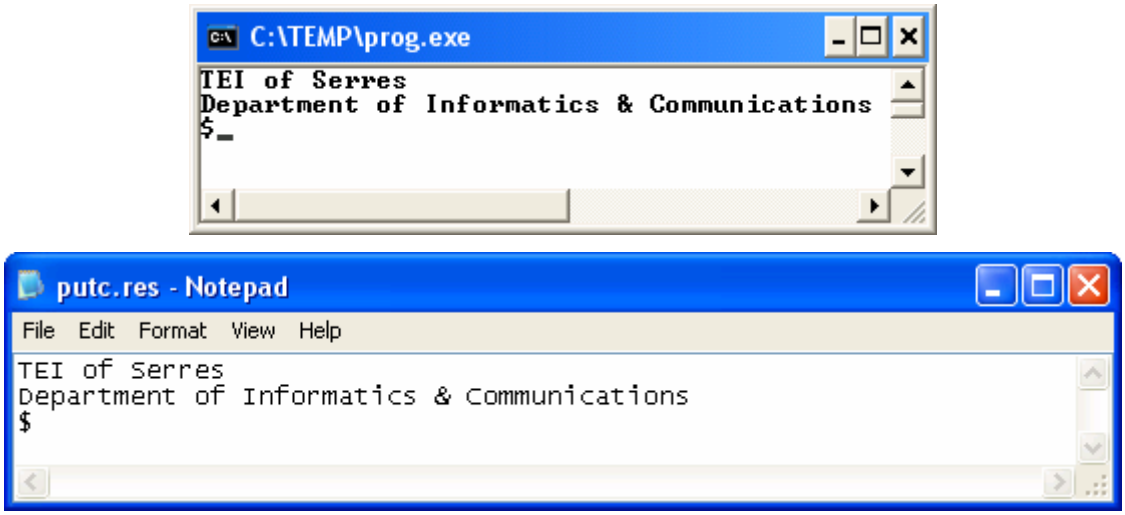
Παράδειγμα 12.1

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**.



12.3.2 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:

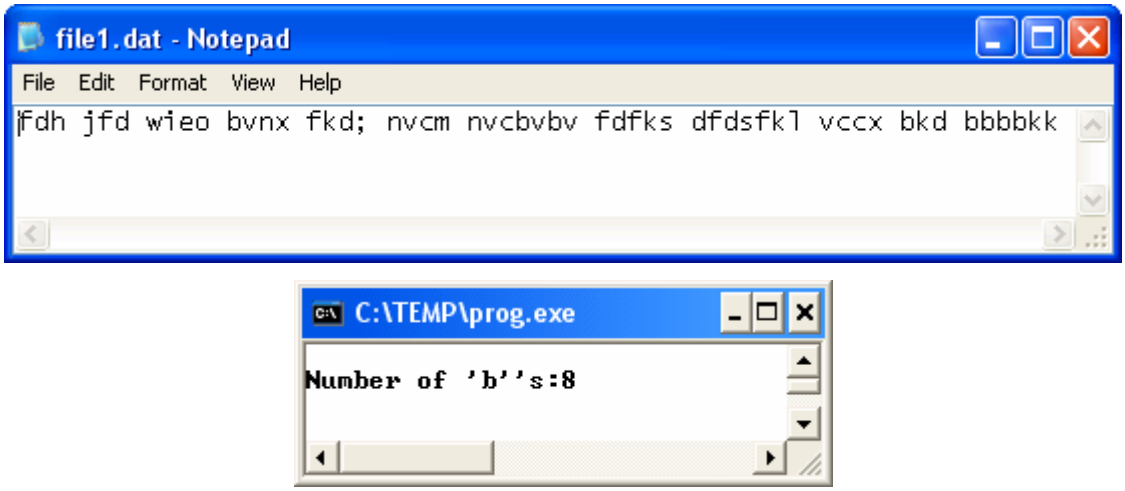
```
ch = getc(pF);
while (ch!=EOF)  ch=getc(pF);
```

Παράδειγμα 12.2

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 12.3

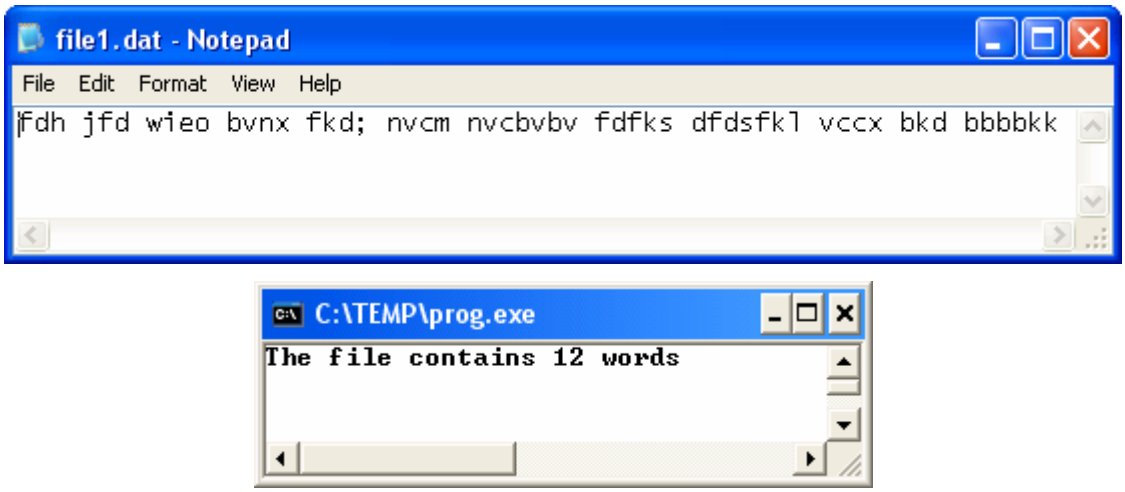
Να καταστρωθεί πρόγραμμα, το οποίο να δίνει τον αριθμό των λέξεων που περιέχονται σε ένα αρχείο ASCII. Το πρόγραμμα θα πρέπει να χειρίζεται τους *λευκούς χαρακτήρες* (κενά, νέες γραμμές, στηλοθέτες) ως πραγματικούς χαρακτήρες. Δηλαδή, εάν υπάρχει μία συμβολοσειρά από κενά ή χαρακτήρες επιστροφής, το 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.



12.4 Μορφοποιούμενες συναρτήσεις εισόδου – εξόδου σε αρχεία

12.4.1 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**.

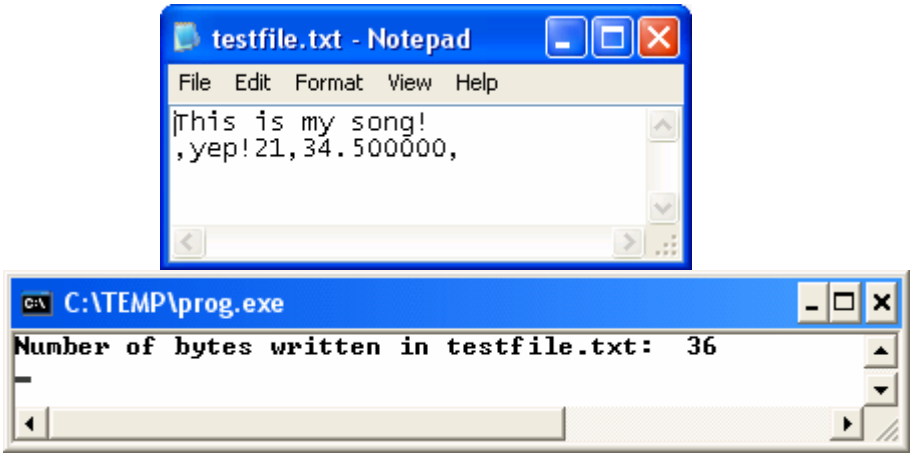
Παρατήρηση: Η **printf(*ορίσματα*);** ισοδυναμεί με την **fprintf(stdout, *ορίσματα*);**, δηλαδή με την ***fprintf()*** που έχει κανάλι εξόδου την οθόνη αντί για το αρχείο.

Παράδειγμα 12.4

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) στο τέλος της οποίας δηλώνεται αλλαγή γραμμής
- Τους χαρακτήρες **",yep!"** (5 bytes)
- Τον ακέραιο **21** (2 bytes)
- Το **κόμμα** (1 byte)
- Τον αριθμό κινητής υποδιαστολής **34.5** (9 bytes)
- Το **κόμμα** και την αλλαγή γραμμής (2 bytes)

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()***.

12.4.2 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.

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

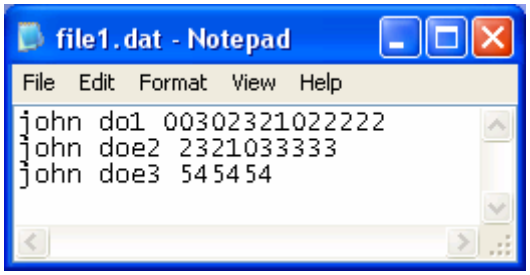
Παράδειγμα 12.5

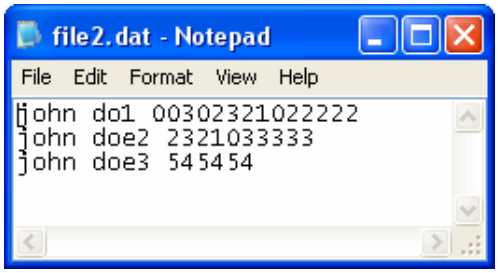
Να γραφεί πρόγραμμα, το οποίο θα δημιουργεί πίνακα 3 στοιχείων, με στοιχεία δομές. Κάθε δομή θα έχει ως μέλη το όνομα, το επώνυμο και το τηλέφωνο ενός ανθρώπου. Το πρόγραμμα θα διαβάζει τα περιεχόμενα του πίνακα από το αρχείο **file1.dat**, θα τα αποδίδει στον πίνακα και θα τα γράφει σε ένα άλλο αρχείο,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





12.5 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 άμεσος 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()***.

12.5.1 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

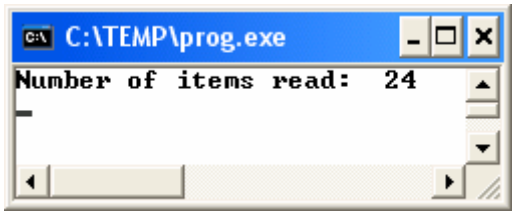
- **buffer** είναι ένας δείκτης σε μία περιοχή της μνήμης η οποία θα δεχθεί τα δεδομένα που διαβάζονται από το αρχείο.
- **length** είναι το μέγεθος του τύπου των δεδομένων που θα αναγνωσθούν. Για τον προσδιορισμό τους συνήθως χρησιμοποιείται η ***sizeof***.
- **num_items** είναι ο αριθμός των στοιχείων (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 12.6

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"**.

12.5.2 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 12.7

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 12.8

Να γραφεί κώδικας για το παράδειγμα 12.5 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 );
}
```

12.5.3 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);
```

12.6 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.

12.6.1 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 12.9

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 )
{
    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 )
{
    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 )
{
    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.

12.7 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 ολόκληρο το buffer 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.

12.8 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 12.10

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 $99^{ο}$ 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 12.11

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));
     }
```

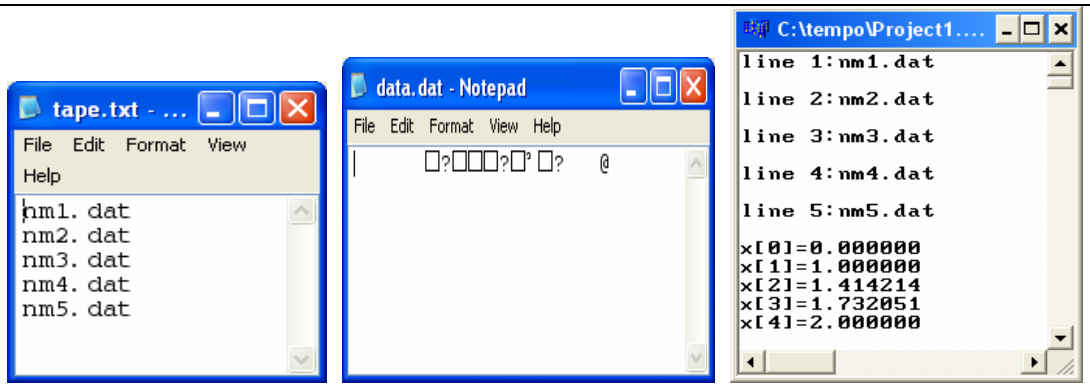
```
}
```

Παράδειγμα 12.12

Στον ακόλουθο κώδικα γίνεται χρήση των διαφόρων τρόπων ανάγνωσης/εκτύπωσης σε αρχείο.

```
#include<conio.h>
#include<stdio.h>
#include<math.h>
#define name "tape.txt"
void main()
{
    FILE *f1;
    char pchar[16];    int i;    float x[5];
    f1=fopen(name,"w");
    for (i=0;i<5;i++) fprintf(f1,"nm%d.dat\n",i+1);
    fclose(f1);
    f1=fopen(name,"r");   //fgets(): ανάγνωση γραμμή προς γραμμή
    for (i=0;i<5;i++) printf("line %d:%s\n",i+1,fgets(pchar,15,f1));
    fclose(f1);
    f1=fopen("data.dat","w");
    for (i=0;i<5;i++) x[i]=sqrt(i);
    fwrite(x,sizeof(x),1,f1);
    fclose(f1);   //Το f1 γίνεται δυαδικό αν και δεν μπήκε το "wb"
    f1=fopen("data.dat","r");
    for (i=0;i<5;i++)
    {
        fscanf(f1,"%f",&x[i]);
        //Η fscanf() διαβάζει τα στοιχεία, εναλλακτικά: fread()
        printf("x[%d]=%f\n",i,x[i]);
```

```
        }
        fclose(f1);
    }
```


tape.txt - ...
File Edit Format View
Help
nm1. dat
nm2. dat
nm3. dat
nm4. dat
nm5. dat
data.dat - Notepad
File Edit Format View Help
C:\tempo\Project1....
line 1:nm1.dat
line 2:nm2.dat
line 3:nm3.dat
line 4:nm4.dat
line 5:nm5.dat
x[0]=0.000000
x[1]=1.000000
x[2]=1.414214
x[3]=1.732051
x[4]=2.000000

ΒΙΒΛΙΟΓΡΑΦΙΑ

Α. Έντυπη

1. A. Alexandrescu, ***Modern C++ Design: Generic Programming and Design Patterns***, Addison-Wesley, 2001.
2. H.M. Deitel, P.J. Deitel, ***Ασκήσεις - Προγράμματα σε C***, Εκδόσεις Γκιούρδα, 2005.
3. H.M. Deitel, P.J. Deitel, ***C Προγραμματισμός***, Εκδόσεις Γκιούρδα, 2003.
4. H.M. Deitel, P.J. Deitel, ***C: How to Program***, Prentice-Hall, 2005.
5. A. Hunt, D. Thomas, ***The Pragmatic Programmer: From Journeyman to Master***, Addison-Wesley, 1999.
6. A. Kelley, I. Pohl, ***A Book on C***, Addison-Wesley, 1997.
7. B.W. Kernighan, Pipe, ***The Practice of Programming***, Addison-Wesley, 1999.
8. B.W. Kernighan, D.M. Ritchie, ***Η γλώσσα προγραμματισμού C***, Εκδόσεις Κλειδάριθμος, 1990.
9. K.N. King, ***C Programming: A Modern Approach***, W.W. Norton & Company, 1996.
10. D.E. Knuth, ***The Art of Computer Programming***, 3rd ed., Addison-Wesley, 1997.
11. R. Lafore, ***Χρήση και Προγραμματισμός Turbo C++***, Εκδόσεις Γκιούρδα, 1992.
12. S. Lohr, ***Go To***, Basic Books, 2001.
13. S. Prata, ***C++ Primer Plus***, 4th ed., SAMS, 2002.
14. E. Roberts, ***Η τέχνη και επιστήμη της C***, Εκδόσεις Κλειδάριθμος, 2005.
15. H. Schildt, ***Εγχειρίδιο εκμάθησης Turbo C***, Εκδόσεις Κλειδάριθμος, 1989.
16. B. Stroustrup, ***The C++ Programming Language***, 3rd ed., Addison-Wesley, 1997.
17. P. Van der Linden, ***Expert C Programming***, Prentice Hall, 1994.
18. M. Waite, S. Prata, ***C Βήμα προς Βήμα***, Εκδόσεις Γκιούρδα, 1996.

19. M. Waite, S. Prata, D. Martin, ***Πλήρης Οδηγός Χρήσης της C***, 6η έκδοση, Εκδόσεις Γκιούρδα, 2000.

---

20. Κλ. Θραμπουλίδης, ***Διαδικαστικός Προγραμματισμός – C (Τόμος Α)***, 2η έκδοση, Εκδόσεις Τζιόλα, 2002.
21. Π. Μαστοροκώστας, ***Προγραμματισμός Ι: Παρουσιάσεις Διαλέξεων***, ΤΕΙ Σερρών, 2005.
22. Π. Μαστοροκώστας, ***Προγραμματισμός ΙΙ: Παρουσιάσεις Διαλέξεων***, ΤΕΙ Σερρών, 2005.
23. Ν. Χατζηγιαννάκης, ***Η γλώσσα C σε βάθος***, Εκδόσεις Κλειδάριθμος, 2005.

Β. 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://www.csd.uoc.gr/~hy150/EssentialC.pdf
5. http://www.cuj.com/
6. http://pw1.netcom.com/~tjensen/ptr/pointers.htm
7. http://www.acm.uiuc.edu/webmonkeys/book/c_guide/